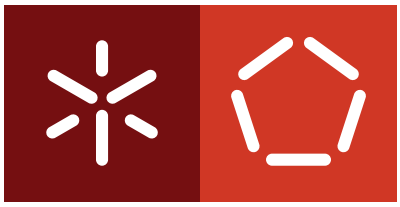
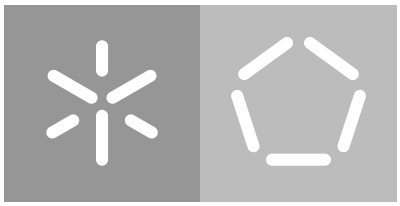


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Marcelo Miranda

Early Validation of System Requirements and Design

December 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Marcelo Miranda

Early Validation of System Requirements and Design

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Jorge Sousa Pinto

December 2019

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ACKNOWLEDGEMENTS

I wish to acknowledge the special moments and all the help provided by everyone during this last year and during my academic journey. A special thanks to Professor Jorge Sousa Pinto for the availability and support he granted, but also for our casual conversations. To United Technologies Research Center for the opportunity to join you during this dissertation, and all the people I met there for the great moments and beers we shared together. To Stelios Basagiannis for the support and to Georgios Giantamidis for the extensive guidance and all the *great* jokes. I want to thank all the professors that guided me during this journey, giving me the tools to better understand this subject that always marvelled me. Finally, I would like to thank my friends for all the truly great moments we shared, as well as the ones that are still to come. For all the support, for putting up with me, and for everything that shall remain unwritten in a dissertation.

ABSTRACT

Modern society is relying more and more on electronic devices, most of which are embedded systems and are sometimes responsible for performing safety-critical tasks. As the complexity of such systems increases due to concurrency concerns and real-time constraints, their design is more prone to errors which can lead to catastrophic outcomes. In order to reduce the risk of such outcomes, a model-based methodology is commonly used. The model describes the behaviour of the system and is subject to verification techniques such as simulation and model checking in order to verify it behaves according to the requirements. Common problems that arise with this methodology is the ambiguity of requirements written in natural language and the translation of a requirement to a property that can be verified along with the model.

This thesis proposes a tool that, after the translation of the requirements to temporal formalism, allows the automatic generation of monitors in order to verify the model. Our target platform is Simulink, which is widely used in this domain to model, simulate and analyze dynamic systems.

Keywords: formal methods, runtime monitoring, temporal logics, SALT, SpeAR

RESUMO

A sociedade de hoje depende cada vez mais de dispositivos eletrónicos, a maioria dos quais são sistemas embebidos e, por vezes, responsáveis pela realização de tarefas críticas. À medida que a complexidade destes sistemas aumenta devido a problemas de concorrência ou restrições de tempo real, o design torna-se mais suscetível a erros que podem levar a resultados catastróficos. A fim de reduzir estes riscos, recorre-se a uma metodologia de desenvolvimento baseada em modelos. O modelo descreve o comportamento do sistema e pode ser sujeito a técnicas de verificação, tais como simulação ou *model checking*, a fim de verificar que este exibe o comportamento descrito nos requisitos. Problemas comuns que surgem com esta metodologia devem-se à ambiguidade dos requisitos, tipicamente escritos em linguagem natural, e à tradução destes para uma propriedade que pode ser verificada em conjunto com o modelo.

Esta dissertação propõe uma ferramenta que, após a tradução dos requisitos para uma linguagem de especificação formal, permite a geração automática de monitores para verificar o modelo. A plataforma para a qual os monitores são gerados é o Simulink, que é tipicamente utilizado neste domínio para modelar, simular e analisar sistemas dinâmicos.

Palavras-chave: métodos formais, lógicas temporais, *runtime monitoring*, SALT, SpeAR

CONTENTS

1 INTRODUCTION 1
1.0.1 Motivation 2
1.0.2 Objectives 2
2 BACKGROUND 4
2.1 System requirements 4
2.1.1 Specifying requirements 5
2.2 Model based engineering 5
2.2.1 Runtime monitoring 6
2.2.2 Transition System 7
2.3 Temporal Logics 9
2.3.1 Linear Temporal Logic 11
2.3.2 Past LTL 12
2.3.3 Property Patterns 14
2.3.4 SpeAR 16
2.3.5 SALT 18
2.4 Simulink 19
3 THE MONGE TOOL 21
3.1 Tool overview - Requirement formalisation 22
3.2 Tool overview - Requirement extraction 23
3.3 Tool overview - Monitor example 24
3.4 Design decisions 25
4 MONGE ARCHITECTURE 27
4.1 Runtime verification with monitors 28
4.1.1 Deterministic Finite Automaton 31
4.1.2 Monitoring with Past Formulas 33
4.2 From a specification language to a monitor 37
4.2.1 SpeAR 38
4.2.2 SALT 40
4.3 Extraction DSL 42
4.4 Final results 43
5 CONCLUSION 46
5.1 Future work 46

LIST OF FIGURES

Figure 1	Comparison between testing and runtime monitoring.	7
Figure 2	Transition System	8
Figure 3	Patterns scopes. Figure reproduced from Dwyer et al. (1999).	15
Figure 4	Patterns hierarchy that includes real-time patterns. Figure based on Konrad and Cheng (2005).	16
Figure 5	Outport and Inport blocks	19
Figure 6	Goto and From blocks	20
Figure 7	Delay block	20
Figure 8	MATLAB Function block	20
Figure 9	Chart block	20
Figure 10	Example of a generated monitor with its different components highlighted.	25
Figure 11	Tool architecture	27
Figure 12	Example of a deterministic finite automaton.	32
Figure 13	Past LTL semantics for formula $H\ p$.	33
Figure 14	Recursive Past LTL semantics for formula $H\ p$.	34
Figure 15	Rough example of the process that formal specifications go through.	37

1

INTRODUCTION

Today's society relies more and more on electronics devices and software systems. In this thesis, we are specially concerned with the category of embedded systems. An embedded system has a very specialised role, limited resources, and is frequently bounded by real-time constraints. They are present in almost every aspect of our daily life, impacting the way we communicate, the way we travel and the way we handle our money.

Traditionally, the *waterfall model* was the reference methodology in systems engineering to develop such devices. This model describes the development of a product as a set of sequential steps where the results of each step serves as the foundations for the next step. Although multiple variations of the *waterfall model* exist in the literature, some phases are common across most of the variations. These are the requirements elicitation, the system design, the system implementation, testing and finally, the system maintenance. The *waterfall model* is based on the fact that careful thinking should go into the requirements and the system design since errors in this early steps are propagated to the next phases of development. As such, the later these errors are discovered, the more steps need to be reverted and hence, the more expensive it is to fix them.

Embedded systems engineering presently faces a big challenge, due to the fact that the complexity of embedded systems is continually increasing. These devices need, for instance, to interact with many other systems, which brings to the design concurrency concerns that are hard to reason about. With such complex systems, the assumption that the design engineer is able to take into account all possible scenarios without testing the system is unreasonable. During the implementation or the testing steps some design errors may be revealed forcing the reiteration of the previous steps. The inflexibility of the *waterfall model*, in which the implementation is the first trial the design must pass, makes it unsuitable for the development of such systems. It may be the case that errors remain undetected during the entire development phase and cause a system failure once the system is deployed. The failure of safety-critical systems such as medical devices, nuclear reactors or avionics, however, can be catastrophic and lead to the loss of both human lives and huge amounts of money.

To avoid these situations, safety-critical systems must be delivered with a high level of confidence. The early detection of errors in both the system requirements and the system design is therefore of the utmost importance.

1.0.1 *Motivation*

Nowadays, the reference methodology for the design of safety-critical systems is *model-based design*. This methodology consists of creating a model of the desired system, allowing the engineer to explore and test its design before starting the implementation. A model is a high-level abstraction of the system in which some details, such as implementation details, are discarded and thus not considered for analysis. This way, we end with a simpler version of our system which is tractable for computer analysis techniques such as model checking and simulation. These techniques can be used to explore the validity of the model early in the project and can detect subtle errors in the design. This has a huge impact in industries such as aerospace, nuclear energy research or medical devices where fully implementing and testing a product has a high cost or can be extremely dangerous.

The common approach to check if the system model is exhibiting the expected behaviour is through *unit testing*. However, in this dissertation, we make use of a more powerful technique to verify the system behaviour, *runtime monitoring*. While unit testing analysis the relation between the system inputs and outputs, runtime monitoring consists in the analysis of state information present in the simulation traces to determine if a given property holds. Our particular interest resides in *online monitoring* in which the state information is extracted from a running simulation and both the simulation and the monitors are executed in parallel. Furthermore, the monitors themselves can be automatically generated from requirements as long as they are written in a formal specification language, thus also checking the compliance of the design with the requirements.

System requirements, however, are commonly expressed using natural language instead. Natural language provides multiple obstacles for computer analysis and, for the purpose of automatic monitor generation, it is therefore necessary to encode the requirements in one of the specification languages supported by the methodology being used.

1.0.2 *Objectives*

The tool we propose in this dissertation intends to assist the engineer throughout the process of formalising system requirements, allowing the automatic generation of runtime monitors. The first problem we handle is the extraction of requirements from a document. To perform this initial task we provide a DSL to build *regular expression* patterns in a readable and reusable manner. The engineer should then be able to select the subset of

requirements he intends to formalise. The tool supports properties written in either Linear Temporal Logic (LTL) or Past Linear Temporal Logic (PLTL). Although these specification languages serve as the foundation for many of the tools used in formal verification, they can make the task of translating the requirements quite challenging, due to the amount of expertise they usually require. Thus, since our tool is directed at requirements engineers, we chose to also support high-level specification languages like Specification and Analysis of Requirements (SpeAR) (A. Fifarek, 2017) and Structured Assertion Language for Temporal Logic (SALT) (Streit, 2006) that have a more natural syntax and provide support for *specification patterns*. Specification patterns aim to reduce the complexity of writing formal properties by reusing known solutions to common requirements. It is an approach similar to the *design patterns* approach that is so widely used in the programming world.

With the chosen requirements formalised, the tool is able to automatically generate a *runtime monitor* for each property. Due to its wide usage for modelling and simulating automatic control systems and digital signal processing, we chose to target *Simulink*, developed by *Mathworks* for our monitor generator.

In Chapter 2 we present some key concepts such as requirements elicitation and requirements specification in Section 2.1. We follow up with Section 2.2 where we discuss the design methodology upon which we lay our work. Section 2.3 and Section 2.3.3 expose the current approaches to formalise requirements so that they become suitable for computable analysis. We finish this chapter with an overview of the *Simulink* features we make use of in Section 2.4.

Chapter 3 describes the solution we propose, going through the features we provide in Section 3.1. Chapter 4 exposes the inner works of the tool, describing how we process the requirements the engineer provides us until we are able to generate runtime monitors. We finish with Chapter 5 where we discuss the results we obtained and the aspects that we can still improve.

2

BACKGROUND

2.1 SYSTEM REQUIREMENTS

Requirements elicitation is the first step of development and establishes the groundwork for the future stages. It is the step in which the problem is stated and the solution is justified.

The purpose of system requirements is to state boundary conditions that will leave room for different possible designs, each with their respective trade-offs. They should frame the problem, gathering everything from functional to performance and security requirements, while avoiding describing how the system is to be implemented, as that is the purpose of the design. When defining requirements it is important to consider the context in which the context of the project is inserted as well as the existing processes that might be replaced by the system. By understanding how the system affects this context, we are able to integrate it seamlessly into its environment.

A well formed requirements document should explain what problem is being solved, what functions the system will have in order to solve that problem, as well as design constraints that specify how the system is to be built. The document is a way for the stakeholders to understand and agree on what is supposed to be developed. It should identify all the involved parts in the project, from consumers to developers, clients, sponsors, marketing and legal experts. It should comprise the expectations that all these parties have from the system and state all the assumptions and relevant facts that are necessary to provide background information for the specification readers. This way, a complete requirements document can serve as an agreement between all the stakeholders.

Requirements are usually *divided* into functional requirements and *non-functional* requirements. Functional requirements describe the behaviours the system should exhibit, detailing how the data it handles is transformed across the system and how the interactions with its environment are expected to take place. Non-functional is a broad term for all the other types of requirements such as *performance* requirements, *security* requirements, *legal* requirements, *architecture* requirements, etc. They include boundary conditions such as the maximum weight of the system or what type of assumptions can be made about the communication channels.

It is important to note that requirement elicitation is a complex process in which the captured requirements can have inconsistencies i.e. requirements that contradict each other. It may also be the case that the system is underspecified leaving room for ambiguities, or even overspecified in which case there is no possible design that meets all the requirements. It is already well known that the cost to fix errors increases as the project matures, which motivates the need for an early analysis of the design and its requirements. The requirements serve as a baseline for verification and validation of the system.

2.1.1 *Specifying requirements*

Requirements documents are typically written in natural language. This is not a surprise if we consider that requirements intend to serve as an agreement between all the stakeholders and natural language is the most common way of communicating ideas. Natural languages, however, are unstructured and have no underlying semantics. As such, they are too hard to analyse and are subject to ambiguities.

Formal languages, on the other hand, have well defined semantics with a mathematical background which, along their simple grammar, makes them suitable for computer analysis. Due to its ability to reason about the evolution of a system over time, temporal logic is the formal language of choice that is typically used to write requirements. A common obstacle for the adoption of temporal logic, however, is the high degree of expertise that is required to understand it. As a contrast to natural languages, formal languages can rapidly become hard to read and even harder to write.

Another approach to the writing of requirements are the structured natural languages which are obtained by restricting the grammar and vocabulary of a natural language in order to eliminate ambiguity and complexity. Some of these structured languages have a formal background and can be mapped to a formal language, enabling all the analysis that is possible with that formal language while keeping the readability of a natural language.

For the purpose of this thesis we consider that requirement documents are written in natural language and thus, to enable the generation of monitors from requirements, a preliminary step in which they are translated to a formal language is required.

2.2 MODEL BASED ENGINEERING

In traditional systems engineering, a design engineer writes the specification of the system and hands it to the software developers for them to implement the system. The specification document, however, may not convey correctly the design engineer's ideas or be misinterpreted, leading to undesired implementations. Such differences are only detected when the implementation is subject to a testing bench, in which errors in either the design or the im-

plementation are revealed. This leads to another design-implementation cycle, increasing the overall cost of the project.

In recent years, model based design is the preferred methodology for the design of embedded systems across multiple industries. The model describes the system at a high-level of abstraction and is not concerned with implementation details, allowing for an efficient analysis of the details with which we are concerned. In order for the model to be formally analysable, its semantics should be well defined and have no ambiguities. As such, modelling languages intended to be used for formal analysis should provide a rigorous underlying formal semantics.

There are multiple ways to represent a model, as well as multiple types of models depending on the characteristics of the system we want to analyse and the ones we want to abstract away.

Functional models are usually concerned with the dynamics of the system, the flow of data and the transformations it is subject to. Simulink, a block diagramming tool developed by Mathworks, is widely used for modelling dynamic systems. It allows engineers to simulate the behaviour of the model or formally verify that it does not violate some desired properties. By having an executable specification, the engineer can easily tweak the model and observe how the changes affect the behaviour of the system.

On the other hand, architecture models deal with properties of the physical system and the interactions between its components. Through an architecture model it is possible to verify that an embedded system does not surpass a certain physical weight or that the latency in flow of data is in an expected band. By modelling our system we have the possibility to better explore our design and detect design errors even before starting the implementation. Tools such as Simulink and AADL even allow for automatic code generation based on the model built. Both the early detection of errors and automatic code generation are characteristics of model-based design which have a great impact on the time to market of the product at hand. In this thesis we dedicate ourselves to automatic generation of runtime monitors, and as such we are concerned with functional models and functional requirements. Due to its wide adoption, we chose to target Simulink.

2.2.1 *Runtime monitoring*

Simulations allow the engineers to predict and explore the behaviour of a system, allowing them to understand it as well as detect potential failures even before starting to implement it. This is especially useful in areas like aerospace or medical equipment where building the real system can be too expensive or when an undetected defect can cause a potential hazard.

When a simulation is running, it is possible to extract information from it and check if certain properties are violated or hold, a technique known as *runtime monitoring*. *Offline* monitoring expects monitors to check formulas against simulation traces, while in *online* monitoring the monitor is running in parallel with the simulation. We classify the monitor as *active* if it also affects the simulation inputs and classify it as *passive* otherwise. Online monitoring increases the overall time of the running simulation but it can be used to stop a faulty simulation once a property is violated or, in real systems, can be used to prevent dangerous behaviour from happening.

The usual approach for runtime monitoring is to synthesise requirement models, i.e. monitors, and couple them to the system model in order to be able to query the execution state. Once the simulation is running the system is verified by the monitors which produce a verdict regarding weather the properties are satisfied or not. To synthesise such monitors, engineers write the properties in a formal specification which can then be automatically translated into the intended monitor.

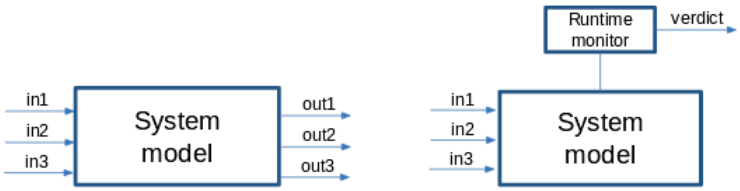


Figure 1: Comparison between testing and runtime monitoring.

The most common approach to detect errors during simulations is *unit testing*. By providing pairs of input/output it is possible to check if for a given input the simulation produces the expected output. For complex systems, however, it is not trivial to discover the correspondent output to a given input and thus this task would not be easily automated. Hence why we rely on runtime monitors instead which, by encoding the behaviour the system is expected to exhibit, is able to produce a verdict for all inputs.

2.2.2 *Transition System*

There are several approaches to modelling systems, depending on the properties of the system that we want to document or analyse. In the context of formal verification, systems are typically modelled with transition systems. A transition system is a graph that represents the behaviour of a system. Formally, it is defined by $TS = (S, Act, \rightarrow, I, AP, L)$ where

- S is the set of states,
- Act is the set of actions through which the system can evolve,

- $\rightarrow \subseteq S \times Act \times S$ is the transition relation that evolves the system by transitioning to the next state through an action,
- $I \subseteq S$ is the set of initial states,
- AP is the set of atomic propositions,
- $L : S \rightarrow 2^{AP}$ is the labelling function that describes which atomic propositions are satisfied at a given state.

The transition system starts in one of the initial states I. Given a state s, if there is an action a that allows the system to evolve to another state s', such action can be taken through the transition relation $s \xrightarrow{a} s'$. $L(s)$ is the set of atomic propositions that are satisfied in state S.

A *path* π is a state sequence that starts in an initial state and is either infinite or ends in a terminal state, i.e. a state with no successors. The set of atomic propositions that are valid along the execution is given by $trace(\pi)$.

$$\pi = s_0 s_1 s_2 s_3 .. s_n$$

$$trace(\pi) = L(s_0)L(s_1)L(s_3)..L(s_n)$$

Temporal logics typically consider only infinite transition systems, i.e. transition systems without terminal states. A finite TS can be transformed into an infinite TS by introducing a new transition from the terminal state to a *trap state*. Trap states have a self-loop and do not satisfy any atomic propositions.

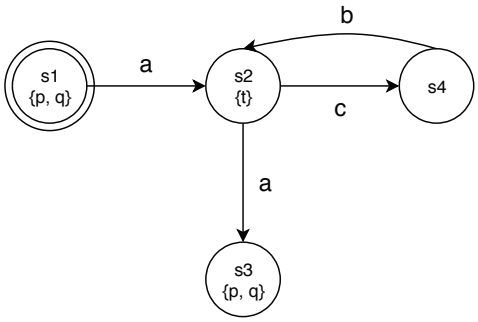


Figure 2: Transition System

In Figure 2 the TS starts in the initial state s_1, where $\{p, q\} \subseteq AP$ evaluates to true. From s_1, the system can only evolve to s_2 through $s_1 \xrightarrow{a} s_2$. Once the system reaches s_2, it can evolve either to the terminal state s_3 or s_4. It is possible for the system to stay forever in the

loop $s_2 \xrightarrow{c} s_4 \xrightarrow{b} s_2$. The action taken is chosen in a non-deterministic way. Some possible paths and the respective traces for this TS are

$$\pi_1 = s_1 s_2 s_4 s_2 s_3 \qquad trace(\pi_1) = \{p,q\}\{t\}\emptyset\{t\}\{p,q\}$$

$$\pi_2 = s_1 s_2 s_4 s_2 s_4 s_2 \ldots \qquad trace(\pi_2) = \{p,q\}\{t\}\emptyset\{t\}\emptyset\{t\}\ldots$$

The temporal logics approached in this document abstract away from actions. They are considered state-based, making only use of the atomic propositions of the states to specify system properties.

2.3 TEMPORAL LOGICS

Temporal logics are extensions of propositional logic that include modal operators to reason about time. In propositional logic, a statement is either true or false and its truth value is constant in time. The truth value of temporal logic statements, however, can vary with time.

Temporal logic is widely used in formal verification to describe the evolution of a system over time. For formal analysis to be possible, functional requirements must be expressed as temporal logic formulae. Once the requirements are expressed in temporal logic, the engineer can check if the behaviour exhibited by the model complies with the one described by the specified properties.

Properties can be classified in two different categories. *Safety-properties* state that something bad should never happen and thus are properties that, once they are not satisfied, are considered to have been violated. A property such as ”no two processes can be on a critical section at the same time” is an example of a safety property.

Liveness-properties state that something good will eventually happen. Since liveness properties always require that something in the future must hold, such properties cannot be verified on finite traces and therefore are not suitable for runtime monitoring. This contrasts with safety properties in which by observing a finite trace it is always possible to check if they hold or not. An example of a liveness property is ”after the button is pressed, the engine will eventually start”.

Different temporal logics consider different properties when reasoning about time. Linear Time Logic (LTL) considers time to be discrete and linear, while Computation Tree Logic (CTL) considers time to be discrete and branching. Metric Temporal Logic (MTL) has a linear and continuous view of time. Let us briefly explain these different notions of time:

1. **Discrete:** Discrete time is viewed as set of points (or moments) that are equally distant. This is the case of digital clocks that have a fixed clock rating.

2. **Continuous:** In continuous time the distance between two points of time is arbitrary and between any two points, there is an infinite amount of other points. This is typically used when trying to model dynamic systems.

3. **Linear:** In linear time, for a formula to hold at a given state, it must hold for all possible paths that arise from that state. There is an implicit universal quantification over all the possible computations.

4. **Branching:** Contrary to a linear view of time, a branching view takes into consideration that a state can have different computations leading to different states and thus different futures. As such, instead of a notion of time based on paths, it considers that for each state there is a tree, rooted in that state, that represents the different computations that can occur.

Although several properties can be expressed both in linear and or branching logics, their expressiveness is incomparable and there are properties that can only be expressed in one of them. An instance of a property that can be expressed with a branching view of time but not with a linear one is the property that states that *the system should always be able to return to one of its initial states*. On the other hand, the property that states *eventually, q will hold forever* is expressible in a linear view of time but not in a branching one.

Temporal logics that consider a discrete view of time such as LTL or CTL are considered *qualitative temporal logics* and deal mainly with the ordering of events, while logics with a continuous view of time such as MTL are considered *quantitative* and deal both with the ordering of events and the distance between them (Koymans, 1990). Real time systems require the expression of properties such as *φ must hold within three miliseconds of property ψ*, which can only be expressed with quantitative timing.

Furthermore, some temporal logics are designed to refer only to the future (Linear Temporal Logic) while others are designed to refer only to the past (Past Linear Temporal Logic). LTL and Past LTL have the same expressive power, i.e they are able to express the same properties. However, some properties can be expressed exponentially more succinctly in a way than the other. This is important not only to ease the job of engineers that are writing properties, but also because the verification time of a property depends on the size of the formula.

The temporal logics that will approached in the next sections are LTL and Past LTL. After that, we will also present two specification languages that are intended to leverage the difficulty of writing formal properties by providing a more natural syntax and pattern support, SpeAR and SALT.

2.3.1 *Linear Temporal Logic*

Through LTL (Pnueli, 1977) it is possible to specify linear time properties that specify the traces the system should exhibit. LTL is considered linear due to the fact that, for each moment in time, the future is already predetermined and there is only a single successor state.

The following grammar describes the syntactic rules that allow the construction of LTL formulae.

$$\varphi ::= \text{ true } \mid \alpha \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \text{ U } \varphi_2$$

where a $\in$ AP.

The formulae are composed of atomic propositions $\alpha \in$ AP; the boolean connectives $\neg$ and $\vee$ which are enough to obtain the missing boolean connectives such as $\wedge$ or $\rightarrow$, hence obtaining the full power of propositional logic; the temporal modalities Next ($\bigcirc$ or **X**) and Until (**U**). As in propositional logic, we are able to derive the missing operators Finally ($\Diamond$ or **F**) and Globally ($\Box$ or **G**).

$$\Diamond\, \alpha ::= \text{ true U } \alpha$$
$$\Box\, \alpha ::= \neg\Diamond\neg\alpha$$

Informally, the meaning of the presented temporal modalities can be described in the following manner.

- $\bigcirc\, \phi$ — the formula ϕ must hold in the next state.
- ϕ U ψ — the formula ϕ must hold until ψ is satisfied.
- $\Diamond\, \phi$ — the formula ϕ must eventually hold in the future.
- $\Box\, \phi$ — the formula ϕ must always hold in the future.

LTL properties operate over paths and their respective traces. For a path π, the satisfaction relation ($\models_{path}$) is defined through the behaviour exhibited by its trace. In turn, for a trace to satisfy ($\models_{trace}$) a property it must include the behaviour described by the property at hand.

$$\pi \models \varphi \qquad \text{iff} \qquad \mathit{trace}(\pi) \models \varphi$$

A state satisfies ($\models_{state}$) a property if, and only if, all the paths starting in that state satisfy the said property.

$$s \models \varphi \qquad \text{iff} \qquad \forall \pi \in \mathit{Paths}(s)\ .\ \pi \models \varphi$$

Finally, a system satisfies ($\models_{TS}$) a given LTL formula if, and only if, all of its initial states satisfy that property.

$$TS \models \varphi \qquad \text{iff} \qquad \forall s_0 \in I \,.\, s_0 \models \varphi$$

The LTL semantics for infinite words over 2^{AP} are defined as follows.

$$\begin{array}{lll}
\sigma \models \mathit{true} & & \\
\sigma \models \alpha & \text{iff} & \alpha \in A_0 \text{ (i.e. } A_0 \models \alpha) \\
\sigma \models \varphi_1 \wedge \varphi_2 & \text{iff} & \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
\sigma \models \neg\varphi & \text{iff} & \sigma \not\models \varphi \\
\sigma \models \bigcirc \varphi & \text{iff} & \sigma[1\ldots] = A_1A_2A_3\ldots \models \varphi \\
\sigma \models \varphi_1 \,\mathrm{U}\, \varphi_2 & \text{iff} & \exists j \geq 0 \,.\, \sigma[j\ldots] \models \varphi_2 \text{ and } \sigma[i\ldots] \models \varphi_1, \text{ for all } 0 \leq \text{ i } < \text{ j}
\end{array}$$

And for the derived operators $\Diamond$ and $\Box$.

$$\begin{array}{lll}
\sigma \models \Diamond\varphi & \text{iff} & \exists j \geq 0 \,.\, \sigma[j\ldots] \models \varphi \\
\sigma \models \Box\varphi & \text{iff} & \forall j \geq 0 \,.\, \sigma[j\ldots] \models \varphi
\end{array}$$

Let us consider a system with two concurrent processes that share a critical section and require mutual exclusion. The safety property *No two processes can be in the critical section at the same time* is easily expressible with LTL by resorting to the set of atomic propositions $crit_n$ that stand for *Process n is in the critical section*

$$\Box\neg(crit_1 \wedge crit_2)$$

and the liveness property that guarantees that at least one of the processes is capable of continuously reaching the critical section.

$$\Box\Diamond crit_1 \vee \Box\Diamond crit_2$$

Unless we guarantee that the system behaves fairly, it is not possible to specify that both processes can reach the critical section infinitely often.

2.3.2 *Past LTL*

According to Barringer et al. (1996) past-time modalities do not add expressiveness power to pure future temporal logics which led to languages dropping such modalities in linear time temporal logic for the sake of simplicity. More recent research (Markey, 2003) shows, however, that although no expressive power is lost, there are classes of properties that can

be expressed exponentially more succinctly with past operators. Succinct formulas, besides being more intuitive for engineers, produce smaller automatas.

The following grammar describes the syntactic rules that allow the construction of Past LTL formulae.

$$\varphi := \text{ true } \mid \alpha \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \text{ Y } \varphi \mid \varphi_1 \text{ S } \varphi_2$$

where a $\in$ AP.

The grammar is similar to that of LTL with temporal modalities Next and Until are replaced by the temporal modalities Previous (**Y**) and Since (**S**). As in LTL, we are able to derive the missing operators Previously (**P**) and Historically (**H**).

Informally, the presented temporal modalities can be described in the following manner.

- Y ϕ — the formula ϕ must have been satisfied in the previous state.
- ϕ S ψ — the formula ϕ must hold since the moment ψ was satisfied.
- P ϕ — the formula ϕ must have been satisfied sometime in the past.
- H ϕ — the formula ϕ must have been satisfied in all the previous states.

The formal semantics of the Past LTL operators for finite traces are defined as follows.

$\sigma \models \mathit{true}$		
$\sigma \models \alpha$	iff	$\alpha \in A_0$ (i.e. $A_0 \models \alpha$)
$\sigma \models \varphi_1 \wedge \varphi_2$	iff	$\sigma \models \varphi_1$ and $\sigma \models \varphi_2$
$\sigma \models \neg\varphi$	iff	$\sigma \not\models \varphi$
$\sigma \models \text{ Y } \varphi$	iff	$\sigma' \models \varphi$, where $\sigma' = \sigma_{n-1}$ if $n > 1$ and $\sigma' = \sigma$ if $n = 1$
$\sigma \models \varphi_1 \text{ S } \varphi_2$	iff	$\sigma_j \models \varphi_2$ for some $1 \leq j \leq n$ and $\sigma_i \models \varphi_1$ for all $j < i \leq n$
$\sigma \models \text{ P } \varphi$	iff	$\exists j \,.\, 1 \leq j \leq n \,.\, \sigma_j \models \varphi$
$\sigma \models \text{ H } \varphi$	iff	$\forall j \,.\, 1 \leq j \leq n \,.\, \sigma_j \models \varphi$

An important observation that has an impact on monitoring algorithms is that the above semantics can also be defined recursively. If the satisfaction relation for a formula and a trace is calculated along its execution, this allows us to calculate the satisfaction relation for the next step by only looking to the previous step.

$\sigma \models \varphi_1 \text{ S } \varphi_2$	iff	$\sigma \models \varphi_2$ or ($n > 1$ and $\sigma \models \varphi_1$ and $\sigma_{n-1} \models \varphi_1 \text{ S } \varphi_2$)
$\sigma \models \text{ P } \varphi$	iff	$\sigma \models \varphi$ or ($n > 1$ and $\sigma_{n-1} \models \text{ P } \varphi$)
$\sigma \models \text{ H } \varphi$	iff	$\sigma \models \varphi$ and ($n > 1$ implies $\sigma_{n-1} \models \text{ H } \varphi$)

2.3.3 *Property Patterns*

Formal verification techniques are still not widely adopted. A common obstacle that is slowing down adoption is the required expertise necessary to write properties. Furthermore, the resulting formulae are often hard to read and even harder to write. To make matters worse, a person entering the field for the first time is usually faced with a lack of both good training materials and good tool support.

The introduction of a pattern system created by expert system designers aims to reduce the background experience required for new users. Design languages, as well as programming languages, are usually very expressive, allowing for a wide-variety of solutions to a problem,with their respective pros and cons that need to be evaluated. Most of the times, however, users prefer guidance over expressiveness. With that in mind and based on the design patterns solution which is widely used in the programming world, pattern systems for specifications that match common requirements into specification templates were created. These systems comprise a collection of high-level abstractions over specifications that can be mapped to many different formalisms such as LTL or CTL.

A pattern captures a solution that appears repeatedly when solving design problems. It seeks to gather information around both the problem and the solution, documenting the context in which it appears, the requirements it addresses, how it solves the problem, and when the pattern should be used. The patterns are formalism-agnostic and can be parameterised by individual states or even nested formulae in order to obtain a concrete specification. It is important to note that even though a pattern can be expressed in multiple formalisms, the resulting formulae are not necessarily equivalent due to the differences in the semantic models of each formalism.

Pattern system by Dwyer

The pattern system described in Dwyer et al. (1999) is a collection of simple patterns defined with the intention of easing the use of formal methods in practice. The patterns are divided according to their nature into two categories dealing with either the *occurrence* of states or the *ordering* of states. In the first category, dealing with occurrence, we have:

- Absence — The state formula does not occur within the scope.
- Existence — The state formula must occur within the scope.
- Bounded existence — The state formula must occur k times within the scope. Variants of this pattern specify *at most k occurrences* and *at least k occurrences*.
- Universality — The state formula occurs throughout the scope.

In the second category, dealing with ordering, we have:

- Precedence — A state P must always be preceded by a state Q within the scope.
- Response — A state P must always be followed by a state Q within the scope.
- Chain precedence — A sequence of states $P_1, .., P_n$ must always be preceded by a sequence of states $Q_1, .., Q_n$. This pattern is a generalisation of the precedence pattern.
- Chain response — A sequence of states $P_1, .., P_n$ must always be followed by a sequence of states $Q_1, .., Q_n$. This pattern is a generalisation of the response pattern.

Each pattern has a scope which can be the entire program or just a fragment and is defined by a starting state and/or an ending state. The scopes described in Dwyer et al. (1999) are the following: *global* (over the entire program), *before* (the execution up to a given state), *after* (the execution starting at a given state), *between* (the execution between two states), and *after-until* (similar to the *between* scope but the ending state is not guaranteed to occur).

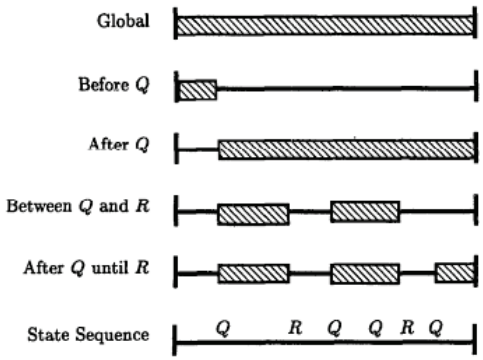


Figure 3: Patterns scopes. Figure reproduced from Dwyer et al. (1999).

A pattern is composed by several fields such as its intent and the problem it addresses, mappings to multiple formalisms such as LTL and CTL, as well as examples and concrete situations in which they were used. Additionally, some patterns describe their relation to other patterns, for instance, the *absence* pattern is the dual of the *existence* pattern and the *chain response* is a generalisation of the *response* pattern. Dwyer provides some notes on property specification using the pattern system he developed. In these notes he explains how to adapt the existing patterns to better express the desired property by addressing topics such as the combination of patterns, variations in the scopes and pattern parameterisation.

Out of the 555 examples of property specifications Dwyer collected to evaluate his pattern system, 92% were considered to be an instance of one of the provided patterns.

The patterns developed by Dwyer et al. (1999) inspired many other works such as a modified version of the pattern system that extends it to support real-time properties by

Konrad and Cheng (2005) or instantiations of the patterns into timed observer automata by Gruhn and Laue (2006).

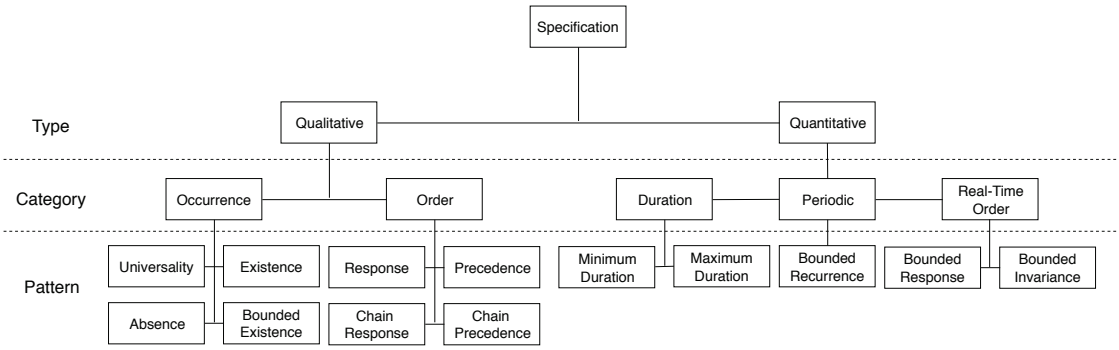


Figure 4: Patterns hierarchy that includes real-time patterns. Figure based on Konrad and Cheng (2005).

This work also contributed to the emergence of high-level specification languages that include both patterns and scopes in its features. Next, we present two examples of such languages, SpeAR and SALT, which provide a grammar close to a structured natural language to improve readability and intuitiveness when reading and writing properties.

2.3.4 *SpeAR*

SpeAR (Specification and Analysis of Requirements) is an open-source tool for capturing and analysing requirements stated in a language that is formal, yet designed to read like natural language (A. Fifarek, 2017).

The SpeAR specification language has the formal semantics of Past LTL and supports basic arithmetic, logical, and relational operators. The SpeAR developers sought to provide a specification language as natural as possible so that engineers could express themselves more naturally. Hence, SpeAR provides english aliases for most logical operators. The following is an example of a requirement written using the SpeAR specification language.

if signal greater than threshold then output equal to ON

Spear documents have a well-defined structure that promotes the grouping of requirements to enable modularity and reuse. A SpeAR document distinguishes *inputs*, *outputs*, *state*, *assumptions*, *requirements* and *properties*.

- **Inputs, Output, State:** These parts of the document describe the data that is handled throughout the system. Inputs represent monitored or observed data from the environment as well as inputs from other components. Outputs represent data sent to

the environment or other components. State represents data that is not visible to the environment or other components.

- **Assumptions:** Assumptions identify necessary constraints on inputs from the environment and other components.
- **Requirements:** Requirements identify constraints that the component must guarantee through its implementation.
- **Properties:** Properties represent constraints that the system should satisfy when operating in its intended environment.

In addition to capturing requirements formally, SpeAR enables multiple analysis approaches such as logical entailment and logical consistency.

Logical entailment allows engineers to prove that stated properties are consequences of the captured assumptions and requirements. Formally, SpeAR proves that a given set of Assumptions (A) and Requirements (R) entails each property (P).

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge R_1 \wedge R_2 \wedge \ldots \wedge R_m \vdash P_i$$

This feature provides early insight into the correctness and completeness of the captured requirements and prove that the captured requirements and assumptions satisfy all of the stated properties.

Logic entailment, however, is only useful when the assumptions and requirements are consistent with each other. In the case of a contradiction, the logical conjunction of the constraints is false and thus the logical entailment is a vacuous proof (i.e. false $\rightarrow$ true). SpeAR provides a logical consistency check for such cases in which it searches for a counterexample to the assertion that the conjunction of the assumptions and requirements cannot be true for N consecutive steps.

$$\neg((A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge R_1 \wedge R_2 \wedge \ldots \wedge R_m) \wedge (Step_{Count} \geq N))$$

In case the requirements are proven inconsistent for the user-selected N steps, SpeAR identifies the set of constraints in conflict. However, in case SpeAR declares the constraints are consistent it does not mean they are consistent at $N + 1$. To prove consistency for all possible N, SpeAR would need to provide a stronger guarantee known as realisability check, which verifies for all steps and inputs that satisfy the assumptions that the system remains responsive and only performs valid transitions (Gacek et al., 2015).

2.3.5 *SALT*

SALT (Structured Assertion Language for Temporal Logic) is a high-level specification language designed for the comfortable writing of concise and readable specifications to use in model checking or runtime verification. Unlike many specification languages, SALT does not target a particular domain (Streit, 2006).

The SALT language consists of three layers, and the generated formulas change according to which layers are used. The *propositional layer* deals with boolean propositions and boolean operators. The *temporal layer* encapsulates the main features of the SALT language that reason about temporal behaviour and is divided into a future fragment and a symmetrical past fragment. The timed layer adds real-time constraints to the language and is divided into a future and a past fragment as well.

If only *propositional layer* is used, the resulting formula will only have propositional operators. If the *temporal layer* or *timed layer* are used, the resulting formula will be a LTL or Timed LTL formula, respectively. Similarly, if only the future fragment is used, the generated formulae will be pure future LTL or Timed LTL. In the cases where the past fragment is used, the formulae will contain past operators.

Besides the common temporal operators, SALT provides several high-level features that improve the easiness with which the engineers can express themselves.

- **Exception operators:** Exception operators define exception conditions for a formula φ. The evaluation of the formula stops when the condition occurs and is either accepted or rejected depending on the exception operator used. If the condition is never satisfied, the operator is ignored.

- **Regular expressions:** Simplified regular expressions (SRE) allow the expression of complex patterns of conditions in a very concise way. Since regular expressions cannot be translated into LTL, SRE have some limitations.

- **Counting quantifiers:** Counting quantifiers allow brief and intuitive statements about conditions that have to hold a certain number of times. Such class of properties are usually very verbose when specified using LTL.

- **Scopes:** The scope operators bring to SALT the scope concept introduced in Dwyer et al. (1999) allowing developers to specify that an expression has to hold before, after or in between some events.

- **Macros:** Parametariseble macros enable user-defined patterns that can be used in a similar way as SALT default operators. This allows users to easily reuse solutions that appear frequently in their requirements.

- **Iteration Operators:** Iteration operators intend to ease the handling of specifications where an expression is repeated several times with only a few parameters exchanged.

All these features are translated into LTL or Timed LTL accordingly. The SALT compiler performs several optimisations on the generated formulae. This compilation step allows SALT to be used as a frontend to existing verification tools.

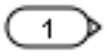
The SALT compiler can be extended via a plugin mechanism in the form of Haskell modules. Two types of plugins are supported. Proposition parsing plugins allow to perform checks or transformations on the atomic propositions that are used within a SALT specification. Printing function plugins allow to define custom output syntax for the LTL formulae.

2.4 SIMULINK

Simulink is a tool developed by Mathworks that allows engineers to perform multidomain modeling, simulation, analysis as well as code generation. In Simulink, the modelling is done using a graphical block diagramming tool with blocks provided by libraries. MATLAB and Simulink are tightly coupled and it is possible to interact with a Simulink environment through MATLAB, which allow us to build our Simulink monitors through MATLAB scripts generated by our tool.

Stateflow extends Simulink with the ability to model reactive systems using state machines. This enables engineers to model control logic through a block that can be connected to other Simulink blocks for inputs and produces a verdict as output. While Stateflow provides many features, our monitors only rely on *Stateflow.States* and *Stateflow.Transition* to build the transition system, as well as *Stateflow.Data* to model the inputs, the output and internal variables.

Next, we present the Simulink blocks that we use to build our monitors.



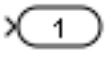


Figure 5: Outport blocks (top) link signals from a system to a destination outside of the system. They can connect signals flowing from a subsystem to other parts of the model. They can also supply external outputs at the top level of a model hierarchy.

Inport blocks (bottom) link signals from outside a system into the system.

Figure 6: The Goto block (top) passes its input to all the matching From blocks. The input can be a signal or vector of any data type. The From block (bottom) accepts a signal from a corresponding Goto block, then passes it as output.

Goto blocks and From blocks are matched by the use of Goto tags.

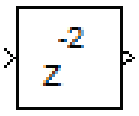


Figure 7: The Delay block outputs the input of the block after a delay. The block determines the delay time based on the value of the Delay length parameter.

Figure 8: With a MATLAB Function block, you can write a MATLAB function for use in a Simulink model.

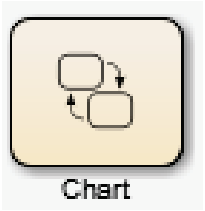


Figure 9: A Stateflow chart is a graphical representation of a finite state machine. States and transitions form the basic elements of the system.

3

THE MONGE TOOL

In this thesis we develop a proof of concept tool to support to the model based engineering workflow. The purpose of the tool is to increase the confidence we have in our model by improving a practice that is already widely used in the design of critical systems, i.e. simulation. For that, we provide a workflow that allows the automatic generation of runtime monitors. Such monitors can be plugged into the system model and run in parallel with the simulation, guaranteeing that the property it models is not violated for that simulation run.

The supported specification languages, SpeAR and SALT, were selected because they provide high-level constructs that enable engineers to write formal specifications with a syntax close to that of natural language. This greatly reduces the amount of expertise required in the formal methods domain and makes it easier for requirements engineers to express themselves since they typically already write requirements in either natural language or a structured version of it.

These languages also provide the ability to specify properties either regarding the future or the past. As explained before, while their expressive power is equivalent, some properties can be more easily expressed in one than the other. This is a factor that is as important to the engineer, who is able to write properties in a more intuitive way, as it is important for the verification algorithms, whose execution times grow with the size of the formulas.

Once the requirements are formalised, the engineer can trigger the generation process. The tool generates MATLAB scripts which, once executed, build Simulink models from scratch that represent the respective monitors and can be plugged into the system model. The Simulink platform was chosen as the target due to its wide use in dynamical systems modelling.

We also provide a feature to help with the task of extracting the requirements from a document so that they can be imported into the tool. Text extraction is usually done by relying on the regular expressions. Regular expressions, however, are hard to read and write, not composable, and targeted at developers. We try to make it easier for requirements engineers to perform requirement extraction by providing a DSL that addresses these issues.

3.1 TOOL OVERVIEW - REQUIREMENT FORMALISATION

It is possible to either add requirements manually or use the extraction feature we provide to load requirements from a document. In order to generate monitors for the listed requirements, the engineer needs to provide a formal specification for the intended requirements with one of the supported specification languages.

The tool is able to infer which of the specification languages is being used to formalise the requirement, avoiding the need to manually specify it. If the monitor generation process is triggered and there are requirements that were not formalised, they will be safely ignored. This allows the engineer to skip non-functional requirements as well as to formalise the requirements incrementally, which is useful in case they have been automatically extracted from a document.

```
REQ_1: When the system is in state_1 and signal_3 + signal_2 > 100, the system shall
     transition to state state_3.
Pattern: historically if previously state equal to STATE_1 and signal_3 + signal_2 >
     100 then state equal to STATE_3

REQ_2: Once the system is in state_3 occurs, then the system will be in state_4
    somewhere in the future.
Pattern: assert always (if "state == STATE_3" then eventually "state == STATE_4")

REQ_3: The system should be able to handle a load of 2mb/s
Pattern: None

REQ_4: When the system is in state state_3, signal_3 shall be calculated as signal_5
     * 8 / 10.
Pattern: assert always (if "state == STATE_3" then "signal_3 == signal_5 * 8 / 10")

REQ_5: Every time signal_5 is between 5 and the previous value of signal_3, the
    system is in state_5.
Pattern: historically if signal_5 > 5 and signal_5 < previous signal_3 then state
    equal to STATE_5

REQ_6: Every time signal_10 > 20, the system should keep itself in state_5 until
    signal_10 < 10
Pattern: after signal_10 > 20 until signal_10 < 10 always state equal to STATE_5
```

Listing 3.1: Example of requirements and their respective specification.

In the example portrayed in 3.1 there are six different requirements. Each requirement has an id, a description and its respective specification. *REQ_1* and *REQ_5* were specified as SpeAR formulas. *REQ_6* was specified as a SpeAR pattern. *REQ_2* and *REQ_4* were specified as SALT formulas. *REQ_3* is a performance requirement and thus was ignored.

3.2 TOOL OVERVIEW - REQUIREMENT EXTRACTION

Requirements are usually written in a natural language and archived in digital formats that focus on readability instead of structure. Hence, requirement documents are rarely suited for analysis and computations and thus there is a necessity to extract requirements into a more treatable form from such documents.

Regular expressions are the common solution to extract segments from a text document. By defining a search pattern that describes the data we seek, regular expressions allow us to find all the substrings of the document that match such pattern. However, we do not consider this technique a good solution for requirement engineers. Regular expressions require some expertise to write, are hard to compose and, due to their succinct syntax, are difficult to read. Thus, we developed a simple DSL that addresses the described issues.

```
identifier = "REQ_"
              followed_by one_or_more digit

text = one_or_more any_char

requirement = capture identifier
              followed_by ":"
              followed_by zero_or_more whitespace
              capture text
              followed_by new_line
```

Listing 3.2: Example of an extractor using the developed DSL.

The extractor in Example 3.2 is compiled to $(REQ\_[0-9]+) : \backslash s*(.+)\backslash n\{2\}$, which is the regular expression we would use underneath to extract the requirements listed in 3.3. Each expression denotes an extractor. The tool expects a final extractor that captures an identifier for the requirement as well as the requirement text.

In this example, the *requirement* extractor starts by capturing the identifier, relying on *identifier* extractor that was previously defined, which expects the keyword `REQ_` followed by one or more digits. In order to match a requirement, the identifier should be followed

by a colon and optional whitespace. All characters are then considered to be part of the requirement text, as defined in the *text* extractor until a new line is met.

```
REQ_1: When the system is in state_1 and signal_3 + signal_2 > 100, the system shall
     transition to state state_3.

REQ_2: Once the system is in state_3 occurs, then the system will be in state_4
    somewhere in the future.
```

Listing 3.3: Excerpt of a fake requirements document that is extractable with the extractors defined in the Listing 3.2.

3.3 TOOL OVERVIEW - MONITOR EXAMPLE

It is important to note that while SpeAR translates to Past LTL, SALT can translate to either Past LTL or LTL depending on what features were used in the specification. While Past LTL based monitors and LTL based monitors are similar in structure, the underlying verification algorithm is different. Past LTL also requires an extra layer to deal with the previous operator **Y**, which looks to the value of an expression in the previous state. More details on the verification algorithms are given in Section 4.1.

Figure 10 shows the structure of the generated monitors. In the following paragraphs the purpose of each component is explained.

- Signal extraction (orange) - The needed signals are extracted from the model and linked into *Goto blocks* so that they can be accessed through *From blocks* in the rest of the monitor. This intends to reduce the number of transitions as well as the reducing the coupling of the monitor to the model by extracting all the required data from the model only once and in a single place.
- Previous layer (purple) - In the case of Past LTL it is possible for expressions to refer to previous values of a signal. Therefore we need an extra layer that links the original signal into a *Delay block* to obtain the values from previous states. As in the *Signal extraction* layer, these blocks are linked into *From* and *Goto blocks* to reduce transitions throughout the monitor.
- Mathematical expressions (green) - Math expressions need to be computed into boolean values before passing them to the runtime monitoring algorithm. For each math expression we add a *MATLAB Function block* that gets their parameters through *From blocks* and outputs a boolean value with the result of the expression. The constants used inside the expressions are defined in the MATLAB script generated by the tool.

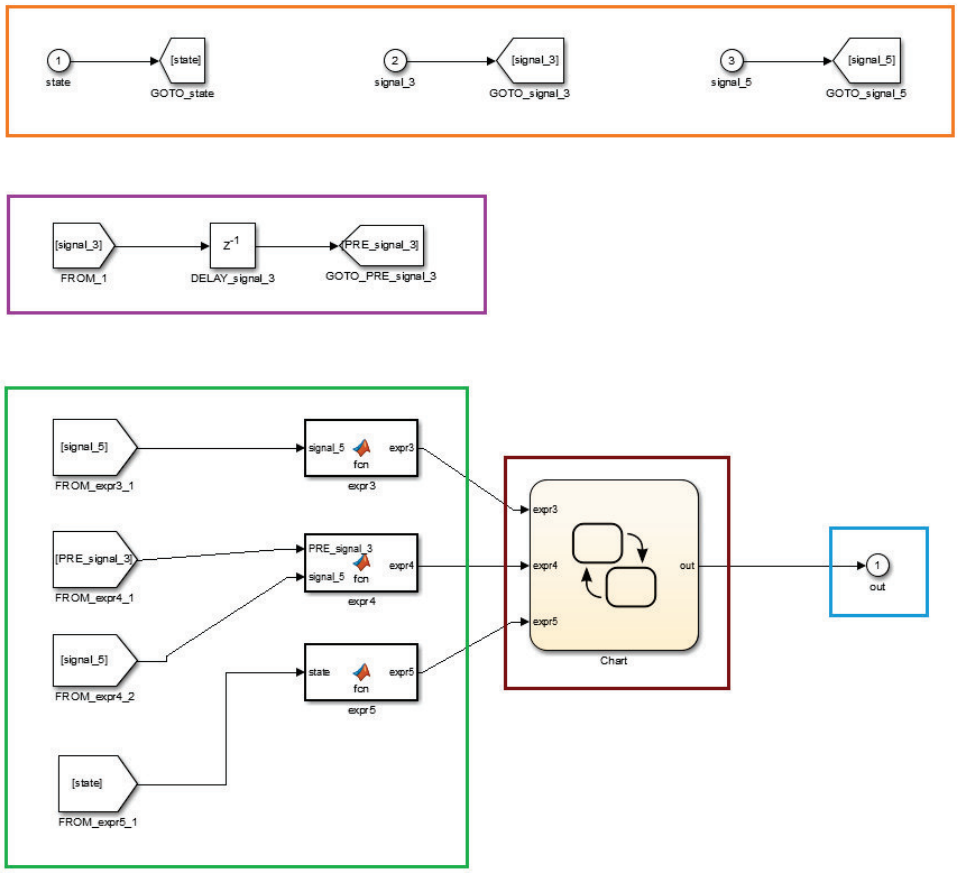


Figure 10: Example of a generated monitor with its different components highlighted.

- Stateflow (red) - The *Stateflow block* encapsulates the runtime verification algorithm and it is where the main difference between LTL and Past LTL monitors lies. The algorithms are described later in Section 4.1. The Stateflow outputs the monitor verdict for that point in time.
- Output (blue) - The *output* layer simply exposes the monitor verdict to other systems in the model.

3.4 Design decisions

While developing the tool we made some design choices that affect the way requirements can be written. We will now elaborate on these.

1. SpeAR and SALT do not offer the same features. We do not intend to introduce the lacking features since the purpose of supporting multiple languages is for them to

cover for each other. However, mathematical expressions are an example of something we want to have support for in all the languages we support.

2. Since the purpose of the tool is to generate runtime monitors, it is important to interoperate with the Simulink model. Thus, when writing the specification the engineer should take care to name the variables present in the specifications with the same names as the equivalent Simulink signals.

3. For simplicity, every specification is handled individually and there is no notion of a SpeAR document or SALT document. This makes it impossible for the engineer to use some of the features SpeAR and SALT provide, such as user-defined patterns. Section 4.2 provides more details on why this decision was necessary.

Temporal logics usually support only boolean values and boolean operators. However, it is common for functional requirements to rely on arithmetic operations to express behaviour. While SpeAR supports such operators, SALT does not. SALT provides, however, *quoted boolean propositions* whose interpretation can be customised. We make use of such propositions to handle mathematical expressions.

```
assert always (if 'state == STATE_3' then eventually 'state == STATE_4')
```

Listing 3.4: SALT specification with quoted boolean expressions and named constants.

```
historically if previously state equal to STATE_1 and signal_3 + signal_2 > 100 then
    state equal to STATE_3
```

Listing 3.5: SpeAR specification with built-in math operators.

It is also common for engineers to use named constants in their requirements to express the set of values a signal can be evaluated to. In order to differentiate constant names from signal names, signal names are expected to be lower case identifiers while named constants are expected to be upper case identifiers.

4

MONGE ARCHITECTURE

The MonGe architecture is divided in two different packages: an *extraction* package and a *backend* package.

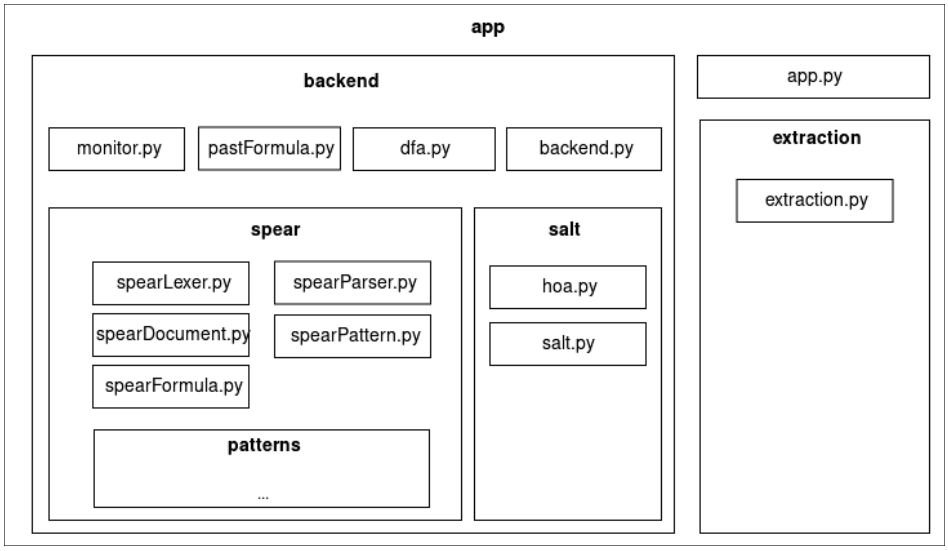


Figure 11: Tool architecture

The extraction package is a simple one. It defines the grammar and the translation from the DSL to regex patterns.

The backend, on the other hand, needs to provide two different abstractions for the verification algorithm: one based on *deterministic finite automatons* (DFA) and another based on the recursive semantics of past LTL. These algorithms are built on top of a *monitor* abstraction that contains all the data obtained by parsing the formula. Based on this data, the *monitor* abstraction is able to render the MATLAB script that builds the runtime monitor for the formula. The rendering of the *Stateflow block* is left outside the scope of this abstraction, allowing for other abstractions to be built on top of this one, implementing different verification algorithms. This is the case of the *PastFormula* abstraction, that implements the rendering of the *Stateflow block* for the recursive semantics of Past LTL and the *DFA* abstraction that does the same for the algorithm based on deterministic finite machines.

Furthermore, the backend module contains two inner packages, one for each of the specification languages supported.

SpeAR always relies on past semantics, however it is possible to write SpeAR specifications by either using their grammar or by using the builtin patterns they provide. The spear package provides the necessary modules to parse the grammar (*SpearLexer* and *SpearParser*) and extract the necessary data required by *PastFormula* (*SpearFormula* and *SpearDocument*). For SpeAR patterns, which have a well know translation into DFAs and thus rely on the DFA based verification algorithm, there is an abstraction (*SpearPattern*) that handles the same data extraction. The patterns package contains the translation from every pattern into a DFA.

A SALT formula can rely either on future or past semantics. The formula is fed to the SALT compiler and then, depending on the semantics used, the next steps may differ. In the case of future semantics, the SALT compiler outputs LTL that is then handled by Spot – a tool that is able to output a DFA in the Hanoi Omega Automata (HOA) format from an LTL expression. Such output is then parsed and converted into a state machine for Simulink. In the case of past semantics, the SALT compiler outputs the formula in an intermediate format which is then subject to a process similar to the one that SpeAR formulas go through.

Finally, the backend provides a *requirement* module that integrates the entire backend, being able to generate a monitor from a text specification that can be either SALT or SpeAR.

4.1 RUNTIME VERIFICATION WITH MONITORS

In this section we will dive again into the monitoring and understand how data flows from the system model into the *Stateflow block* that encapsulates the verification algorithm. These algorithms are responsible for checking, based on the data obtained from the previous layers, if the property holds or not. Thus, this is the part of the monitor that actually encodes the behaviour we expect the system to exhibit, as described in the temporal formula.

As mentioned before, data is extracted from the system model using *From blocks* with the signal names that appear in the formula. In Past LTL it is possible to refer to values of signals from previous states. In such cases, the signals go through a *Delay block* that delays the value as many steps as necessary.

With this, we have all the values we need to evaluate the mathematical expressions that appear in the formula. By evaluating the mathematical expressions we eliminate all arithmetic expressions, ending up with just boolean values which are the only data type that our verification algorithms can reason about.

The monitor abstraction provides a way to describe a runtime monitor and generate a MATLAB script that builds the Simulink model for that monitor. As expected, most of the parameters match the various layers that exist in the model:

1. the set of signals that need to be extracted from the system model;
2. the set of signals that need to be delayed due to the semantics of the Past LTL *PRE* operator;
3. the mathematical expressions that need to be evaluated and their respective inputs;
4. the constants that appear in the temporal formula and need to be defined as a MATLAB value.

This abstraction however is just a building block that must be extended for each verification algorithm we use by providing the logic to render the respective *Stateflow chart* that is encapsulated in the *Stateflow block*. The following is a template of the generated script that is instantiated with the data extracted from the specifications. Note that there are several parts that just deal with the positioning of elements in the chart.

```
sfnew;
rt = sfroot;
model = rt.find('-isa', 'Simulink.BlockDiagram');
ch = model.find('-isa', 'Stateflow.Chart');
set_param('untitled/Chart', 'position', [1000, 500, 1150, 650]);

{% for arg in args %}
add_block('simulink/Sources/In1', 'untitled/{{ arg }}');
add_block('simulink/Signal Routing/Goto', 'untitled/GOTO_{{ arg }}')
add_line('untitled', '{{ arg }}/1', 'GOTO_{{ arg }}/1');

set_param('untitled/{{ arg }}', 'position', [{{ 300 * loop.index }}, 50, {{ 300 *
    loop.index + 20 }}, 70]);
set_param('untitled/GOTO_{{ arg }}', 'position', [{{ 300 * loop.index + 100 }}, 40,
    {{ 300 * loop.index + 140 }}, 80]);
set_param('untitled/GOTO_{{ arg }}', 'GotoTag', '{{ arg }}')
{% endfor %}
```

It starts by creating a new Simulink model with an empty Stateflow block and defines some variables. Next, it goes through all the signals (*args*) that are needed from the system model and creates a pair of *Input block* and *Goto block* for each one, setting the necessary parameters and connecting them. With this we are able to access all signals throughout the rest of the model while avoiding cluttering the model with connections.

```
{% for p in previous %}
add_block('simulink/Signal Routing/From', 'untitled/FROM_{{ loop.index }}')
add_block('simulink/Commonly Used Blocks/Delay', 'untitled/DELAY_{{ p.0 }}')
```

```
add_block('simulink/Signal Routing/Goto', 'untitled/GOTO_PRE_{{ p.0 }}')

add_line('untitled', 'FROM_{{ loop.index }}/1', 'DELAY_{{ p.0 }}/1');
add_line('untitled', 'DELAY_{{ p.0 }}/1', 'GOTO_PRE_{{ p.0 }}/1');

set_param('untitled/FROM_{{ loop.index }}', 'position', [{{ 200 + 400 * loop.index0
    }}, 140, {{ 200 + 400 * loop.index0 + 40 }}, 180]);
set_param('untitled/DELAY_{{ p.0 }}', 'position', [{{ 200 + 400 * loop.index0 + 140
    }}, 140, {{ 200 + 400 * loop.index0 + 180 }}, 180]);
set_param('untitled/GOTO_PRE_{{ p.0 }}', 'position', [{{ 200 + 400 * loop.index0 +
    280 }}, 140, {{ 200 + 400 * loop.index0 + 320 }}, 180]);

set_param('untitled/FROM_{{ loop.index }}', 'GotoTag', '{{ p.0 }}')
set_param('untitled/GOTO_PRE_{{ p.0 }}', 'GotoTag', 'PRE_{{ p.0 }}')
{% endfor %}
```

We now proceed to build the *previous layer* of the monitor. This layer has a set of blocks for each signal that refer to previous states of the system. Each set of blocks consists of a *From block* to obtain the signal value from the previous layer, a *Delay block* to delay this value a parameterised number of states, and a *Goto block* to make it accessible throughout the rest of the model as in the previous layer.

```
{% for id, expr in exprs.items() %}
prop = Stateflow.Data(ch);
prop.Name = '{{ id }}';
prop.DataType = 'boolean';
prop.Scope = 'Input';

{% set outer_loop = loop %}

add_block('simulink/User-Defined Functions/MATLAB Function', 'untitled/{{ id }}');
set_param('untitled/{{ id }}', 'position', [800, {{ 450 + 100 * loop.index0 }}, 900,
     {{ 450 + 100 * loop.index0 + 50 }}]);
add_line('untitled', '{{ id }}/1', 'Chart/{{ loop.index }}');

fHandle = rt.find('-isa', 'Stateflow.EMChart', 'Path', 'untitled/{{ id }}');
fHandle.Script = sprintf(['function {{ id }} = fcn({{ expr.2 }})\n', '{{ id }} = {{
    expr.0 }};']);

{% for arg in expr.1 %}
```

```
{% set index = outer_loop.index0 + loop.index0 %}

add_block('simulink/Signal Routing/From', 'untitled/FROM_{{ id }}_{{ loop.index }}')
set_param('untitled/FROM_{{ id }}_{{ loop.index }}', 'position', [600, {{ 450 + 100
    * index }}, 650, {{ 450 + 100 * index + 50 }}]);
set_param('untitled/FROM_{{ id }}_{{ loop.index }}', 'GotoTag', '{{ arg }}')
add_line('untitled', 'FROM_{{ id }}_{{ loop.index }}/1', '{{ id }}/{{ loop.index }}'
    );
{% endfor %}

{% endfor %}
```

The next layer being built is the mathematical layer. It consists of two loops: the first generates a MATLAB function for each mathematical expression that appears in the specification so that it can be evaluated; the inner loop goes through the function inputs and generate *From blocks* to access the respective signal value that corresponds to each input. All functions output boolean values that are connected to the *Stateflow block*. These values are interpreted as the propositional variables of the formulas.

```
{% for const in constants %}
{{ const }} = {{ loop.index }};
{% endfor %}

{{ stateflow }}

sfsave('untitled', '{{ new_name }}');
```

Finally, we define the necessary constants that were used in the formula. We render the *Stateflow block*, which was generated by a subclass of this abstraction according to the verification algorithm it encodes. The chart is saved with the given name.

4.1.1 *Deterministic Finite Automaton*

A deterministic state automaton (DFA) is a transition system equipped with a set of accepting states. By providing a string of input symbols and checking if the last input symbol

leads into an accepting state, we verify that the string is a valid computation for that transition system.

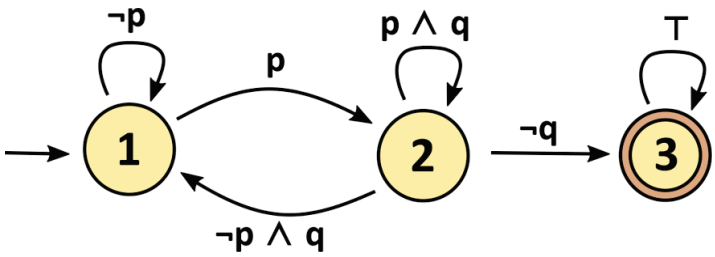


Figure 12: Example of a deterministic finite automaton.

Figure 12 demonstrates an example of a DFA for the property *Every time p holds, q holds in the next step*. The DFA has an initial state **1** and an accepting state **3**. If the execution trace leads the DFA into the accepting state, the system will be trapped in that state and the property will be considered violated.

An example of a trace that violates this property is $\{p\}, \{p, q\}, \{\}$. In the first input symbol **p** holds and therefore **q** must hold in the next input symbol, which leads the system into transitioning to state **2**. State **2** expects **q** to hold, otherwise it transitions to state **3**, meaning that the property was violated. In the next input symbol, **q** holds and so does **p**, so the system will wait again for **q** to happen and thus remain in state **2**. Finally, in the last input symbol, **q** does not hold and the system evolves into state **3**. The system remains trapped in state **3** and the monitor reports that the property was violated.

Stateflow is the feature provided by *Simulink* to introduce deterministic finite automatons in a *Simulink* model. If we know how to transform the formula into a DFA, checking if the accepting state is reachable within a DFA is the default algorithm we use. That is the case for formulas with LTL semantics and SpeAR patterns which, while having Past LTL semantics, have a fixed and well known translation into a DFA. In the case of SpeAR formulas we rely on the recursive semantics of Past LTL to verify the validity of a property.

The DFA module is a simple one that extends the monitor abstraction with the data required to describe a DFA, and takes on the task of generating a *Stateflow block* that describes the respective DFA. The following is a template of the MATLAB script that renders a deterministic finite automaton in Simulink.

```
{% for s in states|sort %}
q_{{ s }} = Stateflow.State(ch);
q_{{ s }}.Name = 'q_{{ s }}';
{% endfor %}

entry = Stateflow.Transition(ch);
entry.Destination = q_{{ initial_state }};
```

```
entry.DestinationOClock = 0;

{% for s in accept_states|sort %}
q_{{ s }}.HasOutputData = 1;
{% endfor %}

add_block('simulink/Sinks/Out1', 'untitled/out');
set_param('untitled/out', 'position', [1300, 575, 1320, 595]);
add_line('untitled', 'Chart/1', 'out/1');

{% for (s1, prop), s2 in transitions.items() %}
tr = Stateflow.Transition(ch);
tr.Source = q_{{ s1 }};
tr.Destination = q_{{ s2 }};
tr.LabelString = '{{ prop }}';
{% endfor %}
```

It starts by defining the set of states, while declaring the initial state and all the accept states. After that it defines the transition relation over the set of states defined previously, determining the set of actions through which the system can evolve.

4.1.2 *Monitoring with Past Formulas*

The process of translating a formula expressed in Past LTL to a deterministic finite automaton is not straightforward and thus we decided to rely on a different algorithm for properties that reason about the past.

This algorithm is based on the recursive semantics of Past LTL, allowing us to evaluate a Past LTL property just by looking at the current and previous states. This is an important improvement over the commonly used semantics which, at every state, require a lookup of every past state. Thus, while the commonly used semantics give us a worst case scenario of O(N), where N is the number of we saw so far, the recursive semantics give us a worst case scenario of O(1).

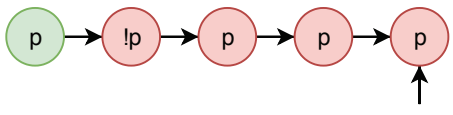


Figure 13: Past LTL semantics for formula $H\ p$.

In Figure 13, in order to discover if $H\ p$ holds in the current state, the algorithm needs to iteratively look into all the previous states and check if p does not hold in any of those states. As the can see in the figure, the property does not hold for the current state, because p did not hold in the past in the past.

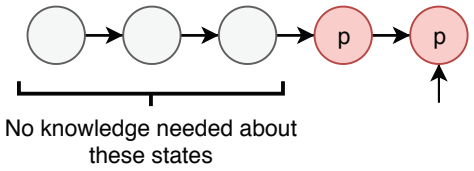


Figure 14: Recursive Past LTL semantics for formula $H\ p$.

The case is different in Figure 14 which illustrates the recursive semantics. Even though the p holds in the current and previous state, we have information that it did not hold sometime in the past, which is enough to conclude that $H\ p$ does not hold in the current state.

In order to generate a *Stateflow block* that can encode these semantics, we must first extract all the subformulas from the temporal formula being verified. So, if we are verifying the property *if p holds, q held in previous step, and the same is true for every step in the past*, denoted as $H\ (p \rightarrow Y\ q)$, after the first step of the algorithm we end with the results shown in the Listing 4.1.

```
sf5: q
sf4: true
sf3: p
sf2: ¬pre(sf3) ∨ pre(sf4)
sf1: pre(sf2)
```

Listing 4.1: Subformulas of $H\ (p \rightarrow Y\ q)$.

We should note that it is important to list the most inner formulas first so that they are according to their evaluation order. The verification algorithm relies on two sets of values.

1. the *previous* set – it holds the values of the subformulas that were evaluated for the previous state.
2. the *now* set – it holds the values that are being evaluated for the current state by relying on the *previous* set and the signal's values in the current state.

Once the *now* set is completely evaluated, we proceed to the evaluation of the next state evaluation, replacing the old *previous* set with the current *now* set. While this loop is the main aspect of the algorithm, there is also an initialisation step for the first state, since it has no *previous* set to rely on and thus must be evaluated differently.

```
INITIALIZATION:
pre(5) = q;
pre(4) = true
pre(3) = p;
pre(2) = ¬pre(3) ∨ pre(4);
pre(1) = pre(2)

LOOP:
now(5) = q;
now(4) = pre(4);
now(3) = p;
now(2) = ¬now(3) ∨ now(4);
now(1) = now(2) ∧ pre(1);
pre(1) = now(1);
pre(2) = now(2);
pre(3) = now(3);
pre(4) = now(4);
pre(5) = now(5);
out = now(1);
```

Listing 4.2: Example of the algorithm for $H\ (p \rightarrow Y\ q)$.

The last line states that the output of the *Stateflow block* matches the value evaluated for the complete formula in the current state. The code presented in Listing 4.2 is actually the MATLAB code that appears inside the *Stateflow block*. The *Stateflow block* for this algorithm is composed by two states, one for the initialisation and the other for the loop as one can see in the following excerpt.

```
initialState = Stateflow.State(ch);
initialState.Position = [200, 200, 350, 500];
initialState.LabelString = sprintf({{ initial }});

mainState = Stateflow.State(ch);
mainState.Position = [600, 200, 350, 500];
mainState.LabelString = sprintf({{ main }});
```

This is followed by the definition of the only three transitions in this chart: the first transition that leads into the initial state, the transition from the initial state to the main state, and the self loop transition in the main state.

```
entry = Stateflow.Transition(ch);
entry.Destination = initialState;
entry.DestinationOClock = 0;

tr = Stateflow.Transition(ch);
tr.Source = initialState;
tr.Destination = mainState;
tr.SourceOClock = 3;
tr.DestinationOClock = 9;

selfLoop = Stateflow.Transition(ch);
selfLoop.Source = mainState;
selfLoop.Destination = mainState;
selfLoop.SourceOClock = 12;
selfLoop.DestinationOClock = 3;
```

Finally, the *pre* and *now* sets are defined, based on the number of subformulas, as well as the output of the Simulink chart.

```
pre_arr = Stateflow.Data(ch);
pre_arr.Name = 'pre';
pre_arr.DataType = 'boolean';
pre_arr.Props.Array.Size = '{{ subformulas|count }}';

now_arr = Stateflow.Data(ch);
now_arr.Name = 'now';
now_arr.DataType = 'boolean';
now_arr.Props.Array.Size = '{{ subformulas|count }}';

out = Stateflow.Data(ch);
out.Name = 'out';
out.DataType = 'boolean';
out.Scope = 'Output';
add_block('simulink/Sinks/Out1', 'untitled/out');
set_param('untitled/out', 'position', [1300, 575, 1320, 595]);
add_line('untitled', 'Chart/1', 'out/1');
```

This behaviour is encoded in the *PastFormula* module that extends the monitor abstraction. The subformulas are extracted while rendering the MATLAB script to build the monitor while both the the initialisation and the loop run in the *Stateflow block* with the rest of the monitor. This

4.2 FROM A SPECIFICATION LANGUAGE TO A MONITOR

After analysing the extracted requirements and formalising all the formal specifications we intend to check, we are ready to trigger the monitor generation process. This process goes through each of the specifications and identifies if they are either SpeAR or SALT formulas. When a SpeAR formula is identified, it is immediately converted into a monitor through a series of transformations. On the other hand, SALT formulas are first collected and only then processed in simultaneous. This is because the first step of this process relies on the SALT compiler which has a big initialisation overhead and thus we want to avoid calling it multiple times. Once the compiler completes, we handle its output and transform it until we are able to generate a runtime monitor.

The parsing process for SALT and SpeAR, while different, end both in the monitor abstraction and thus have many similarities. Figure 15 is a rough process of the process that formal specifications go through. In reality, there are other transformations involved in the process, but the core solution involves parsing the formal specification into an intermediate representation, usually a temporal formula, and then extract the necessary data to build the monitor, namely signal names and mathematical expressions.

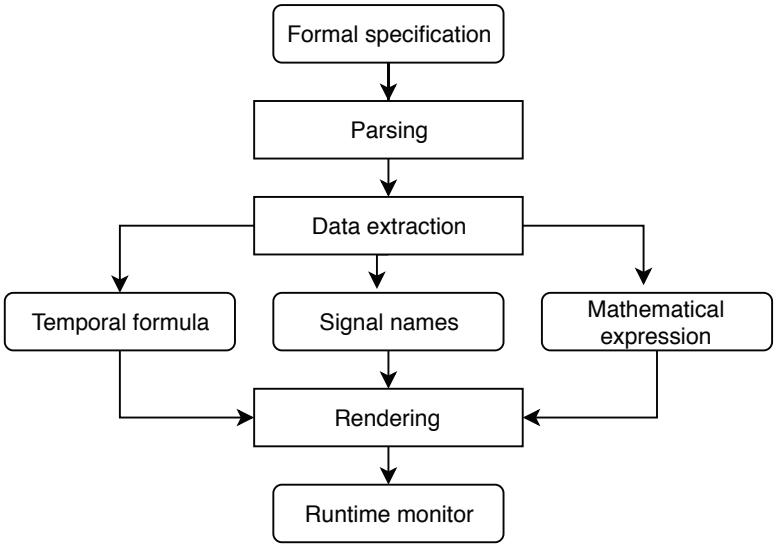


Figure 15: Rough example of the process that formal specifications go through.

This data extraction step heavily depends on the specification we are analysing and thus we have an unique module for each one. The following sections will give more details about how, for each language, we transform the specification into a temporal formula, collect the signal names they refer to, and remove the mathematical expressions from the formulas.

4.2.1 *SpeAR*

The process of transforming SpeAR specifications into monitors starts with a parsing stage. In order to perform this task we had to modify the ANTLR4 grammar provided with SpeAR. This modifications were necessary because:

1. the grammar was too loose and allowed for semantically incorrect specifications.simplifications.
2. we only support a subset of the language and thus we excluded some operators.
3. we are only interested in parsing constraints and not the entire SpeAR documents.

We chose not to focus on a complete document because it is divided in several parts and we did not to deal with the complexity of adding a layer to our tool that would abstract this concepts to the different specification languages: proposition variables are divided in inputs, outputs and state (internal variables) while constraints are divided in assumptions, requirements and properties. While this allows for interesting features such as verifying logical entailment and logical consistency, it would require the end user to handle these concepts manually which would be a cumbersome task when dealing with large requirement's documents..

The parser begins by checking if the constraint is a pattern since those are handled in a completely different manner. They have different data extraction logic and they rely on a deterministic finite automaton for verification in rather than the recursive semantics of past LTL.

For constraints that are not patterns, we start by changing them to prefix notation, following by the normalisation of operators that can be written through multiple tokens such as the **historically** operator which can also be written as **H**.

```
a) H (if p > 2 then previously q)

b) [H [if [> p 2] then [Y q]]]

c) [H [-> [> p 2] [Y q]]]
```

Listing 4.3: a) Normal SpeAR specification b) Constraint after changing to prefix notation and normalisation c) Constraint after desugaring

if $param_1$ **then** $param_2$	$param_1 \rightarrow param_2$
if $param_1$ **then** $param_2$ **else** $param3$	$param_1 \rightarrow param_2 \wedge \neg param_1 \rightarrow param_3$
while $param_1$ **then** $param_2$	$param_1 \rightarrow param_2$
before $param_1$	$\neg(once\ param_1)$
never $param_1$	$historically\ \neg param_1$
$param_1$ **equivalent to** $param_2$	$param_1 \rightarrow param_2 \wedge param_2 \rightarrow param_1$

Table 1: Examples of desugaring complex SpeAR constructs.

Once the pattern is normalised, we proceed to desugar complex constructs such as *while* and *never* operators into a set of simpler operators. Figure 4.3 demonstrates the initial transformations the specification described in *a)* goes through. It is converted to prefix form in *b)* and in *c* the *previously* operator is normalised while the more complex *if* operator is desugared into a logical consequence.

Now that the formulas are in their simplest version, we are able to starting analysing them in order to extract all the necessary data required by the *PastFormula* abstraction which will render the entire monitor, including the verification logic associated with the recursive semantics of past LTL. The core of this step is the analysis we do on the mathematical expressions, gathering metadata such as the constants, variables, references to values on previous states, and the mathematical expressions themselves. Once this analysis is complete, we replace the mathematical expressions, which are evaluated separately, by propositional variables that assume the value of the evaluated mathematical expression in the current state.

This analysis step is recursive over the mathematical expression, descending gradually sub expressions tree. To identify variables and constants we rely on regular expressions, once we reached the roots of the sub expressions tree, and use the convention we specified earlier: variables are lower case, while constants are upper case. For each match we find, we add it either to the *variables* set or the *constants* set.

If any of the sub expressions includes a *previous* operator, which is the only temporal operator that can appear inside a mathematical expression, we replace such expression by a new identifier of type *PRE_signal_name*. These identifiers are added both to the set of *variables* and the set of *previous* values. The math expressions themselves are transformed back into the form, so that they are ready to be evaluated as a MATLAB expression. For every relational expression, we generate an unique identifier and add the expression to the mathematical expressions set. Inside the temporal formula, the expression is replaced by its identifier, assuming the purpose of propositional variable.

All this analysis logic is done by the *SpearFormula* module, which is an extension of *PastFormula* abstraction. With this analysis we obtain all the necessary metadata to build the Simulink monitor.

Listing 4.4 demonstrates the results of applying the processes described above to a SpeAR specification. The *if* condition was transformed into a logic consequence and the mathematical expression no longer appears in the temporal formula, which is now in prefix notation. All the metadata necessary to build the monitor was also collected successfully.

```
Original SpeAR specification:
    - H (if p > MAX_VOLTAGE then previously q)

Temporal formula after analysis:
    - H [-> m1 PRE_q]

Metadata collected:
    - Variables: {p, q, PRE_q}
    - Previous: {PRE_q}
    - Constants: {MAX_VOLTAGE}
    - Mathematical expressions: {m1: p > MAX_VOLTAGE, m2: PRE_q}
```

Listing 4.4: Example of the analysis of a SpeAR specification and the transformations it is subject to.

That said, patterns are handled in a different manner. If we identify that abstract syntax tree of a SpeAR constraints corresponds to a pattern, it goes through the same process of metadata extracting. However, each pattern has a well known transformation into a deterministic finite automaton, so SpeAR patterns rely on a DFA instead of the recursive semantics of past LTL. Different instantiations of the same pattern vary only on the metadata: the variables set, the previous set and the mathematical set.

From this step onward, all the formulas (either free formulas or patterns) are an extension of the monitor abstraction and thus the MATLAB scripts can be rendered.

4.2.2 *SALT*

SALT provides a compiler and thus it is not necessary for us to handle the parsing logic by ourselves. The compiler supports multiple options for its output syntax, such as SMV syntax or SPIN syntax – both well known LTL model checkers. Although none of its options suit us, it still provides us the option to extend the compiler with our own plugins, which allows us to customise its output. SALT can output formulas either in LTL or Past LTL depending on the operators that were used in the SALT formal specification. While we need the resulting LTL formulas in their canonical form, our backend for Past LTL expects formulas in a prefix notation, which is an issue that is simpler to approach while we still have knowledge about the AST of the formula. Thus, we chose to write a plugin for SALT instead of building another parser and transforming the formulas on our backend.

In short, the SALT compiler receives SALT specifications as input and will output either canonical LTL, or Past LTL in prefix form. Our tool distinguishes the format being outputted by first trying to interpret the past formulas and, if this fails, handles it as a future formula.

When our backend successfully parses a Past Formula from the SALT compiler it instantiates it as *PSaltFormula*, an extension over the *PastFormula* abstraction. Similarly to the *SpearFormula*, this mainly deals with the extraction of metadata, and replacement of mathematical expressions by propositional variables. Remember that mathematical expressions are built upon the quoted propositions that allow for arbitrary text inside a formula. Thus, we can not rely on the abstract syntax tree as we did for SpeAR.

Instead, we gradually consume the text while looking for specific patterns: variables, constants, the previous operator, mathematical operators and number literals. We build a new expression by appending every match exactly as it was found on the original expression, except for the previous operator in which case we generate a new identifier and add it in place of the original match. This new expression is added to the set of mathematical expressions that forms the metadata as well as all the variables, constants and previous operators that we were able to match.

In the case of LTL formulas we need to transform them into its equivalent DFA. For this task we rely on a project, named Spot (Duret-Lutz et al., 2016), that provides a tool to translate LTL formulas into their respective DFA representation. This tool outputs the DFA as a *Hanoi Omega Automata* which we parse and then convert into a *FSaltFormula*, the extension of the *DFA* abstraction for SALT.

```
HOA: v1
name: "G(!a | Fb)"
States: 2
Start: 0
AP: 2 "a" "b"
acc-name: Buchi
Acceptance: 1 Inf(0)
properties: trans-labels explicit-labels state-acc complete
properties: deterministic stutter-invariant
--BODY--
State: 0 {0}
[!0 | 1] 0
[0&!1] 1
State: 1
[1] 0
[!1] 1
--END--
```

Listing 4.5: Example of a DFA in the Hanoi Omega Automata representation.

Listing 4.5 shows the output of the tool for the temporal formula $G\ (a \rightarrow F\ b)$. The HOA representation starts by stating some metadata about the DFA such as:

1. the initial state 0 in the field **Start**.
2. the number of atomic propositions and their respective identifiers in the field **AP**.
3. the number of acceptance states, their respective id and some other properties in the field **Acceptance**.

The metadata is followed by the DFA itself, by stating all states and all transitions as well as the conditions that enable such transitions.

In our case, all atomic propositions correspond to mathematical expressions from which we need to follow the usual procedure of metadata extraction, and then replacing them by simple propositional variables that map into their evaluation at runtime.

From this step onward both future and past formal constraints are an extension of the *monitor* abstraction, and thus we are ready to render their respective MATLAB scripts that generate the equivalent monitor.

4.3 Extraction DSL

The extraction DSL is based on parser combinators, high-level functions that receive parsers as arguments and return a new parser, allowing us to define more complex parsers by combining simpler parsers. The *modifiable pattern*, for instance, builds upon the *modifier* and *pattern* parsers.

$$modifiable\_pattern = optional(modifier),\ pattern$$

Each parser encodes a regex construction and this allow us to parse the DSL into a proper regex that can be used to extract requirements. The supported regex features are described below.

- Capture groups, which are essential to extract specific pieces of data from the text.
- Character classes like digit, lowercase, any char, and many others.
- Placement match such as start_with, end_with
- Pattern quantifiers such as *zero or more*, *one or more* and *optional* or *numbered patterns*.

- Range quantifiers such as *between*, *up to* and *at least*.
- Sequences of possible patterns that can be matched, encoded as *one of*.
- *pattern negation*, *start of string* and *end of string* expressions.
- Support for actual regex patterns and custom extractors that can be reused as patterns.

While Table 2 shows simple translation rules that are performed by the parsers, there are more complex translation rules such as the *modifiable pattern*. This parser starts by compiling the pattern and, if there is a modifier, applies the modifier to the result. The compilation process is aware of its environment so that the user can rely on previously defined extractors to build new ones on top of them.

followed_by $expr$	$expr$
starts_with $expr$	$\widehat{e}xpr$
ends_with $expr$	$expr\$$

Table 2: Translation rules performed by parsers.

4.4 FINAL RESULTS

In this section we intend to do an overview of our workflow, this time focusing on the transformations the formal specifications go through instead of how they are performed. Consider the extractor described in Listing 3.2 that compiles down to

$$(REQ\_[0-9]+) : \backslash s * (.+)\backslash n\{2\}$$

and with which we are able extract the requirements we saw previously in Listing 3.1. From these requirements, we are interested in the first two:

```
REQ_1: When the system is in state_1 and signal_3 + signal_2 > 100, the system shall
    transition to state state_3.
REQ_2: Once the system is in state_3 occurs, then the system will be in state_4
   somewhere in the future.
```

We formalise *REQ_1* using SALT and *REQ_2* using SpeAR as previously shown in Listing 3.4 and Listing 3.5, respectively. With both formulas formalised, we can trigger the monitor generation process.

```
REQ_1: assert always (if 'state == STATE_3' then eventually 'state == STATE_4')
```

```
REQ_2: historically if previously state equal to STATE_1 and signal_3 + signal_2 >
    100 then state equal to STATE_3
```

For *REQ_1* this process starts by invoking the SALT compiler. Since *REQ_1* is a future formula, the compiler output will be pure LTL which will then be processed by Spot. Finally, we analyse the mathematical expressions to extract all the necessary metadata. The replacement of mathematical expressions by boolean propositions was already handled by Spot, and it is only necessary to readjust the name of boolean proposition to match the *identifier* we give to the math expressions. Listing 4.6 shows the DFA outputted by Spot after adjusting the propositional variables' names.

```
State: 0 {0}
[!expr0 | expr1] 0
[expr0&!expr1] 1
State: 1
[1] 0
[!1] 1
```

Listing 4.6: Excerpt of Spot output for REQ_1.

For *REQ_2* this means changing it into prefix form, followed by normalising and desugaring its operators, resulting in a temporal formula with just the operators from Past LTL semantics. To finish the processing this formal specification goes through, we extract the mathematical expressions while simultaneously collecting the metadata. Listing 4.7 describes the final result.

```
Temporal formula:
    - H [-> [&& m0 m1] m2]

Metadata collected:
    - Variables: {PRE_state, signal_3, signal_2}
    - Previous: {PRE_state}
    - Constants: {STATE_1, STATE_3}
    - Mathematical expressions: {m0: PRE_state == STATE_1, m1: signal_3 + signal_2 >
        100, m2: state == STATE_3
```

Listing 4.7: Resulting data after processing REQ_2.

With both requirements processed, we are able to automatically build the runtime monitors. These monitors can then be plugged into the system model and run in parallel with

it during simulations. By observing the monitor output, the engineer is able to detect if the any of the properties did not hold and act accordingly by either fixing the design or checking the consistency of the requirements.

5

CONCLUSION

Critical systems require a high level of confidence in their correctness before they can be deployed. Today, the industry already relies on model-based engineering and simulations to improve the confidence in their designs by doing an early exploration of the behaviour their solution exhibits. The proof of concept described in this dissertation builds upon these processes and, in order to verify the system behaviour, it relies on runtime monitoring instead of typical unit testing. As such, instead of verifying the system by relating pairs of input/output, we focus on verifying its correctness over execution traces, allowing us to state properties that the system must satisfy. Since we automatically generate the monitors from the formalised requirements, the runtime monitors we generate are also checking the design compliance with the elicited requirements.

Our workflow is also heavily focused on reducing the obstacles that requirements engineers face when using tools based on formal methods. To achieve this, we chose to support high-level specification languages with a syntax as close to that of natural language as possible. Thus, both SALT and SpeAR specification languages provide expressive operators while not requiring strong formal methods foundations. Furthermore, we do not expect requirements engineers to be versed in writing regular expressions and thus provided a DSL to ease the extraction of requirements.

In the end, we were able to successfully generate runtime monitors for the Simulink environment based on SALT and SpeAR specifications.

5.1 FUTURE WORK

Currently, our tool limits the flexibility provided by SpeAR and SALT by not supporting the definition of user-defined operators. There are several ways in which we could overcome this limitation, for instance, by supporting a library with operators that the users could contribute to.

We also wish to add support for the SALT operators whose semantics are based on Timed LTL. This would allow our tool to support yet another class of properties, i.e. properties with real time constraints.

It would also be interesting to explore the concept of code generation by inspecting the system model and the associated runtime monitors and generating both the application and the executable equivalent of the monitors. Such task would be a challenging one since the monitors would require a low execution overhead in order to keep it from impacting the system. Another problem that would require attention in this approach would be the concurrency between the application and the monitors, which could not impact the correctness of the requirements. This approach would also be interesting to monitor non-functional requirements such as memory consumption or energy available. Such requirements could be follow a similar methodology to the one described in this dissertation but targeting AADL instead of Simulink.

BIBLIOGRAPHY

E. Hoffman B. Rodes M. Aiello J. Davis A. Fifarek, L. Wagner. Spear v2.0: Formalized past ltl specification and analysis of requirements. *NASA Formal Methods Symposium, May 2017*, 2017.

M. Ahmadian. Model based design and sdr. *IET Conference Proceedings*, pages 19–19(1), January 2005. URL https://digital-library.theiet.org/content/conferences/10.1049/ic_20050389.

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*, volume 26202649. 01 2008. ISBN 978-0-262-02649-9.

Howard Barringer, Michael Fisher, Dov Gabbay, Richard Owens, and Mark Reynolds. The imperative future: Principles of executable temporal logic. 01 1996.

Ali Behboodian. Model-based design. 2, 05 2006.

Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, October 2016. doi: 10.1007/978-3-319-46520-3_8.

M.B. Dwyer, George Avrunin, and James Corbett. Patterns in property specifications for finite-state verification. pages 411–420, 02 1999. doi: 10.1109/ICSE.1999.841031.

Andrew Gacek, Andreas Katis, Michael Whalen, John Backes, and Darren Cofer. Towards realizability checking of contracts using theories. 02 2015. doi: 10.1007/978-3-319-17524-9_13.

Volker Gruhn and Ralf Laue. Patterns for timed property specifications. *Electronic Notes in Theoretical Computer Science*, 153:117–133, 05 2006. doi: 10.1016/j.entcs.2005.10.035.

Sascha Konrad and Betty Cheng. Real-time specification patterns. *Proceedings - 27th International Conference on Software Engineering, ICSE05*, pages 372– 381, 06 2005. doi: 10.1109/ICSE.2005.1553580.

Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2:255–299, 11 1990. doi: 10.1007/BF01995674.

Nicolas Markey. Temporal logic with past is exponentially more succinct. *Bulletin of the EATCS*, 79:122–128, 01 2003.

A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Oct 1977. doi: 10.1109/SFCS.1977.32.

Philipp Reinkemeier, Ingo Stierand, Philip Rehkop, and Stefan Henkler. A pattern-based requirement specification language: Mapping automotive specific timing requirements. pages 99–108, 01 2011.

Jonathan Streit. Salt - language reference and compiler manual. 2006.