A Modular Approach Towards the Runtime Verification of Machine Control Applications
A user-friendly toolset aimed at the industry

Master Thesis Computing Science

Author:
Sam Jansen

Supervisor Radboud:
prof. dr. Jozef HOOMAN

Supervisor TNO:
dr. Jacques VERRIET

Second reader Radboud:
dr. Daniel STRÜBER

July 2020
Abstract

Machine control applications (MCAs) are applications that perform and coordinate machine control functionality. Such applications may involve hundreds or thousands of interacting functions. It is practically impossible to get an overview of all these functions running in parallel. Hence, it is a considerable challenge to verify if a machine control application conforms to its requirements. Although many verification solutions exist in the literature, none provide a complete solution suitable to the MCA domain and its users. In this thesis, we propose a proof-of-concept of a modular toolset for the verification of machine control applications. This toolset allows for the verification of behavioural properties using a runtime verification approach. The toolset consists of four well-separated tools, and can, therefore, be extended with new tools to suit the needs of a particular company. The toolset includes a monitoring tool that can detect all property violations during an MCA execution, thereby giving more insight in the MCA behaviour. The toolset also includes property language and result visualisation tools aimed directly at people in the MCA domain. Results of a small user study performed shows that the toolset is deemed easy to use by people from the domain and that it is clear how the toolset can be extended. As we deliver a proof of concept, further testing and development will be required to make the toolset ready for deployment.
Acknowledgements

During the research described in this thesis, I learned a lot about performing scientific research, machine control applications, Domain Specific Languages and the runtime verification technique in general. I would, therefore, like to thank TNO for providing me with the opportunity to perform this research in one of their projects.

Secondly, I would like to thank my supervisor, Jacques Verriet, for the very good guidance and support I received while performing this research. I would also like to thank my Radboud supervisor Jozef Hooman for all his advice and feedback.

Finally, I would like to thank all of my contacts at Cordis Automation for the time they have taken to help me with my research activities.

Sam Jansen,
Eindhoven, 8 July 2020

The research is carried out as part of the ITEA3 18030 MACHINAIDE project under the responsibility of ESI (TNO) with Cordis Automation as carrying partner. The MACHINAIDE research is supported by the Netherlands Organisation for Applied Scientific Research TNO and Netherlands Ministry of Economic Affairs.
# Contents

1 Introduction 3
   1.1 Project Setting 3
   1.2 Problem Statement 3
   1.3 Goal 4
   1.4 Research Questions 5
   1.5 Thesis Outline 5

2 Preliminaries 6
   2.1 Machine Control Applications 6
   2.2 Cordis Automation 7

3 Verification Approach 10
   3.1 Verification Techniques 10
   3.2 Runtime Verification 11
   3.3 Conclusion 13

4 Related Work 14

5 Properties 16
   5.1 Elements 16
   5.2 System Requirements 17
   5.3 System Properties 18
   5.4 Formalism 19
   5.5 Conclusion 21

6 Architecture 22
   6.1 Modular Toolset 22
   6.2 Conclusion 24

7 Property Language 25
   7.1 Previous Work 25
   7.2 Language Experiment 31
   7.3 Language Design 33
   7.4 Abstraction 39
   7.5 Conclusion 40

8 Monitoring system 41
   8.1 Previous Work 41
   8.2 Monitoring System Solution 45
   8.3 Conclusion 56

9 Verification Result Feedback 57
   9.1 Goal of Feedback 57
   9.2 Previous Work 58
   9.3 Feedback Approaches 59
   9.4 Conclusion 60

10 Verification Data and Formats 61
   10.1 Machine Control Application Data 61
   10.2 Toolset Communication Formats 62
   10.3 Conclusion 64

11 Evaluation 65
   11.1 Toolset Implementation 65
   11.2 Evaluation User Role 66
   11.3 Evaluation Maintainer Role 69
   11.4 Conclusion 70

12 Conclusion & Future Work 71
   12.1 Conclusion 71
   12.2 Future Work 73
   12.3 Reflection 74

Bibliography 75

A Cordis SUITE 80

B MCA Requirements 82

C Language Experiment Examples 86

D Language Grammar 88

E Automata 91

F Data Format 108

G Evaluation Exercises 110
Chapter 1

Introduction

1.1 Project Setting

The project described in this thesis is part of the European ITEA Machinaide\(^1\) project. The Machinaide project is a collaboration of 4 countries and 19 partners which aims at the development of digital twins for small and medium-sized companies. In the Netherlands, Cordis Automation (country co-ordinator) and its customers Additive Industries and Lely, together with TNO and the Eindhoven University of Technology (TU/e) are involved in the project. The goal of the Machinaide project in the Netherlands is to apply the expertise of TNO and TU/e regarding verification & validation and design improvements to a case of Cordis Automation. This project will be part of TNO’s efforts regarding Machinaide.

1.2 Problem Statement

Cordis SUITE\(^2\) developed by Cordis Automation, is a platform for low-code software development for machine control applications (MCAs). It provides dashboards, model specification, simulation, and code generation for several PLC platforms. Code is generated based on models, which are specified using a subset of UML. These models can be created using a graphical interface and allow for a platform-independent description of functionality. Among other things, this simplifies the development process and allows for a more unobstructed view and better application maintainability, according to Cordis. As models are used instead of a programming language, users of Cordis SUITE can more easily involve users from other disciplines within the company, who have no coding knowledge, when reasoning about applications.

Although the generated PLC code is ‘bug-free’ according to Cordis, the freedom of design brings some challenges. Cordis SUITE runs static checks on the models, but these are very simplistic and only check if the models are valid UML models. Methods to test if the generated code does what is intended are not supported. Cordis does provide two means of inspecting their application simulations. Namely, a simple dashboard to issue commands and observe variables and a visual representation of the machine in Unity\(^3\). Creating this visual representation, however, is time-consuming. Both methods only provide data of operations which are currently running. Inspecting complex machines, which consist of hundreds of components, is therefore practically impossible. This problem leaves Cordis with a desire for a more robust solution which allows their users to verify that their designed applications are functioning as intended.

\(^1\) https://itea3.org/project/machinaide.html \(^2\) https://cordis.nl \(^3\) https://unity.com
1.3 Goal

This project aims to tackle the challenges described above by providing a first milestone in verifying that an MCA adheres to its requirements. Detecting problems earlier could lead to fewer design iterations, which saves money and time. The solution described in this thesis is not meant to be the be-all and end-all of the verification of machine control applications but is believed to be a valuable first step. The case of Cordis Automation is used as an information baseline. This is done as Cordis and their customers are considered a projection of our targeted users, which are users in the machine control domain.

This project consists of research which aims at a solution that monitors operational MCAs and automatically detects violations of desired behavioural requirements. This is to be achieved using a language which allows users to specify requirements in the form of properties. Thereby replacing the need to monitor the entire machine visually and allowing for a much broader, but also more detailed view of its execution. Execution data originating from MCAs can then be checked against these properties, after which the results are shown to the user. Our focus is on checking internal MCA data. This categorises the solution as a toned down white-box approach. Figure 1.1 shows the basic schematics of the solution.

General requirements of the solution are:

- **Based on domain**  As the user is responsible for creating the properties used for verification, the language should match the concepts present in the machine control domain. This way, someone with expertise in the domain can understand the properties.

- **Ease of use**  User-friendliness is also an essential aspect of the language. A complex language would only add more complexity to the design process, thereby increasing the chances of errors, instead of reducing them. The visualisation of results should also be easy to understand.

- **Generic**  Although the case of Cordis is used as an information baseline, the solution is aimed at the broader domain of the machine control industry. The solution should, therefore, apply to many types of users with many different needs.

- **Extendability**  Although the generic requirement specifies that the solution should apply to many types of users that have different needs, an eventual solution cannot possibly cover every MCA verification wish companies might have. Because of this, the solution should be extendable with new functionality by people who have knowledge of the internal concepts used.

1.3.1 Roles

Two types of roles can be distinguished when talking about the verification solution: users and maintainers. Users use the solution to specify properties and interpret verification results. Maintainers extend the verification solution with more functionality. Users are more interested in the ease of use requirement, whereas maintainers are more interested in the extendability requirement. Users do not have to be familiar with the internal concepts of the verification solution, maintainers do. In our context, the users of Cordis SUITE fulfil the user role, not the maintainer role.
1.3.2 Scope

This research only focuses on verification based on execution data of a machine control application. Therefore, this solution can only confirm something has not gone wrong during a specific application execution (regarding properties), and not that something will not go wrong eventually in another execution. The latter problem revolves around proving the absence of such error. This is something the TU/e will focus their efforts on using mCRL\(^4\). As this research has a limited time budget, other additional verification approaches will not be explored.

1.4 Research Questions

To add structure to the research, the problem explained in the previous sections has been divided into smaller sub-problems. These can be solved by answering the following questions:

- **RQ1**: What properties should be specifiable?
- **RQ2**: How can a property language be structured for user-friendliness?
- **RQ3**: What abstraction level should be used when specifying properties?
- **RQ4**: What machine control application data is needed for verification using properties?
- **RQ5**: How can the modularity of the verification solution be guaranteed?
- **RQ6**: What is a suitable technique to verify machine control application data against properties?
- **RQ7**: What data is required to understand and process the verification results?
- **RQ8**: Does the verification method improve upon the current verification process?

The results of these questions have been combined to answer the following main research question:

*How to define and verify properties of machine control applications in a user-friendly and generic way?*

1.5 Thesis Outline

The goal of this thesis is to describe all aspects of the project that has been carried out. These include all performed research, assumptions and decisions made, designs and deliverables. In Chapter 2, the main background for this research is discussed. Chapter 3 describes the general approach used towards the verification of MCAs. Chapter 4 describes related work which has been done in the past regarding this approach. Chapter 5 describes the properties which should be specifiable and validatable of MCAs. The architecture of our solution is described in Chapter 6. In Chapter 7, the property specification language is discussed. Chapter 8 describes the verification system. In Chapter 9 approaches towards providing users with feedback relating the results of a verification are discussed. Chapter 10 describes the data formats used in our solution. In Chapter 11 the complete solution is evaluated, and Chapter 12 provides a conclusion where all research questions are answered.

\(^4\) https://www.mcrl2.org/web/user_manual/index.html
Chapter 2

Preliminaries

This chapter describes the main background and preliminaries for this research. In Section 2.1 machine control applications (MCAs) are reviewed. Section 2.2 details multiple aspects of Cordis Automation, including its software Cordis SUITE and its customers.

2.1 Machine Control Applications

Machine control applications are, as the name suggests, applications which run on control systems. Such a control system manages, directs or regulates the behaviour of machines [30]. Most of these control systems are implemented using micro-controllers or more specialised programmable logic controllers (PLCs).

2.1.1 Programmable Logic Controller

According to Netto and Bagri [61], PLCs are hardware and software engineered microcomputers, used to provide industrial control operations. A PLC can physically be connected to several I/O devices. Input devices (light sensors, push buttons) give a real-time status of variables in the form of signals. Output devices (motors, actuators) receive signals from the controller.

A PLC works by performing a process known as scanning. This scanning is a continuous cycle and consists of three steps: (1) reading data from input devices, (2) processing this data by executing a program, and based on decisions made during the program execution, (3) administering signals to output devices [61]. This scanning automates a process or machine and is also known as the sense-think-act paradigm. In practice, the scan cycle time mostly ranges from around 0.1ms to 50ms, according to Cordis.

There are several languages in which a PLC program can be written. The IEC 61131-3 standard defines a total of five languages: Function Block Diagram, Instruction List, Ladder Diagram, Sequential Function Chart and Structured Text [48]. Large providers of PLC systems like Beckhoff[6] and Siemens[7] employ their own proprietary languages.

Large industrial machines can contain a multitude of sensors and actuators. Cordis mentions that the control of such a large amount of I/O is in practice often split up across multiple controllers. Here, every controller is responsible for a physical section or aspect of the machine. PLCs can form networks to communicate with each other [61].

2.2 Cordis Automation

In this section, details about Cordis Automation, its customers and its software Cordis SUITE are discussed.

2.2.1 Cordis SUITE

This section illustrates some aspects of the workings of Cordis SUITE. A simplified overview is given, where all details not deemed relevant are left out.

Cordis SUITE can be used to generate MCAs in multiple languages. These include several IEC 61131-3 languages and the languages Beckhoff and Siemens employ. In Cordis SUITE no applications can be generated which span multiple controllers. For machines with multiple PLCs, multiple independent MCAs have to be designed.

Models

Machine control application functionality is specified in Cordis SUITE using one static and multiple dynamic models. The static model is an UML class diagram, it allows users to specify components (also known as machine parts) of a machine in an object-oriented manner, see Figure 2.2. Dynamic models add behaviour to these components in the form of state machines or activity diagrams. As both types of models practically serve the same purpose of adding behaviour to components, only state machines are discussed here. See Figure 2.1 for a state machine in Cordis SUITE. Multiple different state machines can be attached to one component.

Figure 2.1: Cordis SUITE state machine

8 https://www.omg.org/spec/UML/About-UML/
Components

In the static model, relations between components can be defined. Here, a component can have multiple child components, which can have children of their own. Multiple instances of a component can be active at runtime. A conveyor component for example, can have two sensor components and one motor component as children, see Figure 2.2. To be able to manage several instances of a component at runtime, each instance is required to have a unique name. In Figure 2.2 this could resolve to the two sensor instances being named BeginSensor and EndSensor. Relations between components are designed in a tree structure. A static model can only contain one component tree.

In Cordis SUITE, sensors and actuators can also be modelled as components. Here a component acts as the interface of such sensor/actuator. Cordis provides multiple pre-built components of sensors and motors with their software.

Besides state machines, a component can hold multiple variables and one global status. This status is in the form of a user-defined error or warning and is used to signal that something is wrong with the component instance. The status can be triggered from within a state machine.

Communication

Communication between components can be achieved in two ways: by using commands, or by using inputSignals/outputSignals. For more information on the latter, see Appendix A. Commands are events which are broadcast by a component down the tree, to whichever other components are ‘subscribed’ to the commands. A command can contain parameters. Components can only communicate with components lower in the tree. This access is transitive, meaning children, grandchildren, etcetera are all accessible. Sibling and parent components are not accessible. This means the motor component from Figure 2.2, for example, cannot send a command to the conveyor or one of the sensors.

State machines & execution

The state machines in Cordis SUITE are UML state machines with some additional functionality. Internally, during the code generation, state machines are transformed into large if-then-else statements. For more information on the state machines used by Cordis SUITE, see Appendix A.

As stated in Section 2.1.1 the second step in the PLC scan cycle is the MCA execution. During this step, all state machines of all components are executed, starting with the root of the component tree. Child components are executed in a depth-first manner. Multiple instances of the same component are executed in alphabetical order by name. The same holds for the execution order of multiple state machines of a component instance. For more information on the execution of state machines, see Appendix A.

Broadcast commands are received in the same PLC cycle they are sent. This is possible as commands can only be received by components which are further down the component tree, and these components are executed later in the cycle. Confirmation that a command reached its subscribers is not communicated back to the sender.
2.2.2 MCA Execution Data

Cordis collects MCA execution data from physical machines and simulations. Extracting enough data (size and frequency), usable for verification, from physical machines has proven to be difficult. Therefore only data from simulations, which is easily extracted, will be used in this research. Cordis provides a toolset to simulate the functionality of a PLC inside a PC. This simulation accepts MCAs written in C#, which can be generated in Cordis SUITE. The generated program runs in the .NET framework. The toolset allows Cordis to retrieve data of the MCA once a scan cycle. Cordis is not sure at what exact point in the cycle the data is taken, but they mention that all data is taken as a snapshot at the same point every cycle. Component data, as discussed previously, is available in such a snapshot. Multiple of these snapshots form a data-trace which Cordis already uses in their dashboard application.

To gather data regarding component variables, users have to attach observers to these variables. Attaching an observer to a variable activates internal methods which retrieve the value of this variable out of the system. An observer has to be attached via the Cordis SUITE GUI. Too many observers enabled at once (3000-4000) could prolong the execution by such an amount that it will no longer fit inside the PLC scan cycle time. The occurrence of this problem also heavily depends on the cycle time, the size of the generated MCA, and what PLC is used.

2.2.3 Customers

According to Cordis, many of their users are experienced PLC programmers. They are knowledgeable in the Beckhoff, Siemens, or IEC 61131-3 PLC programming languages. Many of those users are, on the other hand, not familiar with more general programming languages like Java or C. Although PLC programmers make up the primary user group, they are during the application development often assisted by users from other disciplines in the field of machine control development. For example, process engineers or electrical engineers. This is where the strength of Cordis SUITE comes in, as it is designed to be understood by people from all these different disciplines. After some workshops, all users can understand the basics of the UML models used in Cordis SUITE. Cordis mentions that no users are familiar with any concepts of formal verification.

Cordis SUITE is used to generate MCA for lots of different machines. A few examples are airport hand-luggage security systems, 3D metal printers, machines which test air-fryers, and machines which assemble car tail lights. Although numbers vary widely, the average size of an application designed in Cordis SUITE for such machines is as follows:

- 1 PLC
- 300 IO devices (sensors/actuators)
- 80 state machines
- 400 unique states
- 3,000 instantiated states
Chapter 3

Verification Approach

In this chapter, a suitable approach towards the verification of machine control applications (MCAs) is selected. In Section 3.1, several verification techniques are discussed. The most suitable technique for the problem at hand is selected and explained in detail in Section 3.2. Section 3.3 provides a conclusion. In this section, the information detailed in this chapter is used to partially answer research question RQ6, What is a suitable technique to verify machine control application data against properties?

3.1 Verification Techniques

Leucker and Shallhart mention four techniques most often considered regarding the verification of systems [50]. These are theorem proving [13], model checking [23, 62], testing [60] and runtime verification [50]. Testing and runtime verification can be categorised under dynamic analyses, which is one of two categories of verification techniques that are usually distinguished [33]. Techniques in this category require an application to be executed if it is to be analysed. The other category is static analysis. Here, the application to be analysed is not executed. Theorem proving and model checking are static analysis techniques.

All four techniques have their advantages and disadvantages. Model checking is exhaustive, meaning it considers all possible execution scenarios of an application when performing verification. This makes detection of violations caused by rare scenarios possible. However, as all possible scenarios are considered, the computational power required to verify large applications can exceed beyond what current state-of-the-art computers are capable of [33]. Theorem proving allows showing the correctness of applications similar to how a proof in mathematics shows the correctness of a theorem [50]. Theorem proving has to be applied manually and can just like model checking suffer from performance problems, as all execution scenarios need to be considered.

Dynamic analysis techniques do not have the same computational problems present with exhaustive techniques, as only one execution scenario is analysed at a time. This, however, allows rare situations which cause violations to escape detection easily [70]. Testing covers a wide field of diverse, often ad hoc, and incomplete methods for showing correctness, or, more precisely, for finding bugs [50]. Runtime verification is performed by monitoring the execution of an application and comparing data of this execution to a set of rules. According to Leucker and Schallhart, runtime verification is closest to a form of testing called oracle-based testing [50]. Oracle-based testing is performed using output-sequences. Here a so-called test-oracle, containing multiple tests, is attached to the application under test. The oracle analyses the output and determines if its tests have passed or failed. The difference between the two techniques is that an oracle is typically defined directly, whereas runtime verification rules are often generated from a higher-level language. A suitable set of input has to be
considered for testing, this is not the case for runtime verification, as it observes an application perform its regular operations. Runtime verification can, therefore, be seen as a form of passive testing. Additionally, testing might fail to properly introduce information only available at runtime, like the environment. Runtime verification does not have this problem.

As mentioned in Section 1.3.2, another sub-project within Machinaide (mCRL2), performed by the TU/e, already focuses on static verification techniques. Therefore, model checking and theorem proving will not be explored further in this research. As the goal of the research is to verify applications which are being executed, using any static verification technique is out of the question. As mentioned above, testing requires a suitable set of inputs to be determined when creating tests. A good tester might come up with inputs which trigger many different scenarios, but this can not be guaranteed. As tests are typically defined directly in a programming language, this makes it harder for non-programmers to specify or contribute to tests. As our verification solution is aimed at a broader audience than only programmers, this is not ideal. Runtime verification does not suffer from these problems. The description of runtime verification, where an application is analysed while it is running, fits the goal of this research perfectly. Because of this, of the four techniques mentioned, runtime verification is deemed the most suitable technique for overcoming the challenges of verifying machine control applications.

3.2 Runtime Verification

This section details the concept of runtime verification by describing its goal, definition and procedure.

3.2.1 Definitions

Falcone et al. describe runtime verification as the study of algorithms, data structures and tools focused on analysing executions of systems [33]. This analysis is aimed at improving the confidence in the behaviour of a system, either by improving program understanding or by checking conformance to specifications. The formal definitions of runtime verification concepts used in this section were taken from the paper of Falcone et al. [33], unless mentioned otherwise.

In runtime verification, the question whether the execution of a system adheres to given correctness properties is answered. Checking whether an execution meets these properties is typically performed using monitors [50], see the definition below. As runtime verification is performed during the execution, the possibility of acting in the form of feedback to the system, whenever incorrect behaviour is detected, exists. [50]. This feature is out of scope as this research only focuses on the verification of systems, not the repair.

Event and trace

The execution of a system can be captured as a finite sequence of events, also known as a trace. An event describes an action taken by the system or a change in the state of the system. In general, an event consists of a name and sequence of data values. \( V \) is a set of all data values of the context of the runtime verification system.

Definition 1: An event is a pair \((e, \overline{v})\) where:

- \( e \): an event name
- \( \overline{v} \): a finite sequence of values in \( V \)

In runtime verification, it is common to consider only finite traces, as monitored systems do not run forever. All executions of a system can be described by a (possibly infinite) set of finite traces.
Property

A property is a rule a system must adhere to. Properties can be specified in different formalisms or languages and can, therefore, describe different things. Properties are modelled as functions from traces to a given verdict domain. A verdict domain contains elements to which a property can evaluate. Such a domain can for example be the truth domain $\mathcal{B} = \{\text{true}, \text{false}\}$, or the probability domain of $[0, 1]$.

Definition 2: Let $\text{Prop}(A, D) = \text{Trace}(A) \rightarrow D$ be the set of all properties from traces over the set of events $A$ to the verdict domain $D$.

Monitor

A monitor is a device which consumes events and produces verdicts in the verdict domain. The relation between monitors and properties is 1:1, meaning one monitor is responsible for verifying the execution trace against one property. As monitoring is performed on a running system, execution traces are continually increasing in size. Therefore, a monitor should be designed to consider executions incrementally. When $[\varphi]$ denotes the set of valid traces given by property $\varphi$, the process of a monitor boils down to checking whether the trace $t$ is an element of $[\varphi]$.

3.2.2 Approach

Runtime verification is performed in the following four stages:

1. **Monitor synthesis**: From every property, a monitor is generated.

2. **System instrumentation**: An execution trace is extracted from the system.

3. **Execution analysis**: The execution trace is analysed by the monitors.

4. **Responses**: The monitors produce verdicts based on if the trace conformed to the properties. The monitors also provide corrective feedback to the system.

Figure 3.1 shows the general scheme of runtime verification. Falcone et al. mention different approaches to how monitoring can be performed:

- **Monitor occurrence**: Online, while the system is running. Or offline by processing a log file, which contains the execution traces.

- **Monitor location**: Inline, where the monitor is included in the code of the monitored application. Or Outline, where the monitor is an external entity.

As runtime verification considers a finite execution trace, monitors regarding liveness properties can be inconclusive when the execution ends. A liveness property specifies that an event should happen eventually. As such, an event might not have occurred yet, the monitor cannot be sure if the property should be violated or validated. What verdict is given in such case depends on the implementation of the monitoring system.
3.3 Conclusion

In this chapter, several techniques regarding application verification have been discussed. Of these techniques, runtime verification has been selected as the most appropriate technique to solve the problem of machine control application verification. This is, as the description of runtime verification, where an application is analysed while it is running, fits the goal of this research perfectly. Runtime verification also does not require any application input to be considered beforehand. Finally, the runtime verification approach supports the possibility that verification rules (properties) are generated from a higher-level, possibly user-friendly, language. Therefore, ‘runtime verification’ is partially the answer to research question RQ6 What is a suitable technique to verify machine control application data against properties? A more detailed approach towards this technique, which is used to answer this question completely, is given in Chapter [8].
Chapter 4

Related Work

This chapter provides an overview of previous efforts regarding verification systems which also cover property specification and result visualisation. As the aim is at the runtime verification technique, discussed in Chapter 3, only systems which perform runtime verification are discussed. The systems discussed here are evaluated using the requirements mentioned in Section 1.3 to determine if they are suitable solutions to the problem of this research. Systems which do not have a specification language aimed at non-experts users in the field of verification are not discussed here. As these already violate the easy to use requirement.

TNO has developed ComMA, a tool used for interface modelling and analysis [47]. ComMA uses Domain Specific Languages (DSLs) to model the interfaces and specify constraints on them. These constraints can be behavioural, timing and data related. Runtime verification is performed using POOSL [70]. As ComMA is focused on interface modelling, it does not match the domain of machine control applications (MCAs). Additionally, ComMA primarily uses a black-box approach, which conflicts with our aspirations of monitoring internal application data. This makes ComMA hard to use for the MCA domain.

TNO also used a virtual prototyping approach based on a domain model consisting of multiple DSLs to specify different aspects of an IoT application [71]. The DSLs are used to define, among other things, the application’s domain and its behaviour. One of the DSLs is used to specify properties the application should adhere to. Co-simulation and runtime verification are performed using the in-house tool JCoSim. Using this IoT approach, the entire application first needs to be modelled before it can be validated. This is time-consuming and not practical, as applications might already have been modelled by a company using a company-specific toolset. The internal co-simulation can only be fed external sensor data, making it not ideal for the verification of complete external application (simulations). Concluding, the IoT approach cannot directly be used for our goals.

Li et al. developed a specification and validation system aimed at web services interaction properties [51]. These properties are used to specify the behaviour between a user of the web service and the web service itself. The validation is performed using a tool called the Component Interaction Property Validator (CIPV [53]). This system uses interceptors to handle messages in the context of CORBA. CORBA is an architecture that computer applications use to work together over networks. This does not match the MCA domain. Additionally, the system of Li et al. uses a black-box approach, which conflicts with our aspirations of monitoring internal application data. This makes their system hard to use for the MCA domain.

---

9 https://www.corba.org
Harel and Marelly created a system that focuses on the runtime verification of GUI applications [38]. Properties of these GUI applications can be specified in time enriched versions of Live Sequence Charts (LSCs [26]). Properties can be defined using GUI element actions (e.g. button click). Sequences of GUI actions can be monitored and verified using a tool called the play-engine. As this entire system is focused on GUI applications, it cannot directly be used for the MCA domain.

In [2], Ahrendt et al. propose the StaRVOOrS framework, which allows the specification of both control-oriented and data-oriented properties of applications. StaRVOOrS uses a specification language called ppDATE, which is an extension of the DATE (Dynamic Automata with Timers and Events) [24] formalism. In a later work, Ahrendt et al. [1] added a translation from ppDATE to the DATE formalism, used as input for the runtime verification tool Larva [25]. Larva only supports the runtime verification of Java applications, making it not suitable for the verification of MCAs.

In [56] Malakuti et al. propose E-Chaser, a language-independent runtime verification system. E-Chaser composes of a front-end and back-end. The front-end enables property specification as regular expressions. The back-end provides a toolset which currently enables the generation of runtime verification modules for the Java, C and .Net implementation languages. This is done by modifying the source code to facilitate a connection to external monitors. E-Chaser’s toolset can be extended with new languages. As this system currently only supports the verification of Java, C and .Net applications, it cannot directly be used for the verification of MCAs.

Malakuti and Aksit do not present a runtime verification toolset in [55], but do present a system to create runtime verification systems in a modular way. The system uses event modules. These modules work by receiving input, processing it, and outputting some results. The output of one module can be the input of another. Malakuti and Aksit provide a EventReactor language composition framework consisting of multiple DSLs to specify the modules. Event modules containing functionality related to the runtime verification domain (e.g. trace inspection, property validation) can be specified using this framework. As the solution of Malakuti and Aksit is not a fully functional runtime verification system but rather a toolset to create one, it is not suitable to be used for verification out of the box. Its functionality is, however, considered in Chapter 6.

As can be read above, multiple systems that perform runtime verification exist. However, none of the systems can directly be applied to the MCA domain, as all systems are limited in functionality or usability. Therefore, these systems cannot directly be applied to fit the challenges of companies like the customers of Cordis Automation. Because of this, our research focuses on developing a new runtime verification solution which suits the machine control application domain better than the systems discussed here. This does not rule out that parts of these systems will be used to form our new runtime verification system. These parts are considered later in this thesis.
Chapter 5

Properties

Before a machine control application can be verified, it first has to be clear what exactly should be checked. In this chapter, it is first determined to what concepts applications generated by Cordis SUITE can be reduced, see Section 5.1. Then requirements of these applications are explored, see Section 5.2 and 5.3. In Section 5.4, collected requirements are grouped and formalised using a pattern system. Section 5.5 provides an overview of our contributions in the form of a conclusion. The information this chapter brings forward is used to answer research question RQ1 *What properties should be specifiable?*

5.1 Elements

As mentioned in Section 2.2.1, the modelling of an MCA in Cordis SUITE is centred around the components of the machine. Because of this centralised approach towards components, it is decided to take components as the centre of attention regarding requirement specification and verification. A component carries many different classes of data, of which a few have been discussed in Section 2.2.1. To properly reason about applications generated by Cordis SUITE, and machine control applications (MCAs) in general, all classes of data are reduced to four generic elements:

- State
- Variable
- Global Status
- Command

These four elements capture the basic behaviour of a component at runtime. The **State** represents the action a component is performing and is directly mapped onto the states of a component’s state machine. **Variables** represent the component’s data. Such variables also include sensor and actuator data, as sensors and actuators are also categorised as components. **Commands** are signals used in communicating between components. The **Global Status** mirrors the condition the component is in. These elements are selected as they provide a simplified view, but combined allow for detailed reasoning about a component.

The reduction to the four elements also abstracts away from the specific concepts of Cordis SUITE, thereby opening up possibilities to reason about component-based machine control applications in general. The reduction of the many classes of data in a component is performed in two steps:

1. The grouping of data classes which serve no purpose as separate entities during verification (e.g. variable/temp-variable).

2. The removal of data classes which are not useful for verification (e.g. internal component settings), or are simply not available in execution traces of simulated Cordis SUITE programs.
Reasoning about components can be done by inserting the elements in propositions (e.g. VARIABLE ‘speed’ \(<\) 10). In \[21\], two types of basic propositions are distinguished. These are event propositions, which hold for an infinitesimal time period, and state propositions, which may hold for longer time periods. We name state propositions as phase propositions to not confuse users with the state element. The state, variable and global status elements categorise as phase propositions. This is because propositions using these elements may hold for longer periods. The command element is an event proposition, as it only holds for a concise amount of time.

5.2 System Requirements

To be able to specify properties of an MCA, it first has to be clear what the requirements of such application are. As mentioned in Section 1.3, this research is aimed at the verification of system behaviour. Because of this, the focus is on behavioural requirements, which specify what a system should and should not do in a given scenario \[21\].

Cordis Automation has no clear picture of the requirements its customers would like to verify. Different customers have different systems and hence different types of requirements. Several methods have been applied to gather these requirements:

1. Examining control applications used by Cordis in their workshops and determining possible requirements of these applications.
2. Exploring anonymised requirements provided by Cordis of an old customer machine.
3. Reviewing short descriptions provided by Cordis of current customer machines.
4. Reviewing literature \[17, 19, 31, 51, 52, 54, 66, 72\] to gather information on what kinds of requirements have already been considered in other domains.

These methods are used to get a broad view of what kind of machine control application requirements exist. We want to ensure that our verification solution can support different types of use cases from all kinds of machines and companies. Due to the COVID-19 pandemic, we could not consult the customers of Cordis Automation about their requirement wishes, see Chapter 12 for more information.

An extensive list of requirements has been constructed using all methods, see Appendix B. A few requirements, specified in Table 5.1, are used as a running example throughout this chapter. All collected requirements are categorisable in three requirement categories: reliability, performance and safety. Reliability is the extent to which a system can be expected to perform the specified task \[59\]. Safety relates to the fact that the system does not reach a hazardous or unsafe state, which may lead to an accident \[59\]. The performance category contains requirements which specify details regarding quantifiable system aspects. This categorisation is used for two things: (1) To create a clearer understanding of what kinds of requirements users would like to check. (2) To define what types of requirements the verification solution presented in this thesis should support.
<table>
<thead>
<tr>
<th>Id</th>
<th>Category</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Safety</td>
<td>Temperature of the conveyor while running should not exceed 30 degrees.</td>
</tr>
<tr>
<td>R2</td>
<td>Safety</td>
<td>No more than one item may be present on a conveyor.</td>
</tr>
<tr>
<td>R3</td>
<td>Reliability</td>
<td>Items are detected using three sensors. As soon as one of them is triggered, the module should enter an error status.</td>
</tr>
<tr>
<td>R4</td>
<td>Performance</td>
<td>Once an item leaves the conveyor, a new item will be delivered in no longer than 6 seconds.</td>
</tr>
<tr>
<td>R5</td>
<td>Reliability</td>
<td>A sensor should enter a triggered state after 500 milliseconds, if it detects something for at least 1 second.</td>
</tr>
<tr>
<td>R6</td>
<td>Performance</td>
<td>The number of sensor events must be at least 100 every hour.</td>
</tr>
</tbody>
</table>

**Table 5.1: Example requirements**

### 5.3 System Properties

The requirements collected in the previous section do not have any connection with MCA concepts. To create this connection, some of the natural language is substituted by the four elements described in Section 5.1. This transforms the requirements into properties that can be used for runtime verification, see Table 5.2. The substitution allows for clear, component-based reasoning about the functionality of MCAs. All requirements were translated into this form. As the requirements are written about machines of which few details are available, some assumptions were made during the translation. We do not see this as a major concern, as these requirements are only used during the development, not for real system validation. All properties together are from here on referred to as the *property set*. See Appendix B for the *property set* in full.

<table>
<thead>
<tr>
<th>Id</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>When conveyor in ‘running’ state, variable ‘temperature’ should never be more than 30 degrees.</td>
</tr>
<tr>
<td>P2</td>
<td>Between conveyor sensorEntry entering the ‘on’ state and conveyor sensorExit entering the ‘on’ state, conveyor sensorEntry should never enter the ‘on’ state.</td>
</tr>
<tr>
<td>P3</td>
<td>After any sensor enters the ‘on’ state, next the module should enter GLOBAL STATUS ‘error’.</td>
</tr>
<tr>
<td>P4</td>
<td>After conveyor sensorExit variable ‘detected’ becomes true, conveyor sensorEntry variable ‘detected’ should become true within 6 seconds.</td>
</tr>
<tr>
<td>P5</td>
<td>When sensor1 variable ‘detected’ becomes true for at least 1 second, sensor1 should enter the ‘triggered’ state within 500 milliseconds.</td>
</tr>
<tr>
<td>P6</td>
<td>Sensor variable ‘detected’ should become true more or equal than 100 times per hour.</td>
</tr>
</tbody>
</table>

**Table 5.2: Example properties**
5.3.1 Proposition Combinations

As mentioned in Section 5.1, one can reason about MCA data using propositions. Dwyer et al. proposed boolean combinations of propositions [31]. Of these, AND, OR, NOT and XOR were used in the property mapping. To provide more extensive reasoning about sets of propositions, the composite propositions PARALLEL and ONE presented by Mondragon et al. in [58] were used. For clarity, we renamed them to ANY and ALL respectively. ANY applies to P3 in the running example, where one or more of multiple sensors can trigger an error. ANY and ALL are the OR and AND operators over a set respectively, and therefore carry the semantics described below. Here, P is a set of propositions, and I is an execution trace section related to a single system execution cycle where the propositions could hold.

\[
\text{Any}(P, I) = \exists p \in P \ [\text{holds}(p, I) ]
\]

\[
\text{All}(P, I) = \forall p \in P \ [\text{holds}(p, I) ]
\]

5.4 Formalism

As the properties in the property set are specified using regular English sentences, they could include ambiguity problems which could make them hard to understand. In this section, the properties are formalised. This allows for a better categorisation of the properties, thereby improving the overview of what a solution should support if it is to verify these properties. This section describes the formalism used to categorise the properties.

5.4.1 Patterns and Scopes

Dwyer et al. proposed a pattern-based approach towards the presentation and reuse of property specifications in [31]. Using a pattern system, they aim at assisting practitioners in categorising descriptions of system behaviour. A pattern is a generalised description of a commonly occurring requirement on the propositions sequences of a system’s execution. The pattern system allows behaviour to be formalised using a number of patterns: Absence, Existence, Bounded Existence, Universality, Precedence and Response. Here, Absence P describes that the proposition P should never hold. See Table 5.3 for more examples.

Each pattern has to be accompanied by a scope, which defines the period over which the pattern must hold. These scopes are defined below [71]. Here Q and R are event propositions. See Figure 5.1 for a visualised overview using a timeline. A grey block on the timeline denotes that the corresponding scope is open (active).

- **Globally**: A pattern has to hold the entire program execution
- **Before Q**: A pattern has to hold up to a state where Q occurs. If Q never occurs, the scope is not considered sensitive, and the pattern holds (trivially).
- **After Q**: A pattern has to hold from a given state where Q occurs.
- **Between Q and R**: A pattern has to hold between the state where Q has occurred and state where R occurs. If Q occurred at some point, but no R occurring follows, the scope is not sensitive, and the corresponding pattern holds trivially.
- **After Q until R**: A pattern has to hold after the state where Q has occurred until a state where R has occurred. If Q occurred at some point, but no R occurring follows, the scope is sensitive indefinitely, and the pattern has to hold forever.
The patterns of Dwyer et al. are qualitative, and therefore do not allow reasoning about timing requirements. Konrad and Cheng [44] and Gruhn and Laue [37] extended the pattern system to include patterns which allow for the reasoning about time. In [9], Autili et al. created a catalogue that combines, among others, these extended pattern systems. They also extended this catalogue with a few newly identified patterns.

The catalogue of Autili et al. is used to formalise the property set by mapping every property to a pattern and a scope. This mapping categorises every property into a pattern-scope combination by matching the semantics of the property to the semantics of the pattern and scope. Table 5.3 shows all patterns which were used in the mapping, grouped by author. As Gruhn and Laue did not name their patterns, we gave the Bounded Absence pattern its name. The catalogue was used as it is based upon a proven categorisation of requirement specifications [32]. Dwyer et al. used a survey to prove that 92% of the 555 qualitative properties they collected can be mapped onto one of their patterns [32]. Because of this, the catalogue of Autili et al. provides a good formal basis for any further reasoning about properties. Its patterns and scopes are also easily mapped onto the property set. Parts of the catalogue are also used in multiple papers on requirement formalisation [19, 40, 51, 66, 71, 72].

Not all properties could be mapped to a pattern of the catalogue. In our running example, only properties P1, P2 and P5 could be mapped. Four new patterns were developed to cover the properties which could not be mapped, see ‘new patterns’ in Table 5.3. The Restricted Recurrence pattern is an adjustment of the already existing Bounded Recurrence pattern. Here, the minimum recurrence time is replaced with a maximum recurrence time. The Strict Response pattern is purely syntactic sugar of the Bounded Response pattern, with the time of one PLC execution cycle filled in. (Although commands within Cordis SUITE are processed in the same execution cycle they are sent, explained in Chapter 2, Cordis is not sure when exactly during a cycle data is extracted. To overcome this and to generalise outside of Cordis SUITE, the time of one cycle is deemed the minimum time required before a response can occur.) Using the Prefaced Bounded Response pattern, one can specify the minimum duration a proposition has to hold before another proposition should occur. This can, for example, be used to ignore inconsistencies in a sensor (e.g. a fly would trigger the sensor for 0.1 seconds, but an actual item always is at least 1 second in front of the sensor). The Duration Bounded Existence pattern could, for example, be used for throughput measurements, by checking that at least ten items pass by a sensor every hour. This pattern is an extension of the one time-bounded chain pattern of Gruhn and Laue [37], which only checks for an exact amount of items.

The four new patterns are required as their semantics could not easily be expressed by (combinations of) other patterns. These patterns therefore also improve the user-friendliness of the entire verification solution, by providing users with more patterns to which their requirements can be mapped. As one is not required to use all patterns, the Strict Response pattern does not affect the compatibility with MCAs which do not function based on a predetermined cycle. All patterns which specify some kind of response bound require quantification using a time unit. This is done to avoid possible confusion among users related to response time, and is possible as no properties in the property set require the notion of eventually. The semantics of all patterns are expressed using timed automata in Appendix E. For further explanation of these automata, see Section 8.2.4.
Patterns of Dwyer et al.
- Absence: P must not hold
- Bounded Existence: P must hold (at least/most) k times
- Universality: P must always hold

Patterns of Konrad and Cheng
- Bounded Recurrence: P must hold at least every x time units
- Bounded Response: P must be followed by S within an x time units
- Minimum Duration: P must hold for at least x time units when it becomes true
- Maximum Duration: P must hold for at most x time units when it becomes true

Patterns of Gruhn and Laue
- Bounded Absence: P must not be followed by S within x time units

Patterns of Autili et al.
- Time-constrained Absence: P must hold for at most x time units in total
- Time-constrained Universality: P must hold for at least x time units in total

New patterns
- Restricted Recurrence: P must hold at most every x time units
- Strict Response: P must be followed in the next system cycle by S
- Prefaced Bounded Response: P which holds for at least x time units, must be followed by S within y time units
- Duration Bounded Existence: P must occur (at least/most) k times during x time units

Table 5.3: Patterns used

<table>
<thead>
<tr>
<th>Property</th>
<th>Pattern</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Absence</td>
<td>After Until</td>
</tr>
<tr>
<td>P2</td>
<td>Absence</td>
<td>After Until</td>
</tr>
<tr>
<td>P3</td>
<td>Strict Response</td>
<td>Globally</td>
</tr>
<tr>
<td>P4</td>
<td>Bounded Response</td>
<td>Globally</td>
</tr>
<tr>
<td>P5</td>
<td>Prefaced Bounded Response</td>
<td>Globally</td>
</tr>
<tr>
<td>P6</td>
<td>Duration Bounded Existence</td>
<td>Globally</td>
</tr>
</tbody>
</table>

Table 5.4: Pattern and Scope mapping

5.5 Conclusion

In this chapter, the requirements of machine control application have been explored. Collected requirements were substituted using four elements, which form an abstracted basis of machine components. This substitution maps the requirements directly onto the concepts of a component, thereby creating properties usable in runtime verification. By categorising the properties by requirement category, pattern and scope, a clear view emerged regarding what properties should be specifiable of machine control applications. These properties together form the property set. The focus during the remainder of the research will be on this set. Therefore this set is the answer to research question RQ1, What properties should be specifiable? Our contribution detailed in this chapter is:

- A set of MCA properties mapped to patterns and scopes, see Appendix [B]
Chapter 6

Architecture

In this chapter, the architecture of our verification toolset is presented. Section 6.1 describes our modular approach and its relation to an existing approach. Section 6.2 provides an overview of our contributions in the form of a conclusion. In this section, the information this chapter brings forward is used to answer research question RQ5 *How can the modularity of the verification solution be guaranteed?*

6.1 Modular Toolset

As mentioned in Chapter 4, Malakuti and Aksit present a toolset to create verification systems using *Event Modules*. Using this toolset, the authors aim at modularity, which indicates that individual concerns of a runtime verification technique must be represented as individual modules with well-defined interfaces. Malakuti and Aksit provide a *EventReactor language composition framework*, consisting of multiple DSLs, to specify the event modules. Event modules containing functionality related to the runtime verification domain can be specified using this framework. An event module does its processing using one or more *reactor classes*. A reactor class is implemented in Java and can, for example, be used for trace inspection or property verification. The toolset does, however, not provide the inspection or verification functionality itself.

The toolset of Malakuti and Aksit is aimed at creating general solutions for the verification of all types of systems. The DSLs of the toolset do not provide functionality for an extensive property specification language. Also, no clear methods are provided for the addition of an external property specification DSL. Even though the toolset provides an interesting approach towards modular runtime verification, options to incorporate parts of existing verification systems into a new verification system created with the toolset, are limited. This would require creating new specification languages and verification systems from scratch using the EventReactor framework, which is not ideal.

To solve these problems, we propose a simplified version of this modular approach and aim it at the runtime verification of machine control applications (MCAs). This approach remains loyal to the modularity requirement of Malakuti and Aksit by splitting up the individual concerns of runtime verification, while still allowing the inclusion of existing solutions. Using this new approach could help our verification solution conform to the *generic and extendability* requirements mentioned in Chapter 1. Our approach is detailed below.
As explained in Section 3.2, a runtime verification system takes execution traces and properties representing the requirements as input and produces verdicts as output. The necessary components of a runtime verification system can be sketched as follows:

1. A language in which requirements can be expressed as properties.
2. An adapter which transforms an MCA execution trace into a verifiable form.
3. A system which can determine if properties are satisfied based on the execution trace.
4. A visualisation of the verification results.

Using our modular approach, we created a toolset that allows for the verification of many different requirements of many different machine control applications. Figure 6.1 visualises the verification process using this toolset. Here, Machine Control Requirements are the requirements of an MCA. In our toolset, a tool exists for each of the four components discussed above. These tools are the Property Language, the Trace Adapter, the Monitoring System and the Result Visualisation. Our approach allows the toolset to be extended with alternative tools if desired. This supports the introduction of additional property languages and result visualisations that suit specific types of users better. By creating custom execution trace adapters, a wide variety of MCAs can be verified using our toolset. It makes the toolset (MCA) language-independent. The four tools are elaborated further below.

To support modularity and extendability, clear interfaces have been developed. These require the property data, execution traces and result data to be presented in predetermined formats. These formats are detailed in Chapter 10. The property + proposition data format supports all properties in the property set. Therefore, no changes to the format are needed if alternative tools are added to the toolset that support a subset of the property set. This is not the case if such an alternative tool supports more properties than are in the property set.

Our toolset is aimed at both runtime verification occurrence approaches (mentioned in Section 3.2), meaning it can handle real-time execution data but also execution traces stored in a log file. Our toolset does not support sending any corrective feedback to an MCA.

Property Language The property language allows for the specification of properties regarding the behaviour of MCAs. Such a language can be textual or graphical, depending on what format suits the given properties and users best. The tool which interprets the properties transforms them to a predetermined format. These can then directly be inserted into the Monitoring System. The tool
that interprets the language does also accept data regarding *component instances* of a given MCA as input, see Section [10.1] for more details. As this is static (non-runtime) MCA data and not mandatory for the verification, it is shown using a dashed line in Figure [3.2]. The usages of this data, together with the property specification language, are discussed further in Chapter [7].

**Trace Adapter**  This tool transforms raw execution traces of MCAs into a format which can be consumed by the *Monitoring System*. As applications designed in Cordis SUITE all share the same design philosophy, using the concepts described in Section [2.2.1], only two trace adapters would be necessary for the customers of Cordis. One that transforms traces from simulated MCAs, and one that transforms traces originating from MCAs running on physical machines. Two adapters are needed as these sources do not serve the same raw execution data. MCAs which are not designed in Cordis SUITE could require additional adapters, as the raw execution data could have different formats. The details of what data a trace adapter should produce are discussed in Chapter [10].

**Monitoring System**  This tool checks if the execution traces adhere to the requirements. These requirements are in the form of properties specified using the *Property Language*. The output of the monitoring system is, among other things, verdicts that reveal whether the application conforms to each property. The monitoring system also has the MCA cycle time as input, see Section [10.1] for more details. As this data is not mandatory for the verification, it is shown using a dashed line in Figure [3.2]. As the monitor system is an external entity, it follows the *outline* approach of runtime monitoring. Being an external entity makes the monitor non-intrusive, is does not slow the execution of the application under verification. The monitoring system is discussed further in Chapter [8].

**Result Visualisation**  This tool is responsible for converting the results produced by the *Monitoring System* into a user-friendly form. Other visualisation tools, which can handle the result format, can be connected to our toolset. Furthermore, adapter tools can be created which translate the results to a format which other existing visualisation tools can understand. This approach allows all results to be visualised in a format which suits a given user best. Approaches towards providing users with feedback relating the results of a verification are discussed in Chapter [9].

### 6.2 Conclusion

In this chapter, we proposed a modular toolset approach to the runtime verification of machine control applications. By using strict interfaces, additional tools that have similar goals as the four tools described above, can be added to the toolset. The use of a trace adapter allows the toolset to support a wide variety of machine control application implementations. Our contribution detailed in this chapter is:

- A modular approach to the runtime verification of a wide variety of MCAs that supports the inclusion of additional property languages and visualisation tools.

Using all information in this chapter, research question RQ5 *How can the modularity of the verification solution be guaranteed?* can be answered in the following way: By using multiple independent tools that each perform one aspect of a runtime verification process and can communicate with each other using clearly defined interfaces.
Chapter 7

Property Language

This chapter describes our language in which the properties, determined in Chapter 5, can be specified. In Section 7.1, a collection of existing property specification languages is discussed and evaluated. Section 7.2 details an experiment used to formulate the basic design of our language. In Section 7.3, the design of the language is discussed further. Section 7.4 describes the additional abstraction functionality of our language. Section 7.5 provides an overview of our contributions in the form of a conclusion. In this section, information detailed in this chapter is used to answer research question RQ2 

\textit{How can a property language be structured for user-friendliness?} and research question RQ3 

\textit{What abstraction level should be used when specifying properties?}.}

7.1 Previous Work

This section discusses several existing property specification languages. To determine an appropriate language for property specification of machine control applications (MCAs), it is essential to know what languages already exist. This makes sure no wheel is reinvented. The languages are evaluated based on a set of requirements, which are described below. This evaluation is performed to find a suitable language for the specification of MCA properties of the \textit{property set}.

7.1.1 Language Requirements

A language for the specification of MCA properties has to meet specific requirements if it is to be suitable for our verification toolset. These requirements are:

\underline{Ease of use} As mentioned in Section 1.3, user-friendliness is an essential aspect of the verification toolset. Therefore, the language should be easy to use and understand by its users. As can be read in Section 2.2, many Cordis SUITE users are experienced PLC programmers who are knowledgeable in one or more of the IEC 61131-3 languages. Many of them are, on the other hand, not familiar with any concepts of formal verification or Computer Science in general. Therefore, the inclusion of these concepts in a language should be avoided. This includes the terminology of the pattern systems (‘absence’, ‘eventually’) on which the \textit{property set} is based. The language should require a minimum number of new concepts to be learned by a user. Failing to do this could discourage the user from wanting to learn and use the language [15].

\underline{Fit for purpose} A language is of little value if it cannot be used to verify properties in the \textit{property set}. Therefore a language should support as many properties of the \textit{property set} as possible.
Extendability  A language should be easily extendable with new functionality so that it supports all properties of the property set.

Both roles of the verification toolset are involved with the property language. Users use the language to specify properties. Maintainers implement property support into the language. Users do not have to be familiar with the details of the language’s underlying formalism, maintainers do. Here, a language acts as a higher-level version of a formalism, where it tries not to expose users to the complexity of that formalism.

7.1.2 Languages

Many specification languages exist. Therefore, a pre-selection has been made to remove languages whose syntax resembles temporal logics. These languages would score badly on the ease of use requirement. As mentioned above, users of Cordis SUITE are not familiar with temporal logics and, as mentioned in multiple sources [1, 31, 40], temporal logics are hard to understand for practitioners who have no experience with it. Because of this, the specification languages Algebra II [66], Eagle [10], Lola [28], Reusable Automation Component (RAC) [52], RuleR [11] and UML extensions [29] are ruled out. We also rule out the mCRL2 language [36], used in a sub-project of Machinaide, as it requires users to be familiar with multiple Computer Science concepts, which most users are not. All other languages have been divided into two categories, textual languages and languages which use a graphical notation. Every language is evaluated based on the requirements described above.

7.1.2.1 Textual Languages

**Bandera Specification Language**

As mentioned in Chapter 4, Hatchill and Dwyer created a tool where properties of concurrent Java programs can be specified using the Bandera specification language (BSL) [40]. The BSL is a source-level language, meaning properties are specified using Java methods and objects. BSL uses the pattern system terminology of Dwyer et al. [31], which does not aid its user-friendliness. Predicates are defined in the Java source code itself and specify for example, that a variable should remain in a given range. These predicates are then used in the properties. This separation of predicates and properties could aid readability. The BSL has a syntax which resembles Java, which could be a problem for the users of Cordis SUITE as few are familiar with its syntax. As this language is heavily integrated with Java source code and does not support timing properties, many modifications are required to express properties of the property set.

**ComMA**

As mentioned in Chapter 4, TNO has developed ComMA [47], a tool used for interface modelling and analysis which uses DSLs to specify constraints on those interfaces. In ComMA, the behaviour of an interface (protocol) first needs to be modelled in the form of a textual representation of state machines. Data and timing constraints relating to the protocol are specified in separate properties, which have a more programming language-like syntax. Overall we consider the language to be reasonably user-friendly. ComMA only supports a few properties of the property set. As ComMA is focused on black-box analysis of interfaces, we deem its extendability with currently unsupported properties of the property set, which are not specifically aimed at interfaces, to be complex.
IoT Requirement DSL

As mentioned in Chapter 4, TNO used a virtual prototyping approach based on a domain model consisting of DSLs to specify different aspects of an IoT application. One of these DSLs, the Requirement DSL, is used to specify properties the application should adhere to. These properties are based on the pattern system of Konrad and Cheng. Properties are specified using elementary English sentences and use pattern system terminology. Just like BSL, this DSL allows propositions to be specified as separately from properties, which aids readability and reduces repeated constructs. We deem the Requirement DSL to be reasonably user-friendly, except for the fact that users are required to be familiar with pattern system terminology. As multiple pattern-scope combinations, to which the properties of the property set are mapped to, are supported, the language is deemed to be easily extendable.

PROPOLS

A language used to define properties of web service applications, named PROPOLS, is presented in [72]. Properties are based on the pattern system constructed by Dwyer et al. Properties are defined as elementary English sentences and use pattern system terminology. PROPOLS is ontology-based, meaning uses standard terminology of a domain in its properties. In practice, this translates in slightly different pattern system terminology (‘leads to’ instead of ‘response’). As most terminology remains identical to the terminology used by Dwyer et al., the language is still deemed less user-friendly. Besides that, the language does not support the specification of timing properties and therefore does not cover a large part of the property set. Properties in PROPOLS are translated to TDFAs (Total and Deterministic Finite State Automaton), which do not support timing properties. Because of this, PROPOLS is not deemed easily extendable.

ST-LTL

As mentioned in Chapter 4, Ljungkrantz et al. used a modified version of LTL, named Structured Text LTL (ST-LTL [53]) to specify relevant properties for safety PLC program components. ST-LTL is a textual variant of LTL where pattern system terminology keywords replace the temporal logic operator symbols. According to the authors, ST-LTL was introduced as no other temporal logic is suitable for people who are not trained in formal verification. ST-LTL should be understandable by programmers who also understand the Structured Text specification language of IEC 61131-3. As most users are not familiar with the Structured Text language and pattern system terminology, this language is not deemed very user-friendly. As ST-LTL is built on top of the LTL language, which does not support timed properties, this language is not considered easily extendable to support all properties of the property set.

Structured English Grammar

As mentioned in Section 5.4, Konrad and Cheng extended the pattern system of Dwyer et al. with timed patterns in [44]. This paper also brings forward a structured English grammar to express properties of this extended pattern system. The grammar allows properties to be written in full grammatically correct English sentences, which Konrad and Cheng think will significantly improve the understanding of the properties. Pattern system terminology is, however, present in the grammar. The grammar was later extended by Autilli et al. to support additional properties [9]. Because of this, it covers the most of the property set of any language. As this grammar is not built on top of another language, like LTL, we see no major obstacles in extending this language to support all properties of the property set. Based on the details above, this approach is considered reasonably user-friendly.
Web Services Interaction

As mentioned in Chapter 4, Li et al. proposed a specification and validation system aimed at the behaviour of service interactions [51]. Their specification language allows behaviour to be specified as a set of properties. Like PROPOLS, this language is based on an ontology and the pattern system of Dwyer et al. The properties are defined as elementary English sentences, but use pattern system terminology. As this language resembles PROPOLS, a lot of the same advantages and disadvantages apply. It is relatively user-friendly, but it does not support the specification of timing properties, and therefore does not cover a large part of the property set. All this language is built on top of FSAs (Finite State Automaton [63]), which do not support timing properties, this language is not deemed easily extendable to support all properties of the property set.

7.1.2.2 Graphical Languages

ppDate

As mentioned in Chapter 4, Ahrendt et al. propose, as part of the StaRVrOrS framework, a specification language called ppDATE [2]. ppDATE extends the DATE (Dynamic Automata with Timers and Events) [24] formalism with pre and post-conditions. As ppDATEs are automata, they are a very expressive form of property specification and do not restrict the user to a fixed set of properties. Because of this, users are responsible for correctly implementing the semantics of a property themselves. Such actions can, based on our own experience, be very challenging, especially when concepts of the pattern system are not fully understood. This would make users of the language also fulfill the maintainer role. The fact that some users of Cordis SUITE are familiar with a custom type of state machine used in Cordis SUITE, see Section 2.2, does not make up for the difficulties. Because of this, ppDATEs are not considered to be easy to use. As ppDATEs are very expressive, they are extendable with all properties of the property set.

Observer automata

Gruhn and Laue present a catalogue for real-time properties in [37]. For each pattern in this catalogue, they constructed observers which can be used for constraint monitoring. These observers are timed automata [7]. The pattern catalogue is an extension of the pattern system of Dwyer et al. [31]. Although the catalogue provides examples, the observer automata still carry the same disadvantages as ppDATEs: the user is responsible for correctly implementing the semantics of a property themselves. We do not consider this user-friendly. Gruhn and Laue assess it is best if these observers are automatically generated from a higher-level language. This way, users do not have to familiarize themselves with the details of timed automata. As these automata are very expressive, they are extendable with all properties of the property set.

PLCspecif modules

Darvas et al. created a formal specification language for PLC programs called PLCspecif [29]. This language allows a PLC component to be specified in a module, which is the main building concept of PLCspecif. The authors mention that there is no single specification language that can conveniently cover all kinds of PLC components. Because of this, a module consists of a state machine, input-output connection descriptions and a constraint section. Although these modules are primarily used for PLC code generation, the concept of specifying details about a PLC component in multiple ways is interesting. The modules can, however, only be used to specify invariant properties and therefore support very little of the property set.
Time-Enriched LSC

As mentioned in Chapter [4], Harel and Marelly extend Live Sequence Charts (LSCs [26]) with timing constructs. This extension makes, according to them, the language suitable for specifying the behavioural requirements of time-intensive systems. Timing properties can be placed inside an LSC, connecting the property to the action it specifies something about. We consider this language to be easy to understand. Time-Enriched LSCs do not support all properties which handle the counting of occurrences. Time-Enriched LSCs also do not support the construction of properties regarding something that should not happen. Because of this and the previous remark, the language does not support a large portion of the property set. Because of the limits of the LSC graphical format, this language is not deemed to be easily extendable.

TimeLine

TimeLine editor allows events to be specified across a wide horizontal bar, with time progressing from left to right [68]. Vertical bars on this timeline represent events. Using arrows and other graphical elements, properties can be specified on these events. We consider this language to be easy to understand. TimeLine can, however, not be used to specify timing properties, as there is no notion of a global clock. Recurring events are also not supported. Because of the limits of a graphical timeline format, this language is not considered to be easily extendable.

Visual Timed Scenarios

In [3], Alfonso et al. present Visual Timed Event Scenarios (VTS), a visual language to define complex event-based properties. These properties are specified using models with nodes and several types of (un)directional edges. According to Alfonso et al. VTS constructs can get large when dealing with complex properties, but they would still be more straightforward to comprehend than the same properties written in a textual language. This language does have a steep learning curve as it is deemed not easy to understand. As mentioned by the authors, this language mostly focuses on event-based properties such as bounded response and event correlation. The language would require additions such as recurrence patterns to be usable for the property set. As mentioned in [37], about VTS, only scenarios which violate the properties can be specified, which they think can be a difficult task. This limits one from specifying things that should happen. Because of this, VTS does not cover a large part of the property set. Because of the limits of the VTS graphical format, this language is not considered to be easily extendable.

7.1.3 Review

Table 7.1 shows all languages rated using the requirements specified earlier. The rating uses a ‘good-avg.-bad’ system. The rating is based on facts and opinions from the scientific community, together with our opinions. All ratings are relative and do not express any precise quantification.

As Table 7.1 shows, most of the graphical languages do not nearly support the specification of all properties of the property set. This is the case as these languages are often aimed at a specific type of properties. For example, TimeLine provides a powerful way of specifying event and timing properties, but its notation is not suited for recurring events. Similar sorts of problems occur for Time-Enriched LSCs and Visual Event Scenarios. These languages are deemed very applicable for the specific use cases they support. Because of this, they could, at a later point in time, be connected to the modular system as a secondary language specifically for such use cases. Where Observer automata and ppDATEs rank low on the ease of use requirement, PLCspecific modules rank low on the fit for purpose requirement. Because of this, these three languages are not deemed to be the best language for the specification of MCA properties.
Although ST-LTL does not use LTL operators, users are still required to have some LTL knowledge. It is also required that every user knows the structured text language, which is not the case for the users of Cordis SUITE. Of the other textual languages, the structured English grammar (SEG) from Konrad and Cheng, and the IoT Requirement DSL from Verriet et al. cover most of the property set. Both languages are also deemed easily extendable. SEG ranks high in the user-friendliness category, whereas the Requirement DSL is considered reasonably user-friendly. Because of this, these are deemed the top two languages best suited for the specification of the machine control application properties defined in Chapter 5.

We decided not to develop an entirely new language from scratch, as two possibly suitable languages have been found. Because of this, we think it would better to modify these languages instead of creating a completely new one from scratch. We decided not to pursue the route of creating a new graphical language, as the majority of users are PLC programmers, who can handle textual languages. This, of course, does not hold for all users. Although our language is also made suitable for these users, they could prefer a graphical language. Our modular verification toolset provides opportunities to connect multiple languages, such a graphical language, or other languages which are not selected now.

<table>
<thead>
<tr>
<th>Name</th>
<th>Ease of use</th>
<th>Fit for purpose</th>
<th>Extendability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandera Specification Language</td>
<td>bad</td>
<td>avg.</td>
<td>bad</td>
</tr>
<tr>
<td>ComMA</td>
<td>avg.</td>
<td>avg.</td>
<td>avg.</td>
</tr>
<tr>
<td>IoT Requirement DSL</td>
<td>avg.</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>PROPOLS</td>
<td>avg.</td>
<td>avg.</td>
<td>bad</td>
</tr>
<tr>
<td>ST-LTL</td>
<td>bad</td>
<td>avg.</td>
<td>bad</td>
</tr>
<tr>
<td>Structured English Grammar</td>
<td>good</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Web Services Interaction</td>
<td>avg.</td>
<td>avg.</td>
<td>bad</td>
</tr>
<tr>
<td>ppDATE</td>
<td>bad</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Observer automata</td>
<td>bad</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>PLCspecif modules</td>
<td>avg.</td>
<td>bad</td>
<td>avg.</td>
</tr>
<tr>
<td>Time-Enriched LSC</td>
<td>good</td>
<td>avg.</td>
<td>bad</td>
</tr>
<tr>
<td>TimeLine</td>
<td>good</td>
<td>bad</td>
<td>bad</td>
</tr>
<tr>
<td>Visual Timed Scenarios</td>
<td>bad</td>
<td>bad</td>
<td>bad</td>
</tr>
</tbody>
</table>

Table 7.1: Language rating
7.2 Language Experiment

An experiment was conducted to determine a suitable structure for a language for the specification of MCA properties. This experiment was conducted with our contacts at Cordis Automaton. Here, the two languages selected in the previous section function as inspiration sources for our contacts at Cordis, who are not familiar with property languages. Our selection of languages in the previous section functioned as a pre-selection, as discussing all languages in this experiment is not feasible.

Even though the Requirement DSL uses a lot of pattern system terminology, which most users do not understand, it is still included in the experiment. This is done as it is still deemed the second-best suitable language overall, and provides a different way of specifying properties to the SEG language, which could aid the inspiration process. In this experiment, we focus on the property specification structure of the Requirement DSL, and not the fact that it uses pattern system terminology.

The experiment was carried out with Remy Fennet, who is one of our contacts at Cordis and has lots of knowledge on the customers of Cordis SUITE. This thesis’ supervisor, Jacques Verriet is also part of the experiment. Verriet has less knowledge of the customers of Cordis SUITE but is experienced in the design of property specification languages for industrial use. As Jacques Verriet is one of the authors of the Requirement DSL, care has been taken to remove any bias. Although both languages do not cover the complete property set we still think a meaningful experiment can be carried out using the properties that are supported.

7.2.1 Methodology

The experiment is structured in the following way: Three properties from the property set are specified in both the IoT Requirement DSL and the Structured English Grammar (SEG). These properties are provided to Fennet and Verriet. At the start of the experiment, all necessary terminology, like the property system (patterns and scopes) and its terminology were explained.

The first property is defined in English as follows: When the cooler is turned on, the temperature of the item to be cooled should be less than 100 degrees after 6 minutes. Listing 7.1 shows this property expressed in the two languages. Words in bold font represent the keywords of the languages. See Appendix B for all other property examples.

The IoT Requirement DSL requires the propositions which are used in the properties to be defined separately. As the SEG does not mention where to define the propositions, the Requirement DSL approach is used.

<table>
<thead>
<tr>
<th>Proposition</th>
<th>coolerOn : cooler.state = 'on'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposition</td>
<td>cooledDown : cooler.temperature &lt; 100</td>
</tr>
</tbody>
</table>

// Requirement DSL
Globally Response cooledDown after coolerOn within 6 minutes

// SEG
Globally it is always the case that if coolerOn holds, then cooledDown holds after at most 6 minutes

Listing 7.1: Language example
Based on the three property examples, the participants were asked the following questions:

1. What is your opinion on both languages regarding the specification of properties?
2. Do you think the languages are easy to use for the users they are aimed at and why?

7.2.2 Results

Before referring to the participants of the experiment, the main differences of the languages can be summed up as follows:

1. Requirement DSL requires fewer words to specify a property than SEG.
2. Requirement DSL uses pattern systems terminology, whereas SEG uses a more natural language.

Fennet mentions that Requirement DSL looks simpler as properties can be defined in shorter sentences, and are therefore easier to read. He notes, however, that the language requires him to be familiar with the pattern system terminology, which he was not at the start of the experiment. This relates to the results of a user study performed by Buit on an earlier version of the Requirement DSL [17]. That study concluded with the fact that although the language was deemed readable, users still found the language too formal. The language remained intertwined with pattern system terminology of Konrad and Cheng [44].

Verriet, who is experienced in the concepts of the pattern system, notes that the SEG language is more straightforward for users who are from other disciplines of a company. These users do not write the properties but do advise on them and read them. This requires these users to be able to understand what is specified. He mentions, however, that the short sentences of the Requirement DSL would be more straightforward for the users which will be writing the properties. As they become familiar with the language, the extra verboseness of the SEG language is no longer useful and becomes a hassle to deal with. According to Verriet, one language could be more useful for one type of users, and the other language for the other type of users. Fennet agrees with this opinion.

7.2.3 Conclusion

The experiment can be concluded with the fact that the structures of both languages are not perfect, and both suit a different type of user better. Therefore, an ideal language structure (for the users of Cordis SUITE, which are a projection of the targeted users) to specify properties of MCAs, is placed somewhere in between the structures of the two languages. See below for a summation of the results of the experiment:

- The short sentence structure of the Requirement DSL is deemed by our participants to be more suited for users who write properties.
- The long sentence structure of the SEG language is deemed by our participants to be more suited for users who are from other disciplines of a company. These users are less familiar with the language and the idioms used.
7.3 Language Design

This section describes our property specification language in detail. Design decisions have been made based on the results of the experiment detailed in the previous section, scientific literature, and feedback from our contacts at Cordis Automation. These people represent a group who are potential users of our language (based on fulfilling such function in the past), or people who know a lot about these users. This group is from here on referred to the feedback group.

The requirements case of use and fit for purpose, defined in Section 7.1.1 still apply to our language. Additionally, as concluded from the experiment, a middle ground in sentence length and verboseness between the structures of the Requirement DSL and the SEG language should be aimed at. As we aim for a textual language that is connected to our toolset using an interface which provides no restrictions on adding additional properties in the future (see Chapter 10), we deem no further actions are necessary to be able to conform to the extendability requirement.

7.3.1 Structure

Our language is based on rulesets. A ruleset is used to specify a combination of properties aimed at one or more MCA components. A ruleset consists of several segments, namely the definition segment, the components segment, the propositions segment and the rules segment. All keywords of the language are highlighted in bold. See Appendix D for the full language grammar.

Definition segment

The first segment of a ruleset is the definition. This segment acts as a block comment and can be used to provide a description of the ruleset, see Listing 7.2. Buse and Weimer mention that comments are a direct way of communicating intent [18]. They discovered that, contrary to popular belief, code comments do not automatically increase the readability of a code block. In [57], Martin mentioned that comments do not make up for badly readable code. One also needs to be wary to keep a comment up to date, so that it does not contradict the semantics of the code.

Nevertheless, we think, together with the feedback group, that if the definition segment is used correctly, it can improve the readability of the language. To minimise the adverse effects comments can have, only one definition segment is allowed per ruleset, and the language does not officially support further commenting in other ruleset segments. In practice, these restrictions might be harder to enforce depending on the editor tool used. The definition also is an optional segment, which a user is not required to use.

Definition: A minimum gap of 2 seconds is kept when triggering a warning light, whenever a system is in an error mode. The light is on for 1 second with a maximum deviation of 50 milliseconds.

Listing 7.2: Ruleset definition segment

Components segment

In the components segment, instances of MCA components which the properties specify something about, are specified. This segment exists for three reasons. Firstly, according to the feedback group, depending on the MCA implementation, components can have long names. In this segment, components are given an alias (see Listing 7.3) which allows them to be used efficiently in other segments of the ruleset. In the component segment, the instance keyword is used to specify an alias of one component instance, see the third line of Listing 7.3.
The second benefit of the component segment is that components can only be defined in this segment and not in others, which aids readability and forces the separation of concerns.

The third benefit of the component segment is that it provides functionality which can dramatically reduce the number of rulesets needed. If multiple instances of an MCA component exist, one does not want to specify the same properties for each instance individually. Using the component segment, our language enables the specification of properties that hold for all instances of a component. See Listing 7.3 where the each instance of keywords are used to query every component instance of the MerryGoRound.Module class. Here, the alias module represents all instances of the MerryGoRound.Module class. By using this alias in a property, the property is specified for all those instances. The component segment has its roots in the Requirement DSL of Verriet et al. [71]. It differs from the Requirement DSL as the component segment only supports the querying of all instances of one component class (per ruleset), whereas the Requirement DSL allows multiple of such queries to be defined. We limit this to avoid a rapid increase in the number of instance combinations, and therefore possibly complex rulesets. Also, such functionality is not required to express the properties of the property set.

<table>
<thead>
<tr>
<th>Components:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each instance of MerryGoRound.Module as module</td>
</tr>
<tr>
<td>Instance MerryGoRound.TrafficLights.RedLight as errorLight</td>
</tr>
</tbody>
</table>

Listing 7.3: Ruleset components segment

Propositions segment

As mentioned in Chapter 5, reasoning about MCA data can be done using phase and event proposition. A phase proposition can hold for a longer time period and an event proposition holds for an infinitesimal time period. The propositions segment supports the specification of these propositions, see Listing 7.4.

A proposition is defined by its type (phase/event), a unique name and an expression. An expression can be made using internal attribute data of component instances. To access this internal data, queries using a dot (\'.\') notation are supported. These queries start with the alias of a component defined in the component segment. A query (module.status in Listing 7.4) can be seen as a unique identifier referring to an attribute in an MCA. What these identifiers exactly are, depends on the identifiers used in the trace transformation performed by the Trace Adapter. Because of this approach, a wide variety of MCA implementations are supported by our language and users are not locked into identifiers like '{...}.status' (e.g. not restricted to the four elements of Section 5.1).

Propositions can be defined using one attribute query and one constant (the first two propositions of Listing 7.4) or using two attribute queries (the fifth proposition). The latter proposition can be used to compare attribute data of different components. The language supports constants of types integer, floating-point number and string. As data types of MCA variables are not known beforehand, the language does not provide any type-checking. Users have to determine the type of a proposition themselves by asking if their proposition can only hold for an infinitesimal period (event) or for a longer period of time (phase).

Propositions are defined separately of the properties they are used in, to aid readability and to facilitate their reuse. Phase propositions can be converted into event propositions using the start/end of keywords, inspired by the Requirement DSL of Verriet et al. [71]. This is useful as not all our patterns, detailed in Chapter 5, can handle both types of propositions. This is discussed further in Chapter 8. What operators can be used in these propositions is also discussed in that chapter.
**Propositions:**

- **Phase error:** module.status == 'error'
- **Phase lightOn:** errorLight.state == 'on'
- **Event lightTurnsOn:** start of lightOn
- **Event receivedOn:** module.command == 'on'
- **Phase diffStates:** module.state != errorLight.state

**Listing 7.4: Ruleset propositions segment**

**Rules segment**

As concluded by the experiment, both the IoT Requirement DSL and the Structured English Grammar had some disadvantages. Namely, the shortness of properties and use of pattern system terminology by the Requirement DSL, and the verbosity of properties of the SEG. Because of this, we propose a new way of specifying properties, which takes the positives of both languages and tries to leave out the negatives, see Listing 7.5.

Although our language is not based on an existing ontology like PROPOLS [51], all of the original pattern system terminology has been changed. Here, an After Until scope is now specified as **between Q and R**, which is much more understandable, according to the feedback group. Words as **absence** and **universality** are replaced by more common words like **never** and **always**, respectively. We also introduce a new keyword **while**, which is the phase proposition equivalent of the After Until scope. Now, this scope can be specified in two ways, **between event and event** and **while phase**. The while keyword eliminates the need to create two event propositions (start/end of) when relating an After Until scope to a phase proposition. As the Globally scope represents the entire execution of an MCA and not a specific part, its keyword is deemed redundant. Because of this, scope keywords are only required for the After Until scope. For more clarity, the first keyword of a scope is capitalised.

Properties are specified using English sentences, which makes them easier to understand. These sentences are longer than the ones used in the Requirement DSL but shorter than the ones used in SEG. All properties also use similar word combinations (i.e. **at least**, **when/then**). This requires fewer unique words to be remembered, which can increase the user-friendliness of the Language.

Colons (’:’) are used to separate the scope section from the pattern section in a property. Our language is still based on the notion that a property consists of a pattern and scope. As these are fundamental attributes (the **when** and the **what** respectively) of a property of the **property set**. By using grammatically correct sentences that use common words, we aim to make the properties readable for people who have no understanding of the notion of patterns and scopes. This approach should only require people who write the properties to be familiar with the notion of patterns and scopes.

**Listing 7.5: Rule formulation**

---

<table>
<thead>
<tr>
<th>Between Q and R: never P  // After Until Absence</th>
</tr>
</thead>
<tbody>
<tr>
<td>P at least k times  // Globally Bounded Existence (at least k)</td>
</tr>
<tr>
<td>P at most k times  // Globally Bounded Existence (at most k)</td>
</tr>
<tr>
<td>when P, then S after at most 6 minutes // Globally Bounded Response</td>
</tr>
<tr>
<td>Between Q and R: P at most every 2 seconds // After Until Restricted Recurrence</td>
</tr>
<tr>
<td>once P, it holds at most 3 minutes // Globally Maximum Duration</td>
</tr>
</tbody>
</table>

35
In our language, properties are specified as rules in the rules segment, see Listing 7.6. This segment can contain multiple rules, as complex requirements might require multiple rules to be expressed. All three rules in Listing 7.6 specify properties which have the After Until scope. From top to bottom, the rules have the recurrence, minimum duration and maximum duration patterns respectively. Rules accept one or more propositions defined in the proposition segment. Every rule has a unique name.

```
Rules:
  Rule lightRecurrence: While error: lightTurnsOn always after at most 2 seconds
  Rule lightDurationMax: While error: once lightOn, it holds at most 1050 milliseconds
  Rule lightDurationMin: While error: once lightOn, it holds at least 950 milliseconds
```

**Listing 7.6: Ruleset rules segment**

**Complete ruleset**

All segments together form a ruleset, see Listing 7.7. A ruleset is used to specify a combination of properties aimed at one or more relating MCA components. This scope is selected as it allows for some reuse of propositions and component aliases while remaining fairly compact.

```
RuleSet errorLightFunctionality {
  Definition: A maximum gap of 2 seconds is kept when triggering a warning light, whenever a system is in an error mode. The light is on for 1 second max with a deviation of 50 milliseconds.

  Components:
    Each instance of MerryGoRound.Module as module
    Instance MerryGoRound.TrafficLights.RedLight as errorLight

  Propositions:
    Phase error: module.status == 'error'
    Phase lightOn: errorLight.state == 'on'
    Event lightTurnsOn: start of lightOn

  Rules:
    Rule lightRecurrence: While error: lightTurnsOn always after at most 2 seconds
    Rule lightDurationMax: While error: once lightOn, it holds at most 1050 milliseconds
    Rule lightDurationMin: While error: once lightOn, it holds at least 950 milliseconds
}
```

**Listing 7.7: Ruleset structure**
7.3.2 Features

This section details some additional features of our language.

Proposition combinations

To cover more complex properties of the property set, our language supports the combination of propositions. As mentioned in Chapter 5, propositions of properties in the property set are combined using the AND, OR, XOR, ANY and ALL operators. Phase propositions can be combined using all these operators. As the probability that different events occur at the exact same time is small, event propositions can only be combined using the OR and ANY operators. When multiple operators are used in one proposition, the default precedence rules apply. Brackets are also supported. Listing 7.8 shows some proposition combinations that can be made using the operators. Here, \(PH\) and \(EV\) are placeholders for unique phase and event propositions respectively.

<table>
<thead>
<tr>
<th>Propositions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase p1: ( PH \text{ and } PH \text{ xor } PH )</td>
</tr>
<tr>
<td>Event e1: ( EV \text{ or } EV )</td>
</tr>
<tr>
<td>Event e2: ( \text{any } {EV, EV, EV} )</td>
</tr>
<tr>
<td>Phase p2: ( \text{all } {PH, PH, PH} )</td>
</tr>
</tbody>
</table>

Listing 7.8: Proposition combinations

For simplicity reasons, the two types of propositions cannot be combined using these operators. For readability reasons, only propositions which have already been defined can be combined. This means propositions cannot be combined while being defined (i.e. \(\text{module.status == error AND light.state == on}\)).

Change keyword

Event propositions trigger at the start or end of a phase proposition holding \((\text{start of conveyor.speed == 30})\). It can be beneficial to not only know when \(\text{conveyor.speed}\) starts equalling 30, but every time that the speed changes value. For this purpose, the keyword \text{change} exists. Event propositions using this keyword trigger every time the data annotated by a query changes value. Properties like: ‘Switching speed can only be done by first stopping the conveyor, changing the actual speed to the desired speed and then starting the conveyor again’ can be expressed using this keyword, see Listing 7.9.

<table>
<thead>
<tr>
<th>Propositions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase notStopped: ( \text{conveyor.state != ‘stopped’} )</td>
</tr>
<tr>
<td>Event stateChanged: ( \text{change conveyor.speed} )</td>
</tr>
</tbody>
</table>

Listing 7.9: The change keyword
Component sets

As mentioned earlier, a ruleset can be defined for multiple instances of a component class using the keywords EACH INSTANCE OF in the `components` segment. To simplify the language when reasoning about sets of components, the keywords ALL INSTANCES OF exist. Using these keywords, an alias can be provided to a set of component instances. Using the ANY and ALL operators discussed before, combinations of propositions regarding these instances can be efficiently created. For example: ‘Items are detected using three sensors. As soon as one of them is triggered, the module assumes an item is present.’, see Listing 7.10.

\[ \text{Components:} \]
- Instance MerryGoRound.Module as module
- All instances of MerryGoRound.TrafficLights as lights

\[ \text{Propositions:} \]
- Event present: \( \text{start of module.present} = \text{’true’} \)
- Event lightTurnsOn: \( \text{for any lights, start of state} = \text{’on’} \)

\[ \text{Rules:} \]
- Rule detectPresence: \( \text{when lightTurnsOn, then present after at most 10 milliseconds} \)

Listing 7.10: Component sets
7.4 Abstraction

This section provides information which can be used to answer Research Question RQ3: What abstraction level should be used when specifying properties? To specify properties which can incorporate a lot of MCA component data in their expression, propositions in our language are specified using attribute queries at a component level, see the previous section. By introducing component grouping, using the keywords EACH INSTANCE OF / ALL INSTANCES OF, a basic level of abstraction has been applied to our language. We explore the possibility of adding an additional level of abstraction, with which we aim to achieve two things:

- Decrease the amount of text needed to specify properties.
- Provide a better overview of what properties have been written down.

Of the languages discussed earlier in this chapter, only the Bandera Specification Language \[40\] tries to create some abstraction in its properties. This can be done by a user assigning an abstract type to a variable and thereby providing an extra classification. This classification gives the verification system of the Bandera toolset more knowledge of the variable it is working with. Such abstract type is, for example, \texttt{Range04}, which simplifies the tracking of a variable to \{LT0, 0, 1, 2, 3, 4, GT4\}. Meaning the system recognises the actual values 0 to 4 and summarises lower and higher values using LT0 and GT4, respectively.

Our abstractions, in the form of Abstraction classes, are inspired by the abstractions of the Bandera language. An abstraction class is initialised using propositions and variables and encapsulates its own propositions and rules in the form of a CustomRule, see Listing 7.11. A custom rule is a combination of regular rules. This makes an abstraction class a sort of interface which exposes rules. Here, the rules are abstracted away behind the notion of such class. These rules can be applied to one or multiple components whose behaviour should comply with them. In Listing 7.11 a \texttt{SingleItemModule} abstraction class is defined. It houses rules which only allow one item to be in such a module, where a new item should arrive every \texttt{repeatSec} seconds.

\begin{verbatim}
Abstraction SingleItemModule (Event: inTrigger, Event: outTrigger, Var repeatSec)
{
Rules:
   Rule neverTooMuch: Between inTrigger and outTrigger: never inTrigger
   Rule recurItems: inTrigger at least every repeatSec seconds
}
\end{verbatim}

Listing 7.11: Abstraction class

Listing 7.12 shows how the \texttt{SingleItemModule} abstraction class is applied to a conveyor component. This is done in a new Abstractions segment in a ruleset. When providing a proper name to an abstraction class, one can have a clearer overview of what rules are defined on a component. The ruleset of Listing 7.12 does not require any other rules to be specified, as all required rules are housed by the abstraction class.

We, together with our contacts at Cordis, think that abstraction classes can be beneficial in a few ways. As Rules in abstraction classes can be reused, there is a smaller probability of introducing errors as fewer rules need to be created. Abstraction classes also decrease the amount of text required to specify similar properties which otherwise would be specified often. This fact could, as mentioned by Buse and Weiner in \[18\], increase the readability of the code. We do think abstraction classes could add a layer of complexity to the language; because of this, they are an optional addition.
We do not provide any finished abstraction classes as standard. As can be concluded by the property set of Chapter 5, different companies have different MCAs and therefore want to specify different properties. Because of this, abstraction classes which suite all companies are not easy to define.

The abstraction classes introduce a third role besides users and maintainers, namely abstraction definers. Abstraction definers require knowledge of what abstractions could be defined of MCAs used by their companies. In practice, experienced people could have both the user and abstraction definer roles.

7.5 Conclusion

In this chapter, we defined a language in which properties of the property set, i.e. machine control application properties, can be specified. This language is primarily based on the IoT Requirement DSL and the Structured English Grammar, with the main focus on usability and readability. Our contributions, detailed in this chapter, are:

- A property specification language, aimed at user-friendliness and users in the machine control application domain, which allows the specification of all of the properties in the property set.

- Additional language functionality not seen in other languages:
  - The change keyword.
  - The all instances of keywords.
  - Operations on propositions sets using the any and all keywords.

- Additional abstraction functionality of the language using abstraction classes.

Using data detailed in this chapter, Research Question RQ3 What abstraction level should be used when specifying properties? can be answered as follows: A basic level of abstraction when defining properties, and an additional optional level of abstraction, provided by abstraction classes, when defining often used sets of properties.

Research Question RQ2 How can a property language be structured for user-friendliness? can be answered in the following way: By defining properties in well-segmented rulesets, where short sentence-like rules are used that use as little pattern system terminology as possible.
Chapter 8

Monitoring system

This chapter describes the monitor system component of our verification toolset. Section 8.1 details the requirements of such a monitoring system and introduces existing runtime verification methods. Based on this data, a suitable solution for the machine control application domain is selected. In Section 8.2, implementation details of the monitoring system are given. Section 8.3 provides an overview of our contributions in the form of a conclusion. In this section, the information detailed in this chapter is used to answer research question RQ6 What is a suitable technique to verify machine control application data against properties?

8.1 Previous Work

This section discusses several existing runtime monitoring systems. The systems are evaluated based on a set of requirements described below. This evaluation is done to find the most suitable verification system to fulfil the goals of this research.

8.1.1 System Requirements

As discussed in Chapter 3, the runtime verification technique will be used to verify machine control applications (MCAs). A monitoring system which performs the runtime verification has to meet specific requirements if it is to be suitable for our verification toolset. These requirements are:

**Extendability** A monitoring system should be easily extensible with new functionality to support currently unsupported properties from the property set. A system would have an advantage over other systems if such new functionality can easily be implemented.

**Fit for purpose** A monitoring system is of little value if it cannot be used to verify properties in the property set. Therefore, a system should support as many properties as possible.

**Modularity** The system should be able to function independently from its original toolset if it has one. This makes it possible to integrate it into our modular verification toolset.

Only the maintainer role of the verification toolset is involved with the monitoring system. Maintainers implement property support into the language.
8.1.2 Monitoring Systems

**CIPV**

As mentioned in Chapter 4, the verification system of Li et al. [51] uses the runtime monitor CIPV of Jin and Han [43]. CIPV monitors CORBA objects contained in the messages between clients and servers. As CIPV is an independent tool, the modularity requirement is easily met. CIPV uses Finite State Automata (FSAs [63]) to monitor properties. FSAs do not support any notion of time, making CIPV not cover much of the property set and not easily extendable.

**ComMA**

As mentioned in Chapter 4, ComMA [46] is a tool used for interface modelling and analysis. Property monitors to perform runtime analysis are generated from ComMA models, which are created in ComMA’s property language.

ComMA supports the verification of interface properties aimed at the ordering, timing and data of events between clients and servers. These properties are, therefore, not focused on the internals of client and server components. Because of this, ComMA’s approach can be categorised as black-box. This conflicts with our white-box approach where internal data of an MCA is to be analysed. Because of ComMA’s focus on interface constraints, many properties of the property set which are not required in that setting, are not supported by the monitors of ComMA. We perceive no significant limitations in using the monitoring system of ComMA outside of its toolset.

**Java monitors**

Several tools exist which focus on the runtime verification of Java programs. These include Java-MOP [20], Java-MAC [49], Hawk [27], TraceMatches [14], Larva [25] and MarQ [64]. These tools use aspects to extract execution events from a system under verification. Aspects are generated in AspectJ, one of the aspect-oriented implementations for Java [25]. These aspects enable automatic code injecting without directly altering the code of the system. As these tools are focused on the verification of Java programs, they would require additional work to make them suitable for MCA verification, which makes them not modular. Besides that, Java-MOP and Hawk do not support timed properties, making them cover only a limited part of the property set.

**JCoSim**

As mentioned in Chapter 4, the IoT DSL solution [67] supports the simulation of systems using a co-simulation tool called JCoSim. According to Gomez et al., a co-simulation enables the simulation of a system via the composition of simulators [35]. Each simulator is a black box mock-up of a component of the system. Every simulator solves its part of the problem without having the coupled system in mind. This abstracts from the entire system. JCoSim is a lean HLA (High Level Architecture) [12] based co-simulation which provides timing and publish-subscribe services to its connected simulators.

JCoSim contains property monitors originally developed by Buit in [17] and later extended by Verriet et al. in [67]. These monitors are integrated into the publish-subscribe network and can, therefore, monitor components of the system. JCoSim contains a total of 35 different monitors. These are based on pattern and scope combinations of the pattern system of Dwyer et al. [31] extended by Konrad and Cheng [44]. The scopes Globally, Before, After, Between and After Until and the patterns Absence, Existence, Universality, Precedence, Bounded Response, Bounded Recurrence and Invariance are supported. Internally, the monitors are based on timed automata [6].
As many pattern-scope combinations are supported, JCoSim covers a large part of the *property set*. We also think that implementing additional monitors to cover the complete *property set* to be feasible in JCoSim as the tool has been developed in-house at TNO, making much knowledge readily available.

**Lola**

D’Angelo et al. [28] present Lola, a monitor generation framework for the runtime monitoring of synchronous systems. The monitoring is handled by an evaluation algorithm which accepts variable streams. A stream consists of a continuous sequence of predicate results in the form of booleans. These predicates are based on variables of system data. The algorithm transforms the proposition expressions into equations. These equations are filled with data from the stream and simplified as much as possible to a single boolean value. If this value is true and the output value is marked as a trigger, a violation is reported. Lola does, however, not support the verification of timing constraints, making the overlap with the *property set* rather small. Adding additional functionality to Lola could be challenging, as according to its authors, Lola does require some syntactic sugar before it can support the specification of common constructs.

**Play-engine**

As mentioned in Chapter [4], the verification system that uses Time-Enriched LSCs to verify GUI applications uses a monitoring system called the *play-engine* [39]. The play-engine uses the Time-Enriched LSCs in its monitors that are also used to specify properties in the language of the toolset. Because of this, the play-engine also only covers some of the properties in the *property set* and is not deemed easily extendable.

**RuleR and RuleR-lite**

RuleR is a highly expressive rule-based runtime verification system [11]. According to Falcone et al. [33], RuleR has its roots in Eagle [10]. Eagle is a general-purpose rule-based, temporal system for defining monitors for complex temporal behavioural patterns. Using the Eagle logic, properties can be expressed in a multitude of underlying logics, including real-time and interval logics. An efficient monitoring algorithm never became a reality, partly due to the complex constructs in Eagle’s logic. RuleR was originally designed as a low-level system into which specification written in languages such as Eagle could be compiled. Eventually, RuleR became a runtime verification system in its own right [33]. RuleR-lite [33] is a cut-down version of RuleR. According to Falcone et al. it removes many of the powerful features while remaining highly expressive.

At the time of writing, only a prototype implementation of RuleR rule parsing and monitoring exists. This can only be used for Java program verification using AspectJ aspects, making it not modular. Timing constraints are also not supported. Therefore, much work is required to make RuleR suitable for MCA verification. The same holds for RuleR-lite, as it is based on RuleR.

**TRACE**

In [41], Hendriks et al. present TRACE, a tool for the analysis of execution traces. Traces can be checked using Metric Temporal Logic (MTL) [45]. MTL can be used to specify, among other things, temporal properties of real-time systems. This makes TRACE support some properties of the *property set*. TRACE is, however, aimed at particular execution traces of a machine, which are focused on claiming and releasing of resources. As we aim at more general traces, using TRACE for our goals could be complex.

43
8.1.3 Review

In Table 8.1, all systems are rated using the requirements specified earlier. The rating uses a ‘good-avg.-bad’ system. The rating is based on facts and opinions from the scientific community, together with our opinions. All ratings are relative and do not express any precise quantification.

<table>
<thead>
<tr>
<th>Name</th>
<th>Extendability</th>
<th>Fit for purpose</th>
<th>Modularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIPV</td>
<td>bad</td>
<td>bad</td>
<td>good</td>
</tr>
<tr>
<td>ComMA</td>
<td>avg.</td>
<td>bad</td>
<td>good</td>
</tr>
<tr>
<td>Java monitors</td>
<td>bad</td>
<td>avg.</td>
<td>bad</td>
</tr>
<tr>
<td>JCoSim</td>
<td>good</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Lola</td>
<td>bad</td>
<td>bad</td>
<td>good</td>
</tr>
<tr>
<td>Play Engine</td>
<td>bad</td>
<td>avg.</td>
<td>good</td>
</tr>
<tr>
<td>RuleR and RuleR-lite</td>
<td>bad</td>
<td>avg.</td>
<td>bad</td>
</tr>
<tr>
<td>TRACE</td>
<td>avg.</td>
<td>avg.</td>
<td>good</td>
</tr>
</tbody>
</table>

Table 8.1: Monitoring systems rating

JCoSim matches our requirements the best and therefore provides the best starting point for our MCA monitoring system. JCoSim already allows the verification of many properties in the property set. We also deem JCoSim, of all systems, to be the easiest to extend with new functionality. This is partly because much knowledge of the tool is available at TNO. We decided not to create our own solution but rather modify an existing one, as JCoSim already contains much useful functionality. A completely new solution would, in our eyes, not be able to differentiate itself enough to be justifiable.
8.2 Monitoring System Solution

This section describes the approach to the monitoring system and covers its implementation details. As JCoSim is deemed the best starting point for a monitoring system, in Section 8.2.1, the functionality of JCoSim is discussed. Section 8.2.2 outlines how JCoSim is adapted to fit the MCA domain. These improvements are discussed in detail in Section 8.2.3.

8.2.1 JCoSim Internals

As mentioned before, JCoSim is a co-simulation tool. It allows all components of a system to be simulated using simulators. Each component has its own simulator instance. Information flow is realised by messages. All simulators are connected via a publish-subscribe network. This network handles the message flow between the simulators by acting as a gateway. It distributes a message published by one simulator to all subscribers of that message. JCoSim also provides the timing mechanism needed for a co-simulation.

JCoSim was originally developed to simulate lighting systems. These systems operate using the same three-stage control cycle as PLCs (using sensors, controllers and actuators). The input of JCoSim is in the form of events which trigger sensor simulators. To execute a co-simulation in JCoSim, sequences of these events are published at their predetermined timestamp by an input simulator. Based on these events, sensor simulators produce messages to which controller simulators are subscribed. Similar interaction occurs between controllers and actuators. Actuator simulators produce messages which act as the output of JCoSim. See Figure 8.1 for a simplified view of the data flow through the JCoSim simulators.

The sensor, controller and actuator simulators also publish messages containing internal information, like variable values. Proposition simulators subscribe to these messages. A proposition simulator could, for example, be listening to the output of a temperature sensor. Here, the proposition simulator has a proposition regarding the sensor temperature value, which evaluates to true or false. What proposition simulators are active depends on what properties of what system components are to be checked. A proposition simulator observes one variable or event. Proposition simulators can also be subscribed to other propositions, where they combine received results using the boolean operators AND and OR. Multiple of these proposition combinations can be chained, allowing for complex propositions. A proposition simulator evaluates its proposition whenever it receives a message it was subscribed to. The result of this message is published via the network.
Requirement simulators are subscribed to messages from the proposition simulators. A requirement simulator represents a property. As mentioned before, JCoSim supports several pattern-scope combinations from the property pattern catalogue of Autilli et al. \[9\]. For every pattern-scope combination, a requirement simulator exists. The number of property simulators a requirement simulator is subscribed to depends on the pattern and scope. Where the \textit{Globally: Absence }P combination requires one proposition, the \textit{Between }R and \textit{S: Absence }P requires three. All requirement simulators internally consist of a timed automaton and are described in \[17\]. Whenever a property is validated or violated, the verdict is published.

Figure 8.2 shows a visual representation of the publish-subscribe and timing mechanism in JCoSim. Here, a few concepts are at play:

- **Simulator Timing:** A map that stores, for each simulator, the earliest time at which that simulator wants to be notified of a time change.

- **Message Queue:** A list in which all messages sent by the simulators in one global time tick are stored.

- **Simulator Execution:** The execution of a simulator can be initiated in two ways: Whenever a certain time is reached (the simulator function \textit{timeAdvancedTo} is called), or whenever a message to which the simulator is subscribed is available (\textit{processMessage} function is called). Both functions can be called in one time tick. During its execution, the simulator publishes messages using the \textit{sendMessage} function. The simulator can also notify the mechanism of the earliest co-simulation time tick that it wants to be notified again.

The duration of a time tick corresponds to the timestamp of events used as the input of JCoSim. The length of a time tick, which can be variable, is therefore defined by the difference between two consecutive unique timestamps. Every time tick, the execution of the publish-subscribe and timing mechanism is handled in the following two phases:

1. **Timing Phase:** This phase starts whenever the previous time tick has been handled completely. In the timing phase, the \textit{Simulator Timing} map is scanned for the earliest time \(t\) that at least one simulator wants to be notified. When \(t\) has been determined, a set \(E\) of simulators that should be notified at \(t\) is determined. For each simulator \(s \in E\) the mechanism calls the \textit{timeAdvancedTo} function. The timing phase ends when \textit{timeAdvancedTo} has been called for all simulators in \(E\).

2. **Message Handling Phase:** This phase starts when the timing phase has finished. A message \(M\) is taken from the \textit{Message Queue}. The mechanism calls the \textit{processMessage} function of each simulator that is subscribed to \(M\). The message handling phase finishes when all messages in the queue have been processed. As new messages can be added to the queue during this phase, the processing of multiple chained simulators can be performed in one time tick.

At the start of the JCoSim execution, all simulators call the \textit{advanceTo} function with time 0. The execution is started at time tick 0. The execution stops whenever all input events have been published and processed.
8.2.2 Approach

In JCoSim, only systems that are simulated can be verified. As our research aims at the verification of MCAs, an additional simulation of such application in JCoSim is not considered useful. Simulating a system in JCoSim would require the entire system to be modelled again. Hence, we are only interested in the publish-subscribe and timing mechanism and the proposition and requirement simulator functionality of JCoSim. Because of this, we propose a custom version of JCoSim, named Monitoro. Figure 8.3 shows a simplified view of the data flow through Monitoro. Here, we renamed most simulators to monitors and requirement simulator to property monitor, to better reflect their functionalities.

![Simplified view of data flow through Monitoro](image)

In Monitoro, proposition monitors are no longer subscribed to messages from sensor simulators. They are now subscribed to an input simulator. This input simulator takes external data in the format explained in Chapter 6 and publishes messages representing this data at their predetermined timestamp. This effectively replaces the co-simulation part of JCoSim. To handle offline and online runtime monitoring, two input simulators are available:

- **Online**: The input simulator receives real-time data from the TraceAdapter, see Chapter 6. Although this data is real-time, it still includes a timestamp of the exact time it happened. Data is published whenever the data is received and the processing of all time ticks which have an earlier time than the timestamp of the new data, has been completed.

- **Offline**: An input simulator is used that reads a log file containing data and publishes the messages at their designated timestamp.

The verdict monitor is subscribed to all property monitors, see Figure 8.3. This monitor has been modified to transform the verdicts into the format discussed in Chapter 6.

To ease the processing load, Monitoro only monitors changes in MCA data. This prevents the re-processing of application data values which have already been observed and processed in a previous time tick and have not changed since then. To facilitate this, application attribute data should be provided in the following ways: During the startup of the monitoring, all data is considered ‘changed’, so all application data should be provided to Monitoro. For every subsequent timestamp, only application data that has changed its value should be provided to Monitoro.

8.2.3 Proposition Monitors

In Chapter 5, two categories of propositions are defined, namely phase and event propositions. Phase propositions can hold for a longer time period, whereas event propositions hold for an infinitesimal time period. Automata to capture these categories of propositions have been developed by Buit [17], see Figure 8.4 and 8.5. Here, holds() and ¬holds() denote a proposition resolving to true or false respectively. The start! and end! are events sent to signal that a proposition started or stopped holding, respectively. This makes the phase proposition automaton (PPA) in Figure 8.4 map the start and end of a phase proposition onto events. This is required as all monitors in Monitoro can only communicate using events (in the form of messages). The event proposition automaton (EPA) in Figure 8.5 only fires an event (message) when its proposition is evaluated to true.
Four types of proposition monitors exist to capture MCA data, see below. These are internally comprised of one of the two types of automata mentioned above. When reasoning about the string data type in a proposition, the operators = and $\neq$ can be used. Number data types can be compared using the $=$, $<$, $\leq$, $>$ and $\geq$ operators.

- **PhaseSinglePropositionMonitor**: Comprised of a PPA, observes one data query of an MCA component and compares it to a predetermined value (for instance, \texttt{component1.state == 'on'}).

- **PhaseDualPropositionMonitor**: Comprised of a PPA, observes two data queries and compares them (for instance, \texttt{component1.state == component2.state}).

- **EventZeroPropositionMonitor**: Comprised of an EPA, observes the value change of a single data query (for instance, \texttt{component1.state}). See change keyword in Section 7.3.2.

- **EventSinglePropositionMonitor**: Comprised of an EPA, observes one data query of an MCA component and compares it to a predetermined value (for instance, the receipt of the command 'on': \texttt{component1.command == 'on'}}).

The *property set* discussed in Chapter 5 includes combinations of propositions. The AND, OR and NOT propositions were already supported in JCoSim, whereas the XOR, ANY and ALL propositions are new. The NOT operator can negate a phase proposition, but one cannot check for NOT an event. Proposition combinator semantics can be explained in terms of timed automata. See [67] for the semantics of the AND, OR and NOT proposition combinators. Figure 8.6 shows the semantics of the XOR proposition. Here, \(L\) and \(R\) represent the left and right proposition respectively, which are combined by the XOR operator. The automaton can be presented like this, as when the monitoring starts, data of both \(L\) and \(R\) is received.

To explicitly only observe the start or end of when a phase proposition holds, a **TypeCast-PropositionMonitor** is available. This monitor can also be categorised under proposition combinators. It is subscribed to one of the phase proposition monitors defined above. The monitor is an extension of a JCoSim simulator which supports similar functionality.

Because of the publish-subscribe network, multiple proposition combinator monitors and property monitors can subscribe to the same proposition monitor. As a proposition monitor observes a trace for one piece of data in one component instance, a new proposition monitor is required for every unique piece of data which is to be observed in an MCA.
8.2.4 Property Monitors

Monitor Improvements

As mentioned before, property monitors can subscribe to one or more proposition monitors. The automaton which the property monitor is comprised of can observe sequences of these messages (e.g. events). Verdicts are produced by the automaton based on whether or not such sequence complies with the property of the automaton.

As mentioned in Section 8.1, JCoSim includes simulators comprised of timed automata to observe some of the pattern-scope combinations to which properties of the property set were mapped to. These do, however, have two limitations:

1. Once an automaton moves to an error state, it gets stuck in this error state. As the automaton cannot leave this error state, it is not able to detect more violations. Because of this, an automaton can only detect the first violation of a property.

2. Due to the nature of runtime verification, the execution of a system can stop at any time. When this happens, an automaton might still be in the middle of determining its verdict (e.g. waiting for a timer to expire).

In [22] Cimatti et al. introduce runtime monitors which can be reset. According to them, this allows the monitor to detect the violation of a property multiple times. The monitors of Cimatti et al. feature more than a simple monitor restart, as the monitor can be reset without completely losing its history (e.g. forgetting a scope is still active). This principle has been applied to enhance the existing automata of the JCoSim simulators. This solves the first limitation mentioned above. Collecting data about every property violation that occurs opens the door for extensive result visualisations.

To combat the problem runtime verification has with finite execution traces (see Section 3.2.2), Buit defined four categories of monitor verdicts, namely OK, MAYBE OK, MAYBE ERROR and ERROR [17], see below. The MAYBE OK and MAYBE ERROR verdicts solve, among other things, the problem of the verification of liveness properties using finite traces. These two verdicts should only be produced when a trace has been completely processed. The OK and ERROR verdicts are to be produced when a property holds or does not hold respectively. When a trace has been completely processed, it is assumed that the MCA situation remains indefinitely, as there is no data to prove otherwise.

- **OK**: Property holds.
- **MAYBE OK**: Assuming the current MCA situation remains indefinitely, the property will hold.
- **MAYBE ERROR**: Assuming the current MCA situation remains indefinitely, the property will not hold.
- **ERROR**: Property does not hold.

These verdict categories were, however, never applied to the automata Buit developed. To be able to handle finite traces, we improved the automata defined by Buit and Verriet et al. by adding support for the four verdict categories. This solves the second limitation mentioned earlier.

For all pattern-scope combinations not supported by JCoSim, we developed new automata. This makes it possible to observe more properties of the property set. Details regarding improvements and the new automata are discussed below.
Monitor Internals

In this subsection, the automata of which the property monitors are comprised, are discussed in detail. These include the automata in JCoSim, improved to mitigate the two limitations described above, and new automata to observe the properties of the property set which were not supported by JCoSim. All these automata combined are from here on referred to as the automata set.

Just like JCoSim requirement simulators, property monitors are comprised of timed automata. Timed automata are, as defined by Alur and Dill in [8], finite state machines extended with clock variables. As one property monitor can be used to observe a property in the form of one pattern-scope combination, the timed automata of the automata set provide formalised semantics of all pattern-scope combinations used by properties of the property set. We developed all automata in the timed automata formalism of the UPPAAL model checker [12]. In this subsection, we only provide an informal description of the semantics of these automata. For the formal semantics of UPPAAL and timed automata we refer to [12] and [8], respectively.

A timed automaton starts in its initial state. Other states can be reached via directional edges. Edges are labelled with synchronisations, guards and actions, which are all optional. Using a synchronisation, an event can be sent (marked with ‘!’) or received (marked with ‘?’). A guard is an expression which resolves to true or false. Expressions can, for example, be clock constraints (i.e. timers) or integer expressions. Whenever an event is received, and the guard expression resolves to true, the corresponding transition is taken. A transition with an event but without a guard is always taken when the event is received.

In UPPAAL timed automata, initial states are marked with an extra inner circle. Committed states are shown with a ‘C’ in their bodies. A committed state is a transient state, meaning an automaton cannot rest on such a state. It needs to transition to a normal state immediately. The verdicts Ok, Maybe Ok, Maybe Error and Error, discussed in the previous subsection are defined in the automata as ok, maybe_ok, maybe_error and error events, fca respectively. The P, S, Q and R events represent propositions. Here, P is the main proposition. In response properties, S (effect) is the response to P (cause). Q and R are the propositions which signal the opening and closing of a scope respectively.

Figure 8.7 shows the UPPAAL timed automaton used in the monitoring of the Globally: Absence P property. This property describes that proposition P should never resolve to true. In this automaton, resets are handled by the loop between the Error and ErrorRep states. Using this loop, the automaton can receive multiple P events, and produce an Error verdict for each of them. To cope with finite traces, an end-of-trace event (eot label) is introduced. This event is sent by the InputSimulator in Monitoro when the end of a trace has been reached. If the Globally Absence automaton in Figure 8.7 receives no P event during the entire processing of a trace, the automaton will remain in its initial state. When the trace is processed completely, noted by a eot event, the automaton transitions to the EndOk state and thereby produces a Maybe Ok verdict. The Maybe Ok verdict is returned here as no P has been received yet and the automaton cannot be sure that a P will never be received.

Figure 8.7: UPPAAL automaton for Globally: Absence P
We propose a new notation to simplify the representation of UPPAAL timed automata. This notation supports a subset of the UPPAAL timed automata functionality, namely the functionality required by the automata set. The notation only changes the visual representation of the automata. Their semantics remains unchanged. The notation simplifies the representation of repeated constructs. Because of this, fewer visual elements are needed to describe the same automaton, which improves its readability. See Appendix E for all automata of the automata set in our notation.

In our notation, states can be coloured, where a state has at most one colour. The state colouring is defined as follows:

1. When a green or red state is reached, an Ok or Error verdict is implicitly produced, respectively.
2. When an automaton is in a yellow or orange state, and an end-of-trace event is received, a Maybe Ok or Maybe Error verdict is produced, respectively.

See Figure 8.8 for the Globally Absence automaton of Figure 8.7 expressed in the new notation. The red state in Figure 8.8 combines the ErrorRep and Error states of Figure 8.7 into one state and removes the need for the labelled err! edge between them. The yellow state in Figure 8.8 combines the leftmost three states of Figure 8.7.

As mentioned earlier, all automata of the automata set can perform resets. The automata can perform these resets without losing their history. This means that an automaton can remember what events it observed in the past. Using this, an automaton can, for example, remember that a scope is currently open. For the Globally Absence automaton displayed in Figure 8.8, the resetting happens as follows: Every time the red loop-back state is reached, the automaton is reset. This means it can observe another occurrence of the $P$ event. As the automaton does not transition back to its initial state, it can keep the knowledge that a $P$ event has already been observed at least once. If this history functionality was not present and the automaton would have transitioned back to the initial state after a reset, it would have produced a Maybe Ok verdict at the end of the trace. This would not be correct, as a $P$ event was most definitely observed, which violates the Absence pattern.

To more clearly visualise the reset functionality, states can be annotated as loop-back states. These are displayed with a double outline. The loop-back annotation is purely a visual aspect in our notation, as annotated states in our notation are regular states in the UPPAAL notation. An automaton can be viewed as reset whenever a loop-back state is reached in one of the following ways:

1. From a state which just produced an Ok or Error verdict, see red state in Figure 8.8.
2. Via a transition which was triggered by the closure of a scope.

Figure 8.8: Globally: Absence $P$
Figure 8.9 shows the UPPAAL timed automaton used in the monitoring of the After $Q$ Until $R$: Bounded Recurrence $P$ property. This property can be described as: After proposition $Q$ holds, every $c$ time units, proposition $P$ should hold, until proposition $R$ holds. In this automaton, $t$ is a timer with bound $c$. In UPPAAL timed automata, timers are started when the automaton starts. Timers are reset using ($t := 0$). Whenever $t$ expires, the guard $t \geq c$ resolves to true. Because of this, a transition via the committed Error state is made whereby an ERROR verdict is produced.

An UPPAAL timed automaton has non-eager semantics. This means the automaton does not automatically take a transition that has no synchronisation but has a guard that resolved to true. To force such transition, an invariant is added to the outgoing state of such transition. Such invariant specifies an expression. As long as this expression resolves to true, the automata can remain in the state. The invariant expression is almost the opposite of the guard expression. Some overlap is required between the expressions as otherwise, a deadlock will occur, meaning the automaton cannot leave the state. In Figure 8.9, the In state has an invariant $t \leq c$, which forces the transition to the Error state to be taken whenever the timer guard $t \geq c$ resolves to true (i.e. when $t$ equals $c$).

See Figure 8.10 for the After Until Bounded Recurrence automaton of Figure 8.9 expressed in our notation. Here, the white, orange, green and red states correspond to the Init, In, Ok and Error states of Figure 8.9. The monitor in Figure 8.10 contains two loop-back states. The orange loop-back state functions as the state to be returned to when an OK or ERROR verdict is given. The white loop-back state is returned to when the After Until scope is closed. The red and green states are committed, annotated by the dashed circle outline.

Figure 8.10: After $Q$ Until $R$: Bounded Recurrence $P$
Figure 8.11 shows the UPPAAL timed automaton used in the monitoring of the *Globally: Time-constrained Absence* $P$ property. This property can be described as: *$P$ should hold for at most $c$ time units in total.* The start or end of when proposition $P$ holds are observed using $P$ and $\neg P$ respectively. This automaton requires a timer that can pause and resume. In UPPAAL, this is handled by invariants that use stopwatches. A stopwatch is the derivative ($t'$) of a timer $t$. When $t'$ equals 0, the timer is paused, when it equals 1, the timer is running. In our notation, the invariants are replaced by `pause()` and `resume()` actions on the incoming edges, see Figure 8.12. As timers in UPPAAL are started whenever they are reset $t := 0$ or when the automaton starts, the default value of $t'$ equals 1. Because of this, in our notation the timer needs to be paused whenever the timer is reset or the automaton is started for the first time.

![Figure 8.11: UPPAAL automaton for *Globally: Time-constrained Absence* $P$](image)

**Figure 8.11:** UPPAAL automaton for *Globally: Time-constrained Absence* $P$

**Figure 8.12:** *Globally: Time-constrained Absence* $P$

**Translation to UPPAAL notation**

The translation from our notation to the UPPAAL timed automata notation is achieved by following steps below in order:

1. For each red state $S$, if present, remove any edge $E$ with source and target $S$ (self-loop). For each $S$ that applies, add a new edge $E'$ without a label, to a new regular state $S'$. Add an edge $E''$ from $S'$ to $S$ with the same label as $E$.

2. For each non-committed green state $S$, add a new edge $E$ without a label, to a new regular state $S'$. For each $S$ that applies, remove all outgoing edges and attach them to $S'$, meaning $S'$ becomes the source state of those edges.

3. Introduce `err!` and `ok!` labels to unlabelled edges that have red or green source states, respectively.

4. Introduce state invariants to states which have outgoing edges that have timer guards without having events. Express the invariant as the opposite of the timer guard expression while keeping some overlap (e.g. ‘$\geq$’ becomes ‘$\leq$’ and vice versa).

5. Introduce invariants that compare the derivative of a timer to zero or one, to states that have an incoming edge with a `pause()` or `resume()` action, respectively. On these automata using the stopwatch, add `pause()` actions to the edges with actions that reset the timer ($t := 0$). Also add a `pause()` action on the edge that has the initial state as its source.

6. For each yellow state $S$, add a new edge $E$ with label `eot?` to a new committed state $S'$. From $S'$, add an edge $E'$ labelled `maybe_ok!` to a new regular state $S''$.

7. For each orange state $S$, add a new edge $E$ with label `eot?` to a new committed state $S'$. From $S'$, add an edge $E'$ labelled `maybe_error!` to a new regular state $S''$.

8. Perform some visual changes: Annotate committed states with a ‘c’ instead of a dashed circle outline. Remove any colouring from the states. Remove the loop-back state annotation. Specify all initial states with an extra inner circle.
Pattern proposition relation in our automata

In this subsection, the relations between patterns, and the two types of propositions (phase/event) are discussed.

When an automaton supports a proposition of the phase type, it can observe the start and end events of such a proposition. This allows it to ‘remember’ if a phase is active or not. When an automaton supports event propositions, it observes individual events without any memorising involved. The enumerations below show which automata of the automata set should support which proposition type. The phase/event distinction is made as most patterns, can logically only handle one type of proposition. The Minimum Duration pattern, for example, specifies that a proposition should hold for at least \( c \) time units. As an event proposition can only hold for an infinitesimal time period, it cannot logically be used with the Minimum Duration pattern. Because of this, the Minimum Duration pattern only supports phase propositions. We designed all automata shown below except the ones denoted by ‘F’, meaning ‘future work’.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Absence</td>
<td>• Absence</td>
</tr>
<tr>
<td>• Bounded/Strict Response (F)</td>
<td>• Bounded/Strict Response</td>
</tr>
<tr>
<td>• Prefaced Bounded Response (F)</td>
<td>• Prefaced Bounded Response</td>
</tr>
<tr>
<td>• Universality</td>
<td>• Bounded Existence</td>
</tr>
<tr>
<td>• Minimum Duration</td>
<td>• Bounded Recurrence</td>
</tr>
<tr>
<td>• Maximum Duration</td>
<td>• Bounded Absence</td>
</tr>
<tr>
<td>• Time-constrained Absence</td>
<td>• Restricted Recurrence</td>
</tr>
<tr>
<td>• Time-constrained Universality</td>
<td>Duration Bounded Existence</td>
</tr>
</tbody>
</table>

One pattern that can be used with both phase and event propositions is the Absence pattern. Therefore, two separate automata have been designed to monitor properties that use the After Until Absence pattern-scope combination. One automaton supports its \( P \) proposition as a phase, and the other one supports it as an event. Due to the nature of the Absence pattern, only one automaton is required to handle both phase and event propositions when combined with the Globally scope. No other automata in the automata set share this ability.

Other patterns that can benefit from being used with both phase and event propositions are the Bounded Response, Strict Response and Prefaced Bounded Response patterns. A generalised response property reads as follows: **When \( P \) occurs, then \( S \) should occur within \( c \) time units.** Consider the property: **When a sensor triggers, the lift should be in the ‘up’ state within 1 second.** Here, \( P \) is a sensor trigger and \( S \) is: \texttt{start of lift.state == 'up'}. If the lift is already up before the sensor triggers and remains there during the second after the trigger, the property is incorrectly violated. To solve this, \( S \) should be defined as a phase. This way, an automaton can track before the trigger even occurs if the lift is up or not. Most properties in the property set can be monitored using response automata that only support the \( S \) proposition as an event. Therefore, only event versions of these automata have been designed. Versions that can handle phase propositions can be created using our new notation. Creating them is future work.

As the Strict Response pattern is syntactic sugar of the Bounded Response, both patterns can be observed by the same automata. Some patterns can be defined in two ways. For example, the Bounded Existence pattern can be defined as **at least \( k \) or at most \( k \)**. In such cases, two separate automata have been designed.
Runtime verification with UPPAAL

Although all automata of the automata set exist as UPPAAL timed automata, we opted not to use UPPAAL for the runtime verification of MCAs. Although it might be possible, we still use Monitoro for four reasons:

- UPPAAL cannot be used for the online runtime verification approach.
- UPPAAL has high licence fees if it is to be used for non-academic purposes. As Monitoro is our own tool, it has no such fees.
- We do not want to be dependable on the UPPAAL ecosystem, as we have no control over its future directions and how these would affect our modular toolset.
- Although a lesser problem than the reasons above, UPPAAL remains a tool with its goal being model checking, not runtime verification.

8.2.5 Monitor Verification

To make sure all automata of the automata set do what they are supposed to do, we created additional verification automata in UPPAAL. See Figure 8.14 for the verification automaton (VA) of the After Until Bounded Recurrence automaton of Figure 8.13. The goal is to gain more confidence in the correctness of the verdict generation of our property monitor automata. Here, we look at soundness and completeness. Soundness means that no Ok or Error verdicts are produced when they should not be produced, and completeness means that when verdicts should be produced, they are produced.

A VA works by producing events that the original automaton observes and listening to the verdicts produced. Whenever an ok or error event is received by a VA when it should not be received, the VA transitions to an error state (ErrorOk and ErrorError in Figure 8.14). When such a state is reached, one can conclude that the Ok or Error verdict produced by the original automaton is a false positive. This means the verdict production is not sound. Whenever the original automaton should produce a verdict, the VA enters a committed state which can only be left if that verdict is received. If at this time the verdict is not received, one can conclude that the original automaton does not produce verdicts when they should be produced. This indicates that the original automaton is not complete.

![Figure 8.13: UPPAAL automaton for After Q Until R: Bounded Recurrence P](image1)

![Figure 8.14: Verification UPPAAL automaton for After Q Until R: Bounded Recurrence P](image2)
Using the original automaton and the VA, both soundness and completeness can be checked by checking that no deadlocks occur in them except one. This one is a deadlock that occurs in the VA state reached when a \textit{maybe} \{\ldots\} verdict has been produced after an \textit{eot} event (the \textit{End} state in Figure \[8.14\]). For the automata above, the check is done using the UPPAAL formula $A[ \text{deadlock} \Rightarrow \text{End} ]$. All automata set automata, except the Bounded/Strict Response automata, have been verified using this method. As we encountered some roadblocks performing this method on the Bounded/Strict Response automata, properly verifying them is future work.

We do not state that this verification proves the soundness and completeness of the automata set automata, and thereby their correctness. Errors could also exist in the VAs. Nevertheless, this verification increases the chances that some problems are discovered. This corresponds to a quote from George E.P. Box: ‘All models are wrong, but some are useful’ \[16\].

8.3 Conclusion

In this chapter, the co-simulator JCoSim was selected as a runtime monitoring tool to provide verification functionality regarding properties in the \textit{property set}. This tool has been adapted, in the form of Monitoro, to make it suitable for our modular verification toolset. Our contributions detailed in this chapter are:

- Added new functionality, in the form of \textit{Maybe Ok} and \textit{Maybe Error} verdict production, to the timed automata of Buit \[17\] and Verriet et al. \[71\], to cope with finite traces.
- Added new functionality, in the form of resets, to the timed automata of Buit and Verriet et al. to handle the detection of multiple compliances or violations of a property.
- Proposition monitors which can handle proposition combinations using the XOR, ANY and ALL operators.
- New monitors, in the form of timed automata, to support all pattern-scope combinations used in properties of the \textit{property set} which were not supported by JCoSim (and multiple other tools discussed in this chapter). These automata also have the functionality to cope with finite traces and the detection of multiple compliances or violations of a property.
- A new simplified notation for all our UPPAAL timed automata.
- An attempt at soundness and completeness validation of all new and improved timed automata in UPPAAL.
- An adapted version of JCoSim, called Monitoro, which can handle the verification of machine control application execution traces against most properties in the \textit{property set}, and also functions in our modular verification approach toolset. In Monitoro, support for new properties can be added just by creating new proposition or property monitors. See the paragraph below which properties are not supported.

As mentioned in this chapter, to support all properties in the \textit{property set}, some patterns should be usable with both types of propositions (event/phase). This fact holds for eight of the 28 total pattern-scope combinations we defined. In Monitoro, six of these eight can currently only be used with event propositions. Therefore, Monitoro currently does not support a small portion of the \textit{property set}. Adding phase proposition support for these six is future work.

Using all information in this chapter, research question RQ6, \textit{What is a suitable technique to verify machine control application data against properties?} can be answered in the following way: Using the runtime monitoring technique which uses monitors, comprised of timed automata representing pattern-scope combinations, handled by Monitoro.
Chapter 9

Verification Result Feedback

In this chapter, the feedback to users relating the results of the verification of machine control application (MCA) properties is discussed. Section 9.1 discusses the goal of providing this feedback to users. In Section 9.2, the feedback functionality of other runtime verification and model checking systems are reviewed. Section 9.3 details an overview of collected feedback approaches suitable for our verification toolset. Section 9.4 provides an overview of our contributions in the form of a conclusion. In this section, the information detailed in this chapter is used to answer research question RQ7 What data is required to understand and process the verification results?.

The goal of the research detailed in this chapter is to define approaches to the feedback of MCA runtime verification. We correlate these approaches to the capabilities of our verification toolset. This information can help people fulfilling the maintainer role of our verification toolset to design appropriate visualisations for their companies.

9.1 Goal of Feedback

Receiving feedback regarding the conformance to, or violations of properties is a vital aspect of a verification toolset. If one can specify many properties, but cannot conclude if an MCA conforms to those properties or not, and why, a verification toolset is not of much use. In this chapter, we try to answer what feedback should be provided to a user (e.g. if and when a property is violated). The question of how this data should be provided (e.g. in a nice coloured timeline), is out of scope. By answering the former question, the interfaces between the tools of our verification toolset can be prepared to make this feedback data available.

We categorise possible feedback data into two types which can both be used by users to answer a question:

1. **Basic feedback**: Does the MCA conform to the property?
2. **Debugging feedback**: Why does the MCA not conform to the property?

The exploration of why something does not conform to requirements is a process known in the software world as debugging. Rössler describes the four steps of a debugging process [65]:

1. **Failure registration**: Finding out a failure has occurred in the system.
2. **Failure reproduction**: Reproducing the failure which allows one to replay the failure scenario.
3. **Failure understanding**: Understanding how the failure comes into existence.
4. **Failure resolvement**: Fixing the defect which causes the failure.
In our verification toolset, the first step of the debugging process is performed by the Monitoring System. Here, the failure is a property violation. By providing detailed information about the violation detected in the first step, the user can be assisted in performing the second and third step. Such information would be a part of the debugging feedback category. Feedback to MCAs instead of users, which can be a part of a runtime verification toolset, is out of the scope of this research. Such feedback could automatically solve problems in an MCA that cause property violations. As such feedback is out of scope, our verification toolset does not perform the fourth step. In this chapter, we focus on helping people fulfilling the user role with the first three steps.

9.2 Previous Work

In this section, the feedback functionality of verification systems discussed in Chapter 8 and of verification systems connected to the languages discussed in Chapter 7 are detailed.

9.2.1 Non-Runtime Verification Feedback

For most non-runtime verification systems, feedback functionality is not discussed in their respective papers, or only very elementary feedback is provided in the form of yes/no (conforms to property / property is violated). The verification systems which use a model checking approach can provide additional feedback of violated properties in the form of counterexamples. These counterexamples can be categorised as debugging feedback as they are traces which lead to the violation of a property and can, therefore, be used for failure reproduction.

9.2.2 Runtime Verification Feedback

Just like non-runtime verification systems, several runtime verification systems only provide feedback in the form of yes/no answers. Because of the nature of runtime verification, a trace that violates a property acts as a free counterexample if it is logged. ComMA expands on this by providing the user with the difference between the expected and observed behaviour. ComMA also tries to provide the user with relevant system data (e.g. variable values) at the time of the property violation. TRACE provides additional feedback in the form of displaying at what time, what propositions resolved to true. The IoT solution only provides feedback using a lighting-specific visualisation tool called Spectrum. This tool visualises, at simulation speed, which properties were violated and which properties the system conformed to. A similar approach is used in Time-Enriched LSCs. Using the Time-Enriched LSCs toolset, a user can view a simulation of a system similar to a step-by-step debugging tool. Here, the user is presented with the Time-Enriched LSCs used to specify the properties. Using visual clues, the user is provided with feedback in the form of system data (e.g. variables) and violations of properties.

Concluding, most verification systems provide very elementary feedback in the form of yes/no answers. Some tools do provide feedback in the form of the difference between the expected and observed behaviour. Some tools also give an insight into data values at the time of the violation. Other tools only provide a more intuitive version of the yes/no answers using a visualisation. All three approaches above can be used for failure understanding.
9.3 Feedback Approaches

This section provides an overview of MCA runtime verification result feedback approaches. We and our contacts at Cordis Automation deem these approaches helpful to users who are processing verification results. The approaches are categorised as basic feedback or debugging feedback. The availability of the data in our toolset which is required by these approaches, is also discussed.

Global overview of verdicts

This basic feedback approach provides an overview of how many times verdicts (i.e. OK/ERROR) have been produced for each property. This allows a user to quickly see which properties have been violated and which the MCA conformed to. This overview only requires data of the verdicts generated by the Monitoring System.

Detailed verdict data per property

This basic feedback approach extends the global overview approach by providing more detailed information about each verdict. Here, per verdict, the time of verdict production and names of relating MCA components are provided. This data is available in the Monitoring System and requires data of the Property Language tool to link verdicts to the correct component names.

Component data

This debugging feedback approach provides internal data of MCA components (e.g. variables, states) over time. This data can be of components used in a property, or of components which might affect other components used in a property. This data is available as output of the Trace Adapter.

Proposition results

This debugging feedback approach provides data of when a proposition changed from satisfied to violated and vice versa, like TRACE [41]. This data is available in the Monitoring System.

Property results

Just like ComMA, this debugging feedback approach shows the difference between expected and observed behaviour relating to a property. For example, if the Bounded Existence pattern was used to specify that something should occur at least five times, but it occurred only three times. The Monitoring System has data available to support this approach for properties relating to counting. Because of the property monitor designs, this data extraction is not supported for timing properties. To retrieve this data, property monitors would need to be able to observe events after a reset (e.g. after a timer expired). This is future work.

Trace differences

This debugging feedback approach provides users with the differences between two MCA traces. This can be used to show the differences between a trace that violated a property and a trace that conforms to that same property. This approach requires the output traces of the Trace Adapter. Determining what exact data of both traces is useful in the comparison of the differences is future work.
9.4 Conclusion

In this chapter, we explored what data of an MCA verification users would like to receive as feedback. This data was categorised into basic feedback and debugging feedback. Basic feedback is used to convey failure registration. For the debugging, we aimed at failure reproduction and failure understanding possibilities. Our contribution detailed in this chapter is:

- A list of six feedback approaches that incorporate data that users want to receive of an MCA verification.

Most of the data required for these approaches can be retrieved from our toolset. Only timing data of property results, used for the comparison between actual and expected behaviour, cannot be retrieved. Extending property monitors of the Monitoring System to support the retrieval of this data is future work.

Using all information detailed in this chapter, research question RQ7, *What data is required to understand and process the verification results?* can be answered in the following way: Data which provides an overview of all verification results and that includes detailed information regarding internal component data, proposition and property results over time. This data can be, among other things, used to compare expected and actual behaviour.
Chapter 10

Verification Data and Formats

In this chapter, data of machine control applications (MCAs) which is required to perform the verification, is discussed. The formats in which this data should flow through the verification toolset are also provided. Section 10.1 discusses what MCA data is required to perform verification. In Section 10.2, the data formats which the tools of the verification toolset use to communicate, are presented. Section 10.3 provides an overview of our contributions in the form of a conclusion. In this section, the information detailed in this chapter is used to answer research question RQ4 What machine control application data is needed for verification using properties?.

10.1 Machine Control Application Data

This section describes restrictions on MCA data used during verification. It also states what data is required for the verification and what data is optional.

Data restrictions

Propositions in the Property Language partially exist of an attribute query. This query can be seen as a unique identifier referring to attribute data in an MCA. As mentioned in Chapter 7, these attribute queries can be chosen freely as long as the identifiers are also used in the trace transformation performed by the Trace Adapter. Because of this, propositions can be defined using any MCA data available.

Even though attribute queries can be freely chosen, the verification toolset issues some restrictions on the data types of these attributes. As mentioned in Chapter 8, only two data types are supported, namely strings and numbers. This is the case as the Monitoring System only has proposition monitors which use these types. By introducing an appropriate Trace Adapter, other types, like booleans, can be mapped to strings (and sometimes numbers).

To have a correctly functioning verification toolset, all tools should use the same notion of time. Therefore all timestamps should be provided in milliseconds.

Required data

To get up to date with the state of an MCA when the verification starts, the Monitoring System requires all values of the MCA data (e.g. variables), which will be observed by the property monitors, to be provided. This is the case as during the startup of the monitoring, all data is considered ‘changed’. For every subsequent timestamp, only application attribute data that has actually changed its value needs to be provided.
Optional data
To facilitate the use of the Strict Response pattern, the cycle time of an MCA should be provided to the Monitoring System. This is the case as the response time of the Strict Response pattern equals the cycle time of an MCA.

As mentioned in Chapter 7, the Property Language supports the notion of each instance of and all instances of. These keywords refer to all instances of a given MCA component class. To facilitate this abstraction, during the parsing of the properties, an instances XML file should be provided. In this file, the component ids of all instances of a given component class are defined, see Listing 10.1. Here, the OrangeLight and WhiteLight components are both instances of the Light component class.

```xml
<classes>
  <class name="MerryGoRound.Light">
    <instance name="MerryGoRound.TrafficLights.OrangeLight"/>
    <instance name="MerryGoRound.TrafficLights.WhiteLight"/>
  </class>
</classes>
```

Listing 10.1: Instances data file format

Conclusion
In conclusion, the verification of machine control applications using our toolset does not require any specific data to be available, as long as all data observed by the property monitors is provided at the start of the verification. This data is, however, restricted to string and number types. Cycle time and component class instances data are only required if one uses the Strict Response pattern and each instance of or all instances of keywords, respectively.

10.2 Toolset Communication Formats

As mentioned in Chapter 6, to facilitate a modular verification approach, the verification toolset lists several clear interfaces. These mandate data travelling between tools to be provided in a fixed format. This allows coupling and uncoupling of tools to and from the verification toolset. The data formats are based on CSV and are discussed below.

10.2.1 Property and Proposition Formats

Output data of the Property Language tool serves as the input of the Monitoring System where it is used to create proposition and property monitors. This output data should be presented to the Monitoring System as two files that use the formats below. These formats support the specification of all properties in the property set.

Proposition format

As discussed in Chapter 8, four propositions are defined (PhaseSingle, PhaseDual, EventZero, and EventSingle). Listing 10.2 shows their formats, where each line represents a different proposition. Values are separated using the ‘;’ delimiter. The first value specifies the proposition type name. The second value specifies a unique proposition id. This id allows the proposition to be referenced in properties or other propositions. After the id, the attribute query represents an identifier to an attribute in an MCA. The next value is the operator of the proposition used to compare the attribute data. EventZero does not have this operator as it is only used to monitor changes in data, which
requires no operator. Finally, the value to compare to is given. This is in the case of PhaseSingle
and EventSingle a string or number value, and in the case of PhaseDual another attribute query.

The last seven propositions in Listing 10.2 are all proposition combinators. These have a unique
name in the form of an id and accept one or more other proposition ids which they combine. The
TypeCast proposition also requires a flank value, which is ‘start’ or ‘end’, representing which point
of a phase proposition should be converted to an event.

// Propositions
PhaseSingle ; <id> ; <attribute_query> ; <operator> ; <value>
PhaseDual ; <id> ; <attribute_query> ; <operator> ; <attribute_query>
EventZero ; <id> ; <attribute_query>
EventSingle ; <id> ; <attribute_query> ; <operator> ; <value>

// Proposition Combinators
TypeCast ; <id> ; <prop_id> ; <flank>
And ; <id> ; <prop_id_left> ; <prop_id_right>
Or ; <id> ; <prop_id_left> ; <prop_id_right>
Xor ; <id> ; <prop_id_left> ; <prop_id_right>
Not ; <id> ; <prop_id>
Any ; <id> ; {<prop_id_1>, ..., <prop_id_n>}
All ; <id> ; {<prop_id_1>, ..., <prop_id_n>}

Listing 10.2: Proposition data file format

Property format
Properties are defined by their pattern-scope combination. Listing 10.3 shows the data formats of
the Globally Absence and After Until Bounded Existence properties. The first value in the property
data format specifies this pattern-scope combination. The second value denotes a unique property id.
All other values differ per property. For Globally Absence, the third value is the id of the proposition
used. The After Until Bounded Existence property also needs a before_prop_id and after_prop_id,
which are ids of propositions which, when they hold, signal the opening and closing of the After Until
scope respectively. The final value can be a ‘<’ or ‘>’ operator and is used to distinguish between
the two versions of the After Until Bounded Existence property, namely at least k and at most k. A
full list of the property formats can be found in Appendix F.

GloballyAbsence ; <id> ; <prop_id>
AfterUntilBoundedExistence ; <id> ; <before_prop_id> ; <after_prop_id>
; <prop_id> ; <amount> ; <operator>

Listing 10.3: Property data file format

When the each instance of and all instances of keywords are used, the Property Language
tool should internally create separate propositions and properties for each affected MCA component
instance.

This proprietary format is not based on any existing logic format like MTL [45] or TCTL [6]. This
has two advantages: Firstly, the property format does not rely on the expressivity of another logic.
Secondly, this does not require the knowledge of these, sometimes considered complex, logics to be
able to add new properties to the verification toolset.

63
10.2.2 Trace Format

The Trace Adapter transforms traces in such a way that they can be understood by the Monitoring System. Listing [10.4] shows the required format of the file containing these transformed traces. Here, each event of a trace is specified using three values. The first value is the timestamp in milliseconds of when the event was produced by an MCA. The second and third value specify the attribute query of an MCA and its value, respectively. This format is also used for an online runtime verification approach, only without the data being stored in a file.

\[ \text{<time> ; <attribute_query> ; <value>} \]

Listing 10.4: Trace data format

10.2.3 Verification Result Format

In Chapter 9, multiple approaches have been described to provide users with feedback of a performed MCA verification. These approaches use data of propositions, properties and traces. Trace data is already available from the Trace Adapter. The Monitoring System outputs two files containing proposition and property data, respectively. The formats of these files are discussed below.

Proposition results are presented as timestamps of when a proposition changed from satisfied to violated and vice versa, see Listing 10.5. This is true for phase propositions, but as event propositions only hold for an infinitesimal time period, they only require one timestamp. The first value represents the timestamp, whereas the second value is the unique id of the proposition. The third value represents the flank of a phase proposition. This flank denotes if the timestamp belongs to the start or end of a period during which a phase proposition is satisfied. As this flank is not required for event propositions, it is optional (marked using *).

Property results are formatted using four values, see Listing 10.6. These are the timestamp of when the verdict was produced, the id of the property, the verdict itself, and one optional value. This optional value can, depending on the property (see Section 9.3), be used to provide the actual value for a comparison between the actual and expected value of a property.

\[ \text{<time> ; <proposition_id> ; *<flank>*} \]

Listing 10.5: Proposition results file format

\[ \text{<time> ; <property_id> ; <verdict> ; *<actual_value>*} \]

Listing 10.6: Property results file format

10.3 Conclusion

In this chapter, we combined the knowledge presented in previous chapters to determine what data is required to perform runtime verification of machine control applications and what the restrictions on this data are. We also presented formats for the communication of the tools in our verification toolset. Therefore, our contribution, detailed in this chapter, is:

- Formats for the communication between the different tools of our modular toolset.

Using data detailed in this chapter, Research Question RQ4 What machine control application data is needed for verification using properties? can be answered as follows: Besides execution traces, no specific machine control application data is required. However, all values of the MCA attributes (e.g. variables), which will be observed by property monitors, should be provided at the start of the monitoring. During the entire monitoring, these values should be of string or number type.
Chapter 11

Evaluation

This chapter describes a test of our runtime verification toolset in the form of an evaluation. Section 11.1 details which parts of the toolset have been implemented to perform the evaluation. Sections 11.2 and 11.3 discuss the procedure and results of the user role evaluation and maintainer role evaluation, respectively. Section 11.4 provides an overview of our contributions in the form of a conclusion. In this section, the information detailed in this chapter is used to answer research question RQ8 Does the verification method improve upon the current verification process?

11.1 Toolset Implementation

To evaluate our verification toolset, we have implemented basic versions of its four tools:

1. A Trace Adapter implemented in Python that can handle logged output of a Cordis SUITE MCA simulation.

2. A Property Language implemented as an Xtext\(^\text{10}\) Domain Specific Language (DSL) that supports a subset of the patterns and scopes used by properties of the property set.

3. A Monitoring System implemented in Java in the form of Monitoro that support a subset of the patterns and scopes used by properties of the property set.

4. A Result Visualisation implemented in Python that supports the two forms of basic feedback mentioned in Chapter 9: Global overview of verdicts and Detailed verdict data per property.

The Property Language and Monitoring System both support the Globally and After Until scopes. They also support the Universality, Absence, Bounded Response, Preface Bounded Response and Bounded Recurrence patterns. The first three patterns have been selected as they are the most commonly used patterns in practice, according to a study by Dwyer et al. \(^\text{[32]}\). Bounded Recurrence is included as it is used often in the property set. Preface Bounded Response is included as it is an interesting version of the Bounded Response.

The Property Language is implemented as an Xtext DSL as Xtext is a popular language engineering framework of which much documentation is available.

\(^{10}\) https://www.eclipse.org/Xtext/
11.2 Evaluation User Role

This section details the procedure and results of an evaluation performed with people who fulfil the user role of our verification toolset.

11.2.1 Goal

This evaluation uses the Property Language and the Result Visualisation tools. These are included as people that fulfil the user role interact with these tools. The evaluation focuses on the following:

- **Phase/event understandability**: Evaluate the understandability of phase and event notions when specifying propositions.
- **Pattern/scope understandability**: Evaluate the understandability of patterns and scopes, as these are two fundamental attributes of a property.
- **Language understandability**: Evaluate the understandability of rulesets and the properties specified in our language.
- **Language modifiability**: Determine the perceived complexity of modifying rulesets and properties.
- **Language creation**: Determine the perceived complexity of creating rulesets and properties.
- **Result understandability**: Evaluate how difficult it is for users to understand basic forms of verification result feedback. This includes the understandability of the four verdicts of the Monitoring System.

11.2.2 Setup

The event/phase propositions, the patterns and scopes, the structure of the property language and the results of a verification are explained to the participants beforehand. The participants are given cheat sheets which contain all necessary information regarding these subjects.

Three exercises have been created to test the six evaluation goals described in the previous section. See Appendix G for all exercises and their solutions in full. At the end of the evaluation, participants were also asked questions related to the evaluation goals.

Participants

This evaluation is carried out using three participants from several backgrounds. One participant is one of our contacts at Cordis Automation. The two other participants are TNO employees involved in the Machinaide project. Our contact at Cordis has previously worked as an MCA developer and therefore has lots of knowledge about all types of disciplines involved in creating MCAs. All participants have programming knowledge, but none have any experience with pattern systems.

Exercise 1: Understanding phase/event and pattern/scope

The first exercise revolves around the understandability of phase/event propositions and patterns and scopes. Participants are presented with several timelines that represent sequences of propositions resolving to true over time, see Figure 11.1. For every timeline, participants are asked if the sequence conforms to one or several properties.

- Property 1: **Between Q and R: when P, then S after at most 40**
Exercise 2: Understanding and modifying rulesets

In this exercise, participants have to understand and modify the ruleset in Figure 11.2. First, they are asked to describe the property the ruleset specifies in their own words. Next, participants are requested to modify the ruleset, proposition, and rule names so that they better express their intent. Finally, participants have to extend the ruleset by incorporating an After Until scope and by using the each instance of keywords to query multiple component instances.

Exercise 3: Creating rulesets and understanding results

This final exercise is used to test the language creation and verification result understandability. Participants are asked to create a ruleset that describes the property below:

During the entire execution of the MerryGoRound machine, the RightLift of the LeftConveyor, should move up at most 600 milliseconds after the sensor in the centre of the LeftConveyor triggered.

- RightLift is up when its variable ‘oUp’ has value ‘True’
- The centre sensor of the LeftConveyor is triggered when the variable ‘iCentreSensor’ of the LeftConveyor has value ‘True’.

This property is used to verify a trace of the MerryGoRound machine simulation. Results of the verification are shown to the participants using the visualisation tool. Participants are asked to describe the results in their own words.

11.2.3 Results

Phase/event understandability

All participants understood the notions of phase and event propositions with relative ease. The concept of changing from a phase proposition to an event proposition using the start/end of keywords in our language was clear to them. One participant noted that in some cases, he would like to use phase propositions in properties where event propositions are required. He reckons a phase could be provided where the start of the phase would be deemed the required event. He noted, however, that this would make the language less clear. Finally, all participants agreed that domain knowledge is required to be able to determine if a proposition should be categorised as a phase or an event.
Pattern/scope understandability

With some practice, all participants understood the discussed patterns and scopes. The participants mentioned that as long as sufficient information is provided, the patterns and scopes are clear. Comprehensive information is primarily needed when trying to understand the detailed semantics of a pattern. For example, the Bounded Recurrence pattern uses an open interval and not a closed one. All participants mentioned that a learning curve would be present but that this curve would not be steep. Participants would also like to see additional examples that show, for a set of common MCA requirements, what patterns and scopes should be used to formalise them.

Language understandability

All participants agree that people who have knowledge of the MCA domain but have no knowledge of patterns and scopes can understand the rules created in the language just by reading them. The participants mentioned the rules require no knowledge of the language syntax as they are specified in regular English. Such rules do not convey the intricate semantics of the patterns, but do convey general requirements. All participants noted, however, that for the rules to be clear, the proposition names should be clear as well (e.g. liftIsUp instead of prop1). This is the case as these names form an important part of the rule sentence.

Language modifiability & Language creation

The only problems when modifying rulesets arose when participants tried to understand the exercise itself. They needed some time to figure what to change to make the ruleset comply with the request of the exercise. When it was clear that the scope should be changed, the change was easily applied to the ruleset. The same holds for the each instance of keywords, where participants first needed to familiarise themselves with the keywords’ semantics. Participants had no problems in identifying the components and propositions needed. Overall, two of the participants deemed the language user-friendly. One participant deemed it not to be, which he blamed on the fact that there is little interaction between the user and the language editor. In a future version, he would like to see an interactive editor that helps the user construct a ruleset in a step-by-step manner (e.g. first choosing a pattern, then a scope, etcetera).

Result understandability

All participants understood the four verdict types the Monitoring System produces and had no trouble with understanding the results of a verification displayed in the visualisation tool.

General remarks

Participants were asked what skills and competencies users should have when writing rulesets in our language. The most prominent answer was that users should be able to think in an abstract way that makes mapping requirements to patterns and scopes possible. All participants agree that our verification toolset improves upon the current verification process (visually inspecting an MCA simulation). They mention that our toolset can observe much more of what is going on during the execution of an MCA. Several participants noted that this approach removes much uncertainty, which could give MCA customers more certainty that the MCA actually complies with the requirements. One participant noted that this way of verification is also much easier to automate.

All participants think that the toolset is definitely usable in practice. Our contact at Cordis mentions, however, that for the language to be usable, most of the requirements should be expressible using the supported patterns and scopes. Which he thinks is the case in general.
11.3 Evaluation Maintainer Role

This section details the procedure and results of an evaluation performed with people who fulfil the maintainer role of our verification toolset.

11.3.1 Goal

This evaluation uses the Monitoring System and the interfaces defined in our toolset and focuses on the following:

- **Tool addition**: Evaluate the feasibility and difficulty of connecting a new tool to our verification toolset.
- **Adding support for new patterns/scopes**: Evaluate the feasibility and difficulty of adding support for a new pattern or scope to our toolset.

11.3.2 Setup

Adding support for a new pattern or scope is considered too time-consuming for participants of this evaluation to perform. The same holds for the development of a new visualisation or property language tool. Therefore, this evaluation focuses on presenting the concepts and steps that are required when performing such actions, and thereby engaging in discussions with the participants. The evaluation goals are answered using the information gained during these discussions.

Participants

This evaluation is carried out using two TNO employees. These participants both have programming and pattern system knowledge. Both also have experience with timed automata and verification systems in general. This makes them ideal candidates to fulfil the maintainer role of our verification toolset.

11.3.3 Results

**Tool addition**

The participants think that adding a new Trace Adapter or Result Visualisation is a relatively straightforward process. If such new tool supports the required interface, it does not matter how it is constructed. Participants added, however, that besides the data format, all other requirements of the interface should be clearly provided to maintainers. For example, the exact time format used in the data formats. The participants also mentioned that data of an MCA should be extractable if a new Trace Adapter is to be easily created. This fact is outside the scope of our toolset. In general, the participants deemed the addition of a new Property Language the most complex task. If such language should be based upon another formalism than patterns and scopes (e.g. LTL/MLT), the translation between these formalisms could be difficult. Finally, participants reckon that skills of timed automata, pattern systems, DSLs and regular programming, are required to extend our toolset with new tools. More skills could be required if other formalisms come into play.

**Adding support for new patterns/scopes**

Adding support for a new pattern consists of creating one or multiple new property monitors in the Monitoring System (one for each scope the pattern is connected to), adding support to the Property Language and the interface between the two tools. The participants think that this process is definitely feasible if the maintainer has knowledge of timed automata, pattern systems, DSLs and programming. The participants do mention that formalising a new pattern, using, for example, an
automaton, is the most challenging step. Implementing this automaton in a property monitor is
considered a straightforward process by the participants, as examples can be taken from already
implemented monitors.

11.4 Conclusion

In this chapter, our verification toolset has been evaluated. The evaluation focused both on the user
and maintainer roles with the goal of evaluating our toolset’s ease of use and extendability, respecti-
vely.

As can be concluded from the results, the property language and visualisation of our toolset seem
easy to use. After some practice, participants were able to apply the notions of patterns and scopes
to create rulesets. Participants also agreed that people who have no knowledge of the patterns and
scopes but do know the domain, should be able to understand the rules. This remark indicates that
the based on domain requirement could be met. As the number of participants of this evaluation is
small, a more extensive user study would be required to properly confirm these statements.

Participants reckoned that with the right knowledge, extending out toolset with new tools or
adding support for new patterns and scopes is relatively straightforward. The hardest part of this
process was deemed to be the formalisation of such new pattern or scope. Because of this, we conclude
that the extendability requirement could be met. As the number of participants of this evaluation
is small, and no actual development has been performed during the evaluation, a more extensive
maintainer study would be required to properly confirm this statement.

Our contributions detailed in this chapter are:

- Results indicating that, according to a small set of people, our language and visualisation
  conform to the ease of use and based on domain requirements.
- Results indicating that, according to a small set of people, our toolset conforms to the extend-
  ability requirement.

Using all information in this chapter, research question RQ8, Does the verification method im-
prove upon the current verification process? can be answered in the following way: According to the
participants of our evaluation it does, but a more extensive study is required to properly confirm this
statement.
Chapter 12

Conclusion & Future Work

This chapter is the closing chapter of this thesis. Section 12.1 provides a conclusion in which the main research question is answered. In Section 12.2 recommendations of future improvements to our verification toolset are described. Section 12.3 provides a short reflection on this research.

12.1 Conclusion

This section details the answers to all research questions. Here, it is also concluded if our verification toolset conforms to the four requirements detailed in Chapter 1.

Our solution supports the verification of machine control applications (MCAs) using properties. We defined a set of properties that represent what MCA properties should be specifiable. This set include safety, reliability and performance properties. The set, see Appendix B, is the answer to research question RQ1, *What properties should be specifiable?* By using several new patterns in combination with existing patterns and scopes, all properties could be formalised, see Chapter 5 for more details.

To make sure our solution supports many types of users with many different needs, we designed it as a modular toolset. The toolset consist of multiple independent tools (*Property Language*, *Monitoring System*, *TraceAdapter* and *Result Visualisation*) that each perform one aspect of the verification process. The tools communicate with each other using clearly defined interfaces. This allows the tools to be replaced by alternatives if desired. See Chapter 6 for more details. The usage of independent tools which are connected using clearly defined interfaces is the answer to research question RQ5 *How can the modularity of the verification solution be guaranteed?*

The toolset employs the technique of runtime verification. We deemed this technique the most suitable verification approach as it can be used to observe MCAs without the need to consider system input beforehand. See Chapter 3 for more details. The verification itself is performed by the *Monitoring System* which uses several monitors. These monitors are comprised of timed automata that observe MCA traces. To perform effective monitoring, the monitors can report multiple violations of a property and can handle finite traces. See Chapter 8 for more details. This answers research question RQ6 *What is a suitable technique to verify machine control application data against properties?*

Using our toolset, an MCA can be validated by only supplying the execution traces of such MCA. However, all values of the MCA attributes (e.g. variables), which are to be observed by property monitors, should be provided at the start of the monitoring. After that, only changes in values should be provided. During the entire monitoring, these values should be of string or number type. See Chapter 10 for more details. These rules are the answer to research question RQ4 *What machine control application data is needed for verification using properties?*
To make a verification solution user-friendly, the components that users interact with should be easy to use. One such component is the property specification language. Research question RQ2 *How can a property language be structured for user-friendliness?* can be answered in the following way: By allowing properties to be defined in well-segmented *rulesets*, where short sentence-like rules are used that use as little pattern system terminology as possible. Research question RQ3 *What abstraction level should be used when specifying properties?* can be answered as follows: A basic level of abstraction when defining properties, and an additional optional level of abstraction, provided by abstraction classes, when defining often used sets of properties.

The other component of the toolset that users interact with is the feedback of verification results. We explored what data users want to see of a performed verification. Research question RQ7 *What data is required to understand and process the verification results?* can be answered as follows: Data which provides an overview of all verification results and that includes detailed information regarding internal MCA data, proposition and property results over time. This data can be, among other things, used to compare expected and actual behaviour. See Chapter 9 for more details.

Our toolset has been evaluated to determine if it conforms to the requirements defined in Chapter 1, see Chapter 11. The evaluation results show that the toolset is considered *easy to use* for the targeted users and that the *based on domain* requirement is met. The toolset is also deemed *extendable* with new tools and support for new patterns and scopes. The fact that the toolset is extendable, combined with the fact that few restrictions are present on what MCA data can be verified, we conclude that the toolset also conforms to the *generic* requirement. Finally, all evaluators think our toolset improves upon the current verification process employed by Cordis SUITE. This answers research question RQ8 *Does the verification method improve upon the current verification process?* As the evaluation was carried out using five participants, a more extensive evaluation study would be required to properly confirm the statements above.

Using all this information, the main research question *How to define and verify properties of machine control applications in a user-friendly and generic way?* can be answered as follows: Using a modular toolset that allows additional tools to be added, where properties, formalised using patterns and scopes, can be defined using sentence-like rules, and the validation is performed using a runtime approach where monitors observe the execution traces of the MCAs.

Below are the main contributions of our research. See the last section of every chapter of this thesis for a more detailed view of our contributions.

- A modular approach to the runtime verification of a wide variety of MCAs that supports the introduction of additional property languages and visualisation tools.
- A user-friendly property specification language aimed at users in the machine control application domain. This language contains additional functionality, not seen in other languages, aimed at supporting all the properties of the *property set*. The language also introduces an extra abstraction layer in the form of abstraction classes.
- New monitors, in the form of timed automata, to support most of the properties of the *property set*. These automata have the functionality to cope with finite traces and the detection of multiple compliances or violations of a property.
- A list of six feedback approaches that incorporate data that users want to receive of an MCA verification.
12.2 Future Work

In this section, the recommendations for future improvements to our toolset are outlined. This coincides with what this research did not achieve.

Involvement of industry

Because of events out of our control (the COVID-19 pandemic), companies who design and use machine control applications (our intended users) could not be involved in our research. This resulted in a lack of real-world use cases and user information, which made designing a fitting verification solution challenging. Because of this, we recommend that such companies are consulted when the pandemic no longer prohibits this. This way, support for additional properties can be added to the toolset. The companies should also be included in a large-scale user study to properly confirm the results of our evaluation.

Verification Result Feedback

In Chapter 9, we defined several feedback approaches. We recommend that research is performed on extending and implementing these approaches, resulting in a tool that can automatically determine what MCA data is relevant to a given property verdict. This way, a user only receives the data necessary to perform effective and efficient debugging of a verdict. Such a tool could make our toolset more valuable to the industry.

We also recommend extending the Monitoring System automata in such a way that data of violated timing properties, used for the comparison between actual and expected behaviour, can be retrieved. To retrieve this data, property monitors need to be able to observe events after a reset (e.g. after a timer expired).

This research did not focus on providing feedback to the machine control application under verification. Functionality to relay property verdict data back to an MCA could be used to solve problems (semi-)automatically. We consider this a valuable addition to our toolset and, therefore, recommend that it is added in the future.

Properties

Our verification toolset supports most of the properties of the property set defined in Chapter 5. As mentioned in Chapter 8, six pattern-scope combinations that should be usable with both phase and event propositions, are currently only usable with event propositions. The design of the verification toolset allows phase proposition support for these six combinations to be easily added. Therefore, we recommend that support for these propositions is added in the future. This would make our toolset support all properties in the property set.

At the end of this research, new ideas arose of possible MCA requirements not included in the property set. These are requirements like ‘The speed of motor1 should always at least be double the speed of motor2’. Supporting such requirements requires supporting arithmetic expressions. If deemed useful by the industry, we recommend that support for such properties is added to our toolset.

Our contacts at Cordis Automation, together with one participant of the evaluation, have mentioned that discovering what properties should be defined of a system can be challenging. Although this was not part of our research, it is an essential aspect of the verification process. Therefore, we recommend that future research is performed on discovering methods that help users in discovering MCA properties.
Scalability

This research did not focus on the scalability aspect of our verification toolset. From a few experiments, the execution time of our Monitoring System implementation currently increases linearly with the number of properties present. A logged trace with around 70,000 events of a small MCA (2 minutes of data, 450 attributes) can be verified using 100 properties and 200 propositions in 1 second. The same events can be verified using 1,000 properties with 2,000 propositions in 9 seconds. This is on an Intel dual-core laptop with 8GB of RAM. Future research should be performed to optimise the toolset for the use with large industrial systems.

Monitor Robustness

In this research, we have attempted to verify the automata used in the Monitoring System. However, new errors could be introduced when these automata are translated to Java property monitor classes. An approach which automatically translates automata to such classes could be beneficial. We also recommend properly verifying the Bounded/Strict Response automata using our VA method.

Property Language

In [69], Smith et al. detail some advanced approaches to the specification of properties. These include pattern template options, which can, for example, be used to deviate from the standard semantics of a pattern. For the response pattern, one could define that multiple actions should be followed by multiple responses, instead of just one response. Such definitions could make the detailed semantics of a property clear. As these details currently cannot be retrieved by reading a ruleset, such definitions could be a useful addition to our language.

We also recommend further developing the idea of applying abstraction in a property language in the form of abstraction classes, as we think more benefits can be gained from this.

Although the industry was not officially involved in this research, one short meeting with a contact of Additive Industries was held. Here, our finished language was briefly shown. The contact mentioned that to increase the usability of our language, functionality should be added to provide a better overview of all the rulesets used. Such functionality could, for example, show which components are used in which rulesets. We recommend that such functionality is explored and added in the future.

Integration with MCA design tools

In this research, the customers of Cordis Automation have been used as a projection of the users of our toolset. For these customers, it could be beneficial if the toolset was integrated into Cordis SUITE. Such an approach would allow for tight integration between Cordis SUITE, the property language and the result visualisation.

12.3 Reflection

This research project started with the goal of designing a tool that suited the verification needs of companies in the MCA domain. Therefore, the idea was to heavily involve Additive Industries and Lely, who are both members of the Machinaide project, into our research. Because of the COVID-19 pandemic, this idea could unfortunately not become a reality. As the national kick-off of the Machinaide project was delayed several times, and many companies shifted into survival mode, opportunities to involve the industry into our research slowly faded away. This hampered us from using a real industrial case in our research. To cope with these events, the project goal was adjusted, and our contacts at Cordis Automation became the primary source of industry information. The new situation resulted in the need for a toolset that allows new tools to be added. This way, companies like Additive Industries and Lely, could modify the toolset, such that it suits their needs.
Bibliography


76


[51] Li, Z., Han, J., and Jin, Y. Pattern-based specification and validation of web services interaction properties. In International Conference on Service-Oriented Computing (2005), Springer, pp. 73–86.


MCA component communication

Communication between components can be achieved in two ways: by using commands, or by using inputSignals/outputSignals. Commands are events which are broadcast by a component down the tree, to whichever other components are ‘subscribed’ to the command. A command can contain parameters. InputSignals and outputSignals are used as setters and getters of component variables. An inputSignal is used to write the value of another component’s variable. The outputSignal is used to read it. Components can only communicate (using the methods above) with components lower in the tree. This access is transitive, meaning children, grandchildren etcetera are all accessible. Sibling and parent components are not accessible. This means the motor component from Figure 2.2 cannot, for example, send its speed value to the conveyor. The conveyor has to retrieve this value from the motor using an outputSignal.

State machine layout

The state machines used in Cordis Suite are UML state machines with some additional functionality. The state machines consist of one initial and multiple normal states, with directional edges between them, see Figure A.1. A state machine can have a pre-state and a post-state, which are not connected to any other states. A user can attach actions to a state. An action contains code which is executed in one of two scenarios: Only once, whenever the state is entered, or every cycle when the state is active. Actions in these scenarios are specified under entry and do respectively. See Figure A.1 where the state WarmUp contains a entry action. The pre and post-state can only contain do actions. An action can, for example, broadcast a command.
Guards can be placed on edges to prevent automatic transitioning from one state to another. These guards consist of one of the conditions seen below. When a guard condition resolves to true, the transition is taken.

- A check if a command has been received.
- A check if a component instance has a certain global status.
- A predicate regarding variable values of a component instance.

In Cordis SUITE a state can also contain a state machine internally, thereby adding a layer of hierarchy. Such a state is called a composite state (see state Stopped in Figure A.1). This internal state machine is a regular state machine only without a pre or post-state. Whenever a composite state becomes active, the internal state machine also becomes active. Subsequent state transitions now take place in the internal state machine, wherein the top-level state machine the composite state remains active. An internal state machine includes edges to exit points. These exit points have edges back to a state in the top-level state machine. Whenever such exit point is reached, the execution of the internal state machine stops.

Internally, during the code generation, state machines are transformed into large if-then-else statements.

State machine execution
As stated in Section 2.1.1, the second step in the PLC scan cycle is the program execution. During this step, all state machines of all components are executed, starting with the root of the component tree. Child components are executed in a depth-first manner. Multiple instances of the same component are executed in alphabetical order by name. Multiple state machines of a component instance are executed based on their ids in numerical order. Such an id is assigned to a state machine when it is first created in Cordis SUITE. The execution of a component instance occurs in the steps seen below. Here, step 2c and 2d are only executed if the guard condition from step 2b resolved to true, and because of that, a transition to the next state is made.

1. Execute the actions of all pre-states.
2. Execute all state machines, that is, for every state machine:
   (a) Execute the do actions of the state currently active.
   (b) Evaluate the edge guard condition (if true, transition to the next state).
   (c) Execute entry actions of the newly active state.
   (d) Execute do actions of the newly active state.
3. Execute the actions of all post-states.

All state machines start in the initial state. As no guard can be placed on the outgoing edge of this state, a transition to the first non-initial state is automatically made. Per PLC cycle, only one transition can be made within a state machine.

Broadcast commands are received in the same PLC cycle they are sent. This is possible as commands can only be received by components which are further down the component tree, and these components are executed later in the cycle. Confirmation that a command reached its subscribers is not communicated back to the sender. InputSignals also take effect in the same cycle, as the value of a variable is set instantly. As multiple components can set the value of a component variable, this value can change multiple times per cycle.
Appendix B

MCA Requirements

This appendix details the machine control application requirements collected during this research. Similar requirements have been combined into one. Table B.1 and Table B.2 show the requirements, whereas Table B.3 and Table B.4 show these requirements as properties.

B.1 Requirements

<table>
<thead>
<tr>
<th>Id</th>
<th>Requirement</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The system makes sure that no unsafe items are ever moved onto the safe conveyor.</td>
<td>Safety</td>
</tr>
<tr>
<td>2</td>
<td>When an error occurs, an error lamp starts blinking.</td>
<td>Safety</td>
</tr>
<tr>
<td>3</td>
<td>If an error is resolved by removing the error cause, the system will resume normal operation after a reset button is pressed.</td>
<td>Reliability</td>
</tr>
<tr>
<td>4</td>
<td>By pressing an emergency stop button, all conveyors will stop.</td>
<td>Reliability</td>
</tr>
<tr>
<td>5</td>
<td>In the oven, an item should not be exposed to temperatures hotter than 800 degrees for more than 20 seconds.</td>
<td>Safety</td>
</tr>
<tr>
<td>6</td>
<td>While the module is on, the sensor should not be in calibration mode for more than 2 minutes in total.</td>
<td>Performance</td>
</tr>
<tr>
<td>7</td>
<td>A green light goes on if there is an item available and no animal is detected.</td>
<td>Reliability</td>
</tr>
<tr>
<td>8</td>
<td>After a configurable amount of time of detection of an animal, the green light goes off.</td>
<td>Reliability</td>
</tr>
<tr>
<td>9</td>
<td>If an item is present on the first conveyor, it will be sent to the second conveyor if that has room for an item.</td>
<td>Reliability</td>
</tr>
<tr>
<td>10</td>
<td>When an item is no longer detected on the conveyor, the conveyor spins one revolution before stopping. It also stops when an item is detected by the sensor.</td>
<td>Reliability</td>
</tr>
<tr>
<td>11</td>
<td>Once an item leaves the conveyor, a new item will be delivered in no longer than 6 seconds.</td>
<td>Performance</td>
</tr>
<tr>
<td>12</td>
<td>The conveyor will start running when an item is detected on it.</td>
<td>Reliability</td>
</tr>
<tr>
<td>13</td>
<td>The conveyor always spins unless it is full with items.</td>
<td>Safety</td>
</tr>
<tr>
<td>14</td>
<td>No more than one item may be present on a conveyor.</td>
<td>Safety</td>
</tr>
<tr>
<td>15</td>
<td>When a module is running its motor may be stopped at most twice per hour.</td>
<td>Safety</td>
</tr>
<tr>
<td>16</td>
<td>Pressing the pause button stops the conveyor and turns on the blue light.</td>
<td>Reliability</td>
</tr>
<tr>
<td>17</td>
<td>Pressing the pause button again resumes the conveyor and turns off the blue light.</td>
<td>Reliability</td>
</tr>
<tr>
<td>18</td>
<td>A minimal gap of 8 cm has to be kept between items which are moving over the conveyor.</td>
<td>Safety</td>
</tr>
<tr>
<td>19</td>
<td>A ping from the heater should be received always after at most 5 seconds from the last received after the last ping, to confirm it is in safe operating conditions</td>
<td>Reliability</td>
</tr>
<tr>
<td>20</td>
<td>The conveyor keeps spinning whenever possible.</td>
<td>Reliability</td>
</tr>
<tr>
<td>21</td>
<td>After a configurable timeout, the sorter moves the item to the unsafe conveyor.</td>
<td>Reliability</td>
</tr>
<tr>
<td>22</td>
<td>The sorter sends safe items to the safe bin and unsafe items to the unsafe bin.</td>
<td>Reliability</td>
</tr>
<tr>
<td>23</td>
<td>When the safety sensor is triggered, the gantry stops immediately. It continues when the sensor is cleared.</td>
<td>Safety</td>
</tr>
<tr>
<td>24</td>
<td>To preserve a motor unit, it should not run at more than 95 percent of its power for more than 30 seconds.</td>
<td>Safety</td>
</tr>
<tr>
<td>25</td>
<td>When the proximity sensor at the end triggers, one of the two lamps should turn on but not both.</td>
<td>Safety</td>
</tr>
</tbody>
</table>

Table B.1: Collected requirements
<table>
<thead>
<tr>
<th>Id</th>
<th>Requirement</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>When the sensor is triggered for a configurable amount of time, the module goes into an error until the sensor is no longer triggered.</td>
<td>Safety</td>
</tr>
<tr>
<td>27</td>
<td>The conveyor starts spinning when the sensor above it is clear and sensor below is triggered.</td>
<td>Reliability</td>
</tr>
<tr>
<td>28</td>
<td>When an item is blocked during unloading, the system will go into an error after a timeout.</td>
<td>Safety</td>
</tr>
<tr>
<td>29</td>
<td>Items are detected using three sensors. As soon as one of them is triggered, the module should enter an error status.</td>
<td>Reliability</td>
</tr>
<tr>
<td>30</td>
<td>There should be a gap of at least 5 second between items entering a module.</td>
<td>Safety</td>
</tr>
<tr>
<td>31</td>
<td>Switching speed can only be done by first stopping the system, changing the actual speed to the desired speed and then starting the system again.</td>
<td>Safety</td>
</tr>
<tr>
<td>32</td>
<td>When the lamp turns on, it should be on for at least 8 seconds.</td>
<td>Reliability</td>
</tr>
<tr>
<td>33</td>
<td>The number of sensor events must be at least 100 every hour.</td>
<td>Performance</td>
</tr>
<tr>
<td>34</td>
<td>When the conveyor is full, detected by a sensor, no additional items will be loaded from the previous conveyor.</td>
<td>Safety</td>
</tr>
<tr>
<td>35</td>
<td>A lift should not receive multiple up commands within 3 seconds of each other.</td>
<td>Reliability</td>
</tr>
<tr>
<td>36</td>
<td>If the sorter sensors are triggered first, then the rollers start spinning. They don’t stop when the bin becomes full.</td>
<td>Safety</td>
</tr>
<tr>
<td>37</td>
<td>Items are detected using three sensors. As soon as one of them is triggered, the module assumes an item is present.</td>
<td>Reliability</td>
</tr>
<tr>
<td>38</td>
<td>While the module is in an error, once the error light is on, it should be on for at most 1050 milliseconds.</td>
<td>Reliability</td>
</tr>
<tr>
<td>39</td>
<td>While the module is in an error, once the error light is on, it should be on for at least 950 milliseconds.</td>
<td>Reliability</td>
</tr>
<tr>
<td>40</td>
<td>Motor1 should always rotate slower than motor2.</td>
<td>Reliability</td>
</tr>
<tr>
<td>41</td>
<td>All stoppers retract during initialisation.</td>
<td>Safety</td>
</tr>
<tr>
<td>42</td>
<td>Always when the pump is running, an orange light should be on serving as a warning signal.</td>
<td>Reliability</td>
</tr>
<tr>
<td>43</td>
<td>An animal should pass the gate sensor at least every 30 seconds.</td>
<td>Performance</td>
</tr>
<tr>
<td>44</td>
<td>During every print, the mould should be cleaned by opening an air pressure valve at least three times.</td>
<td>Safety</td>
</tr>
<tr>
<td>45</td>
<td>When an item is in the cleaner module, it should be exposed to UV light for at least 40 seconds in total.</td>
<td>Reliability</td>
</tr>
</tbody>
</table>

*Table B.2: Collected requirements (continued)*
## B.2 Properties

<table>
<thead>
<tr>
<th>Id</th>
<th>Property</th>
<th>Pattern</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>While checker variable ‘result’ is ‘unsafe’, conveyor never in ‘moveLeft’ state.</td>
<td>Absence</td>
<td>After Until</td>
</tr>
<tr>
<td>2</td>
<td>When module enters global status ‘error’, next error lamp in ‘blinking’ state.</td>
<td>Strict Response</td>
<td>Globally</td>
</tr>
<tr>
<td>3</td>
<td>When module in ‘error’ state &amp; button in ‘pressed’ state, all components should be in ‘stopped’ state after 1 second.</td>
<td>Bounded Response</td>
<td>Globally</td>
</tr>
<tr>
<td>4</td>
<td>When emergency button enters ‘pressed’ state, next all conveyors not in ‘stopped’ state.</td>
<td>Strict Response</td>
<td>Globally</td>
</tr>
<tr>
<td>5</td>
<td>The oven should not be in the ‘occupied’ state while its variable ‘temperature’ is higher than 800, for more than 20 seconds.</td>
<td>Maximum Duration</td>
<td>Globally</td>
</tr>
<tr>
<td>6</td>
<td>Between module entering ‘on’ state until module no longer in ‘on’ state, sensor not in ‘calibrating’ state for more than 2 minutes</td>
<td>Time-constrained Absence</td>
<td>After Until</td>
</tr>
<tr>
<td>7</td>
<td>When sensor item is in ‘on’ state &amp; all sensors are in ‘off’ state, next green light enters ‘on’ state.</td>
<td>Strict Response</td>
<td>Globally</td>
</tr>
<tr>
<td>8</td>
<td>When any sensors for 3 seconds, always green light in ‘off’ state within 1 second.</td>
<td>Bounded Duration Response</td>
<td>Globally</td>
</tr>
<tr>
<td>9</td>
<td>When sensor 2 begins in state ‘off’ state &amp; sensor 1 end in ‘on’ state, next conveyor 1 should be in ‘running’ state.</td>
<td>Strict Response</td>
<td>Globally</td>
</tr>
<tr>
<td>10</td>
<td>After a sensor enters the ‘off’ state, the conveyor should always be in ‘off’ state after at least 10 seconds, until sensor enters the ‘on’ state.</td>
<td>Bounded Absence</td>
<td>After Until</td>
</tr>
<tr>
<td>11</td>
<td>After sensor end enters the ‘off’ state, sensor begin should enter the ‘on’ state within 6 seconds.</td>
<td>Bounded Response</td>
<td>Globally</td>
</tr>
<tr>
<td>12</td>
<td>Always conveyor in ‘running’ state, unless sensor end in ‘on’ state &amp; sensor start in ‘on’ state.</td>
<td>Strict Response</td>
<td>Globally</td>
</tr>
<tr>
<td>13</td>
<td>After sensor begin entering the ‘on’ state until sensor end entering the ‘on’ state, sensor begin can never enter the ‘on’ state.</td>
<td>Absence</td>
<td>After Until</td>
</tr>
<tr>
<td>14</td>
<td>Between module entering ‘running’ state until module no longer in ‘running’ state, motor not in ‘on’ state at most two times</td>
<td>Duration Existence</td>
<td>After Until</td>
</tr>
<tr>
<td>15</td>
<td>When pause button variable ‘pressed’ turns true, next blue light should be in the ‘on’ state &amp; next conveyor should be in the ‘stopped’ state.</td>
<td>Strict Response</td>
<td>Globally</td>
</tr>
<tr>
<td>16</td>
<td>The module should receive the ‘ping’ command always at least 5 seconds after the last ‘ping’ command was received.</td>
<td>Bounded Recurrence</td>
<td>Globally</td>
</tr>
<tr>
<td>17</td>
<td>Conveyor should never be in the ‘stopped’ state.</td>
<td>Absence</td>
<td>Globally</td>
</tr>
<tr>
<td>18</td>
<td>When sensor sorter enters the ‘on’ state &amp; module variable ‘safe’ is ‘unknown’ for 5 seconds, the sorter should be in ‘moveRight’ state after 1 second.</td>
<td>Prefaced Bounded Response</td>
<td>Globally</td>
</tr>
<tr>
<td>19</td>
<td>When sensor sorter enters the ‘on’ state &amp; module variable ‘safe’ is ‘true’, next the sorter should be in the ‘moveLeft’ state.</td>
<td>Strict Response</td>
<td>Globally</td>
</tr>
<tr>
<td>20</td>
<td>The module should receive the ‘ping’ command always at least 5 seconds after the last ‘ping’ command was received.</td>
<td>Bounded Recurrence</td>
<td>Globally</td>
</tr>
<tr>
<td>21</td>
<td>Motor unit should have its variable power more than 95 for only 30 seconds in total.</td>
<td>Time-constrained Absence</td>
<td>Globally</td>
</tr>
<tr>
<td>22</td>
<td>When proximity sensor variable distance is less than 10, next lamp 1 or lamp 2 should enter the ‘on’ state, but not both.</td>
<td>Strict Response</td>
<td>Globally</td>
</tr>
<tr>
<td>23</td>
<td>When sensor prox enters the ‘on’ state, next conveyor should be in ‘stopped’ state.</td>
<td>Prefaced Bounded Response</td>
<td>Globally</td>
</tr>
<tr>
<td>24</td>
<td>When sensor prox enters the ‘on’ state &amp; sensor prox in ‘on’ state for at least 5 seconds, system should be in ‘error’ state in 10 seconds.</td>
<td>Prefaced Bounded Response</td>
<td>Globally</td>
</tr>
</tbody>
</table>

Table B.3: Pattern and scope mapping
<table>
<thead>
<tr>
<th>Id</th>
<th>Property</th>
<th>Pattern</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>After any sensor enters ‘on’ state, next module enters global status</td>
<td>Strict Response</td>
<td>Globally</td>
</tr>
<tr>
<td></td>
<td>‘error’.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Sensor should be in state ‘on’ at most every 5 seconds</td>
<td>Restricted</td>
<td>Globally</td>
</tr>
<tr>
<td></td>
<td></td>
<td>recurrence</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>After module variable ‘speed’ = x, until module variable ‘speed’ = y,</td>
<td>Bounded Existence</td>
<td>After Until</td>
</tr>
<tr>
<td></td>
<td>eventually conveyor in ‘stopped’ state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>When lamp enters ‘on’ state, it should remain for at least 8 seconds.</td>
<td>Minimum Duration</td>
<td>After Until</td>
</tr>
<tr>
<td>33</td>
<td>Sensor variable ‘detected’ becomes true more or equal than 100 times per hour.</td>
<td>Duration Bounded Existence</td>
<td>Globally</td>
</tr>
<tr>
<td>34</td>
<td>Never together sensor2End in ‘on’ state &amp; sensor2Begin in ‘on’ state</td>
<td>Absence</td>
<td>Globally</td>
</tr>
<tr>
<td></td>
<td>&amp; sensor1End in ‘on’ state &amp; conveyor1 in ‘running’ state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>A lift should not receive the ‘up’ command within 3 seconds of having</td>
<td>Restricted</td>
<td>Global</td>
</tr>
<tr>
<td></td>
<td>received that same command.</td>
<td>recurrence</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>Never together sensorEnd in ‘on’ state &amp; sensorBegin in ‘on’ state &amp;</td>
<td>Absence</td>
<td>Globally</td>
</tr>
<tr>
<td></td>
<td>conveyor in ‘running’ state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>When of any sensor the variable ‘detected’ becomes true, next variable</td>
<td>Strict Response</td>
<td>Globally</td>
</tr>
<tr>
<td></td>
<td>‘itemDetected’ is true.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>Between module entering ‘error’ state until module no longer in ‘error’</td>
<td>Maximum Duration</td>
<td>After Until</td>
</tr>
<tr>
<td></td>
<td>state, once errorLight enters its ‘on’ state it can be in this state for</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>at most 1050 milliseconds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>Between module entering ‘error’ state until module no longer in ‘error’</td>
<td>Minimum Duration</td>
<td>After Until</td>
</tr>
<tr>
<td></td>
<td>state, once errorLight enters its ‘on’ state it must in this state for</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>at least 950 milliseconds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>Motor1 variable ‘speed’ should always be less than motor2 variable</td>
<td>Universality</td>
<td>Globally</td>
</tr>
<tr>
<td></td>
<td>‘speed’</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>When module enters ‘init’ state, next all stoppers enter ‘retracted’</td>
<td>Strict Response</td>
<td>Globally</td>
</tr>
<tr>
<td></td>
<td>state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>While pump is in the ‘running’ state, orangeLamp should always be in</td>
<td>Universality</td>
<td>Globally</td>
</tr>
<tr>
<td></td>
<td>the ‘on’ state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>Gate sensor should be in the ‘on’ state within 30 seconds of also being</td>
<td>Bounded Recurrence</td>
<td>Globally</td>
</tr>
<tr>
<td></td>
<td>in the ‘on’ state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>After sensorBegin enters the ‘off’ state until sensorEnd enters the</td>
<td>Strict Response</td>
<td>After Until</td>
</tr>
<tr>
<td></td>
<td>‘off’ state, when sensorBegin enters the ‘on’ state, next the conveyor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>should be in ‘stopped’ state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>While printer in ‘itemInside’ state, the air valve variable ‘open’</td>
<td>Bounded Existence</td>
<td>After Until</td>
</tr>
<tr>
<td></td>
<td>should be ‘true’ at least three separate times.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>While cleanerModule in ‘itemInside’ state, the UVlight should be in</td>
<td>Time-constrained</td>
<td>After Until</td>
</tr>
<tr>
<td></td>
<td>the ‘on’ state for at least 40 seconds in total.</td>
<td>Universality</td>
<td></td>
</tr>
</tbody>
</table>

**Table B.4:** Pattern and scope mapping (continued)
Appendix C

Language Experiment Examples

Example 1

When the cooler is turned on, the temperature of the item to be cooled should be less than 100 degrees after 6 minutes.

**Proposition**
- coolerOn : cooler.state = 'on'
- cooledDown : cooler.temperature < 100

// Requirement DSL
Globally Response cooledDown after coolerOn within 6 minutes

// Structured English Grammar
Globally it is always the case that if coolerOn holds, then cooledDown holds after at most 6 minutes

Example 2

No more than one item can be present in a module.

**Proposition**
- itemEntering : sensorEntry.state = 'on'
- itemExiting : sensorExit.state = 'on'

// Requirement DSL
After itemEntering until itemExiting absence itemEntering

// Structured English Grammar
After itemEntering until itemExiting, it is never the case that itemEntering holds
Example 3

A maximum gap of at most 2 seconds is kept between triggering a warning light whenever a system is in an error mode.

| Proposition lightOn : warningLight.state = 'on' |
| Proposition inError : module.status = 'error' |
| Proposition notInError : module.status != 'error' |

// Requirement DSL
After inError until notInError Recurrence lightOn every 2 seconds

// Structured English Grammar
After inError until notInError, it is always the case that lightOn holds at least every 2 seconds
Appendix D

Language Grammar

See below for the grammar of our property language. The INT keyword denotes an integer value, whereas the ID keyword denotes a string value. The cardinalities zero or one, zero or more and one or more are expressed by the operators ?, * and +, respectively. The cardinality exactly one is expressed by not including an operator.

```
Language
Language := RuleSet+
RuleSet := 'RuleSet:' ID '{' Definition? Components Propositions Rules '}'

Definition
Definition := 'Definition:' ID

Components
Components := 'Components:' Component+
Component := 'Each instance of' ID 'as' ID
| 'Instance' ID 'as' ID
| 'All instances of' ID 'as' ID

Propositions
Propositions := 'Propositions:' Proposition+
Proposition := Phase
| Event

Event := 'Event' ID ':' EventExp
EventExp := CompareExp
| ComboExp
| FlankExp
| ChangeExp
| AnyForExp
| AnyExp

Phase := 'Phase' id ':' PhaseExp
PhaseExp := CompareExp
| ComboExp
| AnyForExp
| AllForExp
| AnyExp
| AllExp
```
Expressions

CompareExp ::= ID CompareExp'
CompareExp' ::= '==' ID
| '!=' ID
| '==' INT
| '!=' INT
| '<' INT
| '<=' INT
| '>' INT
| '>=' INT

FlankExp ::= 'start of' ID
| 'end of' ID

ChangeExp ::= 'change' ID

ComboExp ::= ComboExp2 ComboExp'*
ComboExp' ::= 'or' ComboExp2 ComboExp'*
ComboExp2 ::= ComboExp3 ComboExp2'*
ComboExp2' ::= 'and' ComboExp3 ComboExp2'*
ComboExp3 ::= ComboExp4 ComboExp3'*
ComboExp3' ::= 'xor' ComboExp4 ComboExp3'*
ComboExp4 ::= ID
| 'not' ComboExp
| '()' ComboExp  

Set expressions

AnyExp ::= 'any' '{' ID '}'
AllExp ::= 'all' '{' ID '}'

AnyForExp ::= 'for any' '{' ID '}' ForExp
AllForExp ::= 'for all' '{' ID '}' ForExp

ForExp ::= CompareExp
| FlankExp
| ChangeExp

Auxiliary

TimeUnit ::= 'milliseconds'
| 'seconds'
| 'minutes'
| 'hours'

89
### Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rules</td>
<td>'Rules:' Rule+</td>
</tr>
<tr>
<td>Rule</td>
<td>'Rule' ID ':' (Scope ':')? Pattern</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scope</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Between' ID 'and' ID</td>
<td>'While' ID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absence</td>
<td>Universality</td>
</tr>
<tr>
<td>'never' ID</td>
<td>'always' ID</td>
</tr>
<tr>
<td>ID 'at least' INT 'times'</td>
<td>ID 'at most' INT 'times'</td>
</tr>
<tr>
<td>'once' ID 'it holds at least' INT TimeUnit</td>
<td>'once' ID 'it holds at most' INT TimeUnit</td>
</tr>
<tr>
<td>ID 'at least' INT TimeUnit 'in total'</td>
<td>ID 'at most' INT TimeUnit 'in total'</td>
</tr>
<tr>
<td>ID 'always after at most' INT TimeUnit</td>
<td>ID 'at most every' INT TimeUnit</td>
</tr>
<tr>
<td>'when' ID 'then no' ID 'after at least' INT TimeUnit</td>
<td>'when' ID 'then ID 'after at most' INT TimeUnit</td>
</tr>
<tr>
<td>'when' ID 'then ID 'next cycle'</td>
<td>'when' ID 'at least' INT TimeUnit 'then' ID 'after at most' INT TimeUnit</td>
</tr>
<tr>
<td>ID 'at least' INT 'times in' INT TimeUnit</td>
<td>ID 'at most' INT 'times in' INT TimeUnit</td>
</tr>
</tbody>
</table>

### Abstraction classes

<table>
<thead>
<tr>
<th>Abstractions</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstractions</td>
<td>'Abstractions:' Abstraction+</td>
</tr>
<tr>
<td>Abstraction</td>
<td>'Abstr' ID ': rules of' ID 'applied with' '(' ID (',' ID)* ')'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Args</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Args</td>
<td>Arg (',' Arg)*</td>
</tr>
<tr>
<td>Arg</td>
<td>'Event:' ID</td>
</tr>
<tr>
<td>Arg</td>
<td>'Phase:' ID</td>
</tr>
<tr>
<td>Arg</td>
<td>'Var:' ID</td>
</tr>
</tbody>
</table>

90
Appendix E

Automata

As mentioned in Chapter 5, two types of propositions exist: phase and event propositions. A phase proposition can hold for a longer time period. Because of this, two types of events can be observed of such a proposition, namely \( \text{START}_P \) and \( \text{END}_P \). These are the starting and ending point of a proposition holding, respectively. An event proposition can only be detected as \( P \), as it holds for an infinitesimal time period. It starts and ends at the same time. In the automata edge labelling, the \( \text{START}_P \) and \( P \) triggers are denoted by \( P? \), whereas \( \text{END}_P \) is denoted by \( \neg P? \).

In the automata described in this appendix, \( t \) is a timer, and \( c \) is a value which represents an amount of time. See Section 8.2.4 for an informal definition of the automata.

E.1 Absence

E.1.1 Globally: Absence \( P \)
\( P \) is a phase or event proposition. \( P \) should never hold.

E.1.2 After \( Q \) until \( R \): Absence \( P \)
\( P, Q \) and \( R \) are event propositions. After \( Q \) until \( R \), \( P \) should never hold.
$P$ is a phase proposition. $Q$ and $R$ are event propositions. After $Q$ until $R$, $P$ should never hold.
E.2 Bounded Existence

One example of a property the Bounded Existence automata can observe: \textit{\(P\) should occur at (least/-most) 15 times.}

The Bounded Existence automata produce verdicts for every error that occurs. In the \textit{at most} \(k\) automata for example, an \textsc{Error} verdict is produced for additional \(P\) after the limit \(k\) has been reached. This methodology is not used for the \textsc{Ok} verdicts. In the \textit{at least} \(k\) times automata for example, an \textsc{Ok} verdict is only produced when \(P\) has occurred at least \(k\) times. No \textsc{Ok} verdicts are produced for any additional \(P\) occurring. This difference between the \textsc{Error} and \textsc{Ok} verdict reporting exists as, for this pattern, receiving additional \textsc{Error} verdicts is deemed important whereas receiving additional \textsc{Ok} verdicts is not.

E.2.1 Globally: Bounded Existence \(P\) (at most \(k\) times)

\(P\) is an event proposition. \(P\) may hold at most \(k\) times.

\[
\begin{align*}
\text{start} & \rightarrow P? \\
& x < k - 1 \\
& x := x + 1 \\
& P? \\
& x \geq k - 1 \\
& P?
\end{align*}
\]

E.2.2 Globally: Bounded Existence \(P\) (at least \(k\) times)

\(P\) is an event proposition. \(P\) should hold at least \(k\) times.

\[
\begin{align*}
\text{start} & \rightarrow P? \\
& x < k - 1 \\
& x := x + 1 \\
& P? \\
& x \geq k - 1 \\
& P?
\end{align*}
\]

E.2.3 After \(Q\) until \(R\): Bounded Existence \(P\) (at most \(k\) times)

\(P\), \(Q\) and \(R\) are event propositions. After \(Q\) until \(R\), \(P\) may hold at most \(k\) times.

\[
\begin{align*}
\text{start} & \rightarrow Q? \\
& x := 0 \\
& P? \\
& x < k - 1 \\
& x := x + 1 \\
& P? \\
& x \geq k - 1 \\
& P?
\end{align*}
\]

\[
\begin{align*}
\text{start} & \rightarrow R? \\
\end{align*}
\]
E.2.4 After Q until R: Bounded Existence $P$ (at least $k$ times)

$P$, $Q$ and $R$ are event propositions. After $Q$ until $R$, $P$ should hold at most $k$ times.

\[
\begin{align*}
&\text{start} \\
\rightarrow &\quad Q? \quad x := 0 \\
\rightarrow &\quad x < k - 1 \\
\rightarrow &\quad x := x + 1 \\
\rightarrow &\quad P? \quad x \geq k - 1 \\
\rightarrow &\quad R? \\
\rightarrow &\quad R?
\end{align*}
\]
E.3 Universality

E.3.1 Globally: Universality $P$

$P$ is a phase proposition. $P$ should hold all the time.

E.3.2 After $Q$ until $R$: Universality $P$

$P$ is a phase proposition. $Q$ and $R$ are event propositions. After $Q$ until $R$, $P$ should hold all the time.
E.4 Bounded Recurrence

E.4.1 Globally: Bounded Recurrence $P$

$P$ is an event proposition. $P$ should hold at least every $c$ time units.

\[
\begin{align*}
    &P? \\
    &t := 0
\end{align*}
\]

\[
\begin{align*}
    &t \geq c \\
    &t := 0
\end{align*}
\]

E.4.2 After $Q$ until $R$: Bounded Recurrence $P$

$P$, $Q$ and $R$ are event propositions. After $Q$ until $R$, $P$ should hold at least every $c$ time units.

\[
\begin{align*}
    &Q? \\
    &t := 0
\end{align*}
\]

\[
\begin{align*}
    &P? \\
    &t := 0
\end{align*}
\]

\[
\begin{align*}
    &R? \\
    &t \geq c \\
    &t := 0
\end{align*}
\]
E.5 Bounded / Strict Response

As the Strict Response is based on the Bounded Response pattern, properties using these patterns can be observed using the same automata.

Consider the event sequence $P-P-S$. Here, the first $P$ occurs in timestamp $n$ and the second $P$ and first $S$ occur together in the timestamp $m$, where $n < m$. As $P$ and $S$ occur in the same timestamp, they co-occur. As mentioned in Chapter 5, we assume that the minimum machine control application (MCA) response time is one execution cycle. This means that an MCA could not have had enough time to issue the $S$ as a response to the second $P$. Therefore, a Bounded/Strict Response automaton should watch out for a new $S$ as a response to the second $P$. To achieve this, once a $P$ is handled by the automaton before an $S$ in the same time tick, the automaton returns to its state sensitive to $S$ after processing $S$, instead of returning to a state which is not sensitive to $S$.

E.5.1 Globally: Bounded / Strict Response P, S

$P$ and $S$ are event propositions. $P$ should always be followed by $S$ within $c$ time units. Here, $d$ is the duration of one cycle.
E.5.2 After Q until R: Bounded / Strict Response P, S

$P, S, Q$ and $R$ are event propositions. After $Q$ until $R$, $P$ should always be followed by $S$ within $c$ time units.
E.6 Minimum Duration

E.6.1 Globally: Minimum Duration \( P \)

\( P \) is a phase proposition. When \( P \) holds, it should hold for at least \( c \) time units.

\[
\begin{align*}
\text{start} & \rightarrow (t := 0) \rightarrow P ? \rightarrow t \geq c \\
& \rightarrow \neg P ? \rightarrow \neg P ?
\end{align*}
\]

E.6.2 After Q until R: Minimum Duration \( P \)

\( P \) is a phase proposition. \( Q \) and \( R \) are event propositions. After \( Q \) until \( R \), when \( P \) holds, it should hold for at least \( c \) time units.

Figure E.1 visualises what phases of a proposition \( P \) holding are considered valid and invalid with respect to the Minimum Duration pattern and the After Until scope. The figure shows a timeline where the scope After \( Q \) until \( R \) is active. Grey bars show \( P \) holding across three different executions.

\[
\begin{align*}
P & \rightarrow t := 0 \\
& \rightarrow Q ? \\
& \rightarrow R ? \rightarrow t \geq c \\
& \rightarrow \neg P ? \\
& \rightarrow \neg P ?
\end{align*}
\]

Figure E.1: After Until: Minimum Duration visualised
E.7 Maximum Duration

E.7.1 Globally: Maximum Duration \( P \)

\( P \) is a phase proposition. When \( P \) holds, it should hold for at most \( c \) time units.

\[
\begin{align*}
t &:= 0 \\
\neg P &\Rightarrow t \geq c
\end{align*}
\]

E.7.2 After Q until R: Maximum Duration \( P \)

\( P \) is a phase proposition. \( Q \) and \( R \) are event propositions. After \( Q \) until \( R \), when \( P \) holds, it should hold for at most \( c \) time units.

Figure E.2 visualises what phases of a proposition \( P \) holding are considered valid and invalid with respect to the Maximum Duration pattern and the After Until scope. The figure shows a timeline where the scope After \( Q \) until \( R \) is active. Grey bars show \( P \) holding across three different executions.

\[
\begin{align*}
\neg P &\Rightarrow t \geq c \\
Q &\Rightarrow R
\end{align*}
\]
E.8 Bounded Absence

E.8.1 Globally: Bounded Absence P, S

P and S are event propositions. When P holds, S should not hold for at least c time units.

E.8.2 After Q until R: Bounded Absence P, S

P, S, Q and R are event propositions. After Q until R, when P holds, S should not hold for at least c time units.
E.9 Time-constrained Absence

One example of a property the Time-constrained Absence automata observe: \( P \) can occur multiple times, but in total, \( P \) should have held at most 10 seconds.

This pattern is the opposite of the Time-constrained Universality pattern. Because of this, Time-constrained Absence \( P \) could be specified as Time-constrained Universality \( \neg P \). For clarity, automata of both patterns have been developed.

E.9.1 Globally: Time-constrained Absence \( P \)

\( P \) is a phase proposition. \( P \) should hold for at most \( c \) time units in total. This does not have to be one continuous time period, but can also be multiple short periods of time.

E.9.2 After \( Q \) until \( R \): Time-constrained Absence \( P \)

\( P \) is a phase proposition. \( Q \) and \( R \) are event propositions. After \( Q \) until \( R \), \( P \) should hold for at most \( c \) time units in total. This does not have to be one continuous time period, but can also be multiple short periods of time.
E.10 Time-constrained Universality

One example of a property the Time-constrained Universality automata observe: *P can occur multiple times, but in total, P should have held at least 10 seconds.*

This pattern is the opposite of the Time-constrained Absence pattern. Because of this, *Time-constrained Universality P* could be specified as *Time-constrained Absence ¬P*. For clarity, automata of both patterns have been developed.

E.10.1 Globally: Time-constrained Universality P

*P* is a phase proposition. *P* should hold for at least *c* time units in total. This does not have to be one continuous time period, but can also be multiple short periods of time.

\[ t \geq c \]

E.10.2 After Q until R: Time-constrained Universality P

*P* is a phase proposition. *Q* and *R* are event propositions. After *Q* until *R*, *P* should hold for at least *c* time units in total. This does not have to be one continuous time period, but can also be multiple short periods of time.
E.11 Restricted Recurrence

One example of a property the Restricted Recurrence automata can observe: \( P \) should occur at most \( once \) \( every \) \( 2 \) \( minutes \). These automata limit a \( P \) from recurring too often.

E.11.1 Globally: Restricted Recurrence P

\( P \) is an event proposition. \( P \) should hold at most every \( c \) time units.

\[
\begin{align*}
\text{start} & \quad \rightarrow \\
\text{\( P? \)} & \quad \text{\( t := 0 \)} & & \quad \rightarrow \\
\text{\( t \geq c \)} & \quad \text{\( t := 0 \)} & \quad \rightarrow \\
\end{align*}
\]

E.11.2 After Q until R: Restricted Recurrence P

\( P, Q \) and \( R \) are event propositions. After \( Q \) until \( R \), \( P \) should hold at most every \( c \) time units.

\[
\begin{align*}
\text{start} & \quad \rightarrow \\
\text{\( Q? \)} & \quad \rightarrow \\
\text{\( R? \)} & \quad \rightarrow \\
\text{\( t := 0 \)} & \quad \rightarrow \\
\text{\( P? \)} & \quad \rightarrow \\
\end{align*}
\]
E.12 Prefaced Bounded Response

One example of a property the Prefaced Bounded Response automata can observe: \( P \) should occur for at least for 1 second, before a response in the form of \( S \) is expected within 4 seconds.

In these automata, after \( P \) holds for the required amount of time, \( P \) is no longer observed (the event \( \neg P \) is not observed). The observation of \( P \) restarts when the response, in the form of \( S \) is observed or the scope is closed and reopened.

E.12.1 Globally: Prefaced Bounded Response \( P, S \)

\( P \) is a phase proposition. \( S \) is an event proposition. When \( P \) holds for at least \( c \) time units sequentially, \( S \) should hold within \( d \) time units.

E.12.2 After \( Q \) until \( R \): Prefaced Bounded Response \( P, Q \)

\( P \) is a phase proposition. \( S, Q \) and \( R \) are event propositions. After \( Q \) until \( R \), when \( P \) holds for at least \( c \) time units sequentially, \( S \) should hold within \( d \) time units.
E.13 Duration Bounded Existence

One example of a property the Duration Bounded Existence automata can observe: \( P \) should occur at least 4 times every 30 minutes.

These automata do not handle overlapping time periods. A time period starts when an automaton first starts. A new time period only begins when the first time period has finished. Because of this, the example property above is already satisfied if \( P \) holds 4 times in the first 2 minutes of the 30 minute time period. If more spread out sequences of \( P \) are desired, one should use a combination of the Bounded Recurrence and Restricted Recurrence patterns.

E.13.1 Globally: Duration Bounded Existence \( P \) (at most \( k \) times)

\( P \) is an event proposition. In a time period of \( c \) time units, \( P \) may hold at most \( k \) times.

\[
\begin{align*}
\text{start} & \quad P? \\
\text{at} & \quad x < k - 1 \\
\text{then} & \quad x := x + 1 \\
\text{if} & \quad x \geq k - 1 \\
\text{else} & \quad t := c \\
\text{end} & \quad t := 0
\end{align*}
\]

E.13.2 Globally: Duration Bounded Existence \( P \) (at least \( k \) times)

\( P \) is an event proposition. In a time period of \( c \) time units, \( P \) should hold at least \( k \) times.

\[
\begin{align*}
\text{start} & \quad P? \\
\text{at} & \quad x < k - 1 \\
\text{then} & \quad x := x + 1 \\
\text{if} & \quad x \geq k - 1 \\
\text{else} & \quad t := c \\
\text{end} & \quad t := 0
\end{align*}
\]
E.13.3 After Q until R: Duration Bounded Existence $P$ (at most $k$ times)

$P$, $Q$ and $R$ are event propositions. After $Q$ until $R$, in a time period of $c$ time units, $P$ may hold at most $k$ times.

E.13.4 After Q until R: Duration Bounded Existence $P$ (at least $k$ times)

$P$, $Q$ and $R$ are event propositions. After $Q$ until $R$, on a time period of $c$ time units, $P$ should hold at least $k$ times.
Appendix F

Data Format

Listing F.1 shows the format which is used as an input of the Monitoring System to create property monitors. As the four existence properties all have two versions, namely at least \( k \) and at most \( k \), an extra operator value is used for those properties. This operator can be a ‘\( > \)’ or ‘\( < \)’ value which annotates at least \( k \) and at most \( k \), respectively.

<table>
<thead>
<tr>
<th>Property</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>GloballyAbsence</td>
<td>(&lt;id&gt; ; &lt;prop_id&gt;)</td>
</tr>
<tr>
<td>GloballyUniversality</td>
<td>(&lt;id&gt; ; &lt;prop_id&gt;)</td>
</tr>
<tr>
<td>GloballyBoundedExistence</td>
<td>(&lt;id&gt; ; &lt;prop_id&gt; ; &lt;amount&gt; ; &lt;operator&gt;)</td>
</tr>
<tr>
<td>GloballyDurationExistence</td>
<td>(&lt;id&gt; ; &lt;prop_id&gt; ; &lt;time&gt; ; &lt;amount&gt; ; &lt;operator&gt;)</td>
</tr>
<tr>
<td>GloballyDurationMin</td>
<td>(&lt;id&gt; ; &lt;prop_id&gt; ; &lt;time&gt;)</td>
</tr>
<tr>
<td>GloballyDurationMax</td>
<td>(&lt;id&gt; ; &lt;prop_id&gt; ; &lt;time&gt;)</td>
</tr>
<tr>
<td>GloballyTimeConstrainedAbsence</td>
<td>(&lt;id&gt; ; &lt;prop_id&gt; ; &lt;time&gt;)</td>
</tr>
<tr>
<td>GloballyTimeConstrainedUniversality</td>
<td>(&lt;id&gt; ; &lt;prop_id&gt; ; &lt;time&gt;)</td>
</tr>
<tr>
<td>GloballyBoundedRecurrence</td>
<td>(&lt;id&gt; ; &lt;prop_id&gt; ; &lt;time&gt;)</td>
</tr>
<tr>
<td>GloballyRestrictedRecurrence</td>
<td>(&lt;id&gt; ; &lt;prop_id&gt; ; &lt;time&gt;)</td>
</tr>
<tr>
<td>GloballyBoundedAbsence</td>
<td>(&lt;id&gt; ; &lt;cause_prop_id&gt; ; &lt;effect_prop_id&gt; ; &lt;time&gt;)</td>
</tr>
<tr>
<td>GloballyBoundedResponse</td>
<td>(&lt;id&gt; ; &lt;cause_prop_id&gt; ; &lt;effect_prop_id&gt; ; &lt;time&gt;)</td>
</tr>
<tr>
<td>GloballyPrefacedBoundedResponse</td>
<td>(&lt;id&gt; ; &lt;cause_prop_id&gt; ; &lt;effect_prop_id&gt; ; &lt;cause_time&gt; ; &lt;time&gt;)</td>
</tr>
<tr>
<td>AfterUntilAbsencePhase</td>
<td>(&lt;id&gt; ; &lt;before_prop_id&gt; ; &lt;after_prop_id&gt;)</td>
</tr>
<tr>
<td>AfterUntilAbsenceEvent</td>
<td>(&lt;id&gt; ; &lt;before_prop_id&gt; ; &lt;after_prop_id&gt; ; &lt;prop_id&gt;)</td>
</tr>
<tr>
<td>AfterUntilUniversality</td>
<td>(&lt;id&gt; ; &lt;before_prop_id&gt; ; &lt;after_prop_id&gt; ; &lt;prop_id&gt;)</td>
</tr>
<tr>
<td>AfterUntilBoundedExistence</td>
<td>(&lt;id&gt; ; &lt;before_prop_id&gt; ; &lt;after_prop_id&gt; ; &lt;prop_id&gt; ; &lt;amount&gt; ; &lt;operator&gt;)</td>
</tr>
<tr>
<td>AfterUntilDurationExistence</td>
<td>(&lt;id&gt; ; &lt;before_prop_id&gt; ; &lt;after_prop_id&gt; ; &lt;prop_id&gt; ; &lt;time&gt; ; &lt;amount&gt; ; &lt;operator&gt;)</td>
</tr>
<tr>
<td>AfterUntilMinDuration</td>
<td>(&lt;id&gt; ; &lt;before_prop_id&gt; ; &lt;after_prop_id&gt; ; &lt;prop_id&gt; ; &lt;time&gt; ; &lt;amount&gt; ; &lt;operator&gt;)</td>
</tr>
<tr>
<td>AfterUntilMaxDuration</td>
<td>(&lt;id&gt; ; &lt;before_prop_id&gt; ; &lt;after_prop_id&gt; ; &lt;prop_id&gt; ; &lt;time&gt;)</td>
</tr>
<tr>
<td>AfterUntilTimeConstrainedAbsence</td>
<td>(&lt;id&gt; ; &lt;before_prop_id&gt; ; &lt;after_prop_id&gt; ; &lt;prop_id&gt; ; &lt;time&gt;)</td>
</tr>
</tbody>
</table>
AfterUntilTimeConstrainedUniversality : <id> ; <before_prop_id> ; <after_prop_id> ; <prop_id> ; <time>

AfterUntilBoundedRecurrence ; <id> ; <before_prop_id> ; <after_prop_id> ; <prop_id> ; <time>

AfterUntilRestrictedRecurrence ; <id> ; <before_prop_id> ; <after_prop_id> ; <prop_id> ; <time>

AfterUntilBoundedAbsence ; <id> ; <before_prop_id> ; <after_prop_id> ; <cause_prop_id> ; <effect_prop_id> ; <time>

AfterUntilBoundedResponse ; <id> ; <before_prop_id> ; <after_prop_id> ; <cause_prop_id> ; <effect_prop_id> ; <cause_time> ; <time>

AfterUntilPrefacedBoundedResponse ; <id> ; <before_prop_id> ; <after_prop_id> ; <cause_prop_id> ; <effect_prop_id> ; <cause_time> ; <time>

LISTING F.1: Property format
Appendix G

Evaluation Exercises

G.1 Exercise 1: Phases/events and patterns/scopes

In this exercise, four timelines are sketched which display propositions (P, Q, R, S) resolving to true, over time. For each timeline, one or two properties are specified. Write down for each property, if the sequence of propositions in the timeline violates the property or not. If it does, explain why.

Part 1

1. never P
2. Between Q and R: never P

Part 2

3. when P, then S after at most 15
   (a) HINT: this property can be violated or satisfied by the sequence twice
4. Between Q and R: when P, then S after at most 40

Part 3

5. P always within at most 10
G.2 Exercise 2: Understanding and modifying rulesets

This exercise revolves around the MerryGoRound machine. To simulate that you have knowledge about the components of the machine, their ids are stated below:

Conveyors:
- MerryGoRound.LeftConveyor
- MerryGoRound.RightConveyor

The conveyor instances above implement the component class MerryGoRound.SideConveyor.

Conveyor motors:
- MerryGoRound.LeftConveyor.Motor
- MerryGoRound.RightConveyor.Motor

MerryGoRound module:
- MerryGoRound.Module

1. See language file exercise2.dsl. Here, a ruleset specifies one property. Try to describe this property in your own words.

2. Improve the names used for the ruleset, propositions, and rules so that they better express their intent.

3. Modify the ruleset in such a way, that the property is only in effect when the MerryGoRound.Module attribute ‘state’ is not ‘error’. (HINT: scope change)

4. Modify the ruleset in such a way, that the property affects all conveyor instances of the component class MerryGoRound.SideConveyor.

\[
\text{Figure G.1: Exercise2.dsl} \quad \text{Figure G.2: Exercise2_Solution.dsl}
\]
G.3 Exercise 3: Creating rulesets and understanding results

This exercise also revolves around the MerryGoRound machine. Below some component ids possibly required in this exercise:

Lifts that are part of the conveyors:
- MerryGoRound.LeftConveyor.RightLift
- MerryGoRound.RightConveyor.RightLift

1. Create a new ruleset in file exercise3.dsl, that defines the following property:

   During the entire execution of the MerryGoRound machine, the RightLift of the LeftConveyor should move up at most 600 milliseconds after the sensor in the centre of the LeftConveyor triggers.

   - RightLift is up when its variable ‘oUp’ has value ‘True’
   - The centre sensor of the LeftConveyor is triggered when the variable ‘iCentreSensor’ of the LeftConveyor has value ‘True’.

2. Describe the results of the verification using the ruleset of question 1 in your own words.

```plaintext
RuleSet liftReadyForItem {

Components:
  Instance MerryGoRound.LeftConveyor as conveyor
  Instance MerryGoRound.LeftConveyor.RightLift as lift

Propositions:
  Event itemInCentre:  start of conveyor.iCentreSensor == 'true'
  Event liftUp:        start of lift.state == 'up'

Rules:
  Rule liftResponse:   when itemInCentre, then liftUp after at most 500 milliseconds

Figure G.3: Exercise3_solution.dsl
```