UNIVERSITY OF TWENTE.

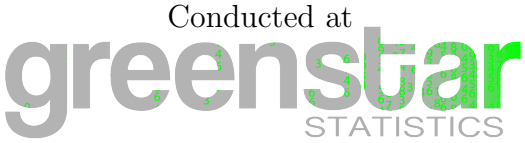
Master's Thesis Computer Science

A monitoring solution for multi-language software

Author:
Ale Strooisma

14 July 2016

Conducted at



Supervisors:
prof. dr. A. Rensink
dr. ir. M. van Keulen
dr. ir. M. van Eenennaam (GreenStar Statistics)

Abstract

Correctness of software is very important. In some applications a software failure may cause serious —even physical— damage, for example when the software manages railroad switches. In other applications the results of a software failure might not be so dramatic, but preventing failures is still essential in order to provide a good user experience.

Runtime monitoring is a method for ensuring software correctness that can be used in addition to the more conventional software testing and model checking. Runtime monitoring is applied at runtime in production environments, as opposed to software testing and model checking which are only applied in a development environment.

Currently there are no runtime monitoring frameworks available for software consisting of multiple components written in different languages, even though runtime monitoring can be very useful for such complex systems. This report presents an architecture and prototype implementation for such a system. This architecture is very flexible and highly extensible in order to support many very different components and programming languages.

Contents

Chapter 1

Introduction

Ovis Telematics develops the Shepherd platform, a system that monitors driving behaviour for taxi companies, leasing companies and other businesses with a significant vehicle fleet. By providing feedback to drivers, their driving behaviour can be improved. For the company this means a reduction in costs and a smaller CO_2 footprint, due to reduced fuel consumption. Additionally, maintenance costs are reduced due to a longer tyre and brake disc service life, and fewer accidents due to safer driving lead to lower repair and insurance costs.

Feedback on driving behaviour is provided to the customers in various ways. First, there is the GreenStar portal, a web application that provides the driving behaviour statistics in various forms, such as graphs showing an overall score over time, or a list of specific cases of bad behaviour such as excessive acceleration or speeding. Additionally, various kinds of periodic reports are sent to the customer to give a quick overview of the current status.

The data from which those statistics are derived is obtained from so-called 'ecoboxes' mounted in the vehicles. The ecoboxes gather data from the on-board diagnostics (OBD) port of the vehicle and from measurement devices in the ecobox itself, such as an accelerometer and a GPS module. Additional data is provided by the customers, such as refuelling data.

Large amounts of raw data are produced each day which have to be processed into driving behaviour statistics. This is a complex process and as such a lot can go wrong. Often problems are encountered by the end-user, which is highly undesirable. To be able to detect failures earlier, Ovis Telematics wants to monitor the Shepherd platform at runtime.

Runtime monitoring is a method of verifying software correctness, complementary to software testing and model checking[1]. The main difference from those techniques is that it is performed after deployment, during regular use, instead of only in a development environment. As such, only a single execution is observed, whereas software testing and model checking aim to cover as many as possible. The major benefit of runtime monitoring in this area is that, while only one execution is checked at a time, all executions occurring in practice are checked.

With runtime monitoring, correctness of the software run is determined by verifying whether the run satisfies a formal specification, just like in model checking. If a property in the specification is violated, this is an indication that

a failure has occurred or is going to occur in the software. When this is detected, action can be taken in order to minimize the damage caused. An important concept in runtime monitoring is instrumentation: the act of modifying the monitored program in order to obtain more information. This process depends strongly on the environment.

1.1 Problem statement

There is currently no runtime monitoring framework available that can be used for the Shepherd platform. The most important barrier to adopting an existing runtime monitoring solution is that the Shepherd platform consists of many different components, implemented in various programming languages, while most runtime monitoring frameworks only support single-component software implemented in a specific language[1]. Therefore a new runtime monitoring framework must be developed that does satisfy the needs of such systems. This section lists the requirements for this solution, which will be called *Sheepdog+*.

R1: Monitor separate components. The Sheepdog+ system must be able to monitor systems consisting of multiple separate components, running in multiple threads and processes.

R2: Be able to monitor components written in multiple languages. Software written in multiple languages must be supported and different components written in different languages must be monitored at the same time.

R3: Support for a new language must be easily added. This is required for the system to be truly language-independent, as illustrated by the following scenario: a system monitored by Sheepdog+ is extended with a component written in a new language. If support for this component cannot be easily added, the usefulness of Sheepdog+ is greatly reduced.

R4: Monitor interaction between separately executing components. To properly monitor a system made up of multiple components, interaction between these components must be taken into account, regardless of the implementation languages of the components, as this is a possible source of problems. If interaction between components cannot be monitored, this would be a serious weakness in the framework. More details are provided in Section 2.5.

R5: Monitor components separately. Apart from interaction, the system must also be able to monitor the behaviour of each component separately, even when multiple instances are running at the same time.

R6: Support various different types of properties. Monitoring should not be limited to a specific class of properties, as the system is intended to work with a class of programs made up out of very different components. Examples of property classes are execution trace correctness, time constraints and liveness/termination checks. What this requirement means is that, although an implementation might be limited, the architecture should be very flexible enough to support many kinds of properties.

[1] An overview of existing solutions is given in Section 2.6

R7: Monitor system performance. Correct behaviour of a system can also be threatened by performance issues, so the Sheepdog+ must also be able to monitor this. Two examples of such properties are a) that a data processing run needs to be finished at the time that the data is needed, and b) that it is unacceptable if serving a web page takes very long.

R8: Writing properties should not be hard. There are two important reasons why this would be a serious problem. Firstly, it would be bad for usability, likely causing the system to not be used to its fullest extent or even barring adoption of the system. Secondly, it increases the likelihood of mistakes being made in the specification, causing false positives or false negatives.

R9: Responding to violations. If a failure is detected the system also needs to act upon this. As indicated in the above requirements, the system needs to be able to work with very different systems and needs to be able to detect very different problems. Different situations may require different reactions to detected, so the mechanism for reacting to detections needs to be very flexible.

R10: Provide a mechanism for alerting staff of detected failures. If a failure is detected, the staff needs to be alerted so any effects of the failure can be remedied and the underlying error can be fixed in order to prevent further failures. To support the latter, information about the failure should be provided, such as the events leading to the failure or the state of the system when the failure occurred.

R11: Prevent failures from occurring. Although this may not be possible in all situations, runtime monitoring can be used to detect failures before they occur (see Chapter 2 for details). The system should provide a method to act upon those detections in order to prevent the failure from occurring.

1.2 Research questions

This report aims to answer the following research question:

How to create a flexible, extensible runtime monitoring framework for systems consisting of multiple separate components implemented in various languages?

This question can be refined into the following subquestions:

Subquestion 1: How to allow monitoring and instrumentation for multiple implementation languages? The main point where Sheepdog+ has to differentiate from other runtime monitoring tools is its support for multiple implementation languages. However, this should not complicate the way the user interacts with the monitoring system: the user actions, such as writing a specification, should be done in one way, independent of implementation languages.

Subquestion 2: How to react to violations and recover from failures in a varied environment? In the ideal case, a detected failure can be prevented, or otherwise handled in such a way that the system can be brought back to

a correct state of operation. Such actions are normally strongly dependent on the specific failure and software system. Because the targeted class of software includes many very different systems, it is unlikely that this can be done in one way for all violations. Therefore, a very flexible way to handle violations must be provided. Again, the user should only need to use one mechanism for specifying this behaviour.

Subquestion 3: How to provide developers with feedback for resolving the cause of an encountered violation? When a violation is detected, the developer must be notified, so the underlying error that caused the failure can be fixed. Additional information should be provided to the developers that helps them determine the cause of the failure.

Subquestion 4: How to measure scalability of the monitored system? As denoted in Requirement 7, the performance of software can also be important for correct behaviour. In the Shepherd platform especially, there are a number of components where scalability to larger amounts of data, due to an increase in customers, is a serious concern.

1.3 Research method

The approach taken during this research is as follows. Before working on the solution, an extensive literature study has been performed. After this, an architecture was designed for a system that addresses the problem statement above, based on the listed requirements and taking inspiration from existing runtime monitoring frameworks and research. A prototype implementation has been built on this architecture that is used with GreenStar Statistics' Shepherd platform in their production environment. The prototype is validated against the requirements while taken in use there. An iterative approach was taken: whenever it turned out that the design was not able fulfil the requirements, the architecture was revisited and the prototype modified. It was ensured that the prototype includes enough functionality to demonstrate that it satisfies the requirements, even though it was not be possible to completely implement the architecture due to time contraints. Instructions on using the software have been provided to the Ovis Telematics developers, so they can make further use of it with the Shepherd platform.

Finally, when the research period drew to an end and the design was considered stable, answers to the research questions have been formulated in the form of an explanation of how the issue addressed in the question is solved by the architecture.

1.4 Overview

This report starts off with a summary of existing research on runtime monitoring in Chapter 2. Chapter 3 describes the architecture of the Sheepdog+ monitoring framework, followed by details on the prototype implementation in Chapter 4. In chapter 5 the prototype is applied to the Shepherd platform and a demonstration program in order to validate the design. Finally, Chapter 6 answers the research question and lists some subjects for further research.

Chapter 2

Runtime monitoring

Runtime monitoring is a method ensuring correct program execution, just as testing and model checking [1]. There are a few key differences though. Most importantly, runtime monitoring is applied during regular use of the monitored software, not only in a development environment [2]. Therefore only the single, current execution of the system is observed [3]. As a result, runtime monitoring can't be used to say anything about the general correctness of the software. Another difference is that the main goal of runtime monitoring is often not finding bugs, but preventing failures from occurring by influencing the control flow of the software when a bug is detected.

Runtime monitoring software is developed separately from the monitored system, as a framework that generates the monitor: a piece of software that checks the correctness of the execution of the monitored system, which is either integrated with the monitored system or running parallel to it[4, 5]. The main advantage of this approach is that the monitored system's source code is not 'polluted' by the monitoring code, keeping it easy to read and improving maintainability. Secondly, the code generation design makes the monitoring software flexible against changes in the monitored system and allows easy reuse in other projects.

Typically, a monitoring framework takes a formal specification and generates code to verify the properties in this specification at various times during the monitored system's execution. If a violation is detected, the monitor can respond and ideally allow the system to continue execution or at least prevent harmful effects from occurring.

Note that this document makes a distinction between the terms 'monitoring framework' and 'monitor'. As described above, based on a specification, the monitoring framework generates the code that makes up the monitor. This generated monitor can still depend on the monitoring framework to run, or it can be completely separate from it, depending on the concrete implementation. Even though this architecture is not a fundamental requirement for runtime monitoring, almost all tools are designed in this way (for example, [4, 5, 6, 7, 8, 9, 10, 11]), and therefore this document assumes this architecture is used to explain the other concepts of runtime monitoring.

To illustrate the various concepts described in this chapter, a simplified digital store application (DSA) will be used as a running example. In this application the user can perform two different actions: authentication and or-

dering an item. For simplicity, it will be assumed that between the moment an item is ordered and the moment the application confirms or refuses the order, additional orders will be ignored. Also, the user cannot log out other than by closing the application. The following properties, expressed informally here, will be considered:

1. Only after authentication can an item be successfully bought.
2. An order can only be completed if the user has enough credits.
3. It should not take more than one minute to process an order.
4. An extension of property 1: Only after authentication and following a request can an order be successfully completed.

The rest of this chapter is organized as follows. First, in Section 2.1 a few general variation points within runtime monitoring are considered. Section 2.2 explains how a monitoring framework can modify the monitored system to allow better monitoring and intervention. Section 2.3 then describes how properties are specified and how a monitor checks whether they are violated. Various methods of responding to violations, including methods to prevent failures are detailed in Section 2.4. Section 2.5 explores how monitoring a system that consists of multiple loosely coupled components should be treated in terms of the theory explained in the earlier parts of the chapter. Finally, an overview of existing monitoring solutions is given in Section 2.6.

2.1 Monitor variation points

This section lists three variation points that must be considered when applying runtime monitoring. These are adapted from [10]. Note that other authors also make these distinctions, but the exact definitions may deviate slightly from those presented here (for example, [3] uses the term "offline" for what is defined as out-line below, and does not mention the concept of offline monitoring as used in this document).

Monitor placement

A monitor can be implemented as part of the monitored system or as a separate program. This is referred to as in-line and out-line monitoring, respectively. An in-line monitor is more powerful, as it can better interact with the monitored system. It can, for example, read the values of certain variables and parameters or can influence the program's execution to prevent failures from occurring, for example by executing extra code. In-line monitoring, however, is very invasive compared to out-line monitoring. Even if the monitor does not influence execution, the monitored program's source code must be modified to provide an in-line monitor.

Synchronicity

A related design choice is between synchronous and asynchronous monitoring. In the case of synchronous monitoring, the monitor evaluation occurs in the monitored system's control flow. This allows the monitor to immediately act

upon detecting a violation. An asynchronous monitor evaluates properties in a separate thread or process, such that the monitored system does not have to wait for the evaluation to finish, reducing performance impact of the monitor. An asynchronous monitor cannot immediately respond to violations, however, as the system has already progressed while the specification was evaluated. This makes preventing failures more difficult.

Offline monitoring

Normally a monitor runs together with the system, this is also referred to as online monitoring. The related notion of offline monitoring refers to checking correctness of a software run after execution by evaluating a stored execution trace, for example in the form of a log file. Although offline monitoring cannot prevent a failure, it can still be useful if online monitoring is not applicable, for example in a multi-component system to determine if the program's output can be used for further processing.

Offline monitoring can be argued not to be a form of runtime monitoring, as the program is not monitored at runtime. However, offline monitoring is still a method of verifying program execution after deployment and it works on the same principles as online monitoring. As such, a runtime monitoring framework can be used for offline monitoring with only minor modifications.

Interdependencies

- Offline monitoring can only be done by an out-line monitor, as the program that is monitored is not running during verification, and is therefore always asynchronous.
- An in-line monitor can only monitor synchronously, as it executes in the same thread as the system code.
- Some of the monitoring concepts described in the rest of this chapter will also depend on some of these variation points.

2.2 Instrumentation

A software run can be considered as a possibly infinite sequence of events occurring in the system. Such a sequence is called an *execution trace* [10]. In runtime monitoring, correctness of a run is determined by evaluating the execution trace. To do this, however, the execution trace must be exposed to the monitor. This is done by sending messages to the monitor. These messages are referred to as *events*. The stream of events to the monitor can be considered a concretization of the execution trace. There are various ways to generate these events, most of which modify the monitored system. This process of modifying the system is referred to as *instrumentation*.

The simplest and least invasive method of instrumentation is to monitor existing input and output channels and to generate an event when certain input or output is observed [5]. This method is very limited, however, because only behavior that is characterized by specific input or output can be verified. The big advantage of this method is that it is completely non-invasive, so it can be very useful if more invasive instrumentation methods are inapplicable.

A more powerful, but more invasive way is *automated instrumentation*, where the framework modifies the source- or byte-code of the monitored system, usually at compile-time. Most commonly this is done by providing the framework with a file containing the formal specification. The positions at which code needs to be inserted is determined from the specification and in some implementations from additional information provided by the user. Often this method makes use of aspect-oriented programming techniques (for example, JavaMOP [4]), but frameworks can also implement a custom method of byte-code insertion, usually based on an instrumentation script (for example, Java PathExplorer [7]). Note that those mechanics are strongly dependent on the language in which the target software is written and can therefore not be reused in projects written in a different language.

In another approach to automated instrumentation, the user provides the specification not in a separate file, but in the form of annotations in the source code that describe the properties. The framework then provides a preprocessor that translates these annotations to monitoring code [6, 4]. This method is based on the principles of *design by contract* [12]. It is more invasive to the source code than the method above, but provides a very intuitive way of specifying properties without the need to provide additional information for the position of code insertion.

Instead of relying on a framework it is also possible to instrument the system manually, writing the monitoring code in-line. This is not advised however, as it makes the existing code harder to understand and provides a very strong coupling between the monitor and the monitored system. Additionally, the extendability and reusability of this method are very bad and the manual coding is prone to errors.

2.3 Specification and verification of properties

As mentioned above, a monitor determines correctness of a run by evaluating its execution trace. The execution trace is possibly infinite, but at any point during execution, only a finite *prefix* of the execution trace is observed [10]. The properties in the formal specification need to be verified based on the available finite prefix. Every time the monitor receives a new event, the event is appended to the prefix. This means that the last monitor verdict (i.e., whether the property is violated or not) may not be representative of the current system state anymore. This shows that the right moment to evaluate the properties is each time a new event is received. This way, the minimum number of checks to get the maximum benefit of the monitor are performed: with fewer checks violations can be missed, and with more checks no more violations will be detected.

The events considered in the DSA example are:

- `auth`: this event is generated after a user successfully authenticates,
- `order`: this event occurs whenever the user places an order,
- `bought`: occurs when an order is successfully processed,
- `failed`: generated when for any reason an order will not be completed.

2.3.1 Monitor verdicts

Because the available prefix based upon which the properties must be evaluated is finite, a property can not only be satisfied or violated, but the monitor verdict can also be inconclusive [10]. To illustrate, consider the example property 1 given above. If a `bought` event is observed before authentication, the property is violated. On the other hand, when the `auth` event is received and the property has not yet been violated, it has been satisfied: the property cannot be violated anymore. When neither of these events have occurred, however, the property is satisfied nor violated.

There are two ways to handle those three values. The first option is to make the framework support multiple-valued logic. The most obvious option would be to use three-valued logic, mapping satisfaction to `true`, violation to `false` and inconclusive verdicts to `unknown` [1, 10], but 4-valued logics have also been proposed [13], where `unknown` is split into `presumably true` and `presumably false`.

The other option is to reduce the problem to a two-valued system. This is actually what most tools do [4]. In this approach, inconclusive verdicts are considered to be either violations, with the reasoning that no proof for correct behaviour has been supplied, or satisfactions, with the reasoning that no definitive indication of failure has been observed. Those methods of interpretation are called the strong and weak view semantics, respectively [14].

2.3.2 Observable properties

Another effect of the finiteness of the available prefix is that not all properties can be observed. Observable properties include, but are not limited to, safety properties and co-safety properties. To define those two classes of properties, let us first define good and bad prefixes:

Definition 1. *Let P be a property and S be the set of traces satisfying P. A prefix p is then called:*

- *a good prefix for P if $px \in S$ for all sequences of events x, or*
- *a bad prefix for P if $px \notin S$ for all sequences of events x.*

In other words, observing a good prefix for property P guarantees that P will not be violated during that run, while observing a bad prefix for property P guarantees that P will be violated. Note that every finite prefix that has a good or bad prefix is, respectively, a good or bad prefix as well. Using those definitions, safety and co-safety properties can be defined:

Definition 2. *A property P, with S being the set of traces satisfying P, is called a safety property if for all traces $t \notin S$ there is a finite prefix that is bad.*

Definition 3. *A property P, with S being the set of traces satisfying P, is called a co-safety property if for all traces $t \in S$ there is a finite prefix that is good.*

From these definitions, it can be seen that safety and co-safety properties can be observed in finite time, because they are defined by their finite prefixes. Note that these definitions have been adapted from [10] to also take finite executions into account.

2.3.3 Evaluation techniques

To maintain the entire execution trace at runtime and to traverse it fully during each property evaluation would be very time- and memory-inefficient. Therefore so-called *non-trace-storing techniques* are used to evaluate properties. Most commonly, this is achieved by implementing property-checkers as automata in the monitor [10]. Each event sent to the monitor brings each automaton to a state where the occurrence of this event is taken into account, thus eliminating the need to save the event history. The type of automaton used depends on the type of properties that the framework must support, on which Section 2.3.5 elaborates. A common and basic automaton is the deterministic finite state machine (FSM):

Definition 4. *A deterministic finite state machine is represented formally by a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$*, where*

- Q *is a finite set of states,*
- Σ *is a finite set of symbols, called the alphabet of the automaton,*
- $\delta : Q \times \Sigma \to Q$ *is the transition function,*
- $q_0 \in Q$ *is the initial state,*
- $F \subseteq Q$ *is the set of final states.*

This automaton is called deterministic, because the transition function specifies a single resulting state for each combination of state and symbol. In non-deterministic automata there can be multiple results for a combination of state and symbol, and which state the automaton transitions to is random. Only deterministic automata can be used for runtime monitoring, as violations do not depend on chance.

In the context of run-time monitoring, the symbols of the alphabet are the events produced by the system. The set of final states is generally interpreted as a set of violation states, made up of the states in which the automaton ends up after the property modelled by the automaton is violated. This means evaluation of properties is very fast when using automata: when the monitor receives an event, the automaton is brought in a new state, as defined by the transition function. If this new state is a final state, the property is violated [11]. As an example, an automaton representing property 4 of the DSA example is shown in Figure 2.1.

The concept of the finite state machine can easily be extended to support more complex properties. For example, to support 3-valued logic, the FSM is extended to a 6-tuple, to include a set of satisfaction states: the states that signify that the property is satisfied. Another option is to modify the transition function, an example of which will be given in section 2.3.5. Note that this will usually also extend the tuple, as the modified transition function will require additional input.

Alternatives to the automaton-based technique described in this section include rewriting-based techniques, for example [15], and transformation-based techniques such as the one described by [16]. Note that just as with automaton-based techniques, these formalisms keep track of the current state of the monitor and have a concept of transitions between those states — they are only represented differently.

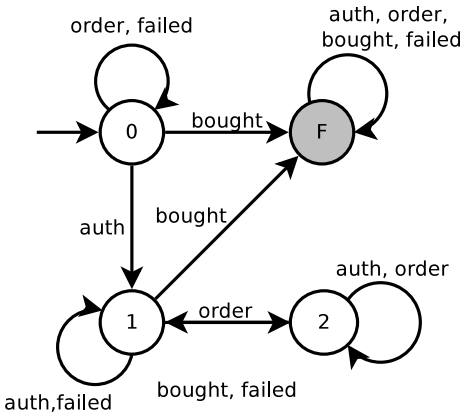


Figure 2.1: An automaton representing property 4 of th DSA example ("only after authentication and following a request can an order be successfully completed"). State '0' is the initial state and the shaded state 'F' is the failure state.

2.3.4 Specification formats

Because the formal specification must be written by a developer, the framework must read the specification in a human-readable format. The properties are then parsed, translated and for each property an equivalent automaton is generated. There are many different formalisms that can be used to specify a property.

Languages for specifying automata have been developed, such as used in the FSM-plugin of JavaMOP [4]. This is a very simple format for the framework, as no translation step is necessary. However, an automaton is not a natural way to directly express or reason about a property, so especially for complex properties it might become a very difficult task to write specifications in this format.

Alternatives are temporal logic languages such as LTL. Those languages are very suitable to express properties of traces, as they were originally developed for model checking, where properties are also verified on paths, albeit many infinite paths instead of one finite path. Many different temporal logic languages exists, including variants designed specifically for runtime monitoring, such as LTL3 [1] and RV-LTL [17]. Example property 1 expressed in LTL may look like "$(\neg$`bought`$)\ \mathcal{U}$ `auth`".

Two other languages that can be used are regular expressions [8] and context free grammars [18]. Although these languages were developed for a very different purpose, they lend themselves very well for specifying properties. Both languages are pattern matching languages and as such it should be specified whether a match means satisfaction or violation. For example, property 1, "only after authentication can an item be successfully bought" is expressed as an extended regular expression as "[`order failed`]* `bought .`*", where matching means violation.

2.3.5 Expressiveness of properties

Until this point it has not been defined what a property can express, and it has been implicitly assumed that only event orders are taken into account. This

type of property can be represented by a basic type of automaton: the finite state machine (FSM). Properties like these are not expressive enough to verify all kinds of behaviour though. This section describes two extensions of this model and their implications on the framework.

Sometimes, more information about the state of the monitored system is required to evaluate a property, for example property 2 of the running example. If the framework must be able to evaluate properties like this, the monitor must have access to properties of the running system, in this example the variable `credits`. One solution is to pass the value as an attribute of the event. If the monitor is in-line, or has an in-line component, an other option is to evaluate it in the observed system's control flow. Additionally, the automaton must be able to handle such more complex events. At the very least, the automaton must have a more complex transition function, where transitions can have a conditional guard. For example, such an automaton would look like the one in Figure 2.2, for property 2. In this case, the current amount of credits can be sent with the events, but in other cases the automaton might need some kind of memory to store some values used in the guards' expressions. Note that adding memory means that the automaton can describe more program states than just the finite set of automaton states Q.

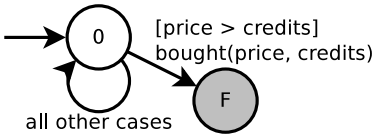


Figure 2.2: An automaton for example property 2. State '0' is the initial state and the shaded state 'F' is the failure state. The image shows the values sent with the event and a conditional guard.

A runtime monitor may also need to monitor timing constraints. A common example is a time-out check to detect possible deadlock. Another simple example is property 3 of the DSA. There are languages to express such properties, such as timed linear temporal logic (TLTL) [1] and metric temporal logic (MTL) [10]. The automaton must also be modified to support timing, which is demonstrated by [19]. The most dramatic effect of this extension is probably that property checks and automaton transitions do not only occur when an event arrives anymore, but also when a timer runs out. Note that this also requires the monitor to be implemented out-line.

2.4 Response to violations

When a property violation is detected, the monitor should act upon it. In the ideal case, the event triggering the violation is generated just before the actual failure, so the monitor can prevent it from occurring. This is not always possible, however, as it requires a high level of invasiveness. This aspect of runtime monitoring is not covered much in existing literature. Still, this section will list a number of possible actions the monitor can take. Their applicability is strongly dependent on the specifics and environment of the monitored system. In all cases however, the monitor should save information that can help developers

find the fault that caused the violation to occur.

- One of the most obvious actions is just killing or restarting the process. This prevents the application from presenting incorrect information to the user and the occurrence of other harmful effects the program can have. The end user still experiences a software failure in the form of a crash, so this is often not desirable.

- Another generic course of action is to freeze execution and notify a system administrator. This administrator must then take some action to resolve the problem. This can include presenting a list of options from which the administrator can choose, possibly made up out of some of the other actions described in this section.

- A common recovery method is the traditional concept of checkpointing and rollback, which allows the program to continue from the last recorded point where the program was in a correct state. If the process has interaction with other software, however, only resetting the failing program might not successfully restore correct execution.

- Certain tools [4, 20] allow the user to specify code that is executed when a certain property is violated. This allows a customizable response and can be very powerful, especially when the code is inserted in-line. Property-specified in-lined response code can be used to recover from specific failures, where more general methods might not be able to let the execution continue. The downside of this method is that a lot of specific code needs to be written, which is in turn prone to errors.

- For specific cases there might be better alternatives, such as the method proposed by Simmonds for web services written in BPEL [11]. A BPEL process is specified by its interactions with other web services, so monitoring this communication gives a complete image of the state of the software. A labelled transition system (LTS) is generated from the BPEL file to maintain this state, and additional transitions are added to the LTS that describe actions that undo the effects of an existing transition. If a violation occurs, these 'compensation actions' can be used to revert to a correct state. As opposed to a simple rollback, this method does bring the environment back to the matching state as well.

2.5 Monitoring multiple-component software

When the system that is monitored consists of various separate components, this has some implications on which of the techniques described above can be used. If those components are not all written in the same language, this becomes even more complex.

The naive method of monitoring a multiple-component system would be to add a separate monitor to each component. This imposes a serious restriction on the properties that can be checked, however, as interaction between the components cannot be monitored this way [8]. To illustrate this, consider a front-end for the digital store application. A property related to both components might be "after the 'buy' button is pressed in the front end, the DSA must respond

with the `order` event, followed by either `bought` or `failed`". This cannot be verified by separate monitors, as the property contains events occurring in both components.

When the system is not instrumented and is therefore only observed from its components' interaction with the environment and other components, most issues described in this section do not apply. In this case, the system can be considered as a whole, except that interaction between the various components can —and probably must— also be observed.

To monitor the interaction between components in an instrumented system, we need an out-line monitor in a separate process, which receives events from all components and can therefore check properties involving multiple components, such as in the example above. Likely a hybrid approach needs to be taken, where inline components of the monitor generate the events and send those to the out-line component via some inter-process communication method, such as CORBA [8, 9]. It is possible that not all properties cover interaction between components, but that some are only related to one component, such as property 2. In this case, the in-line monitor parts can verify those properties, instead of the out-line part. If this design is chosen, communication overhead can be reduced by only sending the events relevant for the properties related to multiple components to the out-line part of the monitor.

If the properties considering multiple components are evaluated remotely, all required state information must be sent with the events to the out-line component (see Section 2.3.5). Mizzi [9] proposes a method where the out-line component can send evaluation requests to the in-line components, which have better access to the state information of the component in which they are embedded. In this situation, the event generator does not need to know anything about the properties, but communication overhead is most likely higher.

2.5.1 Multiple languages

If the various components are written in different languages, all of the above still holds. The main problem, though, is that instrumentation is language dependent. This means that for each language used in the system, separate code generators need to be implemented. How big this problem is, depends on how invasive the monitoring approach is. At the minimum only code-generators for the event-generation code must be provided for various languages, but if the property monitors are also implemented in-line, the automata-generators need to be implemented multiple times for various languages too. Finally, depending on the way the monitor responds to violation, language dependency may also complicate code generation for the violation handlers.

More importantly, it becomes much harder to develop a formalism that can be used to define all properties, because some elements of this formalism may be strongly tied to the language. For example if expressions containing variables in the code need to be verified or when additional information needs to be specified to describe the placement of the generated code.

2.6 An overview of existing solutions

A list of existing runtime monitoring tools and some of their properties are shown in Table 2.1. TraceMatches and J-LO are extensions of AspectJ, an aspect-oriented programming extension for java, that can be used to specify actions to be executed after a sequence of pointcuts, instead of just at a pointcut. TraceMatches uses regular expressions to specify those sequences and J-LO uses LTL.

system	language	logic
TraceMatches	Java	regex
J-LO	Java	LTL
MaC	Java	PastLTL
PathExplorer	Java	LTL
Hawk	Java	Eagle
MOP	Java	many
Temporal Rover	C/C++, Java	MTL
jContractor	Java	contracts
E-Chaser	any	regex
polyLarva	C, Java	custom

Table 2.1: An overview of existing solutions. Of all of those solutions, only MOP seems to see active development.

MaC, PathExplorer, Eagle and MOP are runtime monitoring frameworks that generate outline monitors from a specification. They have implementations in Java called JavaMaC, JPaX, Hawk and JavaMOP, respectively. MOP also has an instance for the Robot Operating System and an instance for monitoring system buses using FPGA-based monitors, but not for other general purpose programming languages. Temporal Rover is a runtime monitoring framework that generates inline monitors from annotations added to the source code of the monitored software, instead of from a separate specification. It also supports offline monitoring by generating tests from the annotations.

jContractor is implemented as a design-by-contract library for Java. Calls to the library methods are detected when the classes containing those calls are loaded. At this point, jContractor does on-the-fly byte code instrumentation to add code for checking the contracts specified in the calls.

Some of the above tools support multiple languages and others are generic frameworks that only have a concrete implementation in Java, but can be extended to more languages. However, none of those tools can monitor a system written in multiple languages. The only tools that are currently capable of doing this are E-Chaser [8] and polyLarva [9]. E-Chaser is built on the Composition Filter Model, a system in which messages that are exchanged between objects (such as method calls) are passed through 'filters' which allow access to the control flow in a similar way as point-cuts. E-Chaser adds verification filters to the system to provide runtime monitoring. As the Composition Filter Model is language-independent, so is E-Chaser. polyLarva takes a more traditional approach: it generates inline and outline monitors from a specification file and uses aspect-oriented programming to add code to the monitored software. It

has multiple code generators in order to support multiple languages.

Chapter 3

Sheepdog+ architecture

This chapter describes the architecture of the Sheepdog+ runtime monitoring framework, designed to satisfy the requirements outlined in Section 1.1. In order to do this, the general runtime monitoring approach described in Chapter 2 is taken: the code for the monitor is generated from a formal specification and the monitored software is instrumented in order for the monitor to obtain information about the monitored software's execution.

Because of this approach, this chapter describes two architectures: firstly, in Section 3.1, the architecture of the monitor and its interaction with the monitored software is explained. The architecture of the code generation framework is outlined in Section 3.2, after which Section 3.3 describes the languages and formats of the specification and other input files. Finally, the main design choices are highlighted in Section 3.4.

3.1 Monitor architecture

Sheepdog+ has to work with software systems made up of many components and needs to monitor the interaction between those components (requirements 1 and 4). As explained in Section 2.5, this requires the system to use outline monitoring. Inline monitoring will not be considered. This choice is discussed in further detail in Appendix A.

For the architecture not to put limitations on the types of properties monitored (requirement 6), the system must be very flexible in what input it can get from the monitored system. Otherwise the data required by a certain property might be unobtainable.

Section 3.1.1 gives an overview of the monitor architecture designed on these principles, which is followed by a more detailed description of its components.

3.1.1 Architecture overview

The monitored software can provide data to the monitor by filing reports to the so-called incident reporter module (see Figure 3.1). The incident reporter is a configurable relaying component: it handles reports based on their type and its configuration, for example by writing a logging request to a file.

Notable events occurring in the monitored system are exposed to the monitor using a report type that contains a single string: the "event label". This event label is a unique name distinguishing the many event types. The incident reporter sends those event labels to the monitor manager. The monitor manager then evaluates all properties by triggering transitions labeled with that event label in all automata. Then it reports any violations to the incident reporter and returns a verdict. Reporting violations to the reporter allows for the flexible handling requested in requirement 9.

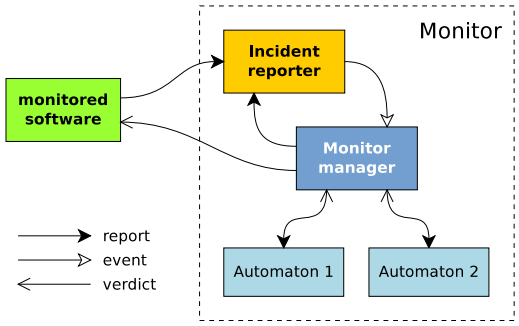


Figure 3.1: The interactions in the system as a result of the monitored software filing a report.

The overview given by Figure 3.1 is a simplified form of the architecture of the system. In practice, the automata need to be evaluated in an other thread or process than the monitored software to monitor multiple components. Therefore, the monitoring system is divided into a client part which is integrated into the monitored software, and a server part that runs in a separate process. The incident reporter is then not called directly by the monitored software, but through a sort of proxy that is provided by the client part of the monitoring system, as shown in Figure 3.2.

As Chapter 2 explained is common in runtime monitoring, additional statements are inserted in the source code of the monitored program (instrumentation). Those statements call a procedure that sends a message to the server process, which is provided by a language binding (the PHP or Shell binding in Figure 3.2) in order to support software written in multiple languages (requirement 2). When the server process receives a message, it is translated into a call to the incident reporter. The incident reporter processes these reports, as described above. Clients receive a reply from the server for every message sent. This reply indicates whether a violation has been found, based on which the client can decide how to proceed. The language bindings can achieve the communication with the server either through bindings to the communications library that is also used in the server, or through a native implementation of the protocol described in Section 3.1.3.

Note that the automata in Figure 3.2, as well as the code that aggregates all server-side components into a server application, are generated by the code generation framework. More details on code generation follow in Section 3.2.

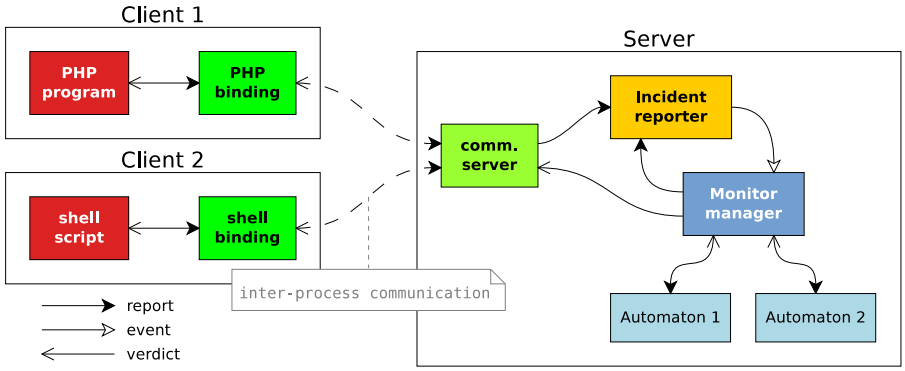


Figure 3.2: The full process resulting from a filed report, from the monitored program to the internals of the Sheepdog+ server.

Uninstrumented observation

In some use cases, it may be preferred to leave the system unmodified and only observe interactions of the system with its environment, instead of instrumenting the monitored software. This can be done with Sheepdog+, but the solution is not ready-made: an external client program must be written that observes the monitored software, translates observed behaviour into events and sends those events to the server, just like an integrated client would. This design choice has been made because this kind of observation is highly case-specific: the interaction can be through a TCP connection, a message broker, API calls, standard output or something completely different.

3.1.2 Incident reporter

The incident reporter is the central component of the Sheepdog+ system. Whenever something significant happens in either the monitored system or in the Sheepdog+ system itself, this is signalled by posting a report to the incident reporter, which decides how to respond.

The reports sent to the incident reporter consist of a report class, a message and optionally a 'context dictionary'. The report class is a string identifying what kind of report it is, based on which the incident reporter decides how to handle the report. Each report contains a message in the form of a simple string, which can be used, for example, to log the report. Additional information about the report can be provided via the context dictionary, an associative array with strings as keys. The values in this array can be of any type. The message is kept separate from the context, so the system can rely on a message being present and being a string. This is useful for logging and error reporting in case something goes wrong while handling the report.

Two other likely useful types of data that might be expected in such a message are a timestamp and information about the origin of the message. The timestamp is not needed, because tracking of time is done server-side (see Section 3.1.4). If the message origin is required, it can be provided in the context dictionary in the required form, be it a process ID, or an indicator of what component type the message originates from.

The incident reporter processes a report by sending it to handlers registered to the class of the report. Which handlers are registered to a report class can be customized through a configuration file. Details on the syntax and semantics of this file can be found in Section 3.3.4. Examples of handlers are a log writer or a handler that connects to a mailer daemon.

By default, the eight traditional log levels as specified by the IETF RFC 5424[1] are available as report classes, as well as the class EVENT, to which by default a handler is registered that interprets the messages as event labels and sends them to the monitor manager. Additional handlers can be registered to the EVENT class as normal. The RFC 5424 classes don't have a default handler specified. Additionally, custom report classes can be specified. Names of custom report classes should only include alphanumeric characters and underscores.

A hierarchy of report classes can be made by specifying a class as a subclass of another class. Additionally classes can be added to groups, to which report handlers can be registered as well. Other groups can also be added to a group. Figure 3.3 illustrates the relations between report classes, groups and handlers.

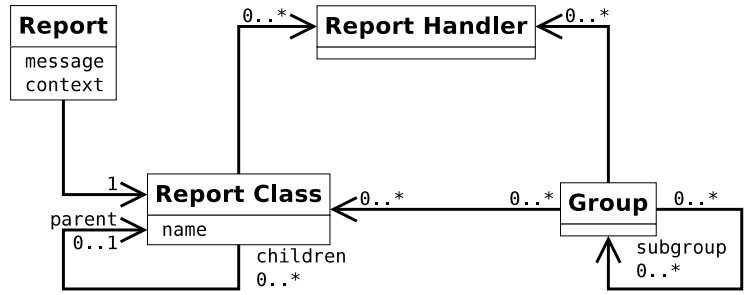


Figure 3.3: A conceptual class diagram showing the relations between report classes and groups.

If a report in a certain class is received, the handlers registered to its parent class and the groups it is in are triggered as well as the handlers registered directly to the class. There are two differences between the subclassing and grouping mechanisms.

- Firstly, the class hierarchy is explicit in the client, whereas groups are only known to the server. This is because subclassing is determined from the class name: MYCLASS:MYSUB is a subclass of MYCLASS.
- The second difference is that, due to the naming method, the class hierarchy only allows "single inheritance", while a class can be in any number of groups.
- Perhaps more important than the technical differences is the difference in usage: subclassing can be used to show intent and is independent of the incident reporter configuration. Groups, on the other hand, are only used to ease configuration.

[1]Debug, info, notice, warning, error, critical, alert, emergency. For more information see `http://tools.ietf.org/html/rfc5424`.

Report processors

A useful application of this incident reporter architecture is to implement a handler that calls some sort of processor with data derived from the report. The monitor manager (see Figure 3.2 and Section 3.1.4) is such a processor, but also for example an outlier detection algorithm can be added to the system via this mechanism. When a problem is detected by a processor, the processor can report this by filing a new report to the incident reporter in a class that indicates a failure[2]. With this mechanism, new verification methods can easily be added to the system, while handling all their logging and error reporting uniformly.

3.1.3 Client-server architecture

Sheepdog+ features a client-server model, where the incident reporter and monitoring component reside in the server process and each separately running monitored component is a client. This client-server model allows a many-to-many relationship between clients and servers. As described in Section 2.5, a server needs to be able to handle multiple clients in order for the monitor to detect issues in the interaction between components (requirement 4), i.e. between clients. Allowing a client to connect to multiple servers is useful when there are multiple instances of the same component running (requirement 5). Events from different instances can be kept separate by sending them to a server specific to the instance from which the events originate. This prevents monitors from giving false positives due to receiving events from different instances. Such components still send events to a central server which can monitor interaction between these and other components.

Language bindings provide routines to the clients that mimic a call to the incident reporter, but actually send a message to all connected servers. Once received by the server this message is translated to an actual call to the incident reporter. The client bindings thus provide a proxy to the incident reporter. The interface between client and server is placed at this point in order to minimize the load on the client, and therefore the overhead incurred by Sheepdog+: all the client has to do is send a simple message.

Communication between client and server is done in an asynchronous request-reply pattern: for every message the server receives, it sends a message back to the client, informing the client of any detected violations. Because the communication is asynchronous, the client does not have to wait for this message, but can continue its execution, minimizing the performance impact of runtime monitoring. The client can, however, synchronize with the server by waiting for the messages to arrive.

Message protocol

The inter-process communication protocol used by Sheepdog+ is based on the ZeroMQ Message Transport Protocol (ZMTP). This is chosen because it is an easy protocol to work with, and has implementations in many languages. This

[2]One of the RFC 5424 classes, such as ERROR can be used for this, but it is usually better to use a custom class specific to the processor, so the report can be handled more flexibly and specific.

makes it much easier to add support for new languages to Sheepdog+ (requirement 3), compared to having to work with low-level inter-process communication systems such as pipes and message queues, and it is much more lightweight than using a message broker.

With ZMTP, a message is sent as raw data, preceded by the length of the data and a byte used for flags. One of those flags is the 'more' flag. If it is set, this indicates that the next message is actually part of the same message, creating a multi-part message. The triplet of length, flags and data is called a frame and a message is made up of one or more of those frames, where the last one does not have the 'more' flag set, while all others do. For a full definition of ZMTP, refer to `http://zmtp.org`.

The messages sent from the client to the server consist of two or three frames, plus a leading empty frame, which is included for compatibility with certain ZMTP implementations. The payload of all of those part is a plain string. For the first message part this string is the report class in which the report is filed and the second part contains the report message. The optional third part contains the context dictionary encoded as plain JSON. This matches the way in which reports are filed locally, which makes it easy to deserialize the messages.

The server's responses consist of one or two message parts. The first part is the verdict, a boolean value that is false when problems were detected as a result of the received message and true otherwise. The optional second part is a string formatted as a JSON object that may contain additional details on encountered problems, similar to the context dictionary in the client-to-server message.

3.1.4 Run-time monitoring

The monitoring component of Sheepdog+ uses an extended type of deterministic finite state machine to monitor properties against the monitored system. The monitor manager maintains a list of all of those automata, and makes sure all are checked for transition when it receives an event label from the incident reporter. For each automaton that has reached a violation state, the monitor manager files a report to the incident reporter. The exact class in which this report is filed indicates the precise violation, but it is always a subclass of VIOLATION, so violations can also be handled in a general way. Finally, the monitor manager returns a verdict, which is simply a boolean value that is true when none of the monitors has entered a violation state and false otherwise.

Expressiveness of the automata

In addition to event labels, the Sheepdog+ monitoring component can also process real-valued numbers. Those numbers can be stored as variables, used in boolean expressions called guards and given a value with an assignment. An assignment $(x, f(X))$, sets the variable x to the value that is given by evaluating the real-valued function f, where X is a set of variables that may include x. The set of all guards using a set of variables X is denoted $G(X)$ and the set of all assignments of variables in X to functions on variables in X is denoted $A(X)$.

In order to satisfy requirement 7, which dictates that Sheepdog+ needs to be able to detect performance problems, Sheepdog+ needs to be to able to

verify timed properties that check whether an operation takes too long. It does this using clocks, a special type of variable that gets incremented over time to represent a time in seconds since an epoch, which is defined by setting the clock to zero (or another value) using an assignment.

The automata used by Sheepdog+, of which an example is shown in Figure 3.4, are timed deterministic finite state machines with variables, defined as follows:

Definition 5. *A Sheepdog+ automaton is represented formally by a 7-tuple* $(Q, \Sigma, C, X, \delta, q_0, F)$*, where*

- Q *is a finite set of states,*
- Σ *is a finite set of symbols, called the alphabet of the automaton,*
- C *is a set of clocks,*
- X *is a set of variables,*
- $\Delta \subset Q \times (\{\tau\} \cup \Sigma) \times G(C \cup X) \times Q \times 2^{A(C \cup X)}$ *is a transition function, where a transition with* τ *instead of a symbol from* Σ *is triggered by the passing of time instead of by reading a symbol.*
- $q_0 \in Q$ *is the initial state,*
- $F \subseteq Q$ *is the set of violation states.*

There are restrictions on the elements that can be in the subset Δ*, in order to guarantee that the automaton is deterministic: if* $\delta(q, e)$*, where* $q \in Q$ *and* $e \in (\{\tau\} \cup \Sigma)$*, denotes the set of transitions* $(q, e, g, q', a) \in \Delta$ *(where* $g \in G(C \cup X)$*,* $q' \in Q$ *and* $a \in 2^{A(C \cup X)}$*), then for each pair* (q, e)*, for all possible values of the variables in* X *and the clocks in* C*, there is precisely one transition* $(q, e, g, q', a) \in \delta(q, e)$ *for which* g *evaluates to true.*

The differences with the deterministic finite state machine described in Section 2.3.3 are the addition of clocks and variables, and changes to the transition function for using those. The symbols making up the alphabet of the automaton are event labels, though not necessarily all event labels received by a server are in the alphabet of each automaton. An element $(q, e, g, q', a) \in \Delta$ represents the transition that is taken when the automaton is in state $q \in Q$, the event $e \in \Sigma$ occurs and $g \in G(C \cup X)$ evaluates to true. This transition brings the automaton to the state $q' \in Q$ and evaluates all assignments in $a \subset A(C \cup X)$.

A transition (q, τ, g, q', a) can be taken without an event occurring. It is taken as soon as the guard g is evaluated and yields true. The guards of τ-transitions from the current state are evaluated whenever the clocks are auto-incremented, so a τ-transition should always have a guard using a clock.

Response to detected failures

Sheepdog+ provides two methods for handling violations, one synchronous and the other asynchronous. In the first method, the client synchronizes with the server by waiting until the server's response has been received. This should be done immediately after sending an event, because otherwise the receiving statements might not be reached in the case of an unforeseen incorrect execution.

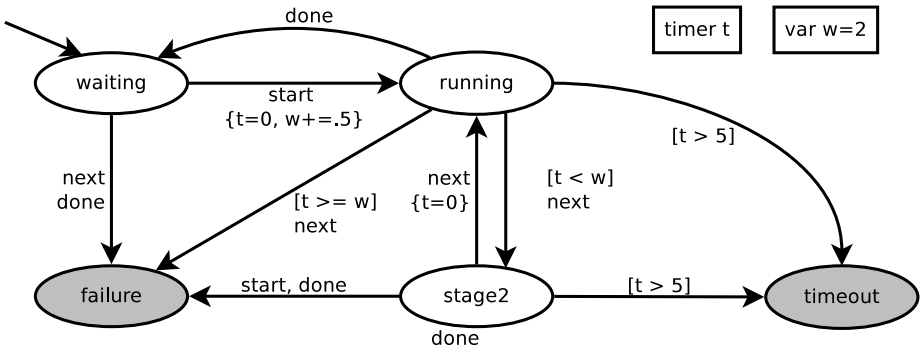


Figure 3.4: An example of a Sheepdog+ automaton. Guards are written between brackets and assignments are written between braces. τ is not written explicitly on the transitions, but every transition without an event label is a τ-transition. The shaded states are violation states.

If the response indicates that a property has been violated, the client can execute some custom code to prevent or recover from the failure (requirement 11). This is the approach taken by tools such as JavaMOP. Although this is a very powerful and flexible method, it also has some serious drawbacks: additional code has to be developed and added to the code base. This is highly invasive and might introduce new errors.

The other, asynchronous method is to add a handler to the incident reporter for reports issued to the VIOLATION report class or a specific subclass. An example of an action that can be executed asynchronously is to kill the process in which the violation occurred. These reports are handled before the server responds, so if the client is known to wait for the server response at some point, i.e. synchronizes with the server, these asynchronous actions are executed before that point in the client's execution.

Because the incident reporter can be configured to handle these reports as the user wishes, the asynchronous method gives great flexibility in handling violations, as is needed per requirement 9. This mechanism can also be used to satisfy requirement 10, by executing actions that notify the staff of the error, such as sending an e-mail.

Figures 3.5 and 3.6 show client-server interactions including an both asynchronous action in the form of logging the event and a synchronous reaction via a synchronization point.

3.2 Code generation framework architecture

The purpose of the code generation framework is to set up the entire monitoring infrastructure, based on one or more specification files. This includes generating the automata and servers, as well as instrumenting the existing code. The code generation framework uses an extensible design based on plugins, to satisfy the need for flexibility.

See Figure 3.7 for a schematic representation of an execution of the code generation framework. The first step is to parse the specification files and generate the source code for the automata. Section 3.2.1 elaborates on these steps

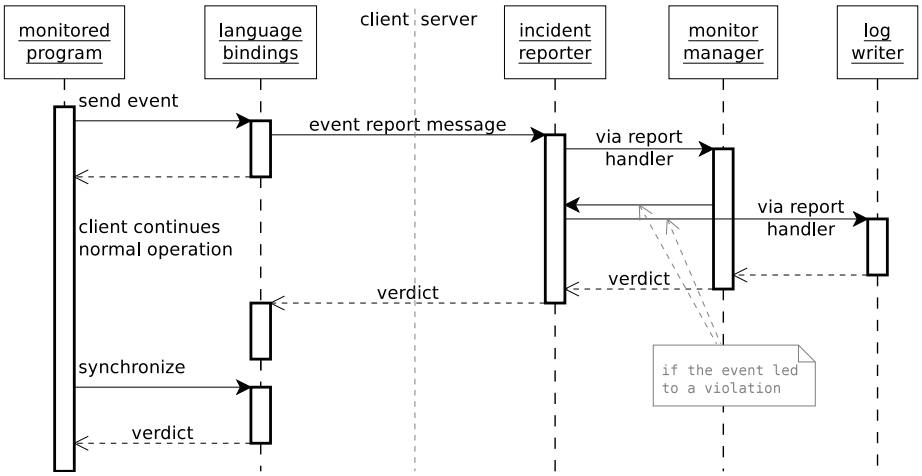


Figure 3.5: An example client-server interaction where the client tries to synchronize with the server, after the server has completed processing the message sent by the client. It shows that the logging call resulting from the failure as an asynchronous action, is completed before the synchronization point. What exactly happens with the message from the server is determined by the ZMTP implementation, but the message is retrieved in the synchronization function.

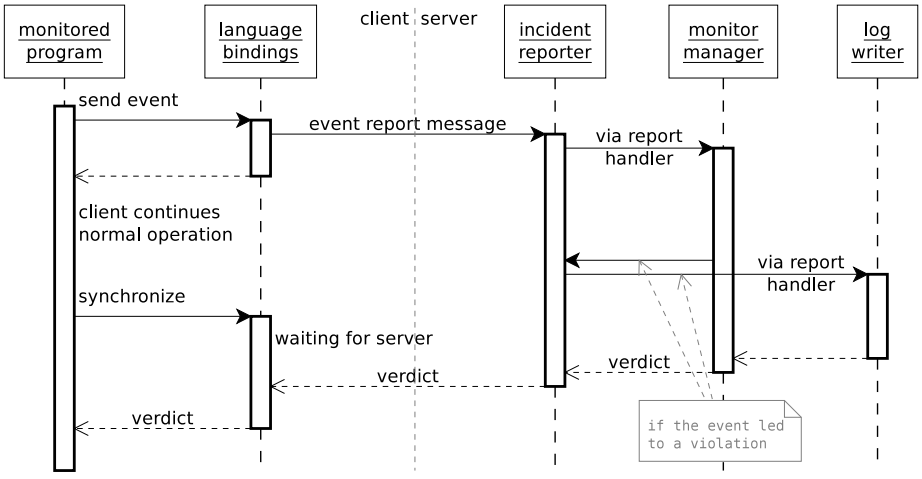


Figure 3.6: An example client-server interaction where the client tries to synchronize with the server, while the server is still processing the event. In this case the logging call is also completed before the synchronization point, because the server response only happens after all actions triggered by the event have completed.

of the process. Then the server is generated, by extending an abstract server class with code that instantiates the newly generated automata, registers them to the monitor manager and creates report classes for all violations that can occur.

If only manual instrumentation or passive observation is used, the code generation framework is done at this point. Otherwise the framework proceeds with instrumenting the source code of the various components using the appropriate language plugins. The input for the instrumentation step is:

- information about which servers to connect to,
- the source code of the program that is instrumented,
- an instrumentation script, which contains information about which events need to be generated under which conditions.

Instrumentation is very language-dependent, so after the instrumentation script is parsed, the actual work is delegated to a language plugin, so many languages can be supported and more can easily be added (requirements 2 and 3). This plugin ensures the corresponding language bindings (mentioned in Section 3.1) can be accessed by the instrumented program, it inserts statements for connecting to the servers and calls to the language bindings that send events and possibly other reports to the servers.

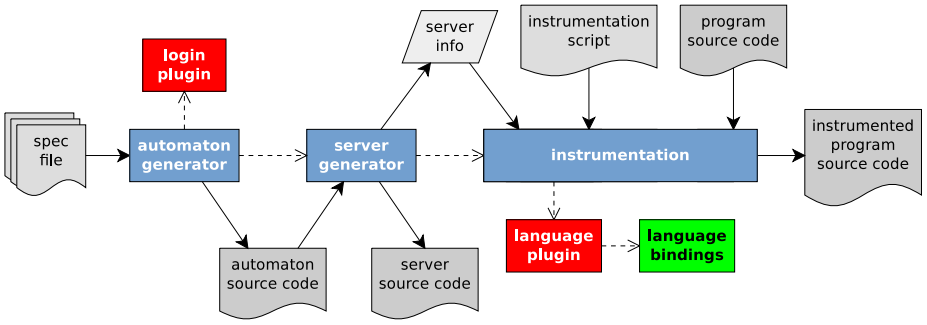


Figure 3.7: The full code generation process.

3.2.1 Automaton generation

Sheepdog+ supports different input languages for properties, and makes adding more languages easy through a plugin-based design, in order to make it easy to specify the wide variety of property types described by Requirements 6 and 7. In a specification file multiple properties can be described, each labelled with the logic in which they are written, so the generation framework knows how to process them. A detailed description of these files is given in Section 3.3.1.

Figure 3.8 illustrates the specification file parsing and monitor generation steps of the code generation framework in more detail. First the file parser decodes the specification file. For each property in the file, a logic plugin is selected based on the logic in which it is specified. The plugin takes two actions: first it parses the plain text property into an abstract syntax tree. Then, using

the other information about the property in the specification file, this abstract syntax tree is transformed into 'automaton representation' — a data structure similar to an abstract syntax tree, that can be used to generate the code. Note that the FSM logic plugin works slightly different from the others: it does everything in one step, because the 'automaton representation' is identical to the abstract syntax tree of a property expressed in FSM.

3.3 Input files and languages

Various files are required to generate and run an instance of the monitoring system, such as specification files and incident reporter configuration files. The contents of some of these files are written in special languages. All of these file formats and the syntaxes and semantics of these languages are described in this section. All grammars listed in this chapter are described using the ANTLR[3] flavor of BNF. Note that, although it is not strictly required by this BNF dialect, the names of terminals are always written fully in capital letters, whereas names of non-terminals don't include capital letters.

3.3.1 Specification files

The properties monitored by Sheepdog+ are described in one or more specification files. The contents of these files are a JSON array of objects representing one property each. An example is shown in Listing 1. JSON has been chosen because it supports many different data structures, is widely supported and is well known by developers. Those JSON objects must at least include the field 'logic' and either 'property' or 'file':

logic specifies in which language the property is specified and is used to determine how to process the other fields,

property a formula expressing the property in the given logic,

file the path to a file containing the property, relative to the specification file itself. This is especially useful for rather verbose logics.

The following sections describe the syntax and semantics of the 'FSM' and 'regex' logics. For properties specified in either of those logics, additional logic-specific fields must be present in the JSON structure. This can be seen in Listing 1 and is detailed in the corresponding sections. The FSM logic is chosen because it is a description close to the inner workings of the automata, and can therefore express any property that can be verified by those automata, while other languages might have limitations. The regex logic was added as a second logic, because it is very easy to use and sufficiently strong in many cases. Other logic can still be added. For example, TLTL [1] would be a good candidate, as the LTL family is used a lot in model checking because it a very powerful logic, and in particular TLTL would be interesting, because it is a strong language capable of specifying timed properties.

[3] `http://www.antlr.org/`

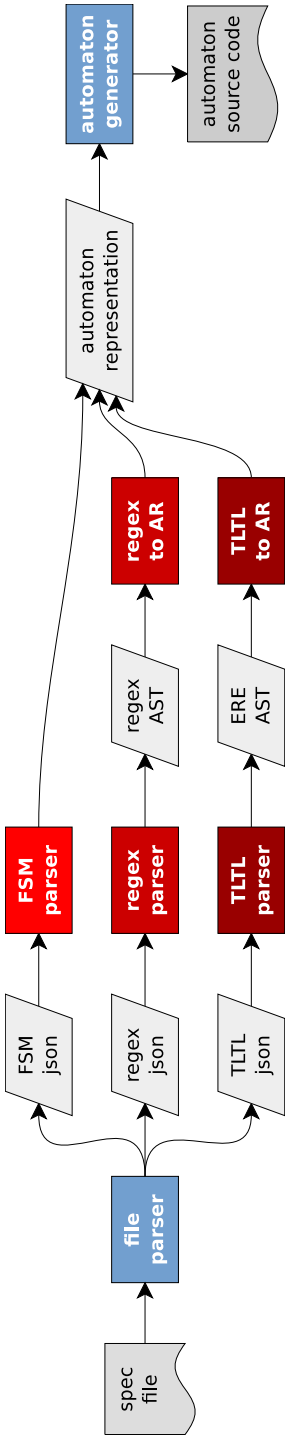


Figure 3.8: The process that generates monitor source code from a specification.

```
[{
    "logic": "regex",
    "property": "(a+ b) | e",
    "match": "violation",
    "filter": ["a", "b", "c", "e"]
},{
    "logic": "FSM",
    "file": "property2.fsm",
    "initial": "waiting",
    "violating": ["failure", "timeout"]
}]
```

Listing 1: An example specification file demonstrating a regex property and an FSM property defined in another file. Possible contents of "property2.fsm" are shown in Listing 2.

3.3.2 FSM logic

With the FSM logic, a property is specified directly as an automaton. In other words, the FSM logic provides a textual representation for automata. Hence the name: FSM stands for Finite State Machine. An example property specified in FSM is shown in Listing 2. Because of the similarity between this logic and the way properties are represented in the system at run time, all properties that can be monitored by Sheepdog+ can be expressed in the FSM logic, including timed properties.

Additional fields required in the specification file for a property specified in the FSM logic are 'initial' and 'violating'. The field 'initial' must be a string and determines the initial state of the automaton, while 'violating', an array of strings, defines the violation states. See Listing 1 for an example.

The grammar for the FSM logic is shown in Listing 3. In addition to specifying states and their transitions, clocks and numeric variables can be declared (respectively using the `@timer` and `@var` keywords) and guards and assignments can be specified on transitions. The numeric variables contain floating point numbers and are guaranteed to be compatible with the values of clocks. For specifying guards and assignments (`action` in the grammar) the FSM logic includes a complete expression language which is the full subset of the C++ expression language that is meaningful in this context. If in one state multiple transitions are specified with the same event, at most one can be specified without guard and it must be specified last. The guards of the other transitions are evaluated in the order in which they appear in the specification.

Note that in the example specification in Listing 2 has transitions from the violation states `fail` and `timeout`. Those are put in place so that, if the program is allowed to continue after a violation, subsequent violations can also be detected. Of course it is not guaranteed this works properly, because this assumes that the monitored software is in a certain state after a violation occurs.

```
@timer t
@var w = 2

waiting [
            start   ->  running {t = 0, w+=.5}
            next    ->  failure
            done    ->  failure
]
running [
            start   ->  failure
    [t < w] next    ->  stage2
            next    ->  failure
            done    ->  waiting
    [t > 5]         ->  timeout
]
stage2 [
            start   ->  failure
            next    ->  running {t = 0}
            done    ->  failure
    [t > 5]         ->  timeout
]
failure [
            start   ->  running {t = 0}
]
timeout [
            start   ->  running {t = 0}
]
```

Listing 2: Example FSM specification demonstrating a timeout check, that matches the second entry in Listing 1. It is the same property as represented by the automaton in Figure 3.4, except that this property has added transitions from the `failure` and `timeout` states.

```
automaton      : element (NL+ element)* ;
element        : declaration | state ;

declaration    : '@' var_type BLANK variable ( ',' variable )*;
variable       : IDENTIFIER '=' addition ;
var_type       : 'timer' | 'var' ;

state          : STATE_NAME '[' NL* transitions? NL* ']' ;
transitions    : transition ( NL+ transition )* ;
transition     : guard?  EVENT? '->' STATE_NAME  action? ;
guard          : '[' logical_or ']' ;
action         : '{' expression '}' ;

expression     : expression , assignment ;
assignment     : ( IDENTIFIER  assignment_operator )?
                                              ternary_expr ;
ternary_expr   : logical_or ('?' expression ':' ternary_expr)?;
logical_or     : ( logical_or  '||' )?  logical_and ;
logical_and    : ( logical_and  '&&' )?  equality ;
equality       : ( equality  ( '==' | '!=' ) )?  comparison ;
comparison     : ( comparison comparison_operator )? addition ;
addition       : ( addition  ( '+' | '-' ) )?  multiplication ;
multiplication : ( multiplication  ( '*' | '/' | '%' ) )?
                                               prefix_expr ;
prefix_expr    : ( '!' | '-' | '++' | '--' )*    postfix_expr ;
postfix_expr   : primary_expr  ( '++' | '--' )* ;
primary_expr   : '(' expression ')' | NUMBER
               | 'true' | 'false' | IDENTIFIER ;

NL             : '\n' | '\r' | '\r\n' ;
BLANK          : ( ' ' | '\t' )+ ;
IDENTIFIER     : LETTER (LETTER | DIGIT | '_')* ;
NUMBER         : (DIGIT* '.')? DIGIT+ ;
STATE_NAME     : IDENTIFIER ;
EVENT          : IDENTIFIER ;
```

Listing 3: The grammar of the FSM logic. Note that newlines are significant, but other whitespace is ignored, except for separating tokens, e.g. `--x` decrements `x`, while `- -x` is parsed as negating `x` twice (essentially a no-op). Also note the required whitespace in the rule 'declaration'.

3.3.3 Regex logic

Sheepdog+ also supports the "regex" logic, which is a version of regular expressions modified to describe sequences of events instead of sequences of characters. The first property in Listing 1 is an example of a property expressed in the regex logic. The main advantages of the regex logic are that (a) it is easy for specifying patterns in sequences of events, (b) it can be used to either specify correct or incorrect behaviour, (c) in general, regular expressions are already well-known by developers. A property specified in the regex logic is expressed as a pattern in a stream of events. A pattern can either describe correct or incorrect behaviour. If it specifies correct behaviour, the property is violated when the execution trace does not match any prefix of the pattern. If the pattern specifies incorrect behaviour, the property is violated when the full pattern is matched. Following weak view semantics, matching a prefix of the pattern is not violating the property. As this logic only does pattern matching, it does not support timed properties and has no access to guards and assignments.

The grammar of the regex logic is listed in Listing 4. There are two important differences with POSIX (extended) regular expressions. Firstly, the terminals are not single characters, but event labels and thus character strings. The second difference is that the only syntactic value of whitespace is to separate event labels. For any other purpose whitespace is ignored. Also note that not all metacharacters in the POSIX standard for extended regular expressions are available. For example, anchors are unavailable, because the specified pattern is always checked against the entire execution trace. Not all regular expressions satisfying the grammar in Listing 4 are correct properties. For example, `a+ a` will never be matched, because the `+` operator is greedy.

```
regex          :  alternation ;
alternation    :  concatenation ( '|' concatenation )* ;
concatenation  :  multiplicity+ ;
multiplicity   :  primary_expr ( '?' | '*' | '+' )? ;
primary_expr   :  '(' regex ')' | set | EVENT ;
set            :  '[' '^'? EVENT+ ']' ;
```

Listing 4: The grammar of the regex logic. Note that whitespace is ignored, except that it separates tokens. Events in a set or concatenation *must* even separated by whitespace, unlike in regular expressions working on character strings.

Specification file fields

In the specification file, two additional fields need to be added to the JSON object: 'match' and 'filter'. The 'property' field or the file to which the 'file' field points only includes the pattern. The field 'match' must be a boolean. If it is false, matching the complete pattern is interpreted as a violation. If 'match' is true, the pattern and all of its prefixes are interpreted as correct behaviour, and any *mismatch* will be a violation of the automaton. The 'filter' field is a list of event labels (that is, strings), that indicates which events need to be

considered: any event not in the list will never cause a mismatch, nor will it be considered a match for a `[^ ]` sub-expression.

3.3.4 Incident reporter configuration files

Just like specification files, incident reporter configuration files are specified in JSON. The root structure must be an object with the fields 'handlers', 'groups' and 'classes', each of which must be an array of zero or more objects as specified below. An example of such a file is given in Listing 5. Report classes and report class groups are not explicitly declared in this file: the first occurence of a name defines a report class or group (depending on the context) and subsequent occurences may modify this definition.

This file will include names of report classes and report class groups. If those did not exist yet they are created, otherwise they are modified.

For each object in the 'handlers' array, a report handler is created of the type given in the mandatory field 'type', which is specified as a string. A string field 'name' must also be present. This name will be use to refer to the handler in the other parts of the file. Other fields are parameters specific to the type of handler created.

The objects in the 'groups' array must have a string field 'name' and may optionally have the fields 'supergroups', 'subgroups' and 'handlers', all of which must be arrays of strings if present. As the namings suggest, this entry creates or modifies a group with the given name by adding it to supergroups and adding subgroups and registering handlers to it.

Similarly, the objects in the 'classes' array must have a string field 'name' and may optionally have the fields 'groups' and 'handlers', both of which must be arrays of strings. These entries create or modify a report class with the given name, add it to the given groups and register the given handlers to it.

```
{
    "handlers": [
        {"name": "logger",
         "type": "FileLogger", "file": "log.txt"}
    ],
    "groups": [
        {"name": "LOGGING", "handlers": ["logger"]}
    ],
    "classes": [
        {"name": "ERROR:EXTERNAL"}
    ]
}
```

Listing 5: Example incident reporter configuration file that adds a handler to the predefined LOGGING group and adds a subclass to ERROR. Note that this class is indirectly a member of the LOGGING group via its parent class, and will thus be logged to "log.txt".

3.4 Summary

This chapter described the design of a runtime monitoring system called Sheepdog+, that uses a code generation framework to create the monitors. The code generation framework as well as the monitors themselves are designed to be very flexible and extensible in order to be compatible with many systems that need monitoring.

Information about the execution of the monitored system is obtained through statements added to the monitored code that send 'reports' to the monitor. This uses a client-server design based on asynchronous communication. A server can take any kind of report and processes them in a configurable manner. Different methods for handling such a report can be added by creating report handler plugins. One type of reports contains the 'events' described in Chapter 2. The stream of events, or execution trace, received by the server is checked against a formal specification using timed deterministic finite state machines with variables. Reports can also originate from within the server, for example to signal the detection of a failure when an automaton reaches an illegal state. This allows for very flexible handling of detections, because of the extensibility and configurability of the report system.

The code generation framework takes a formal specification and generates a monitor that can verify a program's compliance to that specification. The framework uses plugins to parse the properties in the specification, so properties written using various logics can be processed. Support for more logics can simply be added by creating a logic plugin for it. The framework also adds statements to the monitored program to make it send reports. This is also extensible through plugins, in order to support software written in any programming language.

Chapter 4

Prototype implementation

This chapter will go into detail on the prototype implementation of the Sheepdog+ system developed for use with the Shepherd platform. The prototype contains five components, which are discussed in Sections 4.1–4.5: the incident reporter, monitoring core, server core, communication library generator framework. The monitoring core and server core contain some abstract parts, which are extended by code generated by the generator framework. In addition to these components a number of plugins have been developed, which are described in Section 4.6.

All components and plugins, except for the language bindings, are written in C++. The main reason for choosing C++ is that it is a compiled language, which is important for performance. Additionally it is a well-known object oriented language, which is beneficial for maintenance. Additional reasons are that there are a lot of libraries available for C++ and that for most other languages it is relatively easy to write language bindings for C/C++ software.

4.1 Incident reporter

This component implements the incident reporter and related concepts described in Section 3.1.2. A class diagram for this component is shown in Figure 4.1.

The `Reporter` has a member function `log` that takes a report in the form of a report class label, a message and optionally a context dictionary and sends this data to the corresponding report classes. The context dictionary is implemented as the JSON object provided by the JSON library by Niels Lohmann[1]. There are a number of reasons for this choice:

(a) as the context dictionary is encoded in JSON in the messages sent by the client, this makes deserialization very easy as the library provides a method for it,

(b) the JSON objects already support any dictionary value type that can be expressed in the messages sent by the client,

[1]This is an open source library. Its source code and documentation can be found on `https://github.com/nlohmann/json`

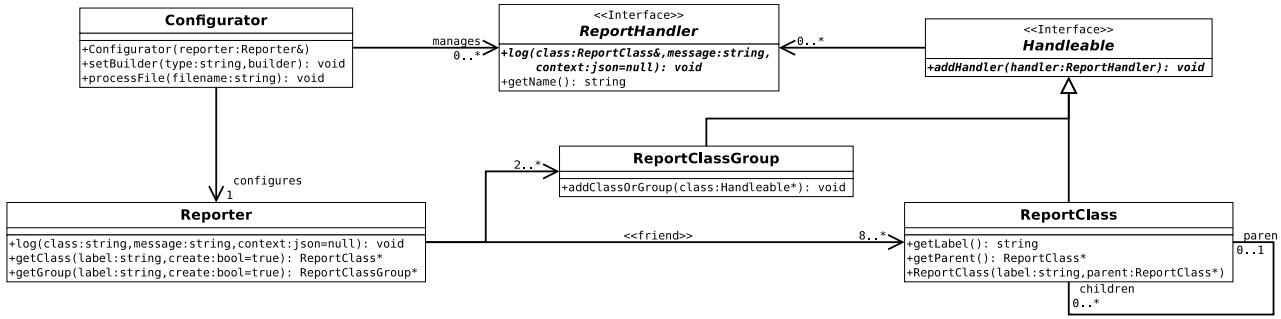


Figure 4.1: Class diagram for the incident reporter component.

(c) using an existing library is preferable over designing and implementing a suitable structure and deserialization method, as the latter would inevitably take a lot of time and introduce bugs into the system.

The report class hierarchy is implemented as a tree of a `ReportClass` objects, which contain pointers to the `ReportHandler`s registered to them. The `ReportHandler`s registered to groups are also in this tree: a `ReportClassGroup` object maintains a list of `ReportHandler`s and `ReportClass`es. Whenever a handler is added, it is registered to all classes in the group, and whenever a report class is added, all handlers registered to the group are registered to the `ReportClass`.

The `log` function traverses the `ReportClass` tree to find all `ReportHandler`s it has to call[2]. If no `ReportClass` is found for the given name, it reports this via the WARNING class. Otherwise, it sends the message and context to all found handlers. If any handler fails, the `Reporter` tries to report this by calling its own `log` function with an error-type class report. If this leads to another handler failure, this is reported via `stderr`. Any exception derived from `std::exception` thrown by `ReportHandler`s is caught this way. The system is designed this way to be as robust as possible, which is important for a system that is in place to monitor an other system.

Creation and execution of the reporter system are separated through the `Configurator` class. There is exactly one `Configurator` instance per `Reporter`. It it responsible for creating instances of `ReportHandler`, `ReportClass` and `ReportClassGroup` for the `Reporter`. The `Configurator` defines two groups by default: ALL, which contains all classes, and LOGGING, which contains the classes corresponding to the log levels specified by RFC 5424. Classes and groups should not be added to ALL manually, but handlers can be added as normal (see Section 3.3.4 for more details on the configuration file).

In this component a lot of pointers to objects are passed around. In the case of `ReportHandler` pointers, these objects are owned by the Configurator, as this is the component that creates the handlers. The `ReportClass` objects on the other hand make up a tree, the root objects of which are owned by the Reporter, as well as the `ReportClassGroup` objects.

4.2 Monitoring core

The monitoring core contains the classes responsible for the actual runtime monitoring. This includes a number of abstract classes which are used by the automaton generator as base classes. A class diagram for this component is shown in Figure 4.2.

The Sheepdog+ automata are implemented as an instance of the abstract `Monitor` class, which contains a number of `State`s. The generators add those states as private data members to the concrete `Monitor` classes. Clocks and variables are also added as private members. The automaton generator implements the transition function in the `input` and `tick` member functions of the concrete `State` subclasses. The concrete `State`s also have a pointer to the

[2]Because of how groups are implemented, only this hierarchy has to be traversed to find all handlers.

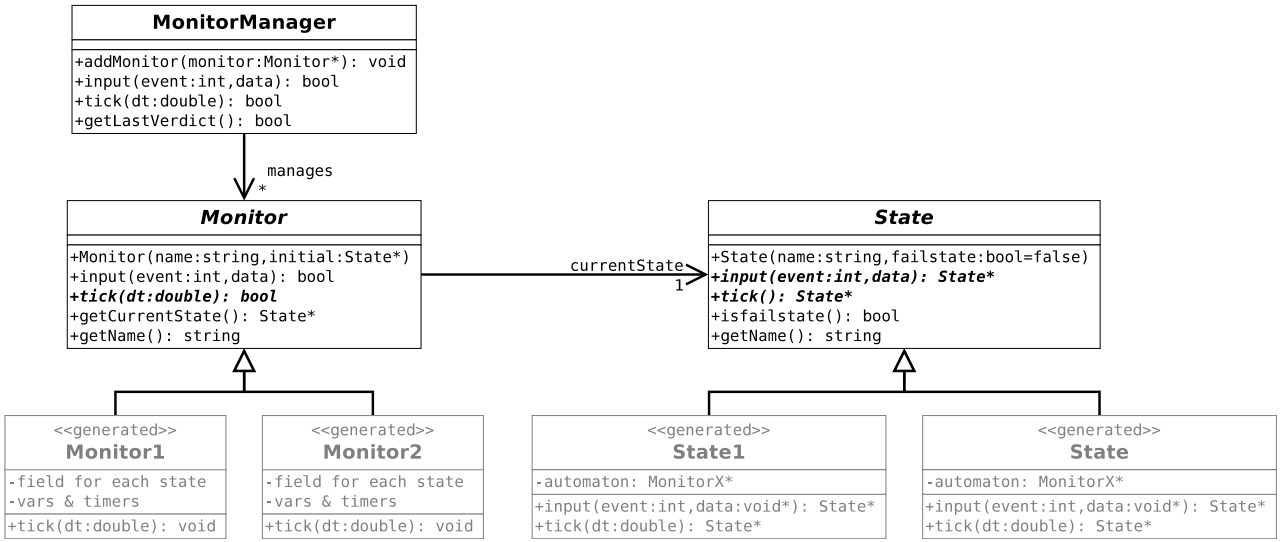


Figure 4.2: Class diagram for the monitoring component.

`Monitor` class they belong to so they can access the variables, clocks and other states of the monitor.

The class `MonitorManager` maintains a list of pointers to `Monitor`s and provides the methods `input` and `tick`, which are used to notify the system of events and to advance the clocks, respectively. Calls to those functions are forwarded to those of all the `Monitor`s maintained by the `MonitorManager`, which update their clocks appropriately and forward the calls to their `currentState`. Clocks are advanced by calling the `tick` function of the `MonitorManager` in a fixed interval, which is by default set to 1 second.

4.3 Communication library

The prototype also includes a library for inter-process communication, which is used by the servers and some of the language bindings. It uses the ZeroMQ[3] C++ bindings as its ZMTP implementation. A class diagram for this component is shown in Figure 4.3.

The `Socket` class is a wrapper for `zmq::socket_t`, the ZeroMQ socket, of which various types exist. There are two specializations of `Socket` in the communications library, each of which has a fixed socket type: the `Server` socket, which uses provides an interface for sending reply messages and the `Client` socket, which allows for asynchronous sending and synchronization. `Server` and `Client` respectively bind and connect to the addresses passed to the constructor, but additional addresses can be bound or connected to using the `bind` and `connect` member functions.

The `Message` class eases creation of messages for sending. It provides functions for adding data of various types to a message, saving the need for the user to convert the data. Each call to one of these functions adds a `MessagePart` to the message, which wraps the actual `zmq::message_t`. As such, it allows sending multi-part messages without having to mess around with flags. The `Message` class also makes it easier to process received messages, by providing easy access to the list of `MessagePart`s making up the multi-part message. `MessagePart` in turn provides conversion functions to the various data types that can be encoded by `Message`.

The `Client` socket does not provide a synchronous send call, because multiple response messages must be processed if earlier an asynchronous send was done. A function that returns the amount of messages that have to be processed and a blocking receive call are available. Synchronizing with the server can therefore be done in a way similar to the example in Listing 6. A synchronous send can be done by synchronizing immediately after a send.

4.4 Server core

The server core consists of an abstract class from which a server is derived by the server generator and a number of utilities used by this class. This abstract class instantiates the server socket, `MonitorManager`, `Reporter` and `Configurator`. Its `start` member function executes a loop in which the server waits to receive

[3] `http://www.zeromq.org/`

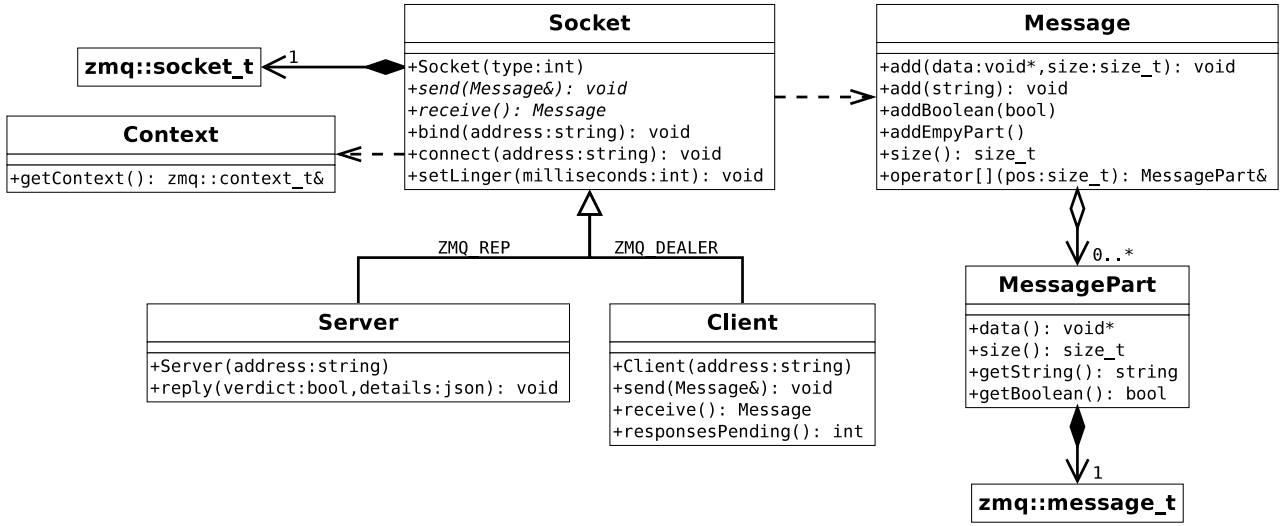


Figure 4.3: Class diagram for the inter-process communications library.

```
while(client.responsesPending()) {
    Message response = client.receive();
    if (response[0].getBoolean() == false) {
        // handle failure
    }
}
```

Listing 6: Example client code for synchronizing with the server. Note that this code directly uses the client interface, while in practice the calls provided by language bindings need to be used, although this should have a similar form.

a message and then processes it. All concurrency issues regarding messages incoming from multiple sources are handled by the ZeroMQ socket.

The other classes in this component are a report handler and `Ticker`. The report handler, `EventForwarder`, decodes event labels and sends the events to the monitor manager. This handler is by default registered to the EVENT class. The `Ticker` class spawns a new thread that periodically sends special 'tick' messages to the server socket. By sending those messages to the same server socket that receives the report-type messages, they are handled in the same thread way as all other messages, ensuring thread-safety.

4.5 Generator framework

This component contains all of the generators and tools making up the generator framework. A class diagram for this component is shown in Figure 4.4. The `run` member function of `Generator` executes the entire process shown in Figure 3.7, by calling all of the specialized generators shown in the class diagram in order.

Note that the `input` functions of `MonitorManager`, `Monitor` and `State` take an integer instead of an event label. `EventsGenerator` creates a mapping from event labels to the integers used in the generated code. This mapping is used by the `EventForwarder` report handler to translate the event labels at run time. This way only one string comparison has to be done per received event.

The generator framework can also be run in debug mode by calling the `run` function of `Generator` with `debug=true`. In this case, the specification files are parsed as normal, but instead of generating code, the `GraphGenerator` is called, which generates graphs representing the automata described by the `MonitorNode` objects, in the form of GraphViz `.dot`-files. Additionally, graphs are created of the abstract syntax trees of regex properties. There is also a tool that generates the abstract syntax trees for expressions in the FSM logic, but this is not automatically called in the debug mode.

For the parsers in the generator framework parser generators were considered, especially boost::spirit, but implementation issues led to a delay, due to which it was decided to quickly get the system working with handwritten parsers. Using an existing parser toolkit may be reconsidered for better extensibility, reliability and proper error handling.

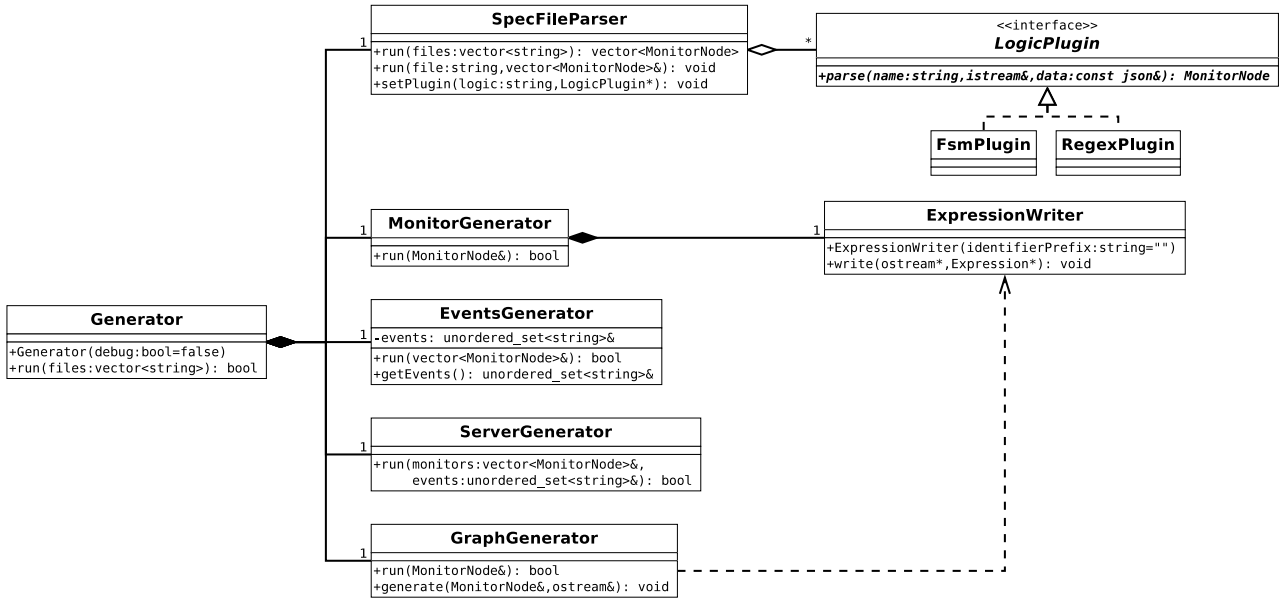


Figure 4.4: Class diagram for the code generation framework, plus two logic plugins.

4.6 Plugins

For the prototype the following plugins were developed:

- logic plugins for the FTL and regex logics described in Section 3.3,
- four report handlers,
- language bindings for PHP, bash and C#.

Each language binding uses a somewhat different approach, but all use source code instrumentation. For the PHP and bash bindings, this is obvious, as they are not compiled languages, but there is another reason to choose for source code instrumentation: as automated instrumentation is not available, instrumentation has to be done manually, which is only feasible in the source code.

The language bindings have a few peculiarities that are worth pointing out. Firstly, the PHP language binding uses a custom PHP extension to couple with the communication library. The message sending statements are provided as PSR-3[4] compatible logging statements, primarily in order to reduce code reading friction, but also to allow a homogeneous way of logging via Sheepdog+ and for integration with third party components conforming to the standard recommendation. The bash bindings are implemented as a small program written in C++ and only allow synchronous send with receive or sending without being able to receive the server response. This is because a bash script consists of a series of separate command line statements and can thus not keep a connection open. Finally, the C# bindings are written on clrzmq4, C# language bindings for ZeroMQ, as this was easier than creating C# language bindings for the communications library. The C# bindings support multi-threaded programs by being able to start a local server for each thread, as described in Section 3.1.3.

The report handlers are `FileLogger`, which is used to write reports to a log file, `DatabaseLogger`, which stores the reports in a database table, and `StdOutLogger`, which writes the reports to `stdout`. The latter was primarily included for debugging purposes, as the server normally runs in the background. All of those three handlers ignore the context of the report. Finally there is the `RethrowHandler`, which sends the report back to the incident reporter in a different class and optionally with a modified message.

Additionally, for GreenStar Statistics, a dashboard page was added to the administrator section of the GreenStar portal, the web application that provides an interface to the Shepherd platform. This dashboard presents a summary of the reports stored in a database table by the `DatabaseLogger`, in order to give a quick insight into the problems that have recently occurred.

[4] A PHP Framework Interoperability Group standard recommendation, see `http://www.php-fig.org/psr/psr-3/`

Chapter 5

Validation

This chapter will demonstrate the Sheepdog+ system by using it with two software systems, and will check whether the system satisfies the requirements set up in Chapter 1. The first system is the GreenStar Statistics Shepherd platform. This is a very large system, so only a few of its components have been instrumented at the time of writing. The second program is an implementation of the Digital Store Application (DSA) that is used as an example in Chapter 2. This system is created to demonstrate some features of Sheepdog+ that are hard to demonstrate with the limited subset of the Shepherd platform, and to show that Sheepdog+ satisfies some requirements related to this functionality. Additionally, it is easier to introduce errors in this demonstration program in order to show successful detections.

The instrumentation process and the specification for the Shepherd platform are explained in Section 5.1. Section 5.2 introduces the DSA and its specification.

5.1 The Shepherd platform

The Shepherd platform is a system that monitors driving behaviour for taxi companies, leasing companies and other businesses with a significant vehicle fleet. The goal of the system is to reduce expenses and CO_2 emissions by improving the driving behaviour of the companies' drivers.

Feedback on driving behaviour is provided to the drivers and managing staff via various kinds of reports (e.g. a daily e-mail to the drivers or a monthly management report) and via the GreenStar portal; a web application that can be used to obtain additional information.

As mentioned in Chapter 1, the data from which the driving behaviour statistics are derived is obtained from so-called 'ecoboxes' mounted in the vehicles. Those devices are not developed by GreenStar Statistics, but are bought from third party manufacturers. The ecoboxes gather data from the on-board diagnostics (OBD) port of the vehicle and from measurement devices in the ecobox itself, such as an accelerometer and a GPS module. The ecobox transmits its data via GPRS to the manufacturer, from which it is obtained by GreenStar Statistics.

5.1.1 Shepherd platform architecture

Large amounts of raw data are produced each day which has to be processed into driving behaviour statistics. This section will give a brief description of how the Shepherd platform processes and stores this data. After giving an overview of the system, this section will go into more detail on the components that are used in this validation process.

Database system

In the Shepherd platform, three database types can be distinguished in which data of various types is stored. The raw data is stored in the 'instance databases', of which there is one per customer. Also other customer-specific data is stored here, such as a list of drivers. For each customer there is also a 'data warehouse', which contains the processed driving behaviour data. This data can at any time be recreated from the raw data, so the data warehouse only exists because processing the raw data takes too long to be done on demand. Finally there is one administration database, which stores general information such as a list of all ecoboxes and at which customer they are in use.

Data processing pipeline

Figure 5.1 shows the data flows in the system and the data processing operations on them. The main data flow is printed in bold. At the time of writing, ecoboxes from two different retailers are in use, called Nazza and Munic. Munic provides its data by pushing to a webhook, while Nazza provides a web service from which the data can be retrieved. For both sources, the data is initially saved in a file. Those files are periodically processed by parsers, which use information from the administration database to insert the data into the instance database of the customer to which the data belongs. Using the "Extract, Transform and Load" (ETL) paradigm, the data in the instance databases is processed into driving behaviour data that is stored in the corresponding data warehouse. The ETL process is executed daily for each instance database. This process again uses additional data from the administration database and consists of many translation and aggregation steps. The resulting statistics can be viewed via the GreenStar web application and are used to send driving behaviour reports via e-mail.

There are a few additional data sources, most notably shift and refuelling data. Shift data shows which driver used which vehicle at what times. In many companies this varies greatly, and thus this data is required to couple driving behaviour data to a driver, as the data is received from a certain vehicle. The refuelling data, which is usually easily available to the customer due to the use of fuel cards, can optionally be used by the Shepherd platform to give more accurate savings numbers. If this data is not available, an estimation is made based on the information gathered from the vehicles. Both of those data types are received from the customer as spreadsheets. Specialized importers insert this data into the database from where it is used by the ETL process, in conjunction with the data from the ecoboxes.

The administration database is largely populated by hand. One partial exception is vehicle information: the GreenStar Statistics staff only has to supply

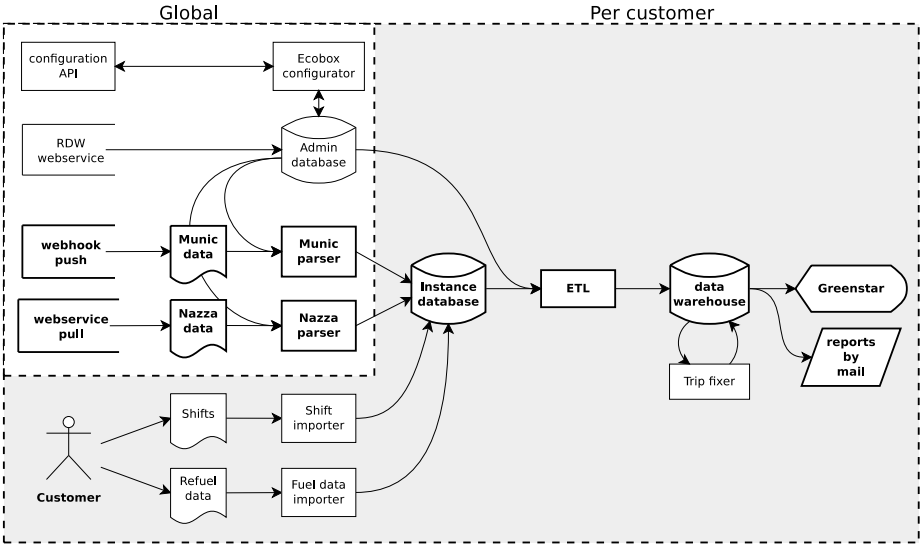


Figure 5.1: The data flows in the Shepherd platform.

a license plate number, and the corresponding details are requested from a web service.

Many processes in the system are automatically executed by a scheduling service (cron), such as the ecobox data parsers, the ETL process and sending reports by mail. Other processes are executed by staff or customers. Note that the many components are built using various technologies. For example, the Munic parser is written in C# and the ETL process consists mainly of MySQL stored procedures executed from Linux shell scripts. Most other processes are implemented in PHP, such as the Nazza parser and the GreenStar web application which is built on the Zend framework.

Real-time Munic parser

The parser for the data from the Munic ecoboxes, described above as a scheduled process, was replaced during development of the Sheepdog+ runtime monitoring system. This new parser processes data immediately as it is received. This is a new, complex, concurrent subsystem being added to the Shepherd platform, making it an interesting candidate for monitoring by Sheepdog+.

The real-time parser works in three steps, starting with an HTTP server:

1. The server listens for HTTP requests on the webhook. If a request is received, it is off-loaded to an available 'listener' and the parser resumes listening for more requests.

2. The listener checks the request for validity and processes it into driving events, each of which are inserted into one of a number of queues.

3. Each queue belongs to a 'parser', which further processes the queued events and stores them in a database.

The system contains multiple listeners and parsers running in separate threads for parallel processing of HTTP requests and queued events. Distribution of requests between listeners is arbitrary, but distribution of events between parsers depends on the customer from which the data originates, because this determines into which database the data must be stored. There are more customers than parser threads, so each parser processes the events for a number of customers. When the real-time parser is shut down, remaining items in the queues are saved to disk to be processed when the parser is started again. The raw requests are saved to disk as well, so the data is not permanently lost if there is an error in the parser.

Ecobox autoconfiguration system

The second Shepherd component that was instrumented is the ecobox autoconfiguration system. In order to give useful data, the ecoboxes need to be configured, depending on various parameters, such as properties of the car in which they are mounted. This took a lot of time for the support staff, so the ecobox autoconfiguration system was developed automate this process, again during the Sheepdog+ project.

The process used by the manufacturer to configure ecoboxes is called a "campaign". The Shepherd platform determines what configuration an ecobox requires, based on the data in the Shepherd databases. It then sends an HTTP request to the manufacturer's configuration API to start a campaign with those settings. Often this causes problems, not in the least place due to problems in the API.

5.1.2 Specification

This section first gives the specification in two parts: first the part concerning the real-time parser, followed by the part applying to the ecobox autoconfiguration system. The specification for the real-time parser focuses on multithreading correctness, as is evident from the properties listed below (the P in the enumeration stands for "parser"):

P1. All parser threads must keep running until stopped normally,

P2. the parser must properly stop when asked to shut down,

P3. all threads must be properly closed when shutting down the parser,

P4. the queue of items that must be processed must be saved before terminating,

P5. each http request must be handled within a specified time.

If a parser thread fails, data will be missing for a number of customers, leading to incomplete or even incorrect reports being sent to those customers. Therefore Sheepdog+ must inform the system administrators of such events, for which requirement P1 is created. On the other hand, when the system is being shut down, all threads must first be stopped before the parser has completely shut down, which is checked by property P3. If this does not happen, problems might occur with the queues. Additionally, Sheepdog+ should issue a warning

when a shutdown takes too long, as this might indicate trouble with terminating one of the threads (requirement P2). Property P4 checks if each parser actually saves its queue, and property P5 checks for problems such as system overload and queue stagnation. The specification files containing these properties is shown in Listings 7 and 8. The specification is divided into two files, because the properties in the first file are monitored by a central server, while the properties in the latter file need to be monitored separately per thread, as supported by the C# bindings.

```
[{
    "logic": "regex",
    "property": "(RTParser_Starting RTParser_Stopped)*",
    "match": true,
    "filter": ["RTParser_Starting",
               "RTParser_Server_DetectedParserCrash",
               "RTParser_Stopped"]
},{
    "logic": "FSM",
    "file": "shutdown_timeout.fsm",
    "initial": "stopped",
    "violating": ["fail", "unexpected_shutdown", "timeout"]
},{
    "logic": "FSM",
    "file": "shutdown_subprocesses.fsm",
    "initial": "stopped",
    "violating": ["fail"]
}]
```

Listing 7: The specification file for monitoring the real-time parser of the Shepherd platform by the central server. The files "shutdown_timeout.fsm" and "shutdown_subprocesses.fsm" are shown in Listing 11 and 12, respectively.

For the ecobox autoconfiguration system there are a few things that need to be monitored, as listed below (the A in the enumeration stands for "autoconfiguration"). Those are all monitored by the FSM property in Listing 14.

A1. Don't start a campaign when no configuration needs to be sent,

A2. include a special "fuel config" if required,

A3. sending the campaign HTTP request must be successful.

5.1.3 Instrumentation

The real-time parser and the ecobox autoconfiguration system are instrumented by inserting calls to the client that send `EVENT`-type messages to the server at various points in the code. This is done, respectively, using the C# and PHP language bindings described in Section 4.6. Additionally, the existing logging facility of the real-time parser was modified to send logging calls to Sheepdog+ in the RFC 5424 classes in addition to its normal logging behaviour.

```
[{
    "logic": "regex",
    "property": "RTParser_Parser_Start
                 RTParser_Parser_SavedQueue
                 RTParser_Parser_Stop",
    "match": true,
    "filter": ["RTParser_Parser_Start",
               "RTParser_Parser_SavedQueue",
               "RTParser_Parser_Stop"]
},{
    "logic": "FSM",
    "file": "process_requests.fsm",
    "initial": "stopped",
    "violating": ["fail", "timeout"]
}]
```

Listing 8: The specification file for monitoring threads of the real-time parser separately. The file "process_requests.fsm" is shown in Listing 13.

The first specification file for the real-time parser (Listing 7) and specification file for the ecobox autoconfiguration system are used to generate one server, which is continuously running on the webserver of GreenStar Statistics. Additionally, a second server is created from the specification file in Listing 8 that is instantiated by each thread of the real-time parser to monitor each thread separately. The servers are configured to write to a log file and to display any errors on the dashboard.

5.1.4 Results

Despite it being hard to introduce errors into the system, some interesting results have been obtained:

- A violation of property P4 was detected during a restart of the system.
- Property P5 ("each http request must be handled within a specified time") was violated after setting the time limit small enough that even with a small peak in the load the automaton timed out.
- The dashboard revealed a number of MySQL errors in the real-time parser. Even though this was not detected by the automata-based monitoring component, it still shows the merit of Sheepdog+, because those errors would have gone unnoticed without the Sheepdog+ system.
- A rare occasion of violating the property for the ecobox autoconfiguration system was observed.

5.2 The Digital Store Application

The Digital Store Application (DSA) provides two user interactions: authentication and ordering items. While processing an order, the system ignores any

other order requests, until the order has been completed. The DSA is implemented as a PHP program that is run from the command line (see Appendix C for the source code), that uses a small database to store things like users and item prices. For simplicity, this database is implemented as a JSON file. Additionally, a bash script is written that allows a user to easily buy a number of products. This script invokes the PHP program to handle authentication and place orders.

The implementation intentionally contains a number of errors that only lead to a failure for certain inputs. This is done so that all properties written for the DSA can actually be violated.

5.2.1 Specification

The properties written for the DSA are, in plain English, as follows (the D in the enumeration stands for "DSA"):

D1. Only after authentication can an item be successfully bought.

D2. An order can only be completed if the user has enough credits.

D3. An item may only be bought as the result of an order.

D4. Every request from the bash script should be processed, i.e. answered by a 'bought' or 'failed' event.

D5. The bash script should terminate within time t_{max}.

The first three properties are taken from Chapter 2, and the fourth property is adapted from Section 2.5. The fifth property demonstrates the use of timed automata to check for termination. The specification file created based on those informal properties is shown in Listing 9. Note that properties two and three are combined into one property in the specification file.

5.2.2 Instrumentation

In order to monitor these properties, statements are inserted in both the bash script and the PHP program that make calls to the languages bindings described in Chapter 4. A synchronization point is inserted just before the transactions are actually completed, so a transaction is canceled if Sheepdog+ detects a problem in the execution.

5.2.3 Results

As mentioned above, there are bugs in the DSA that are encountered for certain inputs only. They can indeed all be found by monitoring the DSA with the given specification, as indicated by the following runs:

1. Using a wrong password, property D1 was violated. This caused the synchronization point to block all transactions, as indicated by the message "Failure detected by Sheepdog+" (line 37 in Listing 16).

```
[{
    "logic": "regex",
    "property": "[^bought auth]* auth bought*",
    "match": true,
    "filter": ["auth", "bought"]
},{
    "logic": "regex",
    "property": "(order (balance_ok bought | failed))*",
    "match": true,
    "filter": ["order", "balance_ok", "bought", "failed"]
},{
    "logic": "regex",
    "property": "(call_buy (bought|failed))*",
    "match": true,
    "filter": ["call_buy", "bought", "failed"]
},{
    "logic": "FSM",
    "file": "timeout.fsm",
    "initial": "waiting",
    "violating": ["timeout", "fail"]
}]
```

Listing 9: The specification file used to monitor the DSA application. The file "timeout.fsm" is shown in Listing 10.

2. When the script is run to buy 6 apples with an account that only has credit for for 5 apples, the sixth order is blocked due to a violation in the automaton monitoring properties D2 and D3, due to the former property.

3. Trying to buy cars also leads to this violation (but now due to the latter property) and to a violation of property D4 that monitors interaction between the bash and PHP scripts.

4. Supplying a negative number to the bash script causes an infinite loop. This is detected by Sheepdog+ because a violation of property D5 is observed.

After each of those runs, the state of the database was evaluated to verify that no illegal transactions have been made. In all cases the database state was correct.

```
@timer t
@var tmax = 5

waiting [
          start  ->  running {t = 0}
          end    ->  fail
]
running [
          start  ->  fail
          end    ->  waiting
     [t > tmax]  ->  timeout
]
timeout [
          start  ->  running {t = 0}
]
fail [
          start  ->  running {t = 0}
]
```

Listing 10: The FSM property described in “timeout.fsm”.

Chapter 6

Conclusion

This thesis has presented a design for a runtime monitoring system capable of monitoring systems consisting of multiple components implemented in different languages, called Sheepdog+. It aims to satisfy the needs of very different systems by being highly flexible and extensible.

Section 6.1 will evaluate whether the system satisfies the requirements, based on the results of Chapter 5 and Section 6.2 summarizes how this system answers the research questions for this project, introduced in Chapter 1. Finally, a number of subjects for further research are listed in Section 6.3.

6.1 Conformation to requirements

This section discusses for each requirement how the use cases of Chapter 5 demonstrate that the Sheepdog+ runtime monitoring solution fulfils it.

R1: Monitor separate components. Both the Shepherd platform and the Digital Store Application are multi-component systems monitored by Sheepdog+, showing that this requirement is fulfilled.

R2: Be able to monitor components written in multiple languages. Satisfaction of this requirement is also demonstrated in both use cases. Shepherd's autoconfiguration component is implemented in PHP, and the parser is implemented in C#. The DSA program is partially implemented in PHP and partially in bash. All of those components are instrumented, showing that programs written in all of those three languages can be monitored.

R3: Support for a new language must be easily added. While the PHP and bash bindings had been created early on, the C# bindings were developed later, for the validation part of the project. This only took a few hours, despite having no experience with C# or the target component, and with minimal assistance of a domain expert.

R4: Monitor interaction between separately executing components. The successful verification of property D4 of the DSA shows that this is possible with the Sheepdog+ prototype, also between components implemented in different languages, as this property monitors interaction between the bash

script and the PHP programs. In the Shepherd platform this kind of monitoring is demonstrated less clearly, but the technically similar case of monitoring interaction between different threads is present there.

R5: Monitor components separately. The real-time parser of the Shepherd platform uses multiple threads to process the incoming data in parallel. Even though those threads all send identical events, the parallel threads can be checked for correctness according to properties P4 and P5 by using the multiple-server approach described in Section 3.1.3.

R6: Support various different types of properties. The properties verified in the Shepherd platform and DSA use cases include normal correctness properties (such as in properties P4, A2 and D1), as well as timing constraints (properties P5 and D5). Additionally, errors detected in traditional ways can be found on the dashboard, because the flexibility of the incident reporter allows Sheepdog+ to process normal error messages, which is in this case achieved by the integration with the existing logging facility. Results have been obtained for all of these property types.

However, the strongest method for allowing a large property variety in the Sheepdog+ monitoring system is not demonstrated. This is the addition of message processors, for example an outlier detection algorithm. This was not demonstrated, because implementing other message processors than the automaton-based runtime monitoring system was outside the scope of this project.

R7: Monitor system performance. For the real-time parser of the Shepherd platform, Sheepdog+ checks whether a request is handled within a certain time (property P5). The primary goal for this property is to issue a warning when the process takes too long, for example due to a very high load or bugs causing the process to take longer than normal. Violations of property P5 were indeed detected, showing that Sheepdog+ can monitor these kinds of properties.

R8: Writing properties should not be hard. The Sheepdog+ design addresses this through a system that allows multiple logics to used. This allows users to use the right tool for the right job. In both use cases the two logics that were implemented are used: where suitable the very easy regex logic was used, while for properties that could not (easily) be expressed as regular expressions, the FSM logic was used, which is much more powerful, but also more verbose.

R9: Responding to violations. In the Shepherd system, the flexible violation handling system of Sheepdog+ is used to list detected errors on the dashboard and to write all logging and monitoring information to a log file. The next section goes into more detail when reviewing the violation handling system.

R10: Provide a mechanism for alerting staff of detected failures. The dashboard is designed to give an easy overview of the issues detected by Sheepdog+. As mentioned above this has already led to the detection of some errors that would otherwise have gone unnoticed until they caused serious harm.

R11: Prevent failures from occurring. Fulfilment of this requirement is demonstrated in the DSA use case, which uses a synchronization point before

the important section of the code. For all detected violations, this mechanism has prevented damage due to a failure.

6.2 Answering the research questions

1. How to allow monitoring and instrumentation for multiple implementation languages?

To support a system consisting of multiple components, the Sheepdog+ monitors are executed in a separate process, i.e. the system uses outline monitoring. Events in the monitored system are exposed to the monitor via messages sent by the monitored software via an inter-process communication system. This way, multiple processes (and threads) and their interactions can be monitored, as they can all send their events to the same monitoring process.

As the messages sent to the monitoring process are not dependent on the implementation language of the sender, components written in any language can be monitored. Of course some code for sending those messages has to be developed to support a new language (creating a language binding), but as sending a message is all that has to be done client-side, this should be fairly straightforward.

Instrumentation is a highly language-specific task. Therefore, the architecture has a plugin-based system to support automated instrumentation for multiple languages. For each language a separate plugin is used that isolates the language specific parts of instrumentation from the rest of the system.

2. How to react to violations and recover from failures in a varied environment?

As explained in Section 3.1.4, there are two ways to react to a violation. For each violation, multiple actions can be triggered, so for one violation both these methods might be utilized.

A failure can be prevented by adding synchronization points to the code of the monitored software. A violation can then be handling at this synchronization point, for example by adding code that restores the program to a valid state, or by preventing the execution of a section of code that might cause harm if the software is in an erroneous state. A good example of the latter option was demonstrated in the Digital Store Application, which cancels a transaction if any failure has been detected.

Sheepdog+ also supports aynchronous handling of violations, by triggering an action via the incident reporter. An asynchronous action cannot guarantee to prevent a failure from happening, because the program continues its execution after the event causing the violation has been sent and thus the failure might already have occurred before the reaction could be executed. This method does also have a number of advantages over the synchronization approach:

- it does not require modification of the source code of the monitored software,
- actions outside the scope of the monitored software can be taken,
- if the violation does not necessarily indicate a failure, it can be handled without disrupting the execution of the monitored software.

An example of the last point is a property intended to warn about performance issues. There is no way to prevent such a 'failure' —adding a synchronization point would only make matter worse! The most important reason for using asynchronous violation handling, however, is the second point: whenever a violation is detected, whether the failure is prevented or not, the developers should be notified in some way, because a violation indicates a problem with the software that must be fixed. Also, intervention of a system administrator is often required when a violation is detected. By detecting failures early and notifying staff, larger problems can be prevented, even if the failure itself is not.

3. How to provide developers with feedback for resolving the cause of an encountered violation?

There are many ways to do this, using asynchronous violation handling. Examples are sending a mail or filing a support ticket, and the prototype uses a dashboard to present issues to the staff. To obtain more information about the cause of a problem reported in this way, developers need to evaluate the log files. More research is required to provide the developers with better tools, as described in Section 6.3.

4. How to measure scalability of the monitored system?

As demonstrated in Chapter 5, timed properties can be used to check whether an operation takes longer than acceptable, so the maintainers of a system can be notified of performance issues. By keeping these time requirements stricter than necessary, bottlenecks can be identified and addressed way before it causes problems. If more advanced information is needed, the system can be extended with a profiler via the incident reporter.

6.3 Future research

The Sheepdog+ design still has a few limitations. This section mentions those limitations as suggestions for further research.

Automated instrumentation

Although automated instrumentation is considered in the Sheepdog+, a few design issues are still open. How the instrumentation should be done precisely is an implementation issue for the plugins, but how to specify what statements need to be added via an instrumentation script, independent of implementation language needs additional research.

Automated instrumentation would greatly improve the usability of the system as it relieves the need to manually modify the code base of the monitored software and also improves reliability for the very same reason (see Section 2.2). Additionally, automated instrumentation opens the door to some other useful techniques described below.

Advanced feedback

If a violation is detected, the sequence of events that led to the violation needs to be distilled from the execution trace, in order to help the developers determine the cause of the violation. Normally this information does not get stored,

as Sheepdog+ uses a non-trace-storing verification method. How to save and present the relevant information needs to be researched.

If automated instrumentation is available, it is easy to add more events to the monitored software, to provide the developers with even more information, for example by listing method calls to provide a sort of "stack trace".

Events carrying data

All parts of the system already allow passing additional data along with events and the automata use variables. However, the automata cannot access the data sent with events yet. Adding this functionality to the design would allow more types of properties to be checked.

Message origin tracking

In some cases it can be interesting what the origin of a message is, for example for distributing logging data over multiple log files. Also, distinguishing multiple instances could be done within one server when the origin of the message is provided. Hence, it might be interesting to research the option of adding a source field to the messages.

Appendix A

Inline servers

The Sheepdog+ architecture only considers out-line servers, while in-line servers could be useful as well, are technically possible and are compatible with the architecture: instead of only sending messages to server processes, a proxy routine could also make a direct call to the incident reporter of an inline server. This section sheds some light on the possibility of in-line servers in Sheepdog+ and the problems due to which they are not considered in this document.

One possible advantage is a reduction in communication overhead, as no inter-process communication is required when addressing an in-line server. This must of course be measured before any concrete statements can be made, and until then performance should not be used as an argument for supporting in-line servers. The overhead reduction is probably not significant, especially with asynchronous monitoring, since the client-server separation is designed in such a way that the load on the client is minimal. Which brings up a big disadvantage of in-line monitoring: it can't be done asynchronously. This means that the overhead might actually increase, because all processing has to be done before execution can continue.

A simple way of creating an in-line monitor would be to provide language bindings for the incident reporter, similar to the current language bindings. However, this implementation lacks the full power of an in-line monitor: the main advantage of in-line monitoring is that the monitor can read the current software state, because it is embedded in the process. To make use of the full power of in-line monitoring, a very strong coupling between the monitored software and the monitor is required and design changes need to be made to access the program state — which is strongly language-dependent. Achieving this requires either a complete re-implementation of the monitor in the target language, or language bindings at a very low level, which would still include a partial re-implementation.

Appendix B

Shepherd platform specification

This appendix contains listings of the property files making up the specification for the Shepherd platform used in Chapter 5.

```
@timer t   = 0
@var   lim = 30

stopped [
    RTParser_Starting                    ->  running
    RTParser_Server_DetectedParserCrash ->  fail
    RTParser_Server_Stopping             ->  fail
    RTParser_Server_Stopped              ->  fail
    RTParser_Stopped                     ->  fail
]
running [
    RTParser_Starting                    ->  fail
    RTParser_RequestedStop               ->  muststop {t = 0}
    RTParser_Server_DetectedParserCrash ->  muststop {t = 0}
    RTParser_Server_Stopping             ->  unexpected_shutdown
    RTParser_Server_Stopped              ->  unexpected_shutdown
    RTParser_Stopped                     ->  fail
]
muststop [
    RTParser_Starting                    ->  fail
    RTParser_Server_Stopped              ->  server_stopped
    RTParser_Stopped                     ->  fail
    [t > lim]                            ->  timeout
]
server_stopped [
    RTParser_Starting                    ->  fail
    RTParser_Server_Stopping             ->  fail
    RTParser_Server_Stopped              ->  fail
    RTParser_Stopped                     ->  stopped
    [t > lim]                            ->  timeout
]
unexpected_shutdown [
    RTParser_Starting                    ->  running
]
fail [
    RTParser_Starting                    ->  running
]
timeout [
    RTParser_Starting                    ->  running
]
```

Listing 11: The FSM property described in "shutdown_timeout.fsm", as referred to in Listing 7.

```
@timer t
@var listeners = 0, parsers = 0

stopped [
                                    RTParser_Starting       ->  running
                                    RTParser_Listener_Start ->  fail
                                    RTParser_Listener_Stop  ->  fail
                                    RTParser_Parser_Start   ->  fail
                                    RTParser_Parser_Stop    ->  fail
                                    RTParser_Stopped        ->  fail
]
running [
                                    RTParser_Starting       ->  fail
                                    RTParser_Listener_Start ->  running {listeners++}
                    [listeners > 0] RTParser_Listener_Stop  ->  running {listeners--}
                                    RTParser_Listener_Stop  ->  fail
                                    RTParser_Parser_Start   ->  running {parsers++}
                      [parsers > 0] RTParser_Parser_Stop    ->  running {parsers--}
                                    RTParser_Parser_Stop    ->  fail
    [listeners > 0 || parsers > 0]  RTParser_Stopped        ->  fail
                                    RTParser_Stopped        ->  stopped
]
fail [
                                    RTParser_Starting       ->  running {listeners=0, parsers=0}
]
```

Listing 12: The FSM property described in “shutdown_subprocesses.fsm”, as referred to in Listing 7.

```
@timer t
@var lim = 60

stopped [
    RTParser_Listener_Start             ->  waiting
    RTParser_Listener_ReceivedRequest   ->  fail
    RTParser_Listener_HandledRequest    ->  fail
    RTParser_Listener_BadRequest        ->  fail
    RTParser_Listener_Stop              ->  fail
]
waiting [
    RTParser_Listener_Start             ->  fail
    RTParser_Listener_ReceivedRequest   ->  running {t = 0}
    RTParser_Listener_HandledRequest    ->  fail
    RTParser_Listener_BadRequest        ->  fail
    RTParser_Listener_Stop              ->  stopped
]
running [
    RTParser_Listener_Start             ->  fail
    RTParser_Listener_ReceivedRequest   ->  running
    RTParser_Listener_HandledRequest    ->  waiting
    RTParser_Listener_BadRequest        ->  waiting
    RTParser_Listener_Stop              ->  fail
                              [t > lim] ->  timeout
]
fail [
    RTParser_Listener_Start             ->  waiting
]
timeout [
    RTParser_Listener_HandledRequest    ->  waiting
    RTParser_Listener_Start             ->  waiting
]
```

Listing 13: The FSM property described in "process_requests.fsm", as referred to in Listing 8.

```
waiting [
    generateCampaign_begin              -> started
    generateCampaign_end                -> fail
    generateCampaign_require_fuelconfig -> waiting
    generateCampaign_added_fuelconfig   -> waiting
    generateCampaign_config_ok          -> waiting
    MunicAPI_postCampaign               -> waiting
]
started [
    generateCampaign_begin              -> fail
    generateCampaign_end                -> fail
    generateCampaign_require_fuelconfig -> reqfuel
    generateCampaign_added_fuelconfig   -> fail
    generateCampaign_config_ok          -> confok
    MunicAPI_postCampaign               -> send
]
reqfuel [
    generateCampaign_begin              -> fail
    generateCampaign_end                -> fail
    generateCampaign_added_fuelconfig   -> hasfuel
    generateCampaign_config_ok          -> fail
    MunicAPI_postCampaign               -> fail
]
hasfuel [
    generateCampaign_begin              -> fail
    generateCampaign_end                -> fail
    generateCampaign_added_fuelconfig   -> fail
    generateCampaign_config_ok          -> fail
    MunicAPI_postCampaign               -> send
]
confok [
    generateCampaign_begin              -> fail
    generateCampaign_end                -> waiting
    generateCampaign_require_fuelconfig -> fail
    generateCampaign_added_fuelconfig   -> fail
    MunicAPI_postCampaign               -> fail
]
send [
    generateCampaign_begin              -> fail
    generateCampaign_end                -> waiting
    generateCampaign_require_fuelconfig -> fail
    generateCampaign_added_fuelconfig   -> fail
    MunicAPI_postCampaign               -> fail
]
fail [
    generateCampaign_begin              -> started
]
```

Listing 14: The FSM property file for the ecobox autoconfiguration system. The initial state is ‘waiting’ and ‘fail’ is the only failure state.

Appendix C

Digital Store Application source code

This appendix contains the source code for the Digital Store Application implementation used for validation in Chapter 5. The Digital Store Application is made up of two PHP scripts, one for authentication and one for processing an order. The source code for those scripts are shown in Listing 15 and 16, respectively, and Listing 17 contains a PHP file included by both. Finally, Listing 18 contains the bash script that can be used to order a batch of items.

```php
<?php
use Ovis\SheepdogPlus\Logger;
include 'common.php';

$username = $argv[1];
$password = $argv[2];

$db = loadDB();

if (array_key_exists($username, $db['users'])
        && $db['users'][$username] == $password) {
    Logger::event('auth');
    $db['session'] = $username;
    storeDB($db);
}
```

Listing 15: The source code for the Digital Store Application's authentication script, 'login.php'.

```php
<?php
use Ovis\SheepdogPlus\Logger;
include 'common.php';

Logger::event('order');

$item = $argv[1];
$db = loadDB();

// Check session
if (array_key_exists('session', $db)) {
    $user = $db['session'];
} else {
    $user = 'henk';
}

// Check if item exists
if (!array_key_exists($item, $db['prices'])) {
    Logger::error('item "' . $item . '" does not exist');
    // Oops, forgetting to send 'failed' event!
    exit(1);
}

// Check price
$cost = $db['prices'][$item];
$balance = $db['balances'][$user];
if ($balance >= $cost) {
    Logger::event('balance_ok');
}

// Synchronous 'bought' call, so the sale is
// only executed if this event is allowed
Logger::event('bought');
while(Logger::responsesPending()) {
    $response = Logger::receive();
    if ($response['verdict'] == false) {
        echo 'Failure detected by Sheepdog+' . PHP_EOL;
        exit(1);
    }
}
$db['balances'][$user] -= $cost;
storeDB($db);
```

Listing 16: The source code for the Digital Store Application's ordering script, 'buy.php'. It selects the user, checks if the order can be processed and then imburses the user. Before executing the imbursement, it synchronizes with the server, and cancels the transaction if the verdict is false.

```php
<?php
use Ovis\SheepdogPlus\Logger;
use Ovis\SheepdogPlus\SimpleLogger;
require_once 'Psr/Log/LoggerInterface.php';
require_once 'Psr/Log/LogLevel.php';
require_once 'Psr/Log/AbstractLogger.php';
require_once '../src/client/php/LoggerInterface.php';
require_once '../src/client/php/AbstractLogger.php';
require_once '../src/client/php/SimpleLogger.php';
require_once '../src/client/php/Logger.php';
require_once '../src/client/php/EchoingLogger.php';

Logger::setLogger(new SimpleLogger());

function loadDB() {
    return json_decode(file_get_contents('db.json'), true);
}

function storeDB($db) {
    file_put_contents('db.json', json_encode($db));
}
```

Listing 17: The source code for the Digital Store Application's 'common.php'. It includes the language bindings (which normally should be done using autoloading) and defines functions for loading and storing the database.

```
#!/bin/bash
if [ $# -ne 4 ]; then
    echo "Illegal number of parameters"
    exit 1
fi

sdp_report event start
php login.php $1 $2

N=$3
item=$4
while [ $N -ne 0 ]; do
    sdp_report event call_buy
    php buy.php "$item"
    if [ $? -eq 0 ]; then
        sdp_report event "$item"
    fi

    let N=N-1
done
sdp_report event end
```

Listing 18: The source code for bash script in the Digital Store Application.

Bibliography

[1] A. Bauer, M. Leucker, and C. Schallhart, "Monitoring of Real-Time Properties," in *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science* (S. Arun-Kumar and N. Garg, eds.), no. 4337 in Lecture Notes in Computer Science, pp. 260–272, Springer Berlin Heidelberg, Dec. 2006.

[2] S. Malakuti and M. Akşit, "Event Modules," in *Transactions on Aspect-Oriented Software Development XI*, pp. 27–69, Springer, 2014.

[3] N. Delgado, A. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, pp. 859–872, Dec. 2004.

[4] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the MOP runtime verification framework," *International Journal on Software Tools for Technology Transfer*, vol. 14, pp. 249–289, June 2012.

[5] M. Viswanathan, *Foundations for the run-time analysis of software systems*. PhD thesis, Dec. 2000.

[6] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim, "Jass — Java with Assertions," *Electronic Notes in Theoretical Computer Science*, vol. 55, pp. 103–117, Oct. 2001.

[7] K. Havelund and G. Rosu, "Java PathExplorer - A Runtime Verification Tool," in *In The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, p. 2001, 2001.

[8] S. Malakuti, C. Bockisch, and M. Aksit, "Applying the Composition Filter Model for Runtime Verification of Multiple-Language Software," pp. 31–40, IEEE, Nov. 2009.

[9] R. Mizzi, *An extensible and configurable runtime verification framework*. PhD thesis, University of Malta, 2012.

[10] S. Muller, *Theory and applications of runtime monitoring metric first-order temporal logic*. PhD thesis, University of Zurich, 2009.

[11] J. Simmonds, S. Ben-david, and M. Chechik, *Monitoring and Recovery of Web Service Applications*. 2010.

[12] B. Meyer, "Applying design by contract," *IEEE Computer*, vol. 25, pp. 40–51, 1992.

[13] A. Bauer, M. Leucker, and C. Schallhart, "Comparing LTL Semantics for Runtime Verification," *Journal of Logic and Computation*, vol. 20, pp. 651–674, Jan. 2010.

[14] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout, "Reasoning with Temporal Logic on Truncated Paths," in *Computer Aided Verification* (W. A. H. Jr and F. Somenzi, eds.), no. 2725 in Lecture Notes in Computer Science, pp. 27–39, Springer Berlin Heidelberg, July 2003.

[15] G. Roşu and K. Havelund, "Rewriting-Based Techniques for Runtime Verification," *Automated Software Engineering*, vol. 12, pp. 151–197, Apr. 2005.

[16] H. R. Andersen and K. J. Kristoffersen, "Temporal Runtime Verification using Monadic Difference Logic," *arXiv:0705.4604 [cs]*, May 2007.

[17] A. Bauer, M. Leucker, and C. Schallhart, "The Good, the Bad, and the Ugly, But How Ugly Is Ugly?," in *Runtime Verification* (O. Sokolsky and S. Taşıran, eds.), no. 4839 in Lecture Notes in Computer Science, pp. 126–138, Springer Berlin Heidelberg, Mar. 2007.

[18] P. O. Meredith, D. Jin, F. Chen, and G. Roşu, "Efficient monitoring of parametric context-free patterns," *Automated Software Engineering*, vol. 17, pp. 149–180, Feb. 2010.

[19] A. Bauer, M. Leucker, and C. Schallhart, "Runtime Verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, pp. 14:1–14:64, Sept. 2011.

[20] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky, "Monitoring, Checking, and Steering of Real-Time Systems," *Electronic Notes in Theoretical Computer Science*, vol. 70, pp. 95–111, Dec. 2002.