FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Cloud-Based Monitoring and Simulation for Fault-Tolerant Event Processing Platforms

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael Strasser
Matrikelnummer 0751134

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Dipl.-Ing. Dr. Waldemar Hummer, BSc (WU)

Wien, 16.04.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Cloud-Based Monitoring and Simulation for Fault-Tolerant Event Processing Platforms

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Michael Strasser
Registration Number 0751134

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dr. Schahram Dustdar
Assistance: Dipl.-Ing. Dr. Waldemar Hummer, BSc (WU)

Vienna, 16.04.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Michael Strasser
Universumstraße 52/43, 1200 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

First of all, I like to appreciate the work of my advisor Waldemar Hummer, who also provided fundamental aspects of this work, like the used domain model. Additionally, I am thanking my friends Andrea, Thomas, Katrin and Christian (no particular order), who supported and motivated me throughout the writting process of this thesis. And last, but not least, I thank my parents, who provided me the possibility to study and live the life I want.

Abstract

Statement of the Problem

Distributed event-based systems [68] are an emerging research field in computer science. The research area of distributed event-based systems is divided into different sub-areas like publish/subscribe systems, event stream processing, complex event processing, or wireless sensor networks. These sub-areas focus on different aspects, yet the challenges in each area share many commonalities. However, there exists no common system model to describe the different varieties in this area, which makes it difficult to derive simulations for such a system since one needs a special simulator for every system and scenario. This also affects the capabilities for monitoring and fault management, as the native code to deal with such issues is different for each of these systems. The result of these circumstances are separate implementations for every system.

The goal of this work is to implement such a common model with respect to all major derivations of the different kinds of event-based systems. The underlying domain model is kept as generic as possible to support adaptability to new systems which might occur in the future. On top of this model, possibilities for monitoring and simulating fault-tolerant behaviour are implemented.

Expected Results

The final outcome of this work is a generic model for different kinds of distributed event-based systems with a runnable monitoring and a simulation environment on top of that model, called EventSim framework. The model synchronizes its state with a real-world application through adaptors, where an example adaptor is generated for an illustrary example . The handling of these adaptors is also done in a generic way, so that more adaptors can be written for different systems. Since they are included in the corresponding real-world system, the framework itself is not changed while adapting to a new system. This provides a generic way of simulating different kinds of distributed event-based systems described in the previous section.

Therefore, a simulator is provided which enables the simulation of several scenarios using different strategies. This simulation provides the possibility to test specified routing algorithms and scaling strategies which can be applied on-the-fly.

Methodology

The system model is based on the preliminary work presented in [44]. At first this domain model gets transformed to a generic programming model. This model is implemented using model-driven development (MDD) as described in [66]. Using MDD makes the model completely independent from current event-based systems, as it is directly generated from the domain model. On top of this, the tool layer is located with connections to the simulator and injection possibilities for real-world adaptors.

The routing in this implementation is a simple forwarding from one event processing agent (EPA) to another. This leaves complex routing algorithms open for future work. The scaling of the model can be handled by different strategies.

The simulator is a standalone program which should provide some sample configurations files for the initialization of the model. The simulation itself can afterwards change the state of the model by applying different event manipulations or strategy changes.

Evaluation of the framework is done by simulating several scenarios on an example event-based system with the help of the simulator. The connection to existing systems is tested using the Storm framework [61].

State of the Art

There have been several approaches to create a generic model for event-based systems like in [83] or [98]. The common patterns of event-based systems have been described in [27]. The results of this previous work has been merged in [44] which is the foundation of the domain model in this work. This works lists the different kinds of event-based systems which are the basis for the domain model. The principle of adaptors is explained in [63], which additionally explains the concepts of events, which was also done by the previous mentioned publications.

The model-driven engineering (MDE) approach using the Eclipse Modeling Framework (EMF) has been described in [81] and [88]. MDE provides the possibility to derive software program code out of given domain models with the use of domain-specific languages (DSL). This work uses the basic features of the EMF which are described in the latter of these two publications.

Topic Relevance

Distributed systems in general are a main part of the Software Engineering-discipline, which has been growing over the last decade. The knowledge of these distributed systems is one of the major requirements for a modern software engineer. Event-based systems are an emerging research field in this area [33]. The innovation of this work is the introduction of a generic simulation model which currently does not exist.

Kurzfassung

Problembeschreibung

Verteilte ereignisbasierte Systeme [68] sind ein aufstrebendes Forschungsfeld in den Computerwissenschaften. Dieses Forschungsfeld ist unterteilt in verschiedene Teilbereiche, wie Publish/-Subscribe Systeme, Event Stream Verarbeitung, Complex Event Processing oder Sensornetze. Diese Teilbereiche legen den Fokus auf verschiedene Aspekte, allerdings haben die Herausforderungen durchwegs einige Gemeinsamkeiten. Es gibt allerdings kein gemeinsames Systemmodel zur Beschreibung dieser verschiedenen Teilaspekte. Dieser Umstand erschwert die Implementierung von Simulationen für diese Systeme, da ein eigener Simulator für jedes System benötigt wird. Dies betrifft auch den Bereich der Überwachung und der Fehlerbehandlung, da auch hier jeweils ein system-eigener Code verwendet werden muss. Aus diesen Umständen folgt eine eigenständige Implementierung für jedes dieser Systeme.

Das Ziel dieser Arbeit ist die Implementierung eines solchen gemeinsamen Model, welches auf die verschiedenen Besonderheiten von ereignisbasierten Systemen Rücksicht nimmt. Dazu wird das zugrundeliegende Datenmodell so generisch wie möglich gehalten um das Hinzufügen neuer Systeme zu erleichtern. Basierend auf diesem Modell, werden Möglichkeiten zur Überwachung und Simulation von fehlertolerantem Verhalten implementiert.

Erwartete Resultate

Das finale Resultat dieser Arbeit ist ein generisches Modell für verschiedene Typen von verteilten ereignisbasierten Systemen mit einer lauffähigen Überwachungs- und Simulationsumgebung, EventSim Framework genannt. Dieses Modell synchronisiert sich mit einer realen Applikation mittels Adaptoren. Ein solcher Adaptor wird als Teil dieser Arbeit für ein Einführungsbeispiel implementiert. Diese Adaptoren sind ebenfalls generisch implementiert, um das Hinzufügen weiterer Adaptoren zu vereinfachen. Nachdem diese Adaptoren Teil der entsprechenden realen Applikation sind, muss das Framework selbst nicht verändert werden um eine neue Applikation hinzuzufügen. Dies ermöglicht die Simulation verschiedener Typen von verteilten ereignisbasierten Systemen.

Dafür wird eine Simulationskomponente entwickelt, welche die Simulation verschiedenartiger Szenarien mithilfe unterschiedlicher Strategien ermöglicht. Diese Komponente ermöglicht das Testen von verschiedenen Pfadalgorithmen und Skalierungsstrategien während der Laufzeit des untersuchten Systems.

Vorgangsweise

Das Model des Frameworks basiert auf der Arbeit von Hummer et al. [44]. Dieses Modell wird in ein generisches Programmmodel transformiert, welches mittels Model-Driven Development (MDD) erzeugt wird [66]. Dieses Verfahren macht das Modell unabhängig von spezifischen Charakteristika heutiger ereignisbasierter Systeme, nachdem es direkt aus dem Datenmodell erzeugt wird. Über dieser Schicht befindet sich eine Tool-Schicht, welche Verbindungen zum Simulator und zu den Adaptoren für Echtsysteme bereitstellt.

Die Pfade in dieser Implementierung sind als simple Weiterleitung von einem Event Processing Agent (EPA) zu einem anderen zu verstehen. Somit können komplexe Pfadberechnungen in zukünftigen Arbeiten behandelt werden. Die Skalierung des Modells kann über verschiedene Strategien sichergestellt werden.

Die Simulationskomponente ist ein eigenständiges Programm, welches eine Standardkonfiguration für die Initialisierung des Standardmodells bereitstellt. Anschließend kann die Komponente den Status des Models direkt über verschiedene Manipulationen oder Strategieänderungen beeinflussen.

Die Evaluierung des Frameworks erfolgt mittels verschieder Strategien, welche an einem exemplarischen ereignisbasierten System durchgeführt werden. Die Anbindung an ein Echtsystem wird über das Storm Framework [61] illustriert.

Stand der Forschung

Es gab schon mehrere Versuche, ein generisches Modell für ereignisbasierte Systeme zu beschreiben, wie in [83] oder [98]. Die gemeinsamen Muster von diesen Systemen sind in [27] genannt worden. Die Erkenntnisse dieser Arbeiten wurden in [44] zusammengefasst. Diese Arbeit ist gleichzeitig auch die Basis, auf der das EventSim Framework erstellt wird. Das Konzept der Adaptoren wurde in [63] erläutert, wo auch eine Erklärung des Begriffes ”Ereignis” erfolgt.

Der Model-Driven Engineering (MDE) Ansatz mittels des Eclipse Modeling Framework (EMF) wurde in [81] und [88] erklärt. MDE ermöglicht die Generierung von Programmcode aus Domain-Modellen mittels domain-spezifischer Sprachen (DSL). Diese Arbeit verwendet die Basis-Funktionalität von EMF, welche in [88] beschrieben wird.

Relevanz des Themas

Verteilte Systeme sind ein großer Bestandteil des Bereichs Software Engineering, welcher im letzten Jahrzehnt immer bedeutender geworden ist. Fähigkeiten in diesem Bereich sind eine der Hauptanforderungen an einen modernen Software-Entwickler. Ereignisbasierte Systeme sind ein wachsendes Forschungsfeld in diesem Bereich. Der innovative Beitrag dieser Arbeit ist ein generisches Simuluationsmodell, welches in dieser Form derzeit nicht existiert.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Formulation . . . 3
- 1.3 Illustrative example . . . 4
- 1.4 Contributions . . . 5
- 1.5 Structure of the Work . . . 7

2 Background and State of the Art **9**
- 2.1 Event Processing . . . 9
- 2.2 Model-Driven Engineering . . . 12
- 2.3 Reflection in programming languages . . . 14
- 2.4 Models@run.time . . . 15
- 2.5 An eventing platform - Storm . . . 16

3 Related Work **21**
- 3.1 Model-Based Testing . . . 21
- 3.2 Fault Tolerance . . . 23
- 3.3 Runtime Monitoring and Adaptation . . . 25

4 Solution Design and Architecture **29**
- 4.1 Architectural Overview . . . 29
- 4.2 Metamodel for fault tolerance testing . . . 33
- 4.3 Simulation actions . . . 36
- 4.4 Validation of Platform Behaviour . . . 40
- 4.5 Heartbeat mechanism . . . 40
- 4.6 Notification mechanism . . . 41
- 4.7 Adding a new platform . . . 43

5 Implementation **45**
- 5.1 The EventSim Framework . . . 45
- 5.2 Model Generation . . . 46
- 5.3 Model & System Storage . . . 50
- 5.4 Simulation . . . 53
- 5.5 Testing platform . . . 57

6 Evaluation **59**
6.1 Eventing example: Correlation indices based on stock market prices 59
6.2 Performance of the EventSim framework 62
6.3 Fault Injection & Detection 64
6.4 Benefits and using the simulation data 71
6.5 Open issues of the current implementation 72

7 Summary and Conclusion **75**
7.1 The Problem revisited 75
7.2 A Solution approach - The EventSim framework 76
7.3 Future work 76

Bibliography **79**

List of Figures

1.1 Sub-Areas of event-based systems [44] . . . 2
1.2 Illustration of the problem formulation . . . 4
1.3 Illustration of a stock market system . . . 5

2.1 Different kinds of EPAs [72] . . . 11
2.2 Abstraction levels of the MOF [37] . . . 13
2.3 Example transformation using MDE [13] . . . 14
2.4 Different Categories of runtime models [95] . . . 16
2.5 Example of a Storm topology . . . 19

3.1 Model-based testing [90] . . . 22
3.2 Fault taxonomy for event based systems [44] . . . 25

4.1 Architecture of the EventSim framework . . . 31
4.2 Adding a new system within the EventSim framework . . . 33
4.3 Class diagram of the metamodel . . . 34
4.4 ”Additional Traffic” action . . . 37
4.5 ”Event emitting” action . . . 37
4.6 ”Event interception” action . . . 38
4.7 ”Kill Event Source” action . . . 39
4.8 ”Kill Worker” action . . . 39
4.9 Notification mechanism . . . 42

5.1 Deployment diagram of the EventSim framework . . . 46
5.2 Artifacts developed in the EventSim framework . . . 47
5.3 Meta-meta model used to describe Metamodels . . . 48
5.4 Storing an event using the EventSim Tool component . . . 52
5.5 Mockup of the implemented simulation GUI . . . 57

6.1 Event-based system used for evaluation . . . 61
6.2 Exemplary deployment of the scenario . . . 62
6.3 Performance analysis for different event rates . . . 65
6.4 Timing diagram in case of killing the event source . . . 69

List of Tables

6.1 Performance evaluation of the scenario 63
6.2 Performance evaluation of the scenario without the EventSim framework 63
6.3 Introduction of additional traffic to component with highest average execution latency (HAEL) 66
6.4 Introduction of additional traffic to component with lowest average execution latency (HAEL) 66
6.5 Test results when intercepting the route between *YStdDev* and *Denom* 67
6.6 Test results when killing the event source 68
6.7 Test results when killing component *Corr* 69
6.8 Test results when killing component *YCurrency* 70
6.9 Test results when killing component *Output* 71
6.10 Test results when killing component *YCurrency* with adapted setup 72

List of Listings

4.1 Example of a query 40
5.1 Stub of the ModelElement class 49
5.2 Creating an event 52
5.3 Interface of the query checker 55

CHAPTER 1

Introduction

The goal of this chapter is to give a proper insight to the subject of distributed event-based systems and outline the contributions gained within this thesis. After an introduction to the field of event-based systems and underlying topics, the concrete problem and the methodological approach are described. The problem tackled within this thesis is then motivated based on an illustrative scenario. As a last step, the structure of the thesis is outlined with a brief outlook on the content of the remaining chapters.

1.1 Motivation

Distributed event-based systems [68] are an emerging research field in computer science. The research area of distributed event-based systems is divided into different sub-areas like publish/-subscribe systems [30], event stream processing [10], complex event processing [18], or wireless sensor networks [54]. The majority of these sub-areas focus on different aspects, yet the challenges in each area share many commonalities. These commonalities include attributes like asynchronous communication, a notification mechanism and decoupled system components, which are subsumed under the term 'event-driven communication paradigm' [75]. Using this paradigm, an event is the single unit of communication in these systems. These events can either be created outside the system and brought into the system through interfaces or generated from nodes within the system because of certain conditions. Higher level semantics are achieved through composition or correlation of events.

Typical event-based systems have an architecture consisting of four layers [67]. The bottom layer is the *event generator*, which is responsible for transferring facts to events. These facts can be heterogeneous and include factors like services, business processes or sensors. Sources for facts include service calls, changes in data and other exogenous triggers. The goal of the *event generator* is to unify these facts to events which can be dealt with by the event-based system. On top of these generators an *event channel* is established, with the responsiblity of transferring the event to an *event processing engine*. This is the component where the business logic is located

and incoming events are matched against certain predefined rules. If a certain rule triggers, a specific action is performed, depending on the matched rule, e.g. a service invocation or an event generation. The top layer is the *downstream event-driven activity* which is initiated because of the event sent before and can, therefore, include the already mentioned actions like event generation for subscribers.

There exist several kinds of event-based systems, which are illustrated in Figure 1.1.

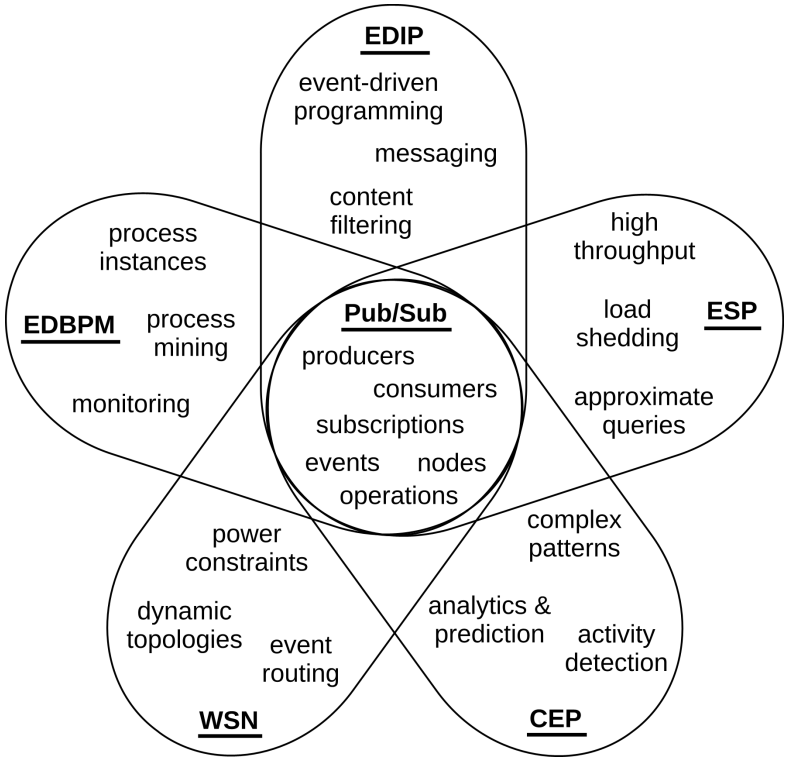

Figure 1.1: Sub-Areas of event-based systems [44]

Event Stream Processing (ESP) [67] deals with event streams and queries performed against these continous streams. This information can be used to monitor the real information flow inside a company and enable decision in real-time according to specific circumstances. *Complex Event Processing (CEP)* [67] is concerned with complex streams, which are often put together out of simple streams, and enables features like event patterns or analysis & prediction. A relatively new field in this area are *Wireless Sensor Networks (WSN)* [77]. They specialise on dynamic topologies, which are a basic requirement of cloud computing for example. Routing is also a fundamental concept for these systems, since nodes are changing on a regular basis and, therefore, routes have to adjust in a dynamic way. *Event-Driven Business Process Management (EDBPM)* [91] is related to business processes as a whole and illustrates workflows out of event logs. Additionally, these processes are then monitored to detect faults and react to them immediately. Other technologies like message-oriented middleware [9] or tuple spaces [4] are referred to as *Event-Driven Interaction Paradigms* in [44]. Most of these technologies deal with

the passing of messages or event-driven programming in specific. The last field in the area, *Publish/Subscribe Systems* [30], combines elements of all other systems. There are two groups of acting entities, producers and consumers. Events and content is produced by the producers and consumed by the consumers, which enables a fast and efficient way of information forwarding.

A common challenge for all these systems is the implementation of a simulation model which is capable of monitoring and fault detection. This is an important issue for fault tolerant systems [24], because the simulation should be able of inserting fault states and check if the system adapts to these circumstances in the right way. Such a system hides failures to the user and operates normally. A key term in this area is 'Failure Masking' [3]. The basic idea behind it is redundancy, for one regarding the structure and also regarding the message passing of the system.

There has been research in this area concerning specific kinds of distributed event based systems, like in [25] for publish/subscribe systems. However there exists no common system model to describe the different varieties in this area, which makes it difficult to derive generic and reusable simulations for testing such systems. The reason for this lies in the fact that a special simulator is required for every system and scenario. This also affects the capabilities for monitoring and fault management [77], as the native code to deal with such issues is different for each of these systems. The result of these circumstances are separate implementations for every system.

1.2 Problem Formulation

Concerned with the different attributes of the specific kinds of distributed event-based systems, the goal of this work is to implement a common model suitable for the different kinds of event-based systems (see Figure 1.2). The underlying domain model is kept as generic as possible to support adaptability to potential new ssytems which might be developed in the future. The current state-of-the-art still lacks the possibility of running different fault scenarios on such a generic model. Another usage is the monitoring of real systems, which should be able to give information about the current status and also provide possibilities to send notifications if a faulty state has been reached or if there is a high possibility to reach one in the near future because of the current status. The problem in implementing such features are the different aspects of distributed event-based systems concerning their architecture or behaviour.

The challenge of deriving such a model is best described by illustrating the differences between the sub-areas of event-based systems which has been done in [44] and is illustrated in Figure 1.1.

The basic mechanism like producers and consumers are similar across all event-based systems, which is illustrated by the circle in the center of the figure. The names of these concepts may be different in the sub-areas ([44] contains a table describing the different terminology for these concepts in the systems). However there are big differences in higher level logic like contraints or monitoring.

These differences have their reason in the different usage scenarios of these systems, e.g. Event stream processing (ESP) on one side deals with the handling of continous event data flows over certain channels whereas service-oriented and event-driven business process management

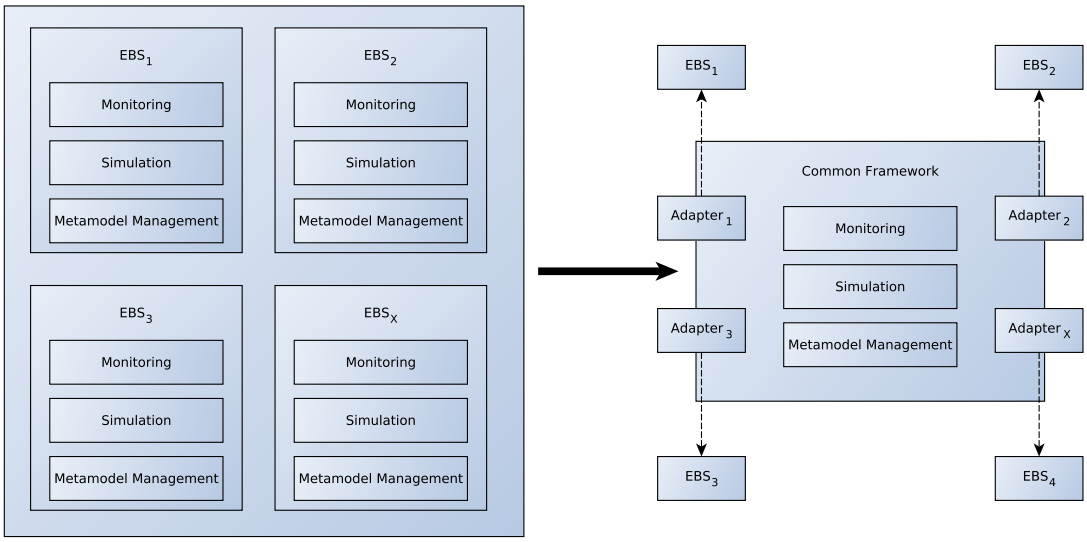

Figure 1.2: Illustration of the problem formulation

(EDBPM) has its focus on the managing of complete business processes [96]. This example shows the differences between these systems in a clear way.

On top of this model, simulations can be performed to ensure the fault-tolerance of the underlying system. Using a generic model, like the one described in this work, makes it possible to define generic simulation actions for different kinds of event-based system without the need of adapting it to the specific aspects of them.

1.3 Illustrative example

To illustrate the problem adressed in this work, the example of a stock market trading platform is used [31]. A simplified structure of such a market is illustrated in Figure 1.3. This illustration uses a sample workflow to illustrate the semantics of the example. The core concept of this mechanism is the virtual trading platform consisting of two areas. The inner area is used to generate quotations based on the offers submitted by customers. Since this is done using a complex algorithm, the task is distributed over several servers. In contrast, the outer area is responsible for receiving events from the customers and sending notifications to them. This area structure is used to divide the workload and separate the customer traffic from the inner logic of the system. The customers, in this case financial institutions, also have applications running on their machines to submit orders and receive notifications about issued orders. This system is a mixture between different event-based system approaches, since the outgoing notification to the customers is a publish/subscribe-like architecture, but the ingoing communication is a typical case of a server-client architecture. Additionally, the traffic between the two areas in the virtual trading platform can be seen as Peer-to-Peer (P2P) system.

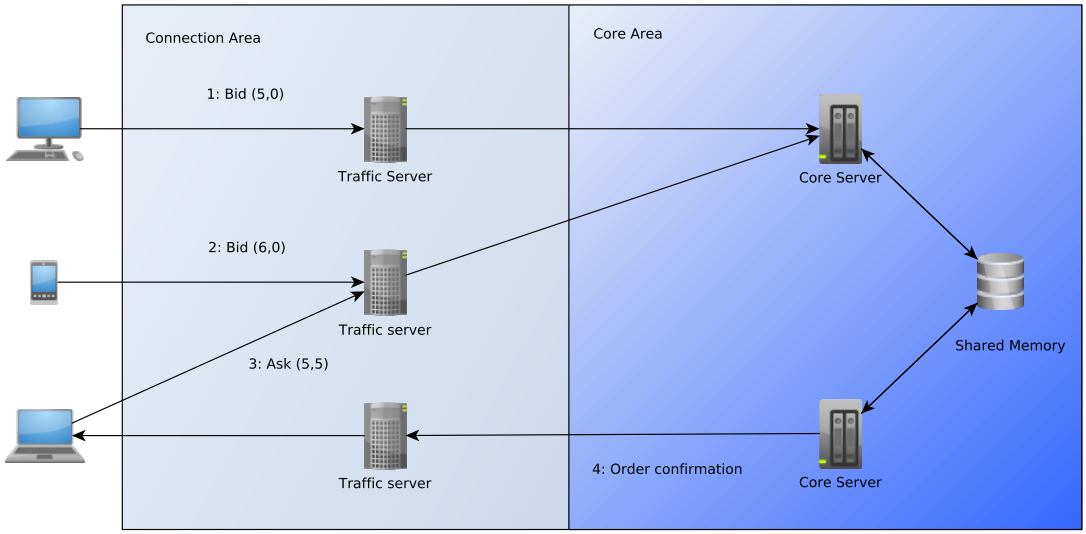

Figure 1.3: Illustration of a stock market system

One key aspect in such a high-risk system, regarding reliability, is fault-tolerance. The servers in the virtual trading platform will most likely be replicated across the network to deal with downtimes of servers. The problem here is that the monitoring of fault tolerance, if there is any, will be modeled to match the aspects of this particular system. If the semantics of the system change over time, the fault tolerance mechanism needs to be adapted with huge effort. To overcome this issue, a robust generic model of the system needs to be generated to support fault-tolerance and monitoring. This is the task of this work and for illustration purposes, the example, presented here, will be used throughout the work to explain the different aspects of it.

1.4 Contributions

The final outcome of this work is a generic model for different kinds of distributed event-based systems with a runnable monitoring and simulation environment on top of that model, which is named the EventSim framework in the following. The contributions of this model are:

- Implementation of the model using model-driven engineering (MDE) tools
- Runtime support for changes to the model
- Synchronisation between the model and a real system
- A simulator to adapt and monitor event-based systems
- Different modules for monitoring and fault injection

The system model used in this work is based on the preliminary work presented in [44]. This model provides an abstraction for the structure of the different kinds of event-based systems and additionally a fault taxonomy for this model is described in this paper. After adaption of this model (since there are certain variables needed for simulation purposes), it gets transformed to a generic programming model. This model is implemented using model-driven development (MDD) as described in [66]. Using MDD makes the model completely independent from current event-based systems, as it is directly generated from the domain model. This also ensures that a 1:1 transformation from the model to the programming model is possible. The concrete implementation used for deriving a programming model is described in [88].

The model synchronizes its state with a real-world application through adaptors, where an example adaptor is generated for the Storm framework [1]. The basic characteristics of software adaptors is provided in [63] with a focus on events. The handling of these adaptors is also done in a generic way, so that more adaptors can be written for different systems. Since they are included in the corresponding real-world system, the simulation model itself is not changed while adapting to a new system. This provides a generic way of simulating different kinds of distributed event-based systems described in the previous section.

On top of this, the EventSim Tool component is located with connections to the simulator and injection possibilities for real-world adaptors. The Tool component itself has no dependency to the underlying model, which enables further adaptions to the model, which may eliminate the need for creating tailor-made simulation solutions for each of these systems. It consists of several standard modules, e.g. for routing [94] or topology [2]. This plugin-like architecture is extendable and can, therefore, be adapted for future, yet unknown, systems.

Additionally, a Simulation component is provided, which enables the simulation of several scenarios using different strategies. This simulator adds the possibility to test specified routing algorithms and scaling strategies, which can be applied on-the-fly. Another usage for the simulator component is inserting predefined failure states into the model and investigate the corresponding model behaviour. On the other side, the simulator is also used to monitor a real-life system and notify the system if certain conditions are met. The granularity of these notifications can be chosen in a flexible way, from high-granularity to notifications for every event.

During development for this work, a testing platform is used to simulate various events and actions in a real system. This platform is implemented using different scenarios, which can be executed based on the given simulation target. An interface for events is provided, giving the Simulation component or any other component the possibility to insert events into the running system. At the end of the workflow, every model operation is communicated to the model.

An extra interface is developed for further use to change the underlying model and invoke a simulation according to this model. This makes it possible to change the inner structure of the model as a whole without the need of changing the simulation code considerablyf. These models are also stored in a persistent way and saved with the relation to a specific real-life system, so that further contributions can build their work on top of them.

Evaluation of the model is done by simulating several kinds of scenarios with the help of the Simulation component and the testing platform. This includes the monitoring of the normal

[1] http://storm-project.net/

system workflow, as well as transforming the model to a failure state. The connection to the testing platform is tested using the Storm framework.

1.5 Structure of the Work

After this Introduction, Chapter 2 describes the current status and work that has been done in areas related to this thesis. To that end, existing models derived for event-based systems are compared and their possible benefits and drawbacks are discussed. Chapter 3 deals with existing work in the sector of the different event-based systems and also papers describing the underlying base of this thesis, especially the used data model.

A concrete description of the model and implementation methods is provided in Chapter 4. The content of this chapter is a short description of the data model used in this work on one side and the architecture of the different components on the other one. The interfaces going out of the model are also described for further work in this sector. On basis of this architecture description, Chapter 5 characterizes the concrete implementation of the different components and which specific methods and patterns have been used for it.

At the end of the work, Chapter 6 analyzes the indicators received by performing simulations on the model. This concludes in an analysis of the usefulness of the simulation process described in this paper. Finally Chapter 7 concludes the work, summarizes the main findings, and points to future research directions.

CHAPTER 2

Background and State of the Art

This section deals with current work in the areas touched by this thesis. Particular areas of interests are event processing as a whole (including a section about the popular event processing language Esper), as well as the used programming paradigms like Model-Driven Engineering (MDE) or Reflection. In the last section, the Storm framwork is described, which is used for evaluation in this work.

2.1 Event Processing

Event Processing is a general term for all routines dealing with events. Since the term *event* can be found in several research areas, event processing is not a term exlusively used in the field of information systems [28]. An example for this is picking up the phone during a phone call, where the phone call would be the respective event. This section, however, concentrates on event processing in information systems.

An early example of event processing in information systems is the programming of a Graphical User Interface (GUI). If the user clicks a certain button or submits a form, this can be transformed to an event and the programmer handles it programmatically [28]. Events in general can have their origin in the real world or can be created automatically, e.g., as reaction to another event. The authors of [28] describe five different event processing applications and also note that a specific system can also be a cross-application of these aspects:

Observation This event processing application has only a passive role. A given system is monitored and if an exceptional behaviour occurs, an alert is issued to the user. Proper handling of this behaviour is left to the user, which results in the event processing system acting like a notification engine [76].

Information dissemination Systems in this sector deal with information forwarding. This does not only mean that information is simply forwarded but it also has to be forwarded to the

right component at the right time. Additionally, the information can be forwarded to different components in different ways, which leads to personalized information [29].

Dynamic operational behavior Basically, this term is a generalization for all systems, in which the output is affected by the input events of the systems. As a result, the behaviour of the respective systems is dynamic and cannot be predicted if one does not know about upcoming input events [39].

Active diagnostics This can be seen as an enhancement to the *Observation* application. A diagnosis of a failure is made based on symptoms received as event. In a fully automated system, resolution of the problem is also done automatically. In other systems, the user is informed about the root problem and, therefore, has better knowledge to fix it [71].

Predictive processing In contrast to the aforementioned applications, systems in this sector try to prevent certain events from occuring at all. They analyze logs, where certain behaviour has led to specific frauds and try to avoid those frauds to happen by dealing with this behaviour before the event can happen. An example of this behaviour is the restriction in some online communitys that you can only perform a search in a forum every five seconds. This way the system tries to prevent itself from Denial-of-Service attacks [41].

In distributed systems, event processing is performed using Event Processing Networks (EPNs) [84], which consists of several Event Processing Agents (EPAs) [26]. An EPN is dynamic, meaning that EPAs can be inserted and removed at runtime. The different types of EPAs are event sources, event processors and event viewers [72]. The interaction between these types of events is illustrated in Figure 2.1. Between those EPAs, events are flowing in one or both directions. Every EPA has an *in* and *out* side. So for a connection between two EPAs, the *out* side of an EPA is connected with the *in* side of another EPA [57]. So the typical workflow for an EPA is receiving an event, performing some operations and sending a new event to one or more EPAs. However, the last part is optional because if a certain activity is finished, there will be no new event. Additionally, a test (so called *guard* [57]) can be performed on the outgoing event before it is finally put in the *in* side of the target EPA.

Esper

Esper [48] [12] is among the most poular event processing languages (EPL), which enables basic and advanced processing features for events, described in this subsection. It has to be noted that Esper is not used by the EventSim framework. This subsection is rather used to describe Esper as an addition to the Eventsim framework. The exact differences and commonalities between these two systems are presented in the following.

The basic concept of an EPL in general and Esper in particular is the reversed idea of a database. While having data stored and queries running on top of them in the world of databases, Esper enables the storage of queries and applying data on them. Another difference between these two worlds is the time flow of these queries. Using databases, queries are executed at one point in time with a specific amount of data stored in the database. In event processing, the queries are

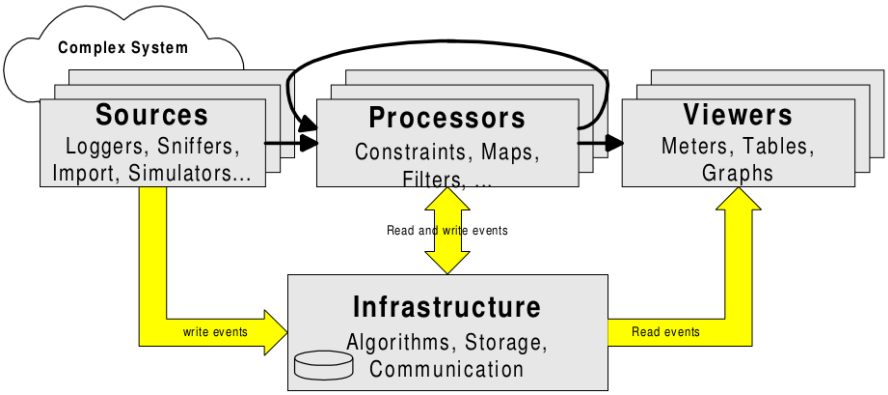

Figure 2.1: Different kinds of EPAs [72]

executed in a continuous way as real-time data (Esper can also handle historical data) comes in. This enables real-time analysis of data and adding new queries on-the-fly for real-time data. The results of the queries, however, can be stored in several formats like a database for further analysis. Therefore, EPLs are designed in a flexible way regarding their use cases and can help in different areas of interest.

Esper supports a rich set of event processing features, including the following:

- description of events using java objects, plain text or own event representation by a dynamic plugin architecture.
- continuous querying, including aggregations like `min`, `max` or `sum` and joins
- timing constraints, which includes rules on the sequential ordering, detection of missing events and assuring that certain events are sent within a given timespan.
- processing models, either via listener or an pull-based iterator
- correlation of events by using attributes joining them together
- support of input and output adaptors for several formats like CSV, HTTP and database binding.
- statements or queries can either be declared in textual format or using objects using the statement object model of Esper.

Example use cases for Esper are financial services, monitoring frameworks and event driven service-oriented architectures in general. The scenario can range from simple activities, like monitoring event exchanges, to more complex activities, like monitoring missing events, to high complex applications, combining all of the features provided by Esper.

In combination with the EventSim framework, presented in this work, Esper can be used to provide an all-in-one solution for event processing and event monitoring. As a result, the EventSim framework does not try to mimic the behaviour and features of Esper but rather complements it. Esper is a good solution, if one is interested in monitoring events and their timing constraints. On the other hand, it lacks support for monitoring the components handling these events. This is the point, where the Eventsim framework steps in and contributes to this field of research.

2.2 Model-Driven Engineering

Model-Driven Engineering (MDE) [80] has its focus on the domain of the underlying system and, therefore, provides an abstraction compared to traditional software development. The main elements of this paradigm are *models* and *metamodels* [32]. In this context a model can be defined as "*a description of (part of) a systems written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer*" [51]. Metamodels on the other hand are a definifition for the corresponding models, so it can be seen as "model of a language of models" [32]. It is used to describe the elements of the model and is therefore one level above the model. Additionally, there can also *meta-metamodels*, which are a description of metamodels, and (potentially) infinitely many levels on top of them.

Model-Driven architecture (MDA) [49] is a set of standards for MDE, established by the Object Management Group (OMG) [87]. On top of these standards, many technologies have been developed. The technologies which are most relevant in the scope of this thesis are briefly discussed in the following [65]:

The Meta Object Facility (MOF) This semiformal standard is used for writing metamodels and models. The problem in this context is that an error in the metamodel is propagated through the chain of models. Therefore, it is important to specify metamodels in a correct way according to the standards. [73]

Mapping functions Mapping functions in MDA are used to transform one model into another one. The advantage using this standardized mechanism is the capturing of expert knowledge, which can afterwards be reused if the model is extended. This also benefits the integration between models. [66]

Marking Models A *mark* can be seen as note on an element of the source model, which enables a differentation between elements during the transformation between models. These marks are not directly part of the source models because in some scenarios this approach would lead to complex source models. On top of that a *marking model* is a description for different types of marks. The combination of mapping functions and marks is called a *bridge* and is, therefore, the structure between two levels of abstraction. [65]

Agile MDA Agile methods in software development are used to deliver small fractions of the whole product to the customer in short development cycles of one or two weeks. Combined with MDE it means that small runnable models are developed in each cycle. The

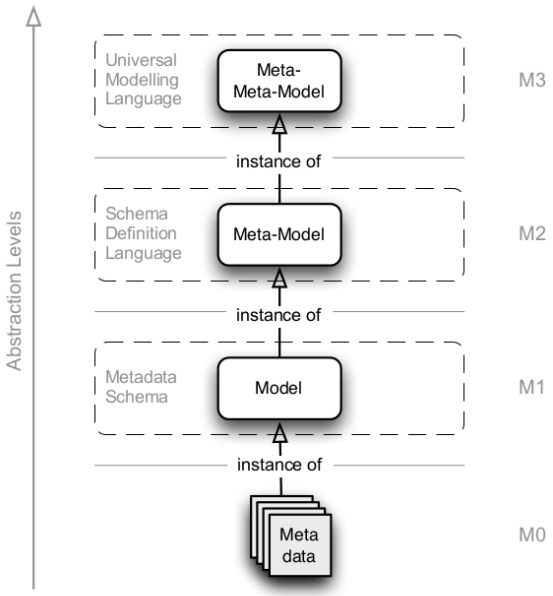

Figure 2.2: Abstraction levels of the MOF [37]

result are *executable models*, which are written in an abstract language. This means on one hand that the model is platform-independent and on the other hand that the customer has a better feeling for the current status of the project since it is easier to understand the abstract model than to understand code snippets. [64]

The core of the MDA is the standard used for describing the highest level of abstraction, i.e., the highest meta-model. Currently the common standards used in practice are MOF, Unified Modeling Language (UML) [78] and Common Warehouse MetaModel (CWM) [62], which is mainly used for data mining across databases. It has to be noted, though, that these three standards are not on the same abstraction level. These abstraction levels for MOF are illustrated in Figure 2.2. MOF and UML are so called M3-models (or meta-metamodels), whereas CWM is a M2-metamodel. On top of the modeling core, several middleware technologies can be placed to build applications for several uses, like Finance or E-Commerce [42]. The technologies mentioned in Figure 2.2 are currently used and therefore subject to change in the future.

The transformation process between different levels of abstraction should match several requirements if it is used in practice. If these requirements are not fulfilled, traditional software development methods seem the better choice for a project. Therefore, defining these transformations is an important part of MDE. The basic requirements are [13]: reuse, composition, genericity, customization and maintenance.
All of these aspects are pointed towards the goal to achieve a generic flexible transformation mechanism, which can be reused and adapted in the future to match new needs. An example for a transformation can be seen in Figure 2.3. The models in this example are written in the XML

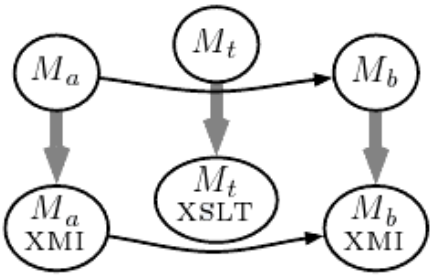

Figure 2.3: Example transformation using MDE [13]

Metadata Interchange (XMI) format [97], whereas the transformation model is written in the Extensible Stylesheet Language Transformations (XSLT) format [50]. So to transform model M_a to model M_b these concrete formats are used for communication between the models and the transformation meta-model M_t.

The complete workflow of MDA contains a number of abstraction levels between the source- and the destination-model. The number of hops is determined by the different problem domains between the source- and destination-level. This work uses one metamodel, which is described in Section 4.

2.3 Reflection in programming languages

Reflection is a term which describes the possibility of a programming language to examine and perhaps manipulate its own behaviour and state. In terms of processes this would take part in the reasoning process [86]. The idea of reflection basically folllows six different properties [86]:

Relation between reflective and non-reflective behaviour The reflective behaviour can be understood as *description*, whereas the non-reflective behaviour can be understood as *reality*. So if one manipulates the *description* of a system, this has to have an effect on the *reality* of this system in the future. Seeing it the other way round, a change of the *reality* now, has to have an effect on the *description* in the future as well. The information flow between those two systems has to be done in a balanced way, since full information flow would make the system too complex in the end.

Theory This property ensures that reflection can only be done with the knowledge of the theory, which describes the underlying system. Since reflection deals with the problem of self-knowledge, the theories of the respective system have to be used for it. This has been part of literature in the field of computer science and in other parts of science as well [19].

Naming The term *reflection* states out the fact that the idea behind it is to get an understanding of the underlying state and behaviour of the system, which means that one basically needs to take a step back from its own to have a perspective on every aspect of the system.

Grade of control With the use of reflection, it is possible to have a more detailed control over a system than otherwise. This provides the possibility to achieve *simplicity* and *flexibility* at the same time. The first one can be achieved by using a strict and simple description of the system, whereas the second concept can be done using reflection. This provides the user the possibility of adjusting the system at runtime.

No complete detachment Although the concept of reflection provides the possibility to change the state and behaviour of a certain system, it is not possible to get a complete separation between the non-reflective and the reflective mode. Both modes share the same background and, therefore, every reflection process has its limitation.

Reflection as core part of the system Reflection cannot be built on top of a given system but has to be built in the core of a system. This has its reason in the theory-relative approach described before. If the system itself shall be manipulated, it is mandatory that reflection is part of the system and not only an extension to the system.

In this work, Java Reflection is used mainly by the EventSim Tool component (discussed in Section 4.1), because the underlying model code is completely unknown at compile time. This means that all of the services attached to the model need to be programmed in a reflective manner to support the generality of the model. Java provides metaobjects to access information about the objects at runtime. The main objects for this task are *Class* and *Method*. Using these metaobjects, one can change or access the state and behaviour of the objects currently loaded at runtime.

2.4 Models@run.time

Dynamic and self-adaptive applications often need to change their underlying data models. In modern applications, the need for this change occurs at runtime and not at design-time. Research in this area is subsumed under the term Models@run.time [14]. The term 'model' in this context refers to a dynamic abstraction of the system and defines the structure and behavior of it. Additionally, these models are implemented in a reflective manner to support the manipulation of it during runtime [11]. According to Vogel et al. [95], models at runtime can be categorized by their level of abstraction using the following categories, which are illustrated in Figure 2.4:

Implementation Models These models are directly coupled to the underlying programming language and used to modify the system over time. Due to this connection, the abstraction level of these models is low compared to the following ones.

Configuration and Architectural Models On a higher level of abstraction, these models describe the configuration and architecture of the underlying system and are, therefore, platform-independent. Concrete use cases for them are monitoring or analysis of a system.

Context and Resource Models Using these models, one can determine the operational environment of the system and, therefore, achieve knowledge about the situation of an entity. This layer resides between the layers mentioned above, depending on the specific system.

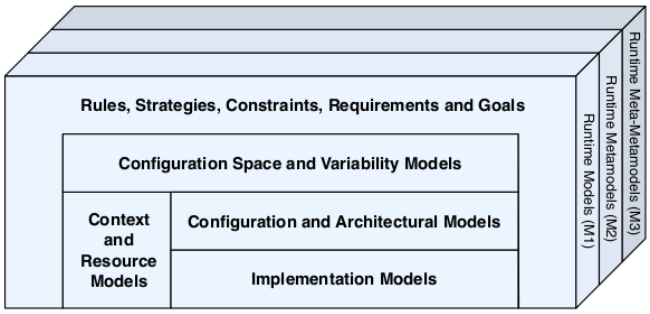

Figure 2.4: Different Categories of runtime models [95]

Configuration Space and Variability Models Compared to the configuration and architectural models, these models are used to describe all possible configuration variants of the system, rather than describing the concrete configuration in use right now.

Rules, Strategies, Constraints, Requirements and Goals This is the highest abstraction layer, which is able to describe every of the above mentioned models and would reside on the relevant layer of the respective model.

The models used in this work are located between the implementation and the configuration layer because they are platform-dependent at the current stage but can be used for analyzing and monitoring the system.

2.5 An eventing platform - Storm

In this section, background information on the Storm framework is provided, which is utilized as evaluation platform for the approach developed within this thesis. Storm [1] is a distributed realtime computation system and can, therefore, be used as example of an event-based system. The version used in this work is Storm 0.8.3-wip3 (as of August 2013). In the following, the characteristics of Storm (regarding architecture and fault tolerance) are highlighted to understand why this framework was chosen.

Architecture

The application code in Storm is packed in a topology, which is composed of Spouts and Bolts [61]. Topologies can either be created in a distributed way, where tasks are distributed across the nodes of a cluster, or in a local mode, where the nodes are simulated with threads. These are the components used for communication. Spouts are the source of a topology and usually use an external medium to transfer events into it. This medium can have every possible format, e.g. a webservice, like the Twitter API, or a file on the local hard disk. Since this external source is theoretically of an infinite amount, topologies in Storm are running until they are killed. These

[1] http://storm-project.net/

spouts are usually located at the beginning of a topology, whereas bolts are used for every other communication and computation.

Messages sent from a spout can either be reliable or unreliable. The case of reliable messages is explained in the last subsection of this section. If messages are unreliable, there is no fault tolerance regarding message sending. Since Storm is normally used in a distributed parallel way, every component can have multiple specific nodes, which are handling its payload. The amount of parallelism for every component can be specified via configuration values and is described in detail in the next subsection.

The communication between spouts and bolts is done using streams. These streams are used to transport tuples, which are an abstraction of any serializable object. The flow of communication between the nodes of the topology can be defined via stream groupings, with the most important being:

Shuffle grouping Using this grouping is the standard method for achieving a high grade of parallelism since the tuples are equally distributed across the nodes of the bolt.

Fields grouping The stream is partitioned by a specific field of the value. All tuples with the same value for this field are emitted to the same node.

All grouping Streams are replicated across all nodes of the cluster, which leads to high redundancy but no parallelism.

Global grouping This is similar to the *Fields grouping* approach with the difference that every tuple of a given stream is emitted to the same node. This is for example used by batch jobs, where one node has to know every event.

Direct grouping This is a special mechanism provided by Storm, where the event producer decides the receiving node.

Every component can have either multiple input or output streams, except spouts, which only have output streams. Because of this flexibility, there is the support for several processing patterns [60]:

Joining Multiple data streams can be joined together to a joined stream. The concrete implementation of this join (e.g. different tuples put on one field or one field per kind of tuple) is left to the developer.

Batching In high-performance systems, tuples can be saved in one component and collected for a batch call. This can for example be used in a database management system, where every event triggers an update and these updates are done in a batch job. The tuples are acknowleged at the time of processing. For reliability, multi-anchoring is used, which is described in the subsection about fault tolerance. The drawback of this method is the lack of parallelism because all of these tuples must be located on the same node.

Caching Some applications use a cache on the nodes to minimize computation time. Storm provides the possibility to assign all values of a field to a specific node. This reduces the

amount of caches throughout the system and optimizes the cache of this node. Otherwise every node of this component would have a cache with some values, which is not an optimal approach regarding performance.

Staged processing If the use case is to get the five highest order amounts in an order management system, this is normally done by saving every order in a node and calculating it on this node. Storm provides the possibility to calculate this for every node, which received orders, and have one node collecting these calculated amounts and determining the highest amounts of the whole system.

Environment

Storm uses Zookeeper [2] as coordination server [45]. It is a service, which provides multiple functionalities for a cloud like naming, maintaining configuration information and providing distributed synchronization. The architecture of Storm is a mixture of a Client-Server and Peer-to-Peer (P2P) system. The main node is called *Nimbus*. This node keeps the information about topologies and the different *Supervisor* nodes, on which the worker processes are started. The communication between these nodes is done through the cluster managed by Zookeeper. Responsibilites of the Nimbus node are monitoring topologies, rebalancing topologies and telling supervisor nodes to start a new worker process if necessary. A supervisor node is started on every physical machine used as a worker. This node is used to directly start and kill worker processes on the machine. Monitoring of the cloud can be done by using a provided Web-GUI or by connecting to the system using Thrift [3].

Storm itself has multiple abstraction levels for the components of a topology, called *worker*, *executor* and *task*. The concept is explained in the following, using an example illustrated in Figure 2.5. This example has three components, one spout and two bolts. The number of workers is set to two, which can be changed for every topology. Each of these workers is a Java process using its own JVM. The *parallelism hint* specifies the number of initial executors per component. Dividing the total amount of parallelism by the amount of worker processes leads to the number of executors per worker process. An executor is a thread inside the corresponding worker process. Tasks are used for the actual data processing. If not specified otherwise, one task is initialized per executor. In this example, the amount of tasks for the blue bolt was set to four. Since the blue bolt uses two executors, every executor has two tasks. The green bolt in this scenario only has one executor, which is normally used if every event workflow must go through the same thread.

Fault Tolerance

Since the Nimbus and Supervisor nodes are basically decoupled, killing one of them does not kill the entire system. If the Nimbus node is killed, topologies can not be removed or added and there is no rebalancing of the topology any more. The computation, however, is running

[2] http://http://zookeeper.apache.org/
[3] http://thrift.apache.org/

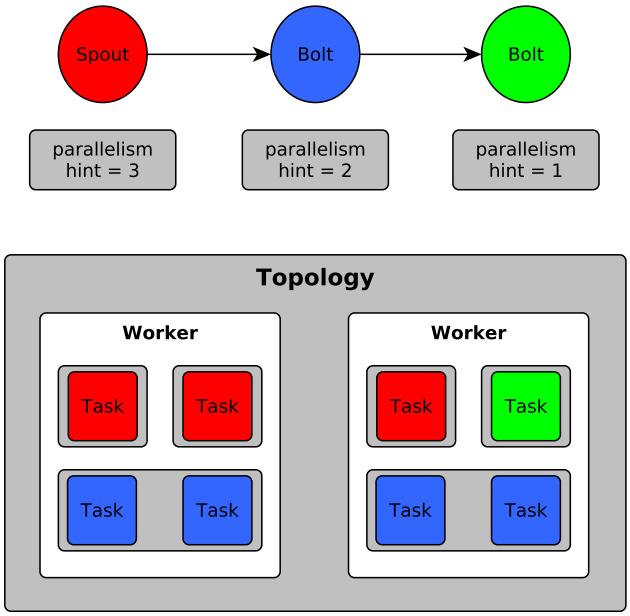

Figure 2.5: Example of a Storm topology

like before the node was killed. The same thing applies for a Supervisor node: new working processes can not be started on this machine any more but the old processes are running like before. Killing a worker process does not affect the topology at all, since a new worker process is started as soon as the change is recognized.

Regarding tuples, they can be reliable and unreliable, as mentioned before. If a tuple is set to be reliable, a tuple tree is created for it. Every new tuple has a random 64bit ID, which is generated by the spout [59]. If a bolt emits a new tuple based on a received one, the old ID is copied onto the new tuple. This makes it possible, to track a tuple throughout its lifecycle. Every bolt is responsible for acknowledging a tuple after it was processed. This ack-message is sent to the relevant source component including the information about new tuples based on the old one. The acking-mechanism is an automatic communication in the Storm framework apart from the application-specific one. In case, a tuple is not acked in a given interval (which can be set by the user), it is replayed by the first component in the tuple tree, which did not receive an ack-message. New tuples can be based on one tuple, but also on multiple tuples. This can be used if tuples are joined together. This mechanism leads to a prevention of data-loss for several failures [59]:

The processing task dies In this case, the spout tuples with the respective ID from the root of the tuple trees are replayed.

Acker task dies This damages the Storm fault tolerance mechanism directly. As a result, every tuple, which was kept track of, is replayed.

Spout dies This is the responsibility of the external medium, the Spout is receiving events from. In an optimal scenario, this medium should recognize the failure and buffer the new events until the spout is back.

Putting this information together leads to the choice of Storm as the primary evaluated framework used in this work. It is used to validate the simulation actions described in Chapter 4 based on an evaluation scenario, the results of the evaluation are presented in Chapter 6. In future work, other frameworks can also be used with the EventSim framework to compare the results between different event processing systems.

CHAPTER 3

Related Work

This section deals with background areas of this work. Therefore, the research field of Model-Based Testing is analyzed, as well as different ways of fault handling, like injection or detection, and monitoring/adaption of systems at runtime. The motivation for this work can be seen on the fact that there are solutions to each of these problems, but there is no framework combining these fields to gain a model-based simulation and monitoring environment for event-based systems.

3.1 Model-Based Testing

Models are not only used for the development of software, as presented in section 2.2, but can also be used to test a given system. The system, which is tested, is then called *system under test (SUT)* [74]. Test cases are input/output combinations of a test model which equal the respective parameters of the SUT. The model is built using the requirements of the system, which are the output of requirements engineering. Since the model is an abstraction of the SUT, it must be simpler than the SUT. If this is not the case, then this approach will be inefficient compared to normal testing. On the other side, the model must have enough information included to make meaningful tests for the system. [90].

Model-Based Testing is done as part of the integration- or system-test phase [70], because the focus of such a test is not the implementation, but rather the system as a whole [5]. The model describes the major transactions in the system and does not evaluate the code basis used for it. Model-based testing techniques can be based on various types of models, for instance finite state machines (FSM) [20]. A FSM describes states and the transitions between them. Every transition has an input event (describing the reason for the transition) and a target state. There can also be multiple outgoing or ingoing transitions from or to a state. A model test, described as test script, consists of several test primitives which can be of one of these types [5]:

Setup of test environment Sometimes the system must be in a specific state to test a given behavior. Additionally, some actions can have multiple outputs and therefore the test environment is adjusted in a way to accomplish a deterministic path through the system.

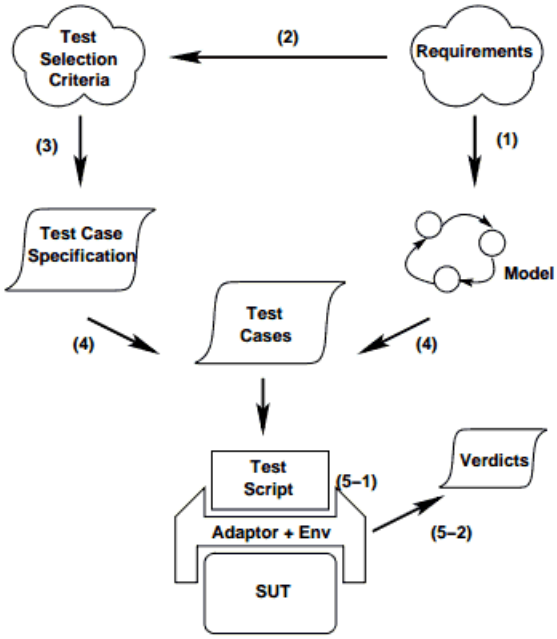

Figure 3.1: Model-based testing [90]

Test stimulus A stimulus is used to start a workflow in the system and can be seen as external event.

Verification This can be done using several methods. The most used are text comparison (in the console or a GUI) or checking the return value of a function. In some cases there is no need for such a check since a given stimulus would not get accepted by the system if the method before was not successful.

Reporting & Logging After a test run, the results can be reported and logged using a normal output, some predefined loggers or interfaces to an automated test system.

The workflow of model-based testing is illustrated in Figure 3.1. First the model is built using the given requirements. These are also used to define the criteria for the test suite, which can be made towards different goals, like coverage or fault detection. Using these criteria, test case specifications are described in a formal way.
These specifications combined with the model are used to derive a test suite, which is written as script. The communication with the SUT is done using one or more adaptors and a predefined environment for the test case. The result of the comparison between the output of the model and the SUT is called *verdict*. The state of it can be *pass*, *fail* or *inconclusive*. The last state is chosen if the result of the comparison cannot be determined at the time of testing.

3.2 Fault Tolerance

In computer science three main terms are used if a software did not work as expected. The definition by Avizienis and Laprie [7] is still used today: *A system **failure** occurs when the delivered service deviates from the specified service, where the service **specification** is an agreed description of the expected service. The failure occurred because the system was erroneous: an **error** is that part of the system state which is liable to lead to failure. The cause in its phenomenological sense of an error is a **fault**. An error is thus the manifestation of a fault in the system, and a failure is the effect of an error on the service.*

Therefore, a fault tolerant system is able to prevent faults from causing failures [1]. It has two components [82]: fault detection (described in the next subsection) and fault repair. Fault repair is very system-specific and thus there exists no general approach for it. Testing of a fault-tolerant system can be done using fault injection, which is also described in the following. The last subsection deals with the specific field of fault tolerance in event based systems.

Fault Detection

Venkatasubramanian et al. [93] described ten characteristics of a fault detection/diagnosis system: Quick detection and diagnosis, Isolability (distinguish between multiple faults), Robustness, Novelty identifiability (decide if the system is working as expected), Classification error estimate, Adaptability, Explanation facility (explanation of the cause), Modeling requirements (minimal modeling effort), Storage and computational requirements and Multiple fault identifiability.

The methods used for fault detection can be divided into model-based (quantitative and qualitative) and process history-based [93]. Model-based methods can be used with a-priori knowledge of the process and its underlying structure. In this case, the knowledge is expressed in a quantitative (e.g. mathematical functions) or qualitative (e.g. descriptions of the relationship between process units) way. Unknown parameters are estimated using direct mathematical functions or iterative procedures. The observation itself can be state- or output-based [46]. The concrete observer component depends on the goal of the detection and the system itself.

Process history-based methods are used if there is no a-priori knowledge about the model, but a large amount of historical data from previous process runs. These methods can also be divided into quantitative (e.g. statistical analysis) and qualitative (e.g. expert systems) measurements [93]. On an higher level, neural networks can also be used, which are able to learn and understand the system as more logs are provided [92]. Additionally, hybrid methods can be used to combine the benefits of two or more approaches.

Fault Injection

Fault injection is used for two major goals: Validation and design aid [6]. Regarding validation, one aspect deals with the validation of the concrete system and building test sets to find faults during development. On the other side, fault injection is also used to detect the dependability of the system at the operational phase. Laprie [53] introduces two goals in this context, which

are also part of the validation phase: *fault removal* and *fault forecasting*. Fault removal is done to minimize the amount of faults using verification and fault forecasting estimates future faults through evaluation.

The design aid goal is touched in early development phases and used to enhance the development process by establishing feedback loops out of negative results from fault injection [6]. Additionally, fault injection in the design phase can be used to build fault dictionaries [15]. These fault dictionaries can afterwards be used in the diagnosis mechanism, like the fault taxonomy described by Hummer et al. in [44].

Hsueh et al. [43] describe three different methods of fault injection. *Simulation-based fault injection* is used to measure the effectiveness of fault-tolerant systems, as well as the dependability of it [35]. Therefore, accurate input parameters of the system have to be estimated. Using *prototype-based fault injection* leads to a better understanding of the system, as bottlenecks of the dependability graph can be identified. For this, a prototype or operational version of the system is used [6]. As a last method, *measurement-based analysis* can be performed if a high amount of real data is available [43]. These historical data are then injected into the system to analyze fault handling and to measure the performance of the system under workload.

Software faults can either be injected at compile-time or runtime [43]. If the fault is injected at compile-time, it can be seen as permanent fault, since it is part of the program, as long as the current compiled version is used. These faults can easily be inserted if one has access to the sourcecode, because it only requires a modification of the source- or assembly-code. On the other side, using only this phase does not allow for a dynamic injection of faults. Runtime injection uses different triggers. A time-based trigger is used for injecting faults at a given time, which leads to unpredictive behavior, as the injection is not based on the current state of the program [36]. Another trigger is exception-based and inserts the fault based on an occurred event. This leads to a more predictive behavior than a time-based trigger. As a last option, code insertion mechanism can be part of the program, which allow the insertion of faults at runtime. This is similar to the insertion using exceptions with the difference that the fault can be inserted at any time using this method. The approaches mentioned in the last paragraphs can also be combined to get a mixture out of compile-time and runtime injected faults. Such a mixture is also used in this work.

Fault Tolerance in Event Based Systems

Fault tolerance in distributed systems generally is defined with the following characteristics [34]: availability, reliability, safety and maintainability. As described by Bruning et al. [17], faults in distributed systems can have its origin in hardware, software, network or operator issues. However, the specific nature of event-based systems, given that they are distributed, requires a special taxonomy of faults. Taxonomies, like those presented in [58] or [17], can be used for an event-based system, but do not pay attention to the characteristics of it. Hummer et al. [44] described a fault taxonomy for event-based systems, which is illustrated in Figure 3.2.

The fault classes in this illustration are based on [8], which introduces 16 classes. Faults can occur either at development time or at runtime. The level of abstraction can either be low (platform-level) or high (business-logic level). It can have its origin from inside the system

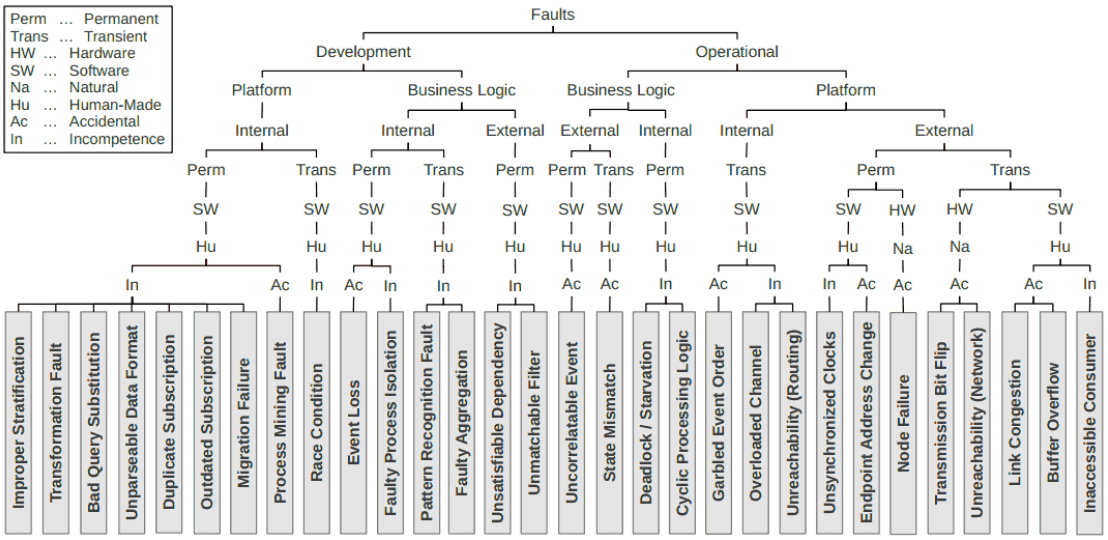

Figure 3.2: Fault taxonomy for event based systems [44]

or be injected from an external source. A fault can be persistent in time or exist for a given amount of time (transient). As with every system, the fault can be hardware- or software-based. The reason for the fault can either be a human mistake or natural. As an extension to this class, faults can either be described as accidental or based on incompetence.

The faults can also be attached to the different kinds of event processing systems, which have been described in Section 1.1 and illustrated in Figure 1.1. The combination of event processing systems and faults is also described in [44].

Based on the work done by Hummer et al. [44], this work uses a modified version of the data model described by them and uses some of the illustrated fault types for testing purposes. The exact fault types in use, are flexible in the framework and can be adapted in the future. This concept is described in Chapter 4.

3.3 Runtime Monitoring and Adaptation

Runtime monitoring is used in the simulation part of this work to examine the behaviour of a SUT. In the literature, there are several requirements to runtime monitoring and adaptation based on the results of monitoring. In the following, current approaches adressing this issue and the approach of this work are discussed in more detail. A monitor of a distributed system needs to handle several issues [47]:

Time of fault detection The system should be able to detect faults in a minimal time after it happened

Number of messages Monitoring of events in an event-based system has to be done using extra messages because they can occur on different nodes of the system. The amount of these

extra messages needs to be small.

Granularity of clocks and timer Since different nodes in a distributed system can have different local times, there needs to be a method of synchronizing the time. This ensures that the time of an event is the same for all nodes.

Resource management Since monitoring is not part of the system itself, it must not harm the timing constraints of the system and allocate too many resources away from the system.

Based on MDE, presented in Section 2.2, monitoring can also be combined with models, resulting in the term *model-driven monitoring* [40]. The model used for monitoring is not the same as the functional model used during development, but a subset of it with all specifications relevant for monitoring. The lifecycle of the model is checked throughout the complete workflow of the system and, afterwards, validated using the specification of the system. This ensures that the overhead of monitoring is small since only a subset of the functional model is needed. In this work, only the functional model is used for monitoring, since it is rather small and leads to no significant overhead.

Depending on the results of runtime monitoring, runtime adaptation needs to be done if the environment of the system has changed or faults have been detected. In modern distributed systems, adaptation is achieved through dynamic insertion, removal or manipulation of components [69]. This requires a component-based software design as described in [38]. In a component-based system, every component executes a subset of the workflow, hides its implementation details from the other components and therefore follows the principle of *separation of concern* [89]. The components communicate with each other using well-defined interfaces. There exist several requirements, which need to be fulfilled for dynamic recomposition [69]:

- The system must provide compositions for the different states of the system.
- The replaceable component must have the same interfaces as the replacing component.
- The pre- and postcondition of all methods of these interfaces need to be the same.
- A persistent state of the old component must be transformable to a persistent state of the new component.
- If a component is removed, it must be replaced or all dependent components are removed.
- If a component is added, every component, it depends on, has to be present or added alongside the new component.
- A new component must be able to continue the work of the component, it replaces.
- The integrity of the communication between components must not be harmed by the recomposition.

The model used in this work meets all these requirements and can, therefore, be used for recomposition of its components if the underlying framework supports it. In the evaluation part of this work (see Chapter 6) recomposition is used to deal with different faults injected into the system.

Since the requirements in the context of runtime monitoring and adaptation are strict (e.g. concerning granularity and monitoring overhead) and the spectrum of EBSs is widespread, there exist a wide variety of approaches in this area. In the following, different aspects of these approaches are highlighted.

Sankar et al. presented a monitoring approach, which is built on top of a formal description of the underlying program [79]. This approach can be used for programs, completely different in their characteristics, because of its generic nature. The checking routine of their approach is implemented using checkpoints, a time where certain checks are performed and the program is stopped if a failure occurs. This is a convenient approach, when dealing with sequential programs, but cannot be applied to event-based systems. Another important aspect is the generic approach of their paper, which is not necessary for the domain of event-based system. This leads to more complexity regarding the formal definition of the program.

The monitoring of timing constraints in distributed systems was discussed in [47]. It is used to state constraints about the time frame an event needs to navigate through a chain of nodes. The constraints are always stated in regard of events. This way, the workflow of different events, as well as maximum time frames for specific actions can be stated. On the other side, constraints for components (e.g. a minimum amount of redundancy) cannot be specified using this system. Therefore, this system can only be used as partial solution if the validation of the SUT should also cover other aspects than timing constraints.

Similar to these approaches, there has been made much research work regarding the application logic or specific attributes of an event-based distributed system (e.g. [22], [85] or [55]). The common aspect of all these worflows is that they are specific to a certain problem: either timing, application concerns or the validity of the node structure. This thesis seeks to unify these different aspects of EBSs and proposes an integrated solution covering the different aspects of them.

CHAPTER 4

Solution Design and Architecture

This chapter explains the characteristics and features of the EventSim framework, which was developed as goal of this work. As a first step, the architecture as a whole and the communication between components in it are presented. Afterwards, the metamodel for event-based systems used in this work is described, followed by an explanation of the different components of the system. The subsequent sections deal with the different aspects of the simulation workflow of the framework and how the interaction between the Tool and Simulation component works. In a concluding section, the necessary steps to add a new platform are described.

4.1 Architectural Overview

The architecture of the system is illustrated in Figure 4.1. This is an overview over the whole EventSim framework, a more detailed view is explained in Section 5.1. According to this illustration, the main components of the system are the following:

Modeling Component This component is the interface accessed by system developers, which have the goal of modifying their event-based system for usage with the EventSim framework. It provides a graphical user interface for the Model Generation component but has no own logic attached to it. Since it is not part of the core of the EventSim framework, this component is subject to future work. For testing purposes and users with experience in the relevant modeling technology, this component is not necessary.

Model Generation The underlying model of the whole EventSim framework is a metamodel for event-based systems. For this work, the metamodel described in Section 4.2 is used. The system developers, however, are free in their decision to change this metamodel or even create a new metamodel from scratch. This component provides the possibilities for these routines, as well as access to a storage system, where different metamodels can be stored for different systems and different versions of these systems. Additionally, this component is responsible for creating the metamodel code out of a given metamodel

created by it. This code is then used in conjunction with the specific platform code to test event-based systems. Concluding, the metamodel, created or edited using this component, is the code base used by all subsequent components. All of the features of this component can be accessed using the Modeling Component because of provided service interfaces.

EventSim Tool This component is used for storage and partly for testing. It consists out of two subcomponents: the *ModelStorage Service* and the *SystemStorage Service*. The ModelStorage Service provides the possibility to create, update and remove persisted models created by the Model Generation component. In its current version, a file storage is used for this. The SystemStorage Service on the other hand is used for the concrete model of an event-based system at runtime. It also provides different operations for persistence with the difference being that this model is kept in runtime memory, because of performance reasons. Additionally, this service notifies registered components about changes in specific systems. This mechanism is used by the Simulation component to keep track of changes regarding the observed event-based system. Every component, which has an service endpoint, can register itself for notifications about state changes. Another important feature for simulation is the heartbeat mechanism, described in Section 4.5. It is used to recognize inactive components. Since this component is under heavy load if different systems are monitored at the same time, it can be replicated, using a cloud, in the future. This distributional approach is possible for the relevant data backends of the services, or for the services as a whole.

Simulation Although the name of the component only refers to the simulation part, it is used for simulation and monitoring. The current state of the system under investigation is monitored using a notification mechanism. Manipulation of the system is done using standardized actions for every system-under-test (SUT) and optional actions specific to the relevant system. Current implemented actions are described in Section 4.3. The Simulation component registers itself at the EventSim Tool component to receive information about ongoing changes concerning the SUT. An additional part of this component is a graphical user interface (GUI), which provides access to every collected information and the different actions defined for a specific system. The concrete user interface of the simulation is adaptable for every system if necessary. If no specific GUI is necessary for a new platform, the standardized GUI can be used without the need to change anything.

Platform In practice, this can be any event-based system, which uses the other components of the system (especially the metamodel generated by the Model Generation component) and their respective interfaces. Every change in the event-based system and every new event is communicated to the SystemStorage Service. So the internal states of the platform and the persisted state in the SystemStorage Service for this system are the same throughout the lifecycle of the system. Additionally, actions can be performed on the platform by the Simulation component. For this work, a test platform is implemented, which provides common features of an event-based system to test and benchmark the EventSim framework. The test platform, used in this work, is designed using Storm, which was presented in Chapter 2. For simulation purposes, it receives events from the Simulation component.

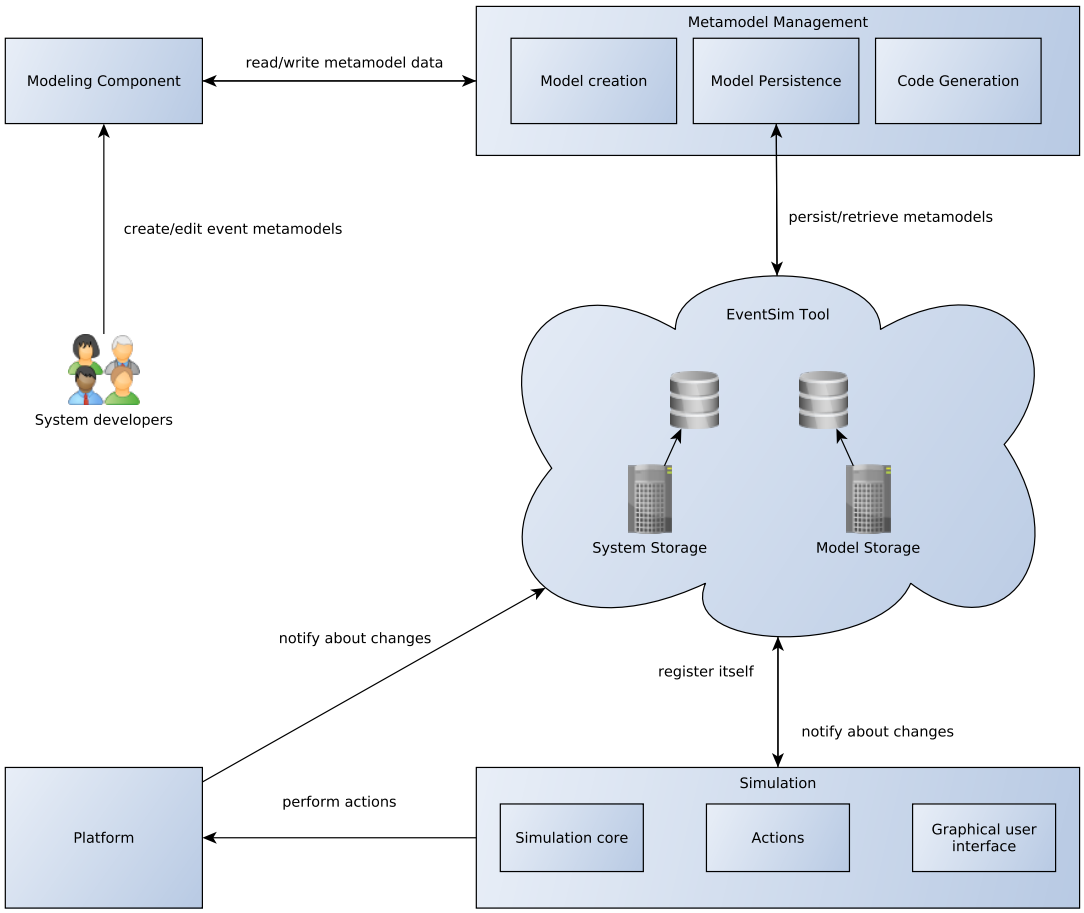

Figure 4.1: Architecture of the EventSim framework

Regarding the stock market example, presented in Section 1.3, the platform is the code base of the virtual trading platform, which is deployed on the relevant servers.

To understand the commmunication flow between the components, Figure 4.2 illustrates the required steps to integrate a new platform. Therefore, the stock market example, presented in Section 1.3, is used for clarification of the individual steps. The steps follow the same sequential ordering as the components in Figure 4.1:

Metamodel and code generation The metamodel is modeled using the ModelingComponent. For experienced users, the ModelingComponent can also be skipped to write the relevant metamodel directly using the underlying modeling language. However, using the ModelingComponent is the more generic approach since the underlying technology is subject to change. In *Step 1.1*, code artifacts are generated, using this metamodel, by the ModelGeneration component. The ModelStorage Service can be used to persist metamodels or load existing metamodels and edit them. As a result of the whole step, the location of

the packaged classes is returned to the user. Currently, the packaged code is transferred to a server and can be downloaded using a simple HTTP download. This package can directly be used in the platform to connect it with the rest of the framework and allow the usage of it for simulation purposes. Specifically this means that components of the SUT described by a certain element of the metamodel must be inherited from this element. In future versions of the framework, the delivery of the code package can be adjusted to the specific needs of the platform developers. However, it is important that all components of the framework and the relevant platform use the same version of the package. Regarding the stock market example, this step can be skipped, because the metamodel, presented in Section 4.2, is sufficient to fulfill the needs of this example.

Adaptation to metamodel The new platform needs adaption to be compatible to the Tool component. Components of the platform have to be connected to the generated classes and the simulation has to be notified whenever there is a change in the system components. This means that the Tool component has to be notified about every noticed change in the element of the system, as well as all node-like components of the platform have to implement the heartbeat-mechanism 4.5. There is no need of an additional adaptation except these described routines. The main components for adaptation in the stock market example are the traffic servers, as well as the computation servers and the events between them. These elements must be defined as children of the respective classes in the metamodel.

Implementation of connectors and actions The simulation GUI is implemented in a generic way and only needs adaptation regarding information retrieval if the platform does not support the sending of notifications or heartbeat-messages. In these cases, it has to be specified how the components and nodes for a specific event based system are retrieved. This information is used for two purposes: On one side the relevant systems of a specific kind can be displayed for choosing in the simulation GUI and on the other side, the list of components is automatically refreshed in the GUI if the SUT allows to retrieve currently active components. If the user has the need for platform specific simulation actions, these are also implemented in this step. Currently, only general actions are implemented. The implementation of connectors and actions for the stock market example depends on the architecture of the example and the specific needs for simulation.

Adapt and start the platform After the adaptation is done, the platform is started. If the simulation is running during start-up time, the notification happens automatically as soon as an action is triggered on the SystemStorage Service. The Simulation component has to subscribe to specific notifications at the SystemStorage Service to receive them afterwards. This is done dynamically at runtime. The notification interface is public and, therefore, every component can subscribe to them, not only the Simulation component. As soon as the platform is started, a mechanism for sending heartbeat messages for every component also needs to be started to avoid inconsistent states for the Simulation component. As for the stock market example, the traffic servers and computation servers need to be extended in a way that they are able to send heartbeat messages. Additionally every creation of such a server and the sending and perhaps the receival of an event must be communicated to the Tool component by the relevant server.

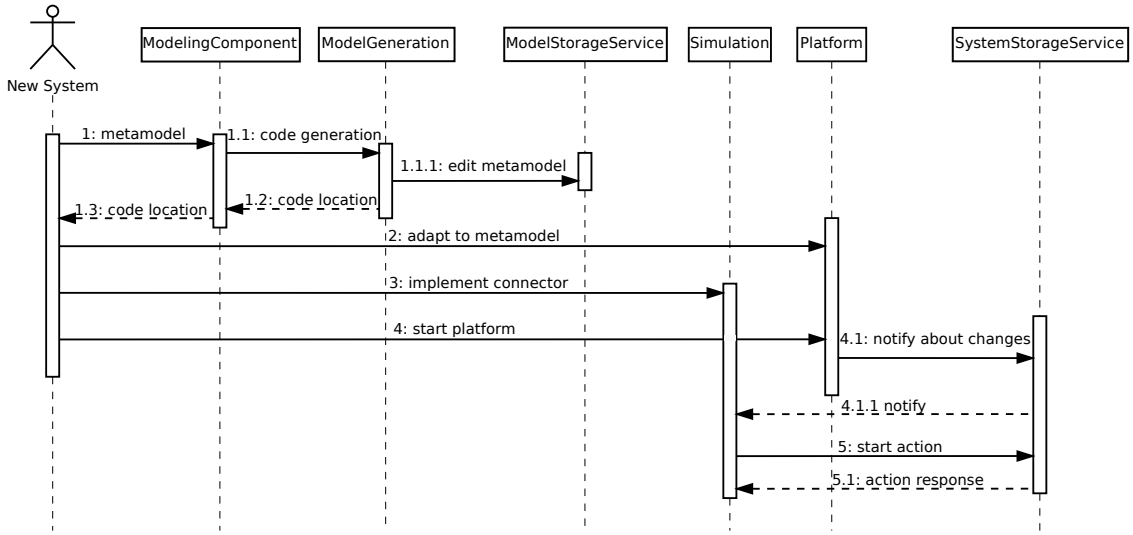

Figure 4.2: Adding a new system within the EventSim framework

Sart the simulation If everything else is running, the simulation can be started as well. Apart from the notification and monitoring part, specific actions can be performed. Some of these actions are continuous, whereas other actions can be seen as triggers. In the case of a continuous action, the action can be stoppped at any time. The actions can be started in a parallel way, which leads to the possibility of combining them for complex scenarios. Such complex scenarios can, however, also be accomplished by calling simple actions in the scope of complex actions. Already implemented actions are described in Section 4.3. Since the simulation component is not directly connected to the platform, there are no additional adaptations necessary for the stock market example.

4.2 Metamodel for fault tolerance testing

The metamodel, used for fault tolerance testing in this work, is an adaptation of the metamodel presented in [44]. Some of the constructs of it were removed, since they are not needed for simulation purposes and would lead to an overload of information. On the other side, some attributes were added to the components because of simulation purposes. The new metamodel is illustrated in 4.3.

The core element of the metamodel is the *SimpleEvent*. Every event has the attributes *source*, *destination* and *time*, which is the timestamp of creation. In the stock market example, events are used for communication between the consumer and traffic servers, traffic servers and computation servers and every other communication between servers. If other attributes are needed in specific systems, they can be specified using the class *Property*. This class describes values of

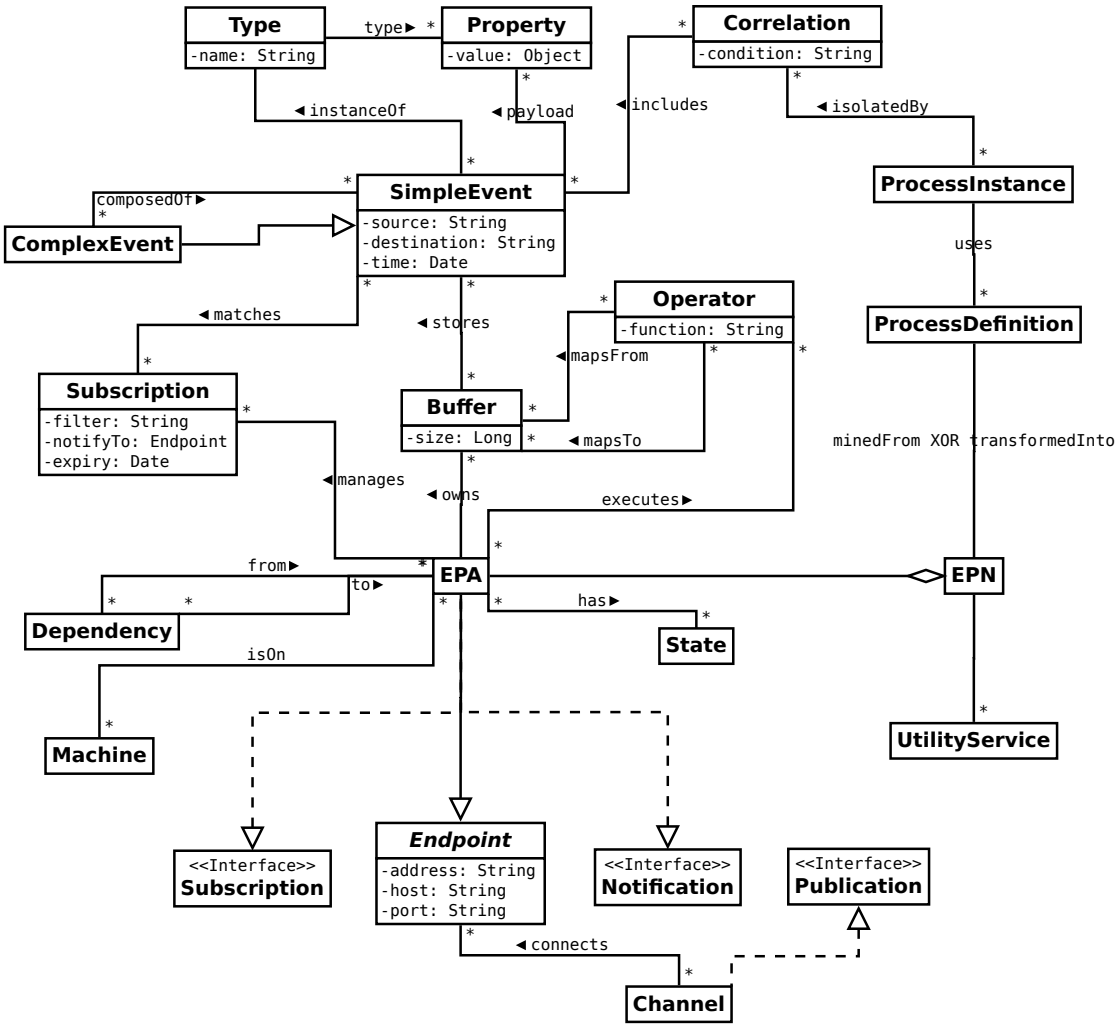

Figure 4.3: Class diagram of the metamodel

a certain *Type*. Events can also have a type, but this is not mandatory. This mechanism would be used in cases, where events may get filtered in the application code because of their nature. In the stock market example, the type could, for example, be used to differentiate between events coming from a consumer or sent to a consumer and events between servers. Events can also be part of another event. Such a composed event is called *ComplexEvent*. Events can correlate with each other, which is described by the condition of the class *Correlation*. Correlation is used to join different events with similar properties and filter them out from the other events. This can be used for higher level processing of events and is not used in the current state of the simluation.

On the other side of the diagram, an event processing network (*EPN*) is described. An EPN can be seen as encapsulated network, consisting of an unspecified number of worker nodes. In the

case of the stock market example, the virtual trading platform can be seen as EPN as a whole, or the two layers can be defined as EPNs. However, since there is also communication between traffic servers and computation servers, one EPN for the whole virtual trading platform would be the right choice for this example. This EPN can logically be transformed into a *ProcessDefinition* or mined from it. This process definiton is used to create new *ProcessInstances*. An EPN uses several services, e.g. for storage and access management, which are subsumed unter the term *UtilityService*. Every EPN can consist of several event processing agents (*EPA*). An EPA is the logical unit of computation or worker node, which can be deployed on several *Machines*. The two EPAs in the stock market example are the traffic server and the computation server. EPAs can be dependent from one another, regarding the flow of communication or the logic of computation. Additionally, an EPA has an internal *State*, displaying the current status of the EPA. Each EPA in its basic variant has two *Buffers*: an input and an output buffer. The metamodel, however, is modeled in a way that supports a flexible number of buffers if applications need them. The input buffer stores an incoming event, uses the function of the corresponding *Operator* to perform computations on this event and, afterwards, submit it to the output buffer. At this stage, an additional function may be executed, before the event is transmitted to its destination. In simple scenarios it may be accurate to only implement one of these functions and leave the other functions empty to increase understandibility of the model. In a minimal scenario, only the two buffers and one functions as a transition point between these buffers is needed. The function for traffic servers in the stock market example would be a data format transformation, which transforms the sent data from the consumer to the right format for the computation servers and vice versa.

EPAs can be either producers, consumers or both, as presented in [44]. This is, however, not illustrated in the class diagram, since multiple inheritance is not supported in Java (which is used as programing language in this work). The common super class for both types is the *Endpoint*, which has the attributes address, host and port. This differentation is made for cases, where the address is not only built out of host and port. It has to be noted, that these three attributes (and especially the address) are used for distinction between different endpoints in the Tool and Simulation component. To support the behaviour of producers and consumers, the interfaces *ISubscription* (for producers) and *INotification* (for consumers) are used. In case of a producer, the EPA manages *Subscriptions* for the consumers. A subscription is characterized by a filter for events, the notification target (a specific consumer) and an expiration date for the subscription. Each consumer in the stock market example has subscriptions to every relevant stock information. The main class for communication is the *Channel*, which connects endpoints. A channel can connect two or more endpoints, depending on the system under test. There can also be more channels leading from one endpoint to another. Such a channel can be anything from the logical representation of a physical unit to a given service endpoint or just a collection of objects. The channel between servers of the same kind in the stock market example is the logical abstraction of a simple network connections, whereas for communication between servers of the two layers, a firewall is used in addition, which leads to another logical abstraction of the channel. The channel for communication from and to consumers is a web service.

To sum it up, this metamodel is capable of describing every kind of event-based system illustrated in Section 2.5. It also supports platforms, with several of these systems at once. For example, the stock market example uses publish-subscribe for consumer notifications and P2P communication between the individual servers. Both of these systems can be described with the metamodel presented here. However, using the Modeling component in combination with the Model Generation component makes it convenient to adapt it specifically to a given system if a new kind of event-based system appears in the future. It has to be noted that the testing platform used in this work does not use every single aspect of this metamodel but these aspects are kept in the metamodel to enable further work on this framework.

4.3 Simulation actions

Simulation actions are used in this work for the purpose of simulating exceptional situations and injecting faults into the target platform (SUT). Therefore, they introduce different flaws into the SUT, which are investigated regarding the behaviour of the SUT following the injection. Simulation actions are of a dynamic nature. They can either be synchronous or asynchronous, have a short or long duration and can have a multiple set of options. These options can be textual, numeric or boolean values. Additionally, every action can have selected worker nodes, for which the relevant action is triggered. The actions are connected to the rest of the simulation component, which means, that they always have access to the current status of the system as retrieved by the notification and heartbeat mechanism. There are five actions currently implemented, which are described in the following. These actions can be used for every different event-based system and are of generic nature. However, as noted before in this work, there is also the possibility to create new actions only for a specific platform.

Additional Traffic

Targeting one worker of the investigated platform with additional data is used to identify potential scaling problems. The additional data will probably lead to scaling problems for every platform, but the amount of additional data for which these problems occur may be different for each of these systems. So one way to see this action is as a benchmark test comparing different platforms. Another problem for systems can be the treatment of this additional data. Some systems may only run into performance issues, whereas other systems may compute wrong results because of the unexpected data. Therefore, this action can be used for multiple purposes. On one side, every additional traffic, which is not parseable by the system, can harm the functionality of it if the system cannot handle this data. The much worse case happens, if the additional data for some reason is parsebale by the platform and may lead to wrong computations. The other option is to emit such a big amount of data, so that the system has performance problems handling it. To accomplish this test, a worker node is selected, which is the data recipient, and the amount of data, sent to this worker, is specified. This routine is illustrated in Figure 4.4. In future versions this may also be applied to several workers of worker groups at once. In the stock market example, this action is useful for both layers. Regarding the connection layer, it is critical to see, how the performance of the individual traffic servers is affected by this additional traffic. The

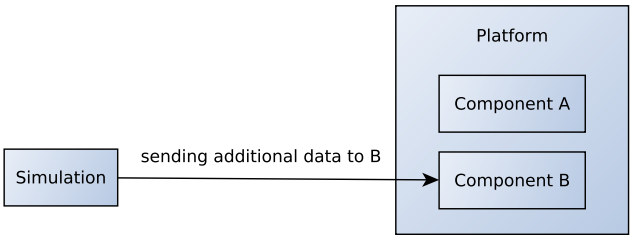

Figure 4.4: ”Additional Traffic” action

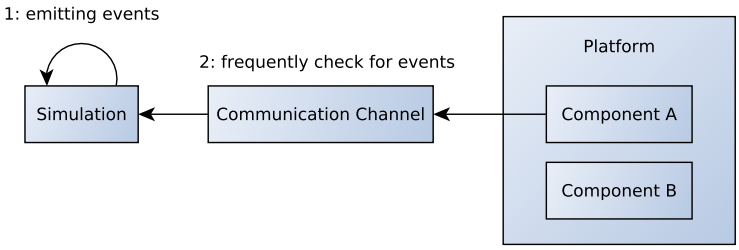

Figure 4.5: ”Event emitting” action

computation servers of the core layer, however, are highly critical regarding the correctness of their computation results. This illustrates, how this action can be used for multiple purposes at once.

Event Emitting

In its current version, the test platform, presented in Chapter 6, is receiving events from the network. This action is responsible for sending events to the platform for testing purposes. Since it is technically done over a network socket, the respective port can be specified. Additionally, the rate of events can be set. The respective communication flow is illustrated in Figure 4.5. This illustration shows that the components receive the events by polling them from the simulation service. This is also used for performance testing, such as the “Additional traffic“ action presented before. The data sent by this action has a predefined format which may be overriden for new platforms as they occur. In real-world scenarios, this action can also be used to see how the platform reacts if there is a certain peak of received events and how this affects performance of the whole platform. Regarding the stock market example, this action is used in its connection layer, since most of the events are emitted by the consumers, which can be simulated using this action.

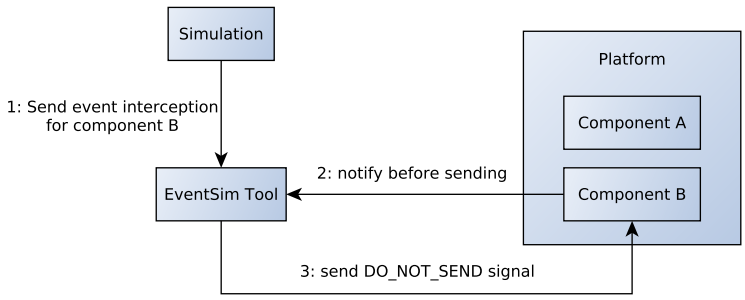

Figure 4.6: ”Event interception” action

Event Interception

The EventSim Tool component provides the possibility to prevent events from getting sent from one node to another. This is accomplished at the point, the event is added in the Tool component by the sending node with the relevant service call. The Tool component answers the call with a predefined value to tell the node not to send the event (see Figure 4.6 for an illustration of this workflow). The corresponding simulation action takes two parameters: the worker node, for which the sending of events is intercepted, and the amount of events from this worker node, which are intercepted. This action, however, also needs little code modifications in the tested platform, because it should react in the right way if the Tool component tells the sending worker to cancel the event sending. In the stock market example, this action can be used in several ways. On one side, the fault-tolerance of the SUT is monitored by evaluating if the SUT recognizes the failed event and tries to send it again. Another use case is the blocking of events from one node at all. This could lead to wrong transactions at the market because of missing asks.

Kill Event Source

Killing the event source can lead to possible deadlocks of other components in a system. Furthermore, it is important to evaluate, how the system reacts in terms of resource handling. Since there will be no computation in the system under the assumption that the event source is taken out for a long time, there should be no high consumption of resources on the indiviual worker nodes. On the other side, the system itself should react in a fast way and restart the event source in a way that the computation time is not increased in real-time systems. One may also distinguish between the killing of one instance of an event source or every event source of the platform. In the first case, a reliable platform should work with no harm as every component is redundant. Concerning the latter case, it is interesting to see, how long it takes for the platform to start the event sources and get back to a normal working state. Event sources are detected by the Simulation component (if possible, otherwise they can be entered via the GUI) and, afterwards, one instance of the event source is killed directly, as illustrated in Figure 4.7. The event sources in the stock market example are the consumers on one hand and the computation servers on the other hand, in the case of an external event on the market. Since consumers can only be killed in

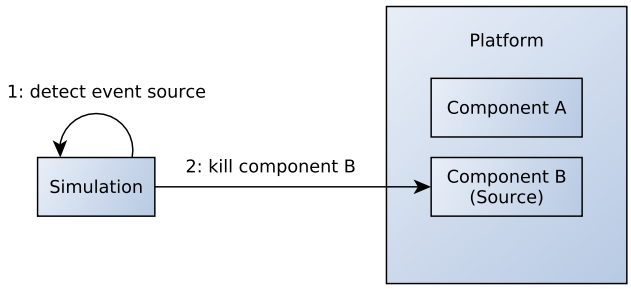

Figure 4.7: ”Kill Event Source” action

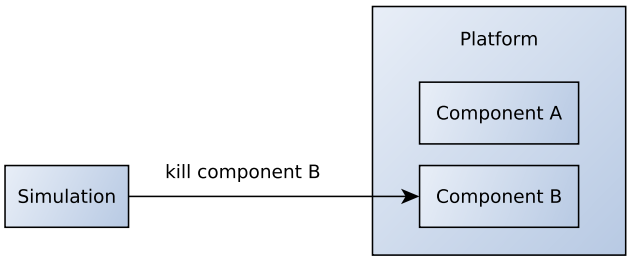

Figure 4.8: ”Kill Worker” action

a testing scenario with stubbed consumers, the most likely scenario is to kill computation servers in a scenario, where there are many external events in the market.

Kill Worker

In contrast to the action ”Kill Event Source”, this action is used to kill any worker node of an event-based system. The difference is that the simulation can target the action at the bottleneck of the system to see, how the performance reacts. These bottlenecks can be identified using the validation mechanism, explained in Section 4.4. Similar to the action before, the reactivation time of the worker node can also be measured. Again, it will be interesting for any platform to see, how long it takes for the given worker group to be up and running again at a normal state. The workflow on the side of the simulation is a bit simplified compared to the ”Kill Event Source” action, which can be seen in Figure 4.8. As for the stock market example, this action can either be used to kill traffic servers or computation servers. The goal of the simulation depends on the kind of server, which was killed. Killing traffic servers can lead to performance issues rather quickly, whereas killing computation servers can lead to performance issues in the case of high load.

4.4 Validation of Platform Behaviour

The purpose of validation is to assert that the target platform reacts properly in response to the applied simulation actions and ensures that constraints of the application logic are still met. The EventSim framework allows to define custom assertions over the system state, which are continuously evaluated throughout the simulation. If any of the assertions is violated during execution of the simulation, a corresponding error message is reported to the user. Validation in the EventSim framework is performed by annotating relevant classes and using a Scripting Language to describe constraints. As part of this work, Groovy [52] is used to describe these constraints. Using this language, it is possible to describe constraints for every workflow, which can be described by the attributes of the relevant class. Additionally, constraints can also be described regarding the current running instances of a class. Therefore, the variable *_componentInstances* is injected into the script, which describes the number of currently running instances of the evaluated class. Fields of the evaluated component are also injected into the script for usage in the query. This is especially useful when it comes to redundancy checks of the components. The validation mechanism allows for constraint checks regarding the structure of the platform, as well as checks regarding the application logic of the platform.

Validation is triggered automatically by the Simulation component as soon as a connection to a platform is established. Every class used in a platform can have certain *constraints* attached to it. An example of such a query, which highlights the different features of the query language, is illustrated in Listing 4.1.

```
_componentInstances >= 1 && _componentInstances <= 3 &&
parameterCache.size() >= 0 && parameterCache.size() <= 1 &&
receivedEvents >= sentEvents
```

Listing 4.1: Example of a query

As described via the constraint examples for the stock market example, constraints could be defined for structural aspects on one side. These constraints deal with redundancy and reliability of the servers. On the other side, constraints dealing with application logic can be defined. These constraints are most useful for the computation server because they encapsulate most of the business logic of this platform.

4.5 Heartbeat mechanism

Every worker node is responsible for sending an alive-message to the EventSim Tool component on a constant basis (the exact frequency for these messages can be configured) [21]. These messages only contain the name and adressing information for this specific component and are used to check the status of the saved components. If a worker node did not send such a message within a certain amount of time (the tolerance for the sending of heartbeat-messages is also configurable), it is removed from the persisted state in the EventSim Tool component. In such a case, the subscribed components of the Tool component (mostly the Simulation component) will be informed about the removal of the element. Additionally, this mechanism is used for simula-

tion because otherwise it would not be possible to provide the EventSim Simulation component with the current state of active components.

The choice of frequency for this messages can be chosen by the platform developers, where both extreme variants have an advantage and a disadvantage. A high frequency leads to a more reliable information about the current state of the application but also increases the amount of network traffic by a huge amount if there are many worker nodes active. Using a lower frequency on the other hand reduces the network traffic but also leads to a more unreliable information about the current state of the nodes, since inactive nodes are recognized rather slowly. The best choice is most likely a mix between these extreme variants, where the real values depend on the likeliness of change in the investigated platform. The adapted components in the stock market example are the traffic servers and computation servers. These two concepts have to be extended with the functionality of sending heartbeat-messages. In this example, sending heartbeat messages over a central component has two disadvantages. First of all, there is no component, which knows every server of the platform. This means that it would require a lot of development time just for the sending of heartbeat messages. The second disadvantage is that such a component would affect the dynamic nature of the platform in a way, that the inner concept of a cloud-like structure would be dismissed by this central component.

4.6 Notification mechanism

As mentioned in Section 4.1, the ModelStorage component keeps track of the current state of an event based system and the actions modifying it. This implies the actions of the system, as well as the current state of it. As a result, every interaction between two system nodes has to be communicated to the ModelStorage component. In addition, the creation and removal of a system component has to be communicated. Afterwards, the information is forwarded to the Simulation component, which handles it, depending on the current configuration of it. This communication workflow is illustrated in Figure 4.9.

During the simulation phase, the Simulation component sends events to the event emitter of the SUT. As a result, a new event is sent to another node of this system. At the time, the other node receives this event, it has to notify the ModelStorage service about the retrieval of this event. As a last step of this workflow, the Simulation component is also notified by the Model-Storage service. This behaviour allows monitoring of the events by the Simulation component. This is the most common workflow for monitoring events in the system.

For this workflow to work in a correct way, the SUT and the simulation component have to fulfill the following requirements:

Connecting The SUT has to be connected with the used metamodel for testing. In this work, every component used for event computation and sending needs to be specified as EPA for the simulation to work in a correct way. In the stock market example, traffic servers and computation servers are affected by this adaptation. The other components of the system,

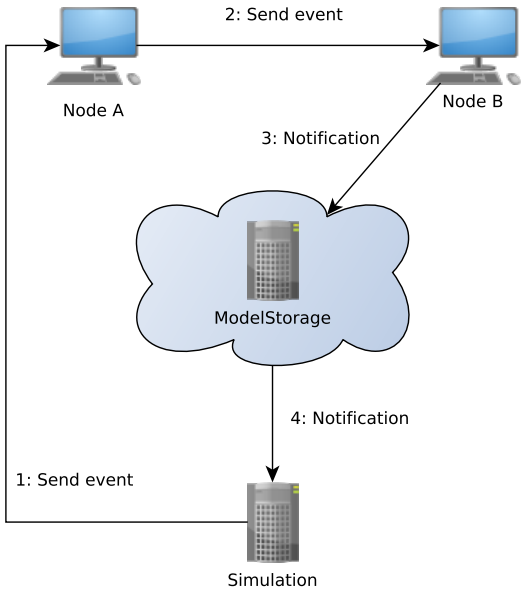

Figure 4.9: Notification mechanism

like events being sent from consumers to servers, vice versa and between servers, have to be inherited from the relevant element of the metamodel as well.

Registering The Simulation component has to register itself at the ModelStorage service. As a result, the ModelStorage service knows that it has to notify the Simulation component about changes. This registration can be done for specific actions, like creation or removal of a component, or for every action. In future versions, these notifications may also be filtered according to filters composed out of the attributes of the relevant element. There is no need for adaptation for platforms, like the stock market example.

Communicating As a final step, the system components have to tell the ModelStorage about every change, e.g. the event receival or the removal of an event. Furthermore, every creation of a new system component and the manipulation/removal of such a component has to be communicated. By definition of the ModelStorage, this notification is then forwarded to the subscribers of this notification. This affects both, the traffic and the computation servers in the stock market example, since these two components are responsible for event creation.

For complete coverage, adding and removing of a node must also be communicated to the Tool component. This is not always possible, because it happens on a layer not controlled by the platform code. Nodes are typically managed on the event-based system hosting the platform, so the platform itself probably does not recognize adding and removing of a node. To deal with this issue, the Simulation component also recognizes new system components if they arise as part of

another notification. Removing of a node cannot be handled with the notification mechanism, but is dealt with using the heartbeat mechanism (Section 4.5).

4.7 Adding a new platform

The steps, applied when adapting an event based system for use with this framework, have partially been described in the last sections of this chapter. This section gives a clear sequential list of tasks, a platform owner has to accomplish before the respective system can be used together with the framework, developed in this work. As with the sections before, the list is illustrated with the stock market example, described in Section 1.3, to support the understandibility of these steps.

1. **Generating a metamodel for the system**: Either the metamodel, provided by this work, is used or a new metamodel is created, which can describe the new system. For small adaptations, the best way is to adapt the metamodel of this work. In the case of the stock market example, the metamodel developed in this work, is sufficient for usage. If the platform developer knows that the simulation needs do not change over the near future, the metamodel could also be simplified, leaving only the relevant classes for simulation in it.

2. **Code generation**: The code for the new metamodel has to be generated and connected to the new system. Therefore, the code base is deployed with the platform code, which encapsulates the code base as part of the platform code. Regarding the stock market example, the code base is put on every machine, a traffic server or computation server is running on.

3. **Connecting the classes**: Connect the classes from the new system with the classes from the generated code by inheriting from them. Connecting in this case means, that the platform code needs to be inherited from the relevant classes of the code base. Traffic servers and computation servers of the stock market example are directly inherited from the class *EPA* to define them as worker nodes. This also means that the attributes for EPAs like *host*, *port* and *address* have to be set in a right way to ensure that the simulation workflow works as expected.

4. **Add notification calls and heartbeat**: Every method in the components of the system, which can manipulate events, has to make a notification call to the ModelStorage. Additionally, every such component is responsible for sending heartbeats via the supplied interface to the EventSim Tool component. This can either be done by each component individually or from a central point in the platform, depending on the structure of the platform.

5. **Annotate the event producers and consumers**: The components used for testing need to be annotated according to their desired behaviour. This ensures correct functionality of the validation. Before annotation, the constraints for every of these components has to be defined. These constraints must be written in a way that makes them transformable to

the query language prestend in Section 4.4. All of these steps are necessary for both, the traffic servers and the computation servers in the stock market example, which should be checked against constraints.

6. **Start the system**: If every previous step is done, the new system can be started and, additionally, the simulation component is started. This automatically triggers registration for notifications at the Tool component. Afterwards, simulation actions can be performed on this platform and the lifecycle of components in it is monitored. For the stock market example, this means that the platform is started as normal, with the differene that the adapted version now automatically sends heartbeat messages and notifications to the Tool component. In a next step, actions like killing certain traffic servers to monitor the performance following this change can be performed.

CHAPTER 5

Implementation

This section is used to describe the implementation details of the EventSim framework. Therefore, every component of the framework is described, as well as the communication between them.

5.1 The EventSim Framework

Architecture

The component architecture of the EventSim framework is illustrated in Figure 5.1. The framework itself consists out of three components, each of them using a Jetty server to deploy the services. Since the EventSim Tool component supports the persisting of models (via the ModelStorage service) and systems (via the SystemStorageService), this component also has two in-memory storage mechanism attached to it. Both of these storage systems are currently implemented as in-memory data structures for performance reasons. In future versions, they can be extended with persistent database support, in order to cope with high amounts of model data which cannot be handled entirely in-memory. The communication between the components is implemented using service calls, as described in Chapter 4. As far as upscaling is concerned, the EventSim Simulation and the EventSim ModelGeneration component can be replicated without requiring additional measures. If the EventSim Tool component is replicated in the future, the data storage part of it will be moved to a separate component to simplify the process of replication. Additionally, synchronisation can be a difficult challenge regarding the EventSim Tool component, since it is the the only stateful component of the framework.

The whole EventSim framework produces five artifacts, used in the components described above, except the eventsim-platform artifact, which is the test platform used for evaluating the framework. These five artifacts and their dependencies with each other are illustrated in Figure 5.2. The common concepts of the framework, like service interfaces and core classes, are defined in the eventsim-common artifact. However, since the service interfaces use the abstract element

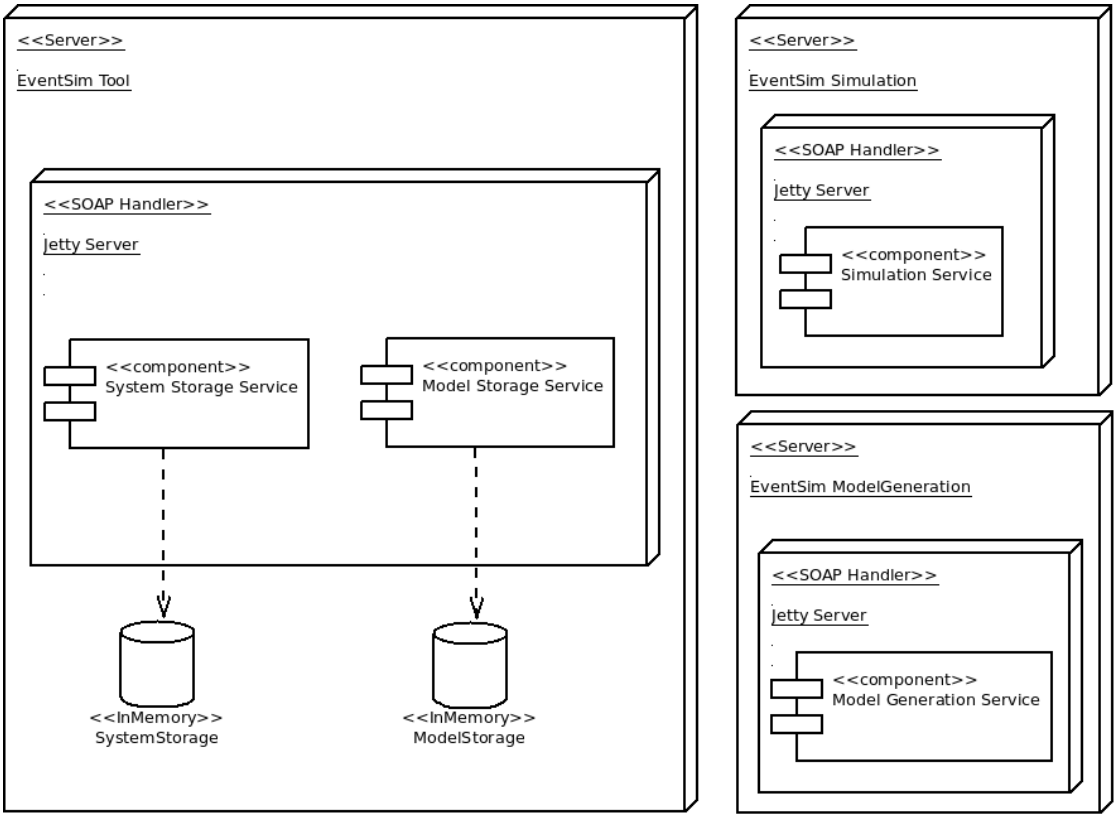

Figure 5.1: Deployment diagram of the EventSim framework

ModelElement, generated in the eventsim-model artifact, it has a dependency to this artifact. The eventsim-tool artifact requires the same class from the eventsim-model artifact and additionally the pojos and service interfaces of the eventsim-common artifact. Both, the eventsim-platform and the eventsim-simulation artifact, need the service interfaces as well. The eventsim-simulation additionally needs the platform classes used in the relevant platform. In a real-world scenario, these classes need to be injected into the simulation project.

5.2 Model Generation

The metamodel for the supported event-based systems of the EventSim framework are generated using the concept of MDE, explained in Section 2.2. In particular, the Eclipse Modeling Framework (EMF) [88] is used. The metamodel can consist of every element defined in the meta-metamodel illustrated in Figure 5.3. This is an adapted version of the Unified Modeling Standard (UML), which features the necessary aspects to model any metamodel for an event-based system. The main element of the model is a *System*. Every *System* can consist out of multiple of these constructs:

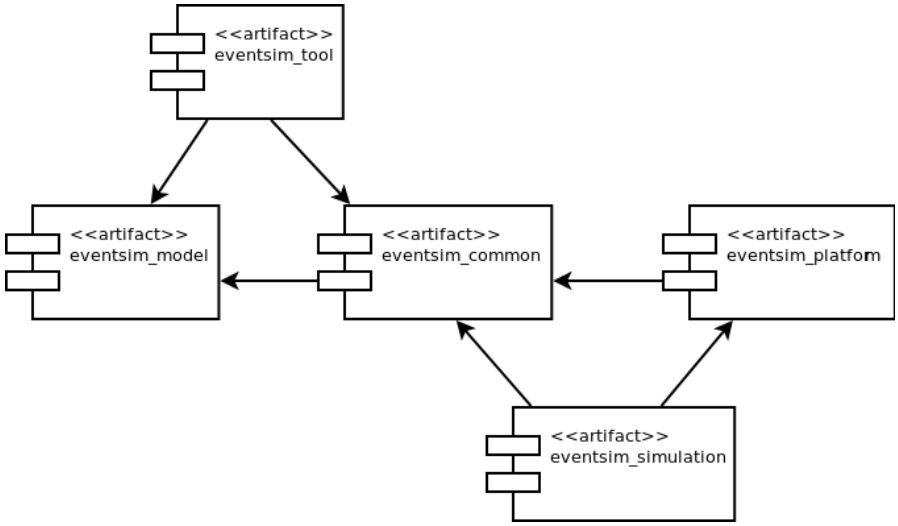

Figure 5.2: Artifacts developed in the EventSim framework

Implementation This describes the implementation of an *Interface* with a *Class*. Interfaces and classes are defined via their name, classes can, additionally, be marked as abstract. An implementation is created for every interface and encapsulates every class implementing this interface in it.

Association An association is the relationship between two *Roles*. A role is an enriched version of a class, in particular it has a class attached to it, like some other attributes, specifying the behaviour of the role. In addition to a name-property, this element also specifies if the role is part of a containment or navigable. The latter is used at the process of code generation, the first one is currently only used as design aspect. Every role is connected to a certain class. Additionally the role has a defined *Multiplicity* with an lower- and upperbound property. The association can also have an additional class property, which represents an association class.

Generalization A *Generalization* is used to describe an inheritance structure. As with the implementations described above, a generalization is described once for every superclass. So one generalization contains every subclass of this superclass.

Every *Class* consists out of multiple *Attributes*, which are described by its name and its *DataType*. The datatypes are only a set of types, which are currently used in the system, and can be adapted for further usage. The value of *ENDPOINTCLASS* is mapped to the datatype *<? extends Endpoint>*, as this is a workaround datatype for the currently used metamodel. *Enumerations* can also be described with this model, having a name and containing multiple *Literals* with a certain value. Classes, Interfaces and Enumerations are subelements of the element *Type*, which is used to describe the elements of a system.

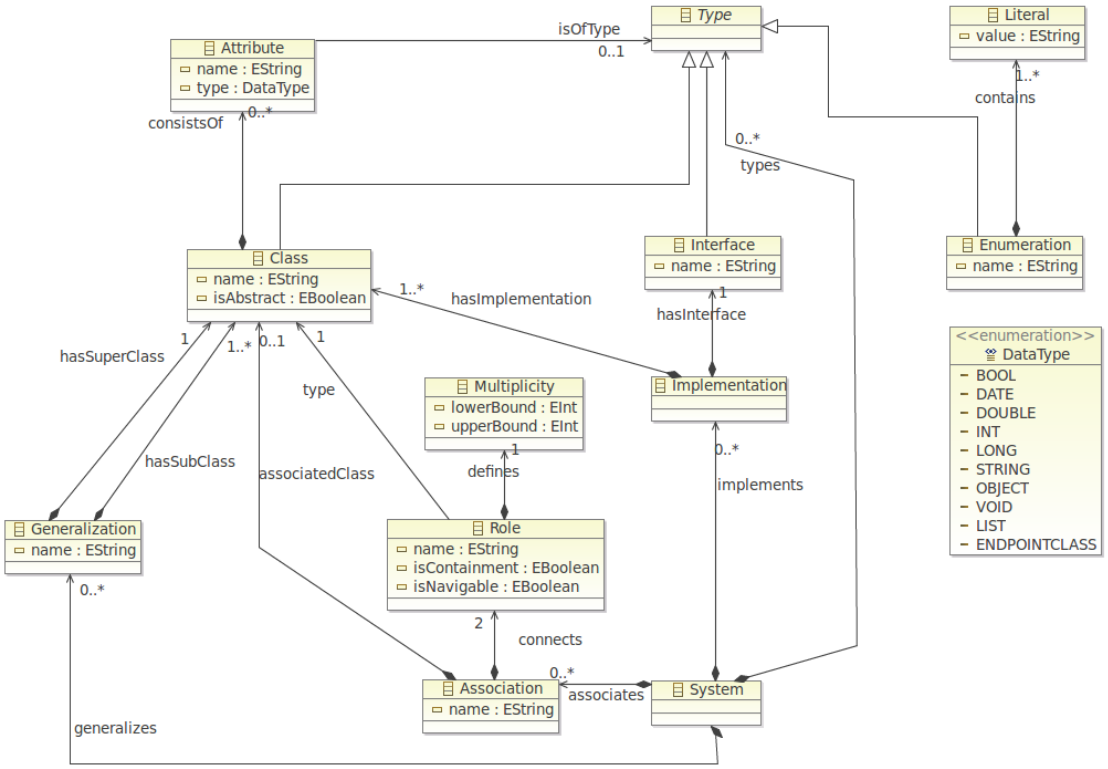

Figure 5.3: Meta-meta model used to describe Metamodels

The model generation is divided into the following phases (the first four phases refer to the same working routine but are handled as different phases, since they are adressed in a different way in the metamodel):

Generating stand-alone classes In this step, every class is generated, which is not part of inheritance or implementing an interface. Creation of a class is a straight-forward process. Every attribute described in the metamodel is declared a field of this class, as well as associations to other classes, if the relevant role is navigable. This procedure is also applied to the next three steps.

Generating sub-classes Every class, which has a superclass other than *ModelElement*, meaning that it was defined as part of an inheritance, is generated.

Generating classes implementing interfaces Classes, which are part of an interface in the metamodel, are created. However, only the classes of an *Implementation*-construct are created in this step, not the interfaces.

Generating super-classes The parent classes of the classes, handled within in the second phase, are generated. The distinction between these two phases is just made out of logical reasons, since treatment of this classes could perhaps be divided in future versions.

Generating interfaces The defined interfaces are generated

Generating enumerations Enumerations defined in the metamodel are generated here.

Creating the base class *ModelElement* This class is used as a parent class for all classes defined in the specified metamodel. It has only one attribute, wich is the ID of the element used for storing it to the database. The important part of this class is the description of the used classes to support marshalling in the web services. Therefore, this part of the class is of dynamic nature. However, this part is generated automatically, based on the elements in the metamodel. A stub of this class is illustrated in Listing 5.1.

Although the sequential order of these steps is not particularly consistent (e.g. interfaces are created after the classes referring to them), this does not mean that the process itself is inconsistent, since the compilation-process of this classes is invoked as soon as all classes, interfaces and enumerations are generated.

```
@XmlSeeAlso({Type.class, Property.class, [...]})
@XmlJavaTypeAdapter(ModelElement.Adapter.class)
@XmlType(name = "modelElement")
public abstract class ModelElement implements Serializable,
Cloneable {

 [...]

 @XmlAttribute
 //getter and setter for id

 public static class Adapter
           extends XmlAdapter<Object, Object> {
  public static Util util = new Util();
     private static Map<String, Class<?>> MAPPING =
     new HashMap<String, Class<?>>();

     static {
       MAPPING.put("type", Type.class);
       MAPPING.put("property", Property.class);
       //additional mappings
     }

     public Object unmarshal(Object v) {
       try {
         if (v instanceof Element) {
              Element e = (Element) v;
              Class<?> clazz = MAPPING.get(e.getLocalName());
              if (clazz != null) {
```

```
                return util.xml.toJaxbObject(clazz, e);
              }
            }
        } catch (Exception e) {
          System.err.println("Unable to unmarshal");
        }
        return v;
      }

      public Object marshal(Object v) {
    return v;
      }
  }

  //clone method
}
```

Listing 5.1: Stub of the ModelElement class

As a result of the model generation part, a JAR-file containing all compiled class-files is returned. In the current version of the framework, this file is transferred to a given server, where it can be downloaded using a simple HTTP-GET request. New platforms just need to declare this jar-package as dependency to be able to connect the platform-specific classes with the elements of the metamodel.

5.3 Model & System Storage

The storage of runtime models is provided via a service named *SystemStorageService*, whereas the *ModelStorageService* of the implementation deals with the metamodels used to generate code.
Storing the objects for each runtime model generates a lot of data, if all the objects are saved with their respective object tree, meaning all the children and siblings element of it. This is the key aspect of the storage feature in this implementation, which has the benefit of storing only the minimum amount of data necessary to provide the same information as with storing all the data. The concrete solution uses two lists at runtime. One is used as a helper list and keeps track of every object with its class and ID saved in the system. This object only has its own data stored in it, as well as objects already in the system at runtime. Referenced objects of a such an object (either childs, parents or siblings) have only the respective ID stored in it. This means that there is an object for every referenced object, but this object is only stored with its ID to link it to the concrete object in the list. Therefore, the parameters of the service methods only use objects with references to other objects and not object trees as a whole. The concrete objects, which are referenced by this object, are loaded from this list. The other list contains the current innerconnected element trees of the runtime model. On a business point of view, this is a snapshot of the SUT. To clarify the process, a sample setup of an event is illustrated in Listing 5.2 and in Figure

5.4, which shows the state of the lists after each of the timestamps specified in Listing 5.2. To sum it up, the complete information of the elements is only saved in the snapshot list, as well as in the latest elements of the helper list. This means that some data is saved redundant, which is on purpose since it provides a significant better performance than in a scenario with every set of data only saved once. Recursive functions are used in a frequent way in this system, but this would be far more extreme if there is no redundant data in the system at all.

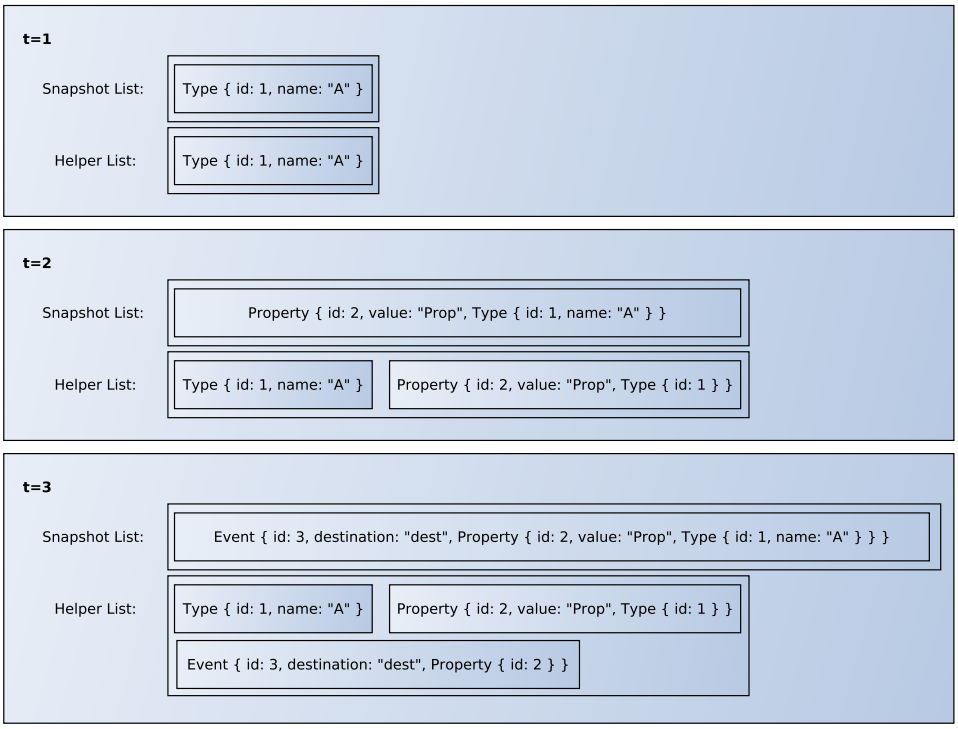

Figure 5.4: Storing an event using the EventSim Tool component

```
Type aType = new Type();
aType.setName("A");

Integer typeId = addElement(aType); //service call (t=1)

Property prop = new Property();
prop.setType(new Type(typeId));
prop.setValue("prop");

Integer propId = addElement(prop); //service call (t=2)

Event evt = new SimpleEvent();
evt.setDestination("dest");
evt.addProperty(propId);

addElement(evt); //service call (t=3)
```

Listing 5.2: Creating an event

A drawback of this approach is the additional code needed on the platform-side as can be seen in this example. This drawback, however, does not compare to the benefit provided by the resource-usage of this approach. Additionally, this example shows that the solution is scaling on a constant basis. The snapshot list is increasing by just the element added to the system and the list of old elements is increased with the object of the passed element (meaning the object including its primitive attributes, but without references to other objects). As a conclusion, every additional element passed to the service leads to two additional elements in the service.

Other operations, like the removal or editing of an existing system component, also use these two lists. Contrary to the adding of an element, elements with child elements can also be directly removed, since the element is decoupled from the snapshot list. Regarding the list of old elements, the removed element and all of its child are removed from that list if it has no connections to existing elements. This mechanism supports a full flexibility of the framework and also provides a basis for further adaptation.

The heartbeat mechanism, described in Section 4.5, saves the individual timestamps in the elements from the old list. This is done out of performance reasons, since traversing the actual snapshot list could take a while in a complex scenario, whereas getting the element from the list of old elements is done with the same performance, irrelevant of the size of the saved data.

5.4 Simulation

The simulation component of the EventSim framework is the part of the framework, where the other parts are combined and the benefit for platform developers is directly visible. On one side it is used to monitor the SUT, on the other side it can be used to execute certain simulation actions, e.g. introduce faulty data into the SUT, and evaluate how the system reacts to it. The component itself consists out of of the monitoring engine, which uses the notification mechanism, see Section 4.6, of the EventSim tool component combined with the heartbeat mechanism, see Section 4.5 (which is not directly visible to the Simulation component), the query engine for constraint checks and a graphical user interface.

Workflow

The main concept behind the simulation component is loose-coupling, which results in the benefit of simplified adaptation in future versions as well as an easier approach to divide development time, since there are no strict dependencies between the different constructs and so they can be developed independent from each other. This can be seen on different aspects:

Actions Actions are implemented as classes and are loaded at startup of the Simulation component. The only constraint is that they are located in a specific package. There exists one package for general actions (applicable to every event-based system) and one package per event-based system for specific actions. Currently, there are only general actions available, since specific actions are not needed for testing purposes. The implementation of

these actions is the same, irrelevant if they are general or specific. The actions themselves are completely independent of one another. However, it is also possible to call simple actions in the context of complex actions. This has the drawback, that dependencies between actions are put into the system, which are not supposed to exist. A better, but also more complex, approach would be to provide interfaces (either as service or locally) in the simple action, so that the complex action does not call it directly. The system of action execution in the Simulation component provides the possibility for both approaches and leaves it up to the platform developer, how actions interact if this behaviour is needed.

User interface The details of the user interface approach are described in Subsection 5.4. Basically the user interface is decoupled from the business logic used for accessing attributes of the SUT. Furthermore, different user interfaces for different devices and SUTs can be defined by adapting a class or creating a new one after a predefined scheme. There exists one general user interface, which can be applied to every known event-based system and extended to the individual needs of the platform developers if necessary. The actions are also loosely-coupled to the user interface, since there is no knowledge of any action at the initialization of the user interface. The actions are retrieved dynamically at the startup of the user interface. This enables the adaptation of the user interface without any knowledge of yet implemented actions, which can be helpful if the development of the SUT is distributed or one developer should not have complete knowledge over the system.

Notification & heartbeat messages This is done using a web service. Therefore, basically every component is able to notify the Simulation component. Using it, makes it possible to use the simulation from outside the EventSim framework as well. Additionally, there is no fixed dependency to existing SUTs, which makes it easy to add a new one because no additional code is necessary regarding notification of the simulation component. The Simulation component itself requires either a working notification or heartbeat mechanism to keep track of changes in the SUT.

Regarding the workflow itself, the notification mechanism is started as soon, as the connection to an event-based system is established. After that, the notification mechanism is running until the connection to this system is closed. The same concept is true for validation of the system. During establishing the connection to a specific system, the current status of this system (regarding its running components) is loaded from the Tool component. This is accomplished with the heartbeat messages, which give the Tool component the necessary information about the current status of the platform. Every action is executed in an own thread, which enables the parallel execution of them. This way, one could combine two or more actions to achieve one goal. Furthermore, it provides complete independence of the execution time of different actions. However, the execution of many simulation goals can lead to performance issues, depending on the individual architecture of the machine, the EventSim simulation component is running. As a result, the Simulation component could be replicated over the network, as suggested in Section 4.1.

Validation

As described in the section 4.4, the check functions are described in an interface. Additionally, the level of the check (either *VARIABLE*, *CLASS* or *MAP*) is directly annotated over the respective check method. In its current implementation, the levels *VARIABLE* and *MAP* are treated the same way, since this difference was introduced for further adaptability of the validation process. A snippet of the interface is illustrated in Listing 5.3. The function *checkQuery* is executing the query and is called from within the Simulation component. The other functions are the relevant mapping functions to the query functions presented in Section 4.4.

```
public boolean checkQuery() throws CheckException;

@LevelOfCheck(level = CheckLevels.CLASS)
public boolean minimumInstances(Object... params)
                        throws CheckException;

@LevelOfCheck(level = CheckLevels.VARIABLE)
public boolean greaterThan(Object... params)
                        throws CheckException;

@LevelOfCheck(level = CheckLevels.MAP)
public boolean minimumSize(Object... params)
                        throws CheckException;
```

Listing 5.3: Interface of the query checker

The right choice of the check level is important, since it is relevant for the resolution of the query method. An implementation of this interface is implemented as part of the EventSim Simulation component. For an adaptation of the interface, e.g. adding a new check method, it is sufficient to add a new method to the interface and the implementation. The arguments of the individual check methods are all of variable length, since the matching to the correct number and type of arguments is done in the individual method (in particular the mapping is done in a private method but the method is called from within the individual check method with the expected parameter types). This routine takes the parameters supplied to the method and tries to match it to the needed types for this method, which provides a generic approach to the problem of querying information of the underlying classes and guarantees the further adaptability of the mechanism. As already discussed in Section 4.4, every method, dealing with the information of currently active elements in the platform or with specific fields of the relevant class, is possible for checking. A combination of these two aspects is also possible, e.g. there must be a minimum amount of instances of a certain class, where the amount of instances is defined in a field of this class.

User Interface

The user interface of the Simulation component can be divided into at least two layers. Everyone of this layers, however, can be divided into more layers to achieve clearer code if necessary. The basic layers are:

Graphic Layer This layer describes the user interface itself, meaning the graphical components and how they are used. In this version, a GUI implemented using Java Swing [56] is used to interact with the user. Using this layer approach, however, it is possible to change the user interface completely and adapt it to the needs of the users, without touching the business logic functions of the user interface. Handling and displaying of simulation actions (described in Section 4.3) is also described on this layer, since it is generic for every event-based system. As the user interface itself is independent from the concrete SUT, this layer only needs to be adapted once for every different user interface and not for every SUT. For small changes in the user interface, the inheritance structure can be expanded to prevent duplicate code.

Business Logic Layer The logic of the user interface, which is specific to a certain event-based system, is described in this layer. This layer is defined for every SUT, so there is one implementation of it per SUT. Regarding similar SUTs, which only differ in minor aspects, one implementation can be done in a generic way for all of them. As part of this work, one implementation of this layer was developed, which should provide basic functionalities for every known event-based system.

The GUI, implemented in this work, is illustrated in Figure 5.5. It consists out of four areas. At the top of the view, the connection to a specific system can be established. Therefore, the *platform* and *version* of the relevant system are specified and submitted to the EventSim Tool component to retrieve the right information. This information is retrieved over a service call to the Tool component, which queries the list of active systems for the relevant information. The next sections in the center of the interface are used to execute actions on the whole system or specific workers. Therefore, actions can be chosen with the *action list*. These actions are loaded dynamically, based on the implemented general actions and the actions implemented for the currently active SUT. Next to this component, the *option area* for the chosen action is displayed, as well as a button to start and stop the relevant action. The option are configurable for every action and, therefore, the option area can look different for every action. Currently, the supported options are textual options (visualized using a textbox), boolean options (visualized using a radiobutton) and numeric options (visualized using a slider). At the right side of the application, the different worker nodes are displayed in a *component list* (displaying the different kinds of components) and a *worker list* (displaying the concrete workers for every component). The bottom of the window contains a *logging area*, which records notifications from the EventSim Tool component and messages regarding actions. In the current version, error messages are also communicated in this area.

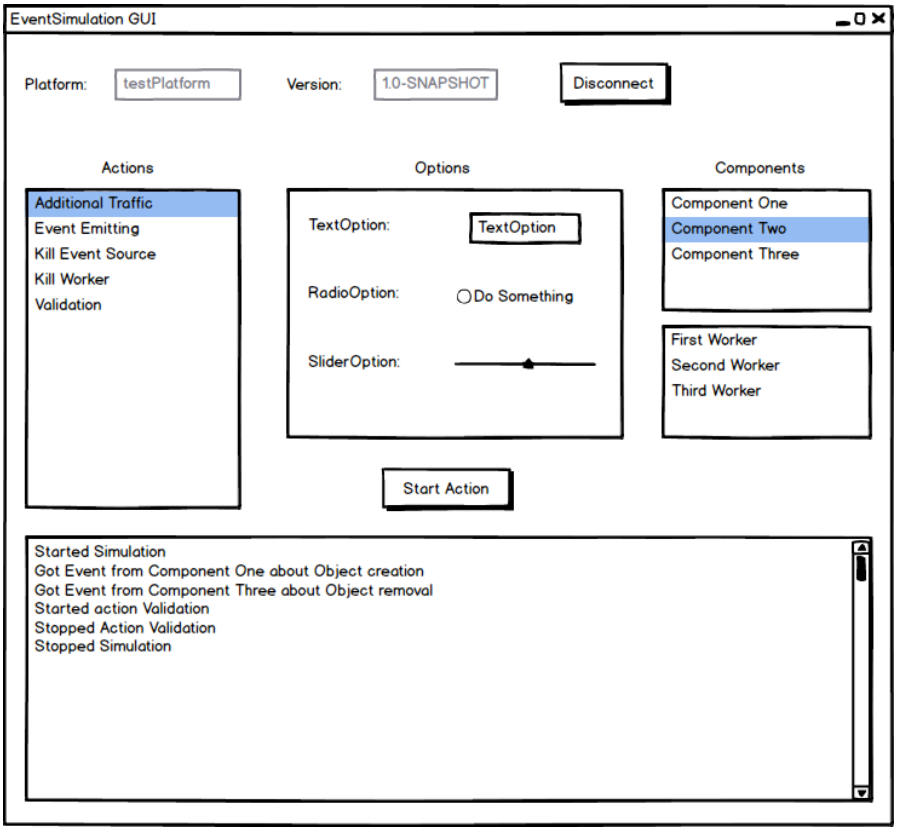

Figure 5.5: Mockup of the implemented simulation GUI

5.5 Testing platform

Since the testing platform, developed in combination with the EventSim framework, is mainly used for evaluation purposes, the application logic of it is described in 6. This section, however, deals with the implementation of the notification- and heartbeat mechanism, which is a best-practice description for connecting an platform to the EventSim framework.

As soon as a worker node of the platform is created, an instance of the class *HeartbeatSender* is created. This is a simple thread, which sends heartbeat-messages at a constant rate (depending on a configuration value). It has the following parameters, which allow the EventSim Tool component to identify the sender of the heartbeat messages: *platform*, *version*, *id*, *name*, *host* and *port*. The *HeartbeatSender* checks the state of its creating component before sending messages to ensure consistency. This way, it is ensured that the information the EventSim Tool component has about the state of the platform is nearly real-time.

The notification mechanism on the other side does not need extra implementation on the side of the platform. The only significant change is that the platform needs to inform the EventSim Tool component about every change in its elements, e.g. creation of a worker node or sending

an event. This involves small changes in every component of the platform, but these changes have to be made only because of adapting to the code base, generated with the EventSim Model Generation component.

CHAPTER 6

Evaluation

This section provides an experimental evaluation and a critical reflection of the EventSim framework and its subcomponents. Therefore, an example eventing application is introduced in the first Section, which is used in the subsequent sections. The following sections deal with the performance of the framework in normal state and the fault injection & detection used to evaluate the fault-tolerance of event-based systems. In the last section, the open issues of the current state of the framework are discussed.

6.1 Eventing example: Correlation indices based on stock market prices

The eventing scenario used for evaluation, is a distributed calculation of correlation indices based on stock market prices. The formula used for correlation is the population Pearson correlation coefficient [16]. This formula is used to describe the correlation between two variables, where a value of *+1* means perfect positive correlation and *-1* means perfect negative correlation. Basically, it is defined as the covariance of the two input variables divided by the product of their standard deviations:

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}}$$

In this scenario, a special distribution of this formula is used to calculate to correlation for two stock market prices of different currencies. The transformed system is illustrated in Figure 6.1. The text on the labeled arrows describe the information sent by the event source, which it received beforehand from another node. Each of the illustrated nodes has a different task:

Receiver This node is responsible for retrieving pairs of stock market prices (X_t,Y_t) and submit them to its successors for calculation.

XStdDev Calculates the standard deviation for X ($\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}$). Previous values for X are kept in memory to enhance performance.

YCurrency Transforms the stock market price of Y to the local currency, since it will be transmitted to the system in a foreign currency.

YStdDev Calculates the standard deviation for Y ($\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}$). Previous values for Y are kept in memory to enhance performance.

DevProd Product of the two standard deviations and, therefore, the denominator of the correlation coefficient.

Dev Calculates the standard deviations and the product of them directly to ensure correctness of the previous result. This component is only used in a test scenario and will be removed in a production environment. In this scenario, it is used to cover more paths and make the system more complex.

Compare Compares the result from component *DevProd* and *Dev* and transmits the denominator to component *Corr* if they match.

DBRec Receives previous stock market prices from the DB to calculate the covariance in the next step. The DB is kept in-memory to support enhanced measurement of the performance of the EventSim framework, see Section 6.2.

Cov Calculates the covariance (nominator of the formula), which is $\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})$.

Corr Puts nominator and denominator together to get the correlation coefficient.

Output This component is only used to output the result to necessary recipients.

As seen in the illustration, this system features nodes sending events to multiple other nodes and nodes receiving events from multiple other nodes. These aspects are used for different simulation actions and provide a good abstraction of a real-world system. The system has one entry point for external events, the node *Receiver*, which receives pairs of stock market prices (X_t,Y_t). This entry point is also used for the simulation to inject events into the system. The standard configuration used for this system is:

Machines There are three machines active for development. This configuration value can be changed each time the system is deployed to the Storm framework. One of the machines hosts the zookeeper instance, as well as the Storm nimbus. The other two machines have one supervisor each and the allocated workers. Considering characteristics, the machines have 5 GB RAM and 3 Virtual CPUs.

Deployments The event source *Receiver* is only deployed once to prevent inconsistent states of the application. The nodes *ComCov*, *Denom* and *Corr* are also only deployed once because they are only processing events if they received two correlated events of two different types. Every other node is deployed twice using the current configuration. This setting can also be changed at the time of deployment to the Storm framework. The number of workers is set to 4. An exemplary deployment of the different components is illustrated in Figure 6.2. It has to be noted, however, that this deployment can change

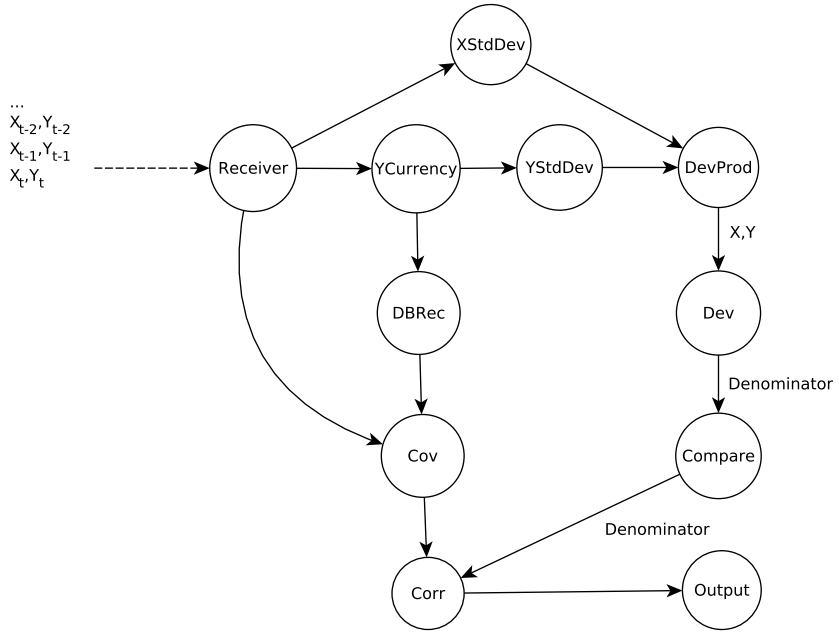

Figure 6.1: Event-based system used for evaluation

every time the topology is started, since Storm assigns the worker in a random way. This also illustrated that targeting one component of the system also affects other components running on this worker node. These side effects are part of the evaluation scenario and do not affect real world systems, where every component is isolated in one worker node.

Timing constraints Heartbeat messages are sent every 30 seconds by the worker nodes, whereas activity-checks for these nodes are performed every 60 seconds by the Tool component. This ensures that the network is not overloaded with these messages but information is almost real-time. Validation checks are also performed every 60 seconds.

The terms used in the following tables describe the following metrices:

SL - Succesful lifecycles This describes the number of completed lifecycles, which lead to the computation of the right correlation index on node *Output*. In a working environment, this number should be as high as the number of events emitted.

FL - Failed lifecycles This describes the number of failed lifecycles (e.g. because of lost events), which lead to no computation of the right correlation index on node *Output*. In a working environment, this number should be zero. Reasons for a failed lifecycle can be the loss of events over the network or bugs in the single nodes.

LAEL - Lowest average execution latency (ms) Execution latency describes the amount of time it takes from the moment, an event arrives, until the method processing it finishes

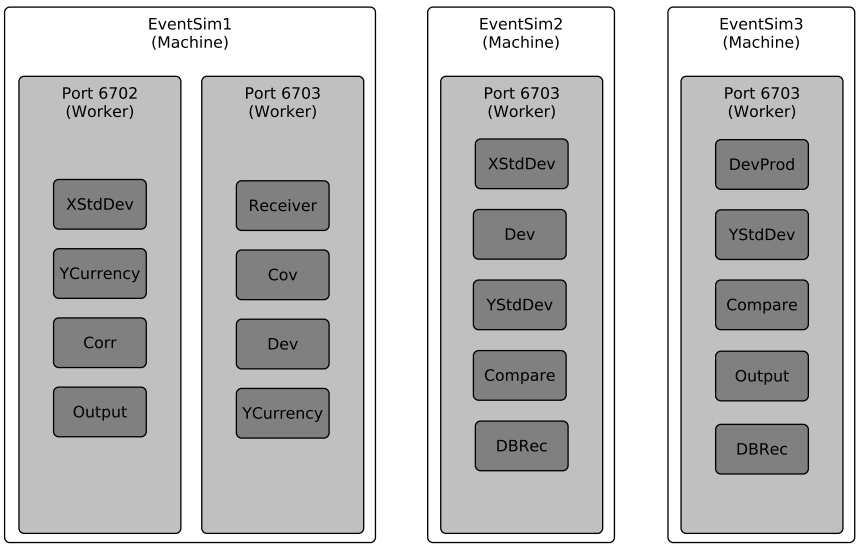

Figure 6.2: Exemplary deployment of the scenario

its work. The lowest average execution latency is the average execution latency of the component with the lowest execution latency in the system, which is listed in parentheses. It has to be noted, that the component *Output* is not eligible for this metric since it does not communicate with the Tool component and has, therefore, clearly the lowest execution latency.

HAEL - Highest average execution latency (ms) This metric describes the component with the highest average execution latency. For evaluation purposes, this node can then be regarded as the most critical node for the platform in terms of performance.

ART - Approximated runtime This is the time elapsed from sending the first event until the last events have been processed. In all the following experiments, this time can be twice the time of the experiment (a limit used for evaluation, in reality the ART can be of any value). The other statistical values are detected either when all events have been processed or the maximum allowed value for the runtime has been reached.

6.2 Performance of the EventSim framework

Using the base configuration, described in Section 6.1, the performance of the framework is monitored, focusing on different aspects. For this evaluation, events are emitted from within the Simulation component to test the performance in a normal state. This section deals with the monitoring part of the EventSim framework, since there are no simulation actions performed during the lifecycle of the system. It serves as a reference for the following sections, where different faults are introduced into the system and the behaviour of it is evaluated. Therefore,

different rates for event sending were set and the results are documented in Table 6.1. To evaluate the performance aspect of the EventSim framework, the same experiments without using the EventSim framework are illustrated in Table 6.2. It is clearly shown that the overhead of the EventSim framework is significant and the business-logic of the scenario is kept simple and, therefore, not performance-critical. The performance-critical part of the EventSim framework are the service calls, which are performed for every outgoing connection of a worker node. Therefore, components with two outgoing connections have worse performance values in combination with the EventSim framework compared to the values without the use of the EventSim framework. Additionally, components with two ingoing connections, like component *Corr* only perform calculations on every second call (in the relevant first call, the correlating event is stored in the cache), so the performance values are better in average. For the following explanations, the results from Table 6.1 are used.

Events/min	Minutes	SL	FL	LAEL (ms)	HAEL (ms)	ART (sec)
100	1	100	0	4.144 (Corr)	21.149 (YCurrency)	60
100	5	500	0	2.938 (Corr)	14.493 (YCurrency)	300
100	10	1000	0	1.994 (Corr)	9.862 (YCurrency)	600
500	1	500	0	2.418 (Corr)	11.353 (YCurrency)	60
500	5	2500	0	1.705 (Corr)	8.273 (YCurrency)	300
500	10	5000	0	1.562 (Corr)	7.446 (YCurrency)	600
5000	1	5000	0	3.353 (ComCov)	13.166 (YCurrency)	85
5000	5	25000	0	3.211 (ComCov)	20.853 (YCurrency)	420
5000	10	50000	0	3.243 (ComCov)	24.282 (YCurrency)	870

Table 6.1: Performance evaluation of the scenario

Events/min	Minutes	SL	FL	LAEL (ms)	HAEL (ms)	ART (sec)
100	1	100	0	0.978 (Corr)	3.720 (DBRec)	60
100	5	500	0	0.551 (Corr)	2.848 (DBRec)	300
100	10	1000	0	0.453 (Corr)	2.141 (DBRec)	600
500	1	500	0	0.655 (Corr)	2.747 (DBRec)	60
500	5	2500	0	0.291 (Corr)	1.542 (DBRec)	300
500	10	5000	0	0.247 (Corr)	1.300 (DBRec)	600
5000	1	5000	0	0.261 (ComCov)	1.248 (DBRec)	60
5000	5	25000	0	0.182 (ComCov)	0.889 (DBRec)	300
5000	10	50000	0	0.193 (ComCov)	0.853 (DBRec)	600

Table 6.2: Performance evaluation of the scenario without the EventSim framework

The timing diagrams for the lowest and highest event-rate (each running 10 minutes) are illustrated in Figure 6.3. This diagram shows the amount of events received by the component *Output* every second during the simulation time, where the red dotted line on the right side

represents the the ten minute mark. So everything after this line is received with a delay. The exact results can be interpreted in different ways:

- The rate of 5000 events/min was chosen for the table because this was the lowest amount (in steps of thousand events/min), which lead to a significant delay of worfklows. This can be seen in the last column, where the runtime of all the events is significantly higher than the runtime in the runs with a lower event rate. Additionally, Figure 6.3 shows that at a low event-rate, the events are received at a constant rate visualizing that the load can easily be handled by the framework. The problem with the high event-rate in this case is not the performance of the individual nodes themselves but that there is only one entry node *Receiver*. In the case of 5000 events/min, the next event comes in, when the current one is executed. This can be seen in Figure 6.3, where the remaining events are put into the system by *Receiver* at the end. This load can be handled by the other nodes but the entry node was not made redundant as part of the configuration.

- For the first two event rates, the values for LAEL and HAEL is decreasing with the runtime of the simulation. This can be explained with caching, as the first requests are taking longer and this effect is normalized after time, depending on the events produced during simulation time. For the highest event-rate, the HAEL of *YCurrency* is increasing with the runtime. This can be explained by the high throughput of events, which can be seen at the right end of the second diagram in Figure 6.3.

- The components with the LAEL and HAEL are the same for nearly every experiment, with the exception of *ComCov*, having the LAEL with the highest event-rate. This circumstance can be explained by the fact that at the end of the simulation, events are executed more frequently, which as more effect on nodes, which are nearer to the event source. The effect of this high-frequency is partially absorbed for *ComCov* by its preceeding nodes. It has to be noted, however, that the values for *Corr* and *ComCov* are much alike throughout the experiment for every event-rate and duration.

6.3 Fault Injection & Detection

This section is used to apply the actions, described in Section 4.3, to the scenario and evaluate the reaction of the metrices to them. Therefore, the values, detected in Section 6.2, are used as reference and checked regarding their relevance for detecting critical nodes and paths. The action ”Event Emitting” is not part of this section, since it is the basis of the whole evaluation process and used as part of every other action. For all of the subsequent actions, the lowest and highest event-rate from Section 6.2 are used with a runtime of 10 minutes. Events on individual nodes (in this section every action with the exception of ”Interrupt events”) also effect other nodes running on the same host and port. This is due to the fact, that not every node of the platform runs on its individual host and port because of resource reasons. As the runtime of the experiments is 10 minutes, this effect should balance itself over the experiment.

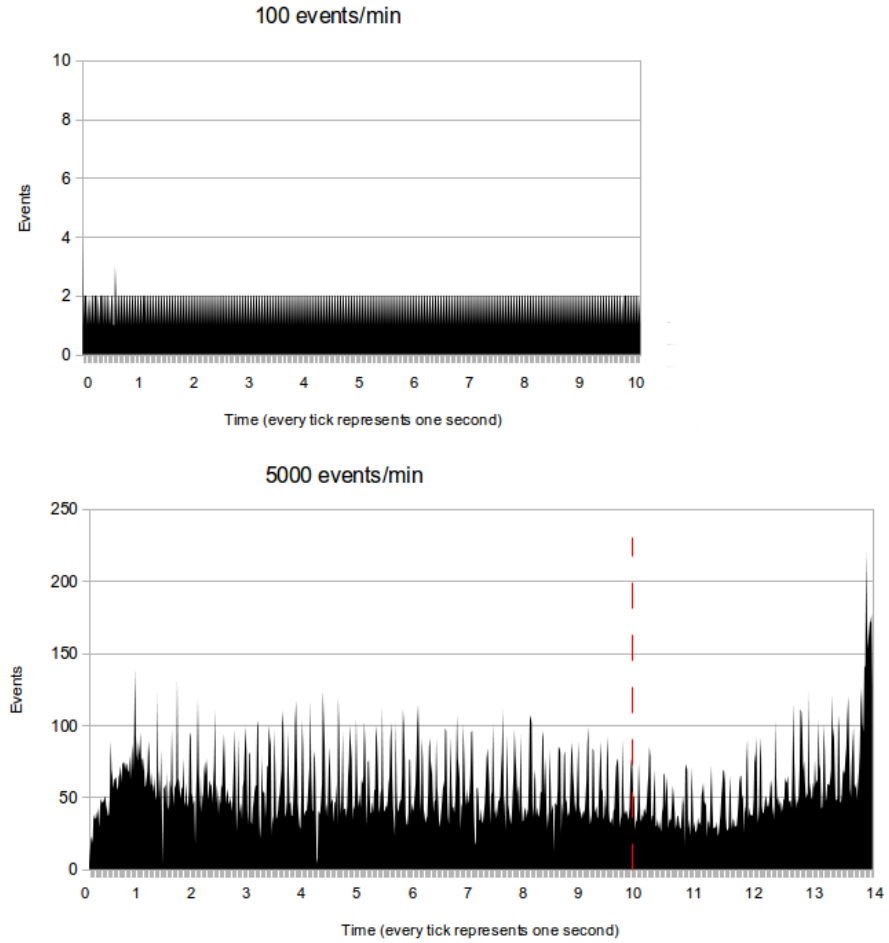

Figure 6.3: Performance analysis for different event rates

Additional Traffic

Additional traffic can be used to introduce unexpected effects into the platform. For evaluation purposes, additional data is sent to the nodes with the lowest and highest average execution latency. In case of redundant nodes, one of the nodes is selected at random. The rate of sending this traffic is changed during the experiment to detect the impact of this additional data. The amount of data is 10KB, since much higher data amounts would lead to performance issues for the simulation. The results are illustrated in Table 6.3 for component *YCurrency* and in 6.4 for component *Corr*/*ComCov*, which had the highest and lowest latency in the fault-free scenario.

A big difference, compared to the results from the normal test, is the existence of failed lifecycles. Those can be explained by the fact that some of the events can not be handled when sending additional traffic. This effect increases with the amount of events sent per minute, because the additional traffic affects more events, when more events are sent as a whole. The number of failed lifecycles is smaller, when the component with the LAEL is targeted compared with the

E/min	Rate (sec)	SL	FL	LAEL (ms)	HAEL (ms)	ART (sec)
100	60	964	**36**	2.577 (Corr)	13.580 (YCurreny)	600
100	30	736	**264**	3.174 (Corr)	20.148 (YCurreny)	600
5000	60	14,772	**35,228**	2.463 (ComCov)	28.506 (YCurreny)	600
5000	30	14,616	**35,384**	2.413 (ComCov)	16.131 (YCurreny)	600

Table 6.3: Introduction of additional traffic to component with highest average execution latency (HAEL)

E/min	Rate (sec)	SL	FL	LAEL (ms)	HAEL (ms)	ART (sec)
100	60	947	**53**	2.467 (Corr)	13.248 (YCurrency)	600
100	30	796	**204**	3.075 (Corr)	14.439 (YCurrency)	600
5000	60	32,895	**17,105**	3.136 (ComCov)	24.810 (YCurrency)	720
5000	30	18,547	**31,453**	4.089 (ComCov)	14.517 (YCurrency)	600

Table 6.4: Introduction of additional traffic to component with lowest average execution latency (HAEL)

effects of targeting the component with the HAEL. This confirms the results of the monitoring values illustrated in Table 6.1.
The values for latency themselves are also particularly higher for the simulations run during this action than for the normal test. This effect, however, is not as dramatic as the effect to the lifecycles. A possible explanation can be the fact that there are less events handled by each component (because of the failing lifecycles) and, therefore, the load on the components is not as high as in the normal scenario. This explanation is also supported by the comparison of the two tables for this action. In case of sending the traffic to component *YCurrency*, the LAEL is smaller than in the other scenario with the side-fact, that the number of succesful lifecycles is also smaller. This is not true for the HAEL, which can be explained with the relatively early occuring of *YCurrency* in the lifecycle. Therefore, lost events have less effect on this component. The action in the first case is also targeted at component *YCurrency*, which can affect its latency.
The rate of sending the additional traffic has an interesting effect on the other values. Opposed to the expected behaviour, the performance is better, if the additional traffic is sent in a more frequent way. This can perhaps be explained by characteristics of the Storm framework, since the ART is also longer in these cases. It can be assumed that Storm is able to deal with more frequent shutdowns (sending the additional traffic sometimes leads to the shutdown of a Storm node) in a better way and keeps the events buffered. This would also explain the longer ART.
It has to be noted, however, that these results are not optimal regarding their significance, since targeting one component may also have effects on other components, since several components are deployed on one Storm node. If enough hardware capacities are available, however, the optimal scenario would be to put every component on an individual node. This would ensure that the triggered actions only affect the relevant nodes.

To sum it up, this part of the evaluation illustrated that the monitoring part of the EventSim

Simulation component can be useful in identifying critical nodes of a lifecycle. The action itself can afterwards be used to measure the importance of individual nodes and how the risk of failing lifecycles is spread among the different components. In an optimal distributed platform, every component should be replicated in a way that the effect of sending additional traffic to components should be nearly the same for all components.

Event interception

This action is used to interrupt the communication between two specific components. It can be used to evaluate how a system handles missing events and how components are influenced by such a fault. In this section, it is used to intercept the path between component *YStdDev* and component *Denom*. This is interesting, since component *Denom* needs the correlating events from *Receiver* and *YStdDev* to compute its own result. However, since *Denom* is not actively waiting for the events (instead it is looking the correlating event up, if a certain event arrives), timing should not be affected by this interception. For the test runs, a variable percentage (P) of the totally sent events are intercepted after five minutes. This timestamp was not varied during the simulation and is chosen without specific intention, since this would not directly affect the outcome of this evaluation. The results of this simulation are illustrated in Table 6.5.

Events/min	P	SL	FL	LAEL (ms)	HAEL (ms)	ART (sec)
100	1	990	**10**	2.209 (Corr)	11.513 (YCurrency)	600
100	5	950	**50**	2.177 (Corr)	11.522 (YCurrency)	600
100	10	900	**100**	2.114 (Corr)	11.705 (YCurrency)	600
5000	1	49,500	**500**	3.092 (ComCov)	19.515 (YCurrency)	870
5000	5	47,500	**2,500**	3.215 (ComCov)	20.700 (YCurrency)	860
5000	10	45,000	**5,000**	3.123 (ComCov)	19.677 (YCurrency)	800

Table 6.5: Test results when intercepting the route between *YStdDev* and *Denom*

The number of affected events is the same as the number of interrupted events. This is an indication that the platform itself is capable of dealing with those interrupted events in the right way and that there are no side-effects. Additionally, it shows that the simulation action has no side-effects and only does what it should. The latency values are a bit smaller compared to the normal performance analysis but are not really affected by the number of interrupted events. This seems logical, as the components *ComCov* and *YCurrency* are not directly affected by the interruption (since they are placed on another path of the system) and component *Corr* does not have a high load at all.

This action is very useful for fetching information about the performance and the importance of a certain path in a platform. Since the number of interrupted events can be chosen freely, it can also be used for load testing if one path of a platform is nearly disabled at all. It has to be noted that during these tests the automatic resending, which is part of the normal Storm configuration, was turned off to see the direct effects of the simulation action. In combination with the action

”Kill worker”, this action can be used to detect the fault tolerance of an event-based system regarding its reliabilty (”How does the platform deal with lost messages?”) and its availability (”How fast does the platform recognize shutdown events and react to it properly?”).

Kill event source

Killing the event source (*Receiver* in the evaluation platform) can lead to bottlenecks at the beginning of the lifecycle, which can afterwards lead to lost events and missing lifecycles as a whole. The experiments performed using this action are illustrated in Table 6.6. The table features two new fields: The number of kill-signals sent to the event source (K) and the number of lost events (LE). Evaluating the platform is done by killing the event source in different frequencies.

Events/min	K	SL	LE	LAEL (ms)	HAEL (ms)	ART (sec)
100	1	923	**77**	2.411 (Corr)	**12.229 (YCurrency)**	600
100	2	835	**165**	2.413 (Corr)	**12.504 (YCurrency)**	600
100	4	689	**311**	2.593 (Corr)	**13.563 (YCurrency)**	600
5000	1	40,861	**9,139**	3.559 (ComCov)	**22.448 (YCurrency)**	760
5000	2	37,745	**12,255**	3.186 (ComCov)	**18.764 (YCurrency)**	690
5000	4	14,859	**35,1410**	2.729 (ComCov)	**14.225 (YCurrency)**	620

Table 6.6: Test results when killing the event source

Regarding successful lifecycles, it is clearly shown in the results that the amount is smaller if the event source is killed more often. This effect is higher, the more events are produced in one minute. On the other side, there are no failing lifecycles throughout the whole experiment, since the action only affects the beginning of the lifecycle.

The latency values are increasing with every additional killing for the low event rate. On the contrary, with the high event rate the effect is vice versa. Looking at the number of succesful lifecycles for this test runs explains this effect, e.g. if the event source is killed four times, the number of succesful lifecycles is nearly a third of the number of succesful lifecycles in the scenario with one event source killing. This leads to a reduced load for the nodes of the platform. The same effect is visible for the ART of the test runs, which is decreasing with every additional killing. For an easier understand of the workflow during such a lifecycle, the timing diagram for the last test run is illustrated in Figure 6.4. The breaks after each kill-attempt is approximately 30 seconds, which is also enlarged by the configuration of Storm, which could be optimized for such failing scenarios. Such an optimization, however, would also lead to increased traffic volumes in the system. A similar effect to the action ”Additional traffic” is visible, where more events are computed at the end because the event source is the bottleneck and computation of events is faster at the time all events passed the event source.

This action can be used for real-world applications to investigate how the platform deals with killing its event source and how fast the recovery process is performed. Additionally, the num-

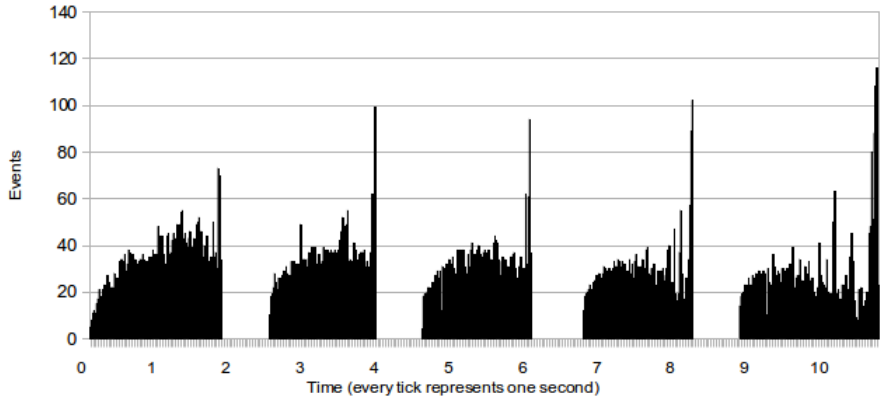

Figure 6.4: Timing diagram in case of killing the event source

ber of lost events because of the unexpected shutdown can be checked. In such a real-world application, the event source would be replicated but this also leads to issues regarding timing and synchronization, depending on the business logic of the platform.

Kill worker

This action is similar to the action ”Kill event source”, with the exception that in this case every worker can be forced to shutdown. For evaluation purposes, several tests are made throughout this section to investigate different aspects of the platform.

In a first step, component *Corr* is shutdown with the same frequencies and test setup as the event source in action ”Kill event source”. Following the results of the normal performance evaluation, this should not have a huge impact since this component is not performance-critical. On the other side, it is not made redundant, so there will be failing events during the lifecycle. Storm is able to start a new instance of a crashed node but this takes some time, which leads to failing lifecycles. The results for this test are illustrated in Table 6.7.

Events/min	K	SL	FL	LAEL (ms)	HAEL (ms)	ART (sec)
100	1	**938**	62	3.019 (ComCov)	13.085 (YCurrency)	600
100	2	**965**	35	3.232 (ComCov)	15.878 (YCurrency)	600
100	4	**926**	73	3.028 (ComCov)	13.805 (YCurrency)	600
5000	1	**47,276**	2,724	3.192 (ComCov)	20.360 (YCurrency)	900
5000	2	**39,304**	11,696	2.644 (ComCov)	19.523 (YCurrency)	700
5000	4	**26,753**	23,247	2.330 (ComCov)	15.487 (YCurrency)	630

Table 6.7: Test results when killing component *Corr*

The results seem logical, as the amount of successful lifecycles is decreasing with the amount of

kill attempts. The only exception in the first two test runs can be explained by the fact that not only the component *Corr* is killed but every component deployed on the specific node. This has of course more impact, the less events are sent. It has to be mentioned that component *ComCov* now has the lowest latency for all test runs, which seems also logical, as component *Corr* was killed during the tests. The ART is also decreasing with kill attempts, resulting out of the also decreasing amount of successful lifecycles.

The next test runs are performed by killing component *YCurrency*. This component is chosen because it seems to be the most performance-critical. On the other hand, it is replicated, in contrast to component *Corr*. Since these two facts are compensatory, it is interesting to compare the measured values with the values from the test run with component *Corr*. The test values themselves are illustrated in Table 6.8.

Events/min	K	SL	FL	LAEL (ms)	HAEL (ms)	ART (sec)
100	1	**954**	46	3.196 (ComCov)	16.894 (YCurrency)	600
100	2	**620**	380	2.734 (ComCov)	21.900 (YCurrency)	600
100	4	**484**	516	3.088 (ComCov)	28.681 (YCurrency)	600
5000	1	**34,079**	15,921	3.161 (ComCov)	21.996 (XCalc)	800
5000	2	**28,497**	22,503	2.555 (ComCov)	23.173 (YCurrency)	690
5000	4	**17,886**	32,114	2.447 (ComCov)	25,620 (YCurrency)	660

Table 6.8: Test results when killing component *YCurrency*

As for the results of this test run, they are very similar to the results of the previous test run. As one would guess, since component *YCurrency* is critical to the functional correctness of the scenario, the number of successful lifecycles is lower than in the previous test. The ART for the test runs is also a bit smaller than in the previous test, since the nodes had less load to handle. An interesting difference to the previous test is component *XCalc* being the component with the highest latency in one test run. In the other test runs with 5000 events per minute, component *XCalc* had almost the same latency as component *YCurrency*. The explanation for this behaviour is the fact that component *YCurrency* does not handle as many events as component *XCalc* in this scenario, because component *YCurrency* is frequently killed.

The last test of this section deals with final event receiver, component *Output*.As it is only used for the output of the result and not for computation, the number of succesful events should not be affected to the same extent as by killing of other components. Some lifecycles, however, will still not succeed, as the component is killed during event transmitting. The results are illustrated in Table 6.9.

The comparison with the action ”Kill event source” reveals that for a low number of kill-attempts, the number of successful lifecycles is higher by targeting the event source instead of the event target *Output*. The contrary effect is seen for a high number of kill-attempts. This seems logical as killing the event source on a frequent basis harms the performance of com-

Events/min	K	SL	FL	LAEL (ms)	HAEL (ms)	ART (sec)
100	1	**946**	54	2.802 (Corr)	15.145 (YCurrency)	600
100	2	**561**	439	2.737 (Corr)	14.281 (YCurrency)	600
100	4	**511**	489	3.044 (Corr)	14.425 (YCurrency)	600
5000	1	**37,988**	12,012	3.911 (ComCov)	18.803 (YCurrency)	870
5000	2	**32,911**	17,089	2.601 (ComCov)	19.845 (YCurrency)	680
5000	4	**27,683**	23,317	3.953 (ComCov)	17.366 (YCurrency)	700

Table 6.9: Test results when killing component *Output*

putation, whereas killing the event target harms only the final event receival, depending on the number of events coming in during the recovery phase of the component. Additionally, component *Output* is replicated, whereas the event source has a single instance.

This action can be used to identify bottlenecks of the platform, as well as performance-critical components. It also showed that the monitoring information, retrieved in Section 6.2, is valid because the performance is stressed more when targeting a performance-critical component. This section also showed that the action is more useful, when less components are deployed on one node, since it takes down the entire node and not only the component, which is targeted.

To sum it up, different actions can be used to inject faults into the SUT. In addition to the monitoring of system data, this fault-injection is used to identify critical components or paths of the system and adapt to changing conditions at runtime. These faults are also detected by the EventSim framework, which makes it convenient to adapt to them in real-time. As a conclusion, the EventSim framework is used to monitor regular system behaviour, detect faults and inject faults on purpose for information or learning reasons.

6.4 Benefits and using the simulation data

As shown in this Chapter, the EventSim framework has several benefits to support different EBSs:

- The number of currently active nodes is constantly monitored and notifications about changes in the node structure are automatically received.
- Validation of the system, as described in Section 4.4, is done periodically and irregular states are communicated to the user of the EventSim Simulation component.
- Using fault-injection, critical paths and components of the SUT are identified regarding different aspects. This information can afterwards be used to adapt the SUT or at least pay attention to these aspects in future versions of the SUT if adaption is not suitable. An example for this usage is shown in the following.

- These faults are also detected in real-time using the fault-detection mechanism of the EventSim framework, which uses the same backend as the normal monitoring mechanism.

In the following, a concrete configuration adaptation based on the results of fault-injection is described. Section 4.3 showed that component *YCurrency* is the most critical component in terms of latency. As said in Subsection 6.3, killing one component also affects other components in a scenario with only four nodes. Therefore, another series of tests is used to perform tests in an environment with four nodes but the additional constraint that component *YCurrency* is always executed in a separate process. Such a constraint can be configured in Storm if the underlying business logic requires such a behaviour. This constraint harms the flexibility of the whole system, so it should only be used if necessary and for critical components. The results of this test are illustrated in Table 6.10.

Events/min	K	SL	FL	LAEL (ms)	HAEL (ms)	ART (sec)
100	1	**943**	57	2.922 (Corr)	13.954 (YCurrency)	600
100	2	**991**	9	3.069 (Corr)	14.309 (YCurrency)	600
100	4	**855**	145	3.060 (Corr)	14.429 (YCurrency)	600
5000	1	**48,181**	1,819	3.804 (Corr)	13.777 (YCurrency)	810
5000	2	**46,208**	3,792	3.760 (Corr)	14.864 (YCurrency)	770
5000	4	**44,941**	5,059	4.746 (ComCov)	13.555 (YCurrency)	780

Table 6.10: Test results when killing component *YCurrency* with adapted setup

The results show a big difference compared to the previous results (see Table 6.8), since the number of successful lifecycles is significantly higher. Additionally, the latency values (regarding the highest latency) are smaller than before, which seems logical as only smaller parts of the platform are targeted compared with the action executed in Subsection 6.3 and illustrated in Table 6.8.This example shows that the data collected during monitoring after fault injection can effectively be used to target specific aspects of the system to enhance performance or fault-tolerance.

6.5 Open issues of the current implementation

As the evaluation in the previous section showed, the EventSim framework can be used for fault injection and detection in its current state. It is capable of showing critical paths or components in an EBS. However, there exist open issues, which can be adressed in the future:

- An issue, which occured during several simulation actions, is the granularity of the SUT. If every worker node only hosts one component, the simulation action can be applied in an optimal way and the effects are as expected. In the configuration used for evaluation, every worker node hosted multiple differenct components. Therefore, every action against one component also affected other components on this node, since the components can only

be targeted by their hostname and port on the worker node. To deal with this effect, more specific simulation actions for every SUT are needed, which can be implemented using the EventSim framework. However, this would not suit the general approach implemented by the framework and should only be done as a workaround. Testing real world systems should not be affected by these circumstance, since, in such a scenario, each worker node normally hosts one component or at least hosts its components on different ports.

- The communication with the EventSim Tool component is reduced to sending notifications about EPAs and events during the simulation test runs. Information like properties and buffers are not communicated to the service, since this would affect the runtime of the simulation significantly. This results out of the fact that each of these classes is submitted via a separate web service call, which culminates in a long request time. A workaround for this behaviour would be to submit the entire object tree as a whole but this could also bring update policy problems with it if parts of the object tree are updated. Therefore, this approach was chosen out of simplicity and since it is sufficient for basic simulation purposes.

CHAPTER 7

Summary and Conclusion

This section provides a summary to the work done in this thesis. At the beginning, the problem formulation is shortly explained again. Afterwards, the solution presented in this work is summarized, followed by a discussion of future work based on this thesis.

7.1 The Problem revisited

A good fault-monitor for EBS is able to model, monitor and provide possibilities for fault detection and injection. In addition, it should be implemented as generic as possible to support all known types of EBSs.

As mentioned in Chapter 1, there exist multiple types of different event-based systems (EBSs) in computer science today. In the context of these systems, fault-tolerance is an important issue, since most of the systems are critical and should not crash under exceptional situations (e.g. load bursts) or in the presence of faults. Based on this paradigm, there has been much research in the field of fault-tolerant EBSs but most of these approaches focused on a specific type of EBS.

The reason for the specialization of research in this area are key differences between different types of EBSs regarding their architecture or communication flow. This makes it hard to develop a common model for all of them and keeping the specific features in mind at the same time. In addition to the problem of model creation for these systems, monitoring is also beneficial for fault-tolerance. This involves monitoring of the normal workflow, as well as fault detection. In this context, the reaction of the system-under-test (SUT) needs to be measured in addition to normal monitoring. This is an important performance indicator of an EBS. A fault-monitor should also be able to inject faults in the SUT, which target different areas of it. On one side, specific nodes can be targeted, and on the other side, injections targeting specific paths of the system can also be used to detect critical parts of the system. The main challenge in this field also comes from the differences between the types of EBSs.

7.2 A Solution approach - The EventSim framework

The components implemented in this work are summarized under the term *EventSim Framework*. The different parts of this framework adress the problem of monitoring a fault-tolerant EBS and simulate the handling of faults.

Modeling of the SUT is done using a generic metamodel for the different types of EBSs. If necessary, this metamodel can also be adapted or rewritten. Every other component of the EventSim framework uses the model generated with *Metamodel Management* component. The current state of the model, as well as different versions of the metamodel, are saved using the *Tool* component of this framework. Fault detection and injection of the SUT are implemented in the *Simulation* component. The adaption to a new system requires little effort, which makes the framework extensible for future use and adaption. This is also true for the currently implemented simulation actions, which are designed in a generic way.

Additionally, a *testing platform* was developed as part of this work, which is used for evaluation purposes. The results, presented in Chapter 6, suggest that the monitoring of events works as expected and detected weaknesses of the SUT are more critical, when it comes to fault injection, than other parts of the SUT. The evaluation revealed several aspects regarding simulation. The load information of event sending components was clearly shown with full information about critical nodes and paths. Additionally, the simulation actions are more effective, when targeting detected critical parts of the system. This showed that the information gained from monitoring is valid. The actions also revealed aspects of the system code and configuration, which can be optimized regarding performance. All in all, it has been shown that the simulation of an event-based system can be done in real time. On top of that, fault detection and injection mechanisms provided useful information about critical parts of the system.

7.3 Future work

As for model generation, the *Modeling* component is a task for future work, which was shortly presented in Section 4.1. It is used to develop new metamodels or adapt existing metamodels using a Graphical User Interface (GUI).

Open issues regarding evaluation of an SUT have been presented in Section 6.5. The granularity of the SUT is not changeable for some real-world systems but the different simulation actions can be optimized to target only the node, which should be attacked, and produce no side-effects. The persisted data can also be optimized in future versions of the framework, since the current implementation is optimized with the focus on performance but can get complex as the SUT grows. A possible solution for this problem is persisting objects using a small data format directly in the memory of the *Tool* component. Another enhancement to this component is the possibility to make it redundant and, therefore, have a shared memory used by the single instances.

Regarding the *Simulation* component, future work can deal with the extensibility of simulation actions. The currently implemented actions are designed in a generic way but target the specific component directly. A more generic way would be to handle these actions over the *Tool* component. On the other side, this would lead to tighter coupling of these two components, so

the optimal way for generic actions can be discussed. Another aspect, which is subject to future work, focuses on the visualization of simulation results. Currently, they are illustrated in a textual way but future versions of the framework can visualize them graphically or use historical data for statistics. Additionally, the service calls to the EventSim framework can be injected using Aspect Oriented Programming (AOP) [23] in the future. Currently, they are called as part of the business logic.

Bibliography

[1] Russell J. Abbott. Resourceful systems for fault tolerance, reliability, and safety. *ACM Comput. Surv.*, 22(1):35–68, March 1990.

[2] Réka Albert and Albert-László Barabási. Topology of evolving networks: local events and universality. *Physical review letters*, 85(24):5234, 2000.

[3] David G. Andersen and Rohit N. Rao. Principles for end-to-end failure masking, 2003.

[4] Bernhard Angerer. Space-based programming. `http://onjava.com/pub/a/onjava/2003/03/19/java_spaces.html`, last visited on 06.03.2013.

[5] Larry Apfelbaum and John Doyle. Model based testing. In *Software Quality Week Conference*, pages 296–300, 1997.

[6] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, J-C Fabre, J-C Laprie, Eliane Martins, and David Powell. Fault injection for dependability validation: A methodology and some applications. *Software Engineering, IEEE Transactions on*, 16(2):166–182, 1990.

[7] A. Avizienis and J.-C. Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, 1986.

[8] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004.

[9] Guruduth Banavar, Tushar Deepak Chandra, Robert E. Strom, and Daniel C. Sturman. A case for message oriented middleware. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 1–18. Springer-Verlag, 1999.

[10] Roger S Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. *arXiv preprint cs/0612115*, 2006.

[11] Nelly Bencomo, Amel Bennaceur, Paul Grace, Gordon Blair, and Valérie Issarny. The Role of Models@run.time in Supporting On-the-fly Interoperability. *Computing*, 2012.

[12] Thomas Bernhardt and Alexandre Vasseur. Complex event processing made simple using esper. http://www.theserverside.com/news/1363826/Complex-Event-Processing-Made-Simple-Using-Esper, last visited on 02.11.2013.

[13] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. Reflective model driven engineering. In *«UML» 2003-The Unified Modeling Language. Modeling Languages and Applications*, pages 175–189. Springer, 2003.

[14] G. Blair, N. Bencomo, and R.B. France. Models@ run.time. *Computer*, 42(10):22–27, 2009.

[15] Douglas C. Bossen and MY Hsiao. Model for transient and permanent error-detection and fault-isolation coverage. *IBM Journal of Research and Development*, 26(1):67–77, 1982.

[16] Royal Society (Great Britain). *Proceedings of the Royal Society of London*. Number Bd. 58. Taylor & Francis, 1895.

[17] Stefan Bruning, Stephan Weissleder, and Miroslaw Malek. A fault taxonomy for service-oriented architecture. In *High Assurance Systems Engineering Symposium, 2007. HASE'07. 10th IEEE*, pages 367–368. IEEE, 2007.

[18] Alejandro Buchmann and Boris Koldehofe. Complex event processing. *it-Information Technology*, 51(5):241–242, 2009.

[19] Peter Carruthers. Simulation and self-knowledge: A defence of theory-theory. *Theories of theories of mind*, pages 22–38, 1996.

[20] Tsun S. Chow. Testing software design modeled by finite-state machines. *Software Engineering, IEEE Transactions on*, (3):178–187, 1978.

[21] Wim A Coekaerts. Heartbeat mechanism for cluster systems, September 15 2009. US Patent 7,590,898.

[22] Christian Colombo, Gordon J Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Formal Methods for Industrial Critical Systems*, pages 135–149. Springer, 2009.

[23] Constantinos A Constantinides, Atef Bader, Tzilla H Elrad, P Netinant, and Mohamed E Fayad. Designing an aspect-oriented framework in an object-oriented environment. *ACM Computing Surveys (CSUR)*, 32(1es):41, 2000.

[24] Flavin Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, 1991.

[25] Patrick Crowley. A dynamic publish-subscribe network for distributed simulation. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, PADS '08, pages 150–, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Opher Etzion. Event processing agent. *Encyclopedia of Database Systems*, pages 1052–1053, 2009.

[27] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[28] Opher Etzion and Peter Niblett. *Event processing in action*. Manning Publications Co., 2010.

[29] Patrick T Eugster, Rachid Guerraoui, A-M Kermarrec, and Laurent Massoulié. Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67, 2004.

[30] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[31] Eugene F Fama. The behavior of stock-market prices. *Journal of business*, pages 34–105, 1965.

[32] Jean-Marie Favre. Towards a basic theory to model model driven engineering. In *3rd Workshop in Software Model Engineering, WiSME*. Citeseer, 2004.

[33] Ludger Fiege, Gero Muehl, and Felix C. Gaertner. Modular event-based systems. *THE KNOWLEDGE ENGINEERING REVIEW*, 17:359–388, 2006.

[34] Yevgeniy Gershteyn. Fault tolerance in distributed systems. *IEEE Concurrency*, 4(2):83–88, 1996.

[35] Jens Guthoff and Volkmar Sieh. Combining software-implemented and simulation-based fault injection into a single fault injection method. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 196–206. IEEE, 1995.

[36] Seungjae Han, Kang G Shin, and Harold A Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 204–213. IEEE, 1995.

[37] Bernhard Haslhofer and Wolfgang Klas. A survey of techniques for achieving metadata interoperability. *ACM Comput. Surv.*, 42(2):7:1–7:37, March 2010.

[38] George T Heineman and William T Councill. *Component-based software engineering: putting the pieces together*, volume 17. Addison-Wesley Reading, 2001.

[39] Y-C Ho. Introduction to special issue on dynamics of discrete event systems. *Proceedings of the IEEE*, 77(1):3–6, 1989.

[40] Richard Hofmann, Rainer Klar, Bernd Mohr, Andreas Quick, and Markus Siegle. Distributed performance monitoring: Methods, tools, and applications. *Parallel and Distributed Systems, IEEE Transactions on*, 5(6):585–598, 1994.

[41] Masaaki Honda, Fumitada Itakura, and Nobuhiko Kitawaki. Adaptive predictive processing system, August 27 1985. US Patent 4,538,234.

[42] Pavel Hruby. *Model-driven design using business patterns*. Springer, 2006.

[43] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[44] Waldemar Hummer, Christian Inzinger, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar. Deriving a unified fault taxonomy for event-based systems. In *6th ACM International Conference on Distributed Event-Based Systems*, 2012.

[45] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[46] Rolf Isermann. Model-based fault-detection and diagnosis–status and applications. *Annual Reviews in control*, 29(1):71–85, 2005.

[47] Farnam Jahanian, Ragunathan Rajkumar, and Sitaram CV Raju. Runtime monitoring of timing constraints in distributed real-time systems. *Real-Time Systems*, 7(3):247–273, 1994.

[48] Paul Jensen. Complex event processing with esper. `http://esper.codehaus.org/`, last visited on 06.03.2014.

[49] Martin Kempa and Zoltán Adám Mann. Model driven architecture. *Informatik-Spektrum*, 28(4):298–302, 2005.

[50] Hong J Kim and Evan S Huang. Extensible stylesheet designs using meta-tag and/or associated meta-tag information, December 5 2006. US Patent 7,146,564.

[51] Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA explained, the model driven architecture: Practice and promise*. Addison-Wesley Professional, 2003.

[52] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in action*. Manning, 2007.

[53] Jean-Claude Laprie. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, pages 2–11, 1985.

[54] Franck L Lewis. Wireless sensor networks. *Smart environments: technologies, protocols, and applications*, pages 11–46, 2004.

[55] Zheng Li, Yan Jin, and Jun Han. A runtime monitoring and validation framework for web service interactions. In *Software Engineering Conference, 2006. Australian*, pages 10–pp. IEEE, 2006.

[56] Marc Loy, Robert Eckstein, Dave Wood, James Elliott, and Brian Cole. *Java swing*. O'Reilly, 2012.

[57] David C Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

[58] Leonardo Mariani. A fault taxonomy for component-based software. *Electronic Notes in Theoretical Computer Science*, 82(6):55–65, 2003.

[59] Nathan Marz. Guaranteeing message processing in storm. `https://github.com/nathanmarz/storm/wiki/Guaranteeing-message-processing`, last visited on 15.03.2014.

[60] Nathan Marz. Common patterns of storm. `https://github.com/nathanmarz/storm/wiki/Common-patterns`, last visited on 16.08.2013.

[61] Nathan Marz. Concepts of storm. `https://github.com/nathanmarz/storm/wiki/Concepts`, last visited on 16.08.2013.

[62] Enrique Medina and Juan Trujillo. A standard for representing multidimensional properties: The common warehouse metamodel (cwm). In *Advances in Databases and Information Systems*, pages 232–247. Springer, 2002.

[63] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 178–187, New York, NY, USA, 2000. ACM.

[64] Stephen J Mellor. Agile mda. *MDA Journal*, 2004.

[65] Stephen J Mellor. *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional, 2004.

[66] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20:14–18, 2003.

[67] Brenda M. Michelson. Event-driven architecture overview. `http://www.elementallinks.com/2006/02/06/event-driven-architecture-overview/`, last visited on 08.02.2013.

[68] Gero Muehl, Ludger Fiege, and Peter R. Pietzuch. *Distributed event-based systems*. Springer, 2006.

[69] Arun Mukhija and Martin Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Systems Aspects in Organic and Pervasive Computing-ARCS 2005*, pages 124–138. Springer, 2005.

[70] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[71] Andrea Paoli, Matteo Sartini, and Stéphane Lafortune. Active fault tolerant control of discrete event systems using online diagnostics. *Automatica*, 47(4):639–649, 2011.

[72] Louis Perrochon, Walter Mann, Stephane Kasriel, and David C Luckham. Event mining with event processing networks. In *Methodologies for Knowledge Discovery and Data Mining*, pages 474–478. Springer, 1999.

[73] Iman Poernomo. The meta-object facility typed. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1845–1849. ACM, 2006.

[74] Gregory M Pope, Jeffrey F Stone, and John A Gregory. Automated software testing system, August 2 1994. US Patent 5,335,342.

[75] Christoph Rathfelder, David Evans, and Samuel Kounev. Predictive modelling of peer-to-peer event-driven communication in component-based systems. In *Proceedings of the 7th European performance engineering conference on Computer performance engineering*, EPEW'10, pages 219–235, Berlin, Heidelberg, 2010. Springer-Verlag.

[76] David S Rosenblum and Alexander L Wolf. *A design framework for Internet-scale event observation and notification*, volume 22. ACM, 1997.

[77] Linnyer Beatrys Ruiz, Isabela G. Siqueira, Leonardo B. e Oliveira, Hao Chi Wong, José Marcos S. Nogueira, and Antonio A. F. Loureiro. Fault management in event-driven wireless sensor networks. In *Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, MSWiM '04, pages 149–156, 2004.

[78] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[79] Sriram Sankar and Manas Mandal. Concurrent runtime monitoring of formally specified programs. *Computer*, 26(3):32–41, 1993.

[80] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[81] Bran Selic. The pragmatics of model-driven development. *IEEE Softw.*, 20(5):19–25, September 2003.

[82] Upasana Sharma. Fault tolerant techniques for reconfigurable platforms. In *Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India*, A2CWiC '10, pages 60:1–60:4. ACM, 2010.

[83] G. Sharon and O. Etzion. Event-processing network model and implementation. *IBM Syst. J.*, 47(2):321–334, April 2008.

[84] Guy Sharon. Event processing network. In *Encyclopedia of Database Systems*, pages 1053–1058. Springer, 2009.

[85] Jocelyn Simmonds, Yuan Gan, Marsha Chechik, Shiva Nejati, Bill O'Farrell, Elena Litani, and Julie Waterhouse. Runtime monitoring of web service conversations. *Services Computing, IEEE Transactions on*, 2(3):223–244, 2009.

[86] Brian Cantwell Smith. *Procedural reflection in programming languages*. PhD thesis, Massachusetts Institute of Technology, 1982.

[87] Richard Soley et al. Model driven architecture. *OMG white paper*, 308:308, 2000.

[88] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[89] Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999.

[90] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. *Computing and Mathematical Sciences Papers*, 2006.

[91] Wil van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. on Knowl. and Data Eng.*, 16(9):1128–1142, September 2004.

[92] Venkat Venkatasubramanian, Raghunathan Rengaswamy, Surya N Kavuri, and Kewen Yin. A review of process fault detection and diagnosis: Part iii: Process history based methods. *Computers & Chemical Engineering*, 27(3):327–346, 2003.

[93] Venkat Venkatasubramanian, Raghunathan Rengaswamy, Kewen Yin, and Surya N Kavuri. A review of process fault detection and diagnosis: Part i: Quantitative model-based methods. *Computers & chemical engineering*, 27(3):293–311, 2003.

[94] Antonino Virgillito, Roberto Beraldi, and Roberto Baldoni. On event routing in content-based publish/subscribe through dynamic networks. In *Distributed Computing Systems, 2003. FTDCS 2003. Proceedings. The Ninth IEEE Workshop on Future Trends of*, pages 322–328. IEEE, 2003.

[95] Thomas Vogel, Andreas Seibel, and Holger Giese. Toward megamodels at runtime. In *Proceedings of the 5th International Workshop on Models@ run. time at the 13th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo, Norway*, volume 641, pages 13–24, 2010.

[96] Rainer von Ammon. Event-driven business process management. In *Encyclopedia of Database Systems*, pages 1068–1071. Springer, 2009.

[97] Michael Weiss. Xml metadata interchange. In *Encyclopedia of Database Systems*, pages 3597–3597. Springer, 2009.

[98] Utz Westermann and Ramesh Jain. Toward a common event model for multimedia applications. *IEEE MultiMedia*, 14(1):19–29, 2007.