Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Event-Driven Simulator for the Palladio Component Model

Julijan Katić

Course of Study:	Informatik
Examiner:	Prof. Dr. Steffen Becker
Supervisor:	Floriment Klinaku, M.Sc.
Commenced:	July 7, 2020
Completed:	January 7, 2021

Abstract

Context. Most often, developers and architects model their component-based softwares but are not aware what impact their design decision can have to the system. For this reason, the Palladio Component Model was created in order to model such softwares in advance, but at the same time predict certain software qualities. The quality prediction is done by the simulators for the PCM.
Problem. However, in recent years the PCM is constantly extended with new meta-models, semantics, tools, and further more. In order to include these new features into the existing simulators, they generally have to change and adapt their existing components and code which can lead to the ripple effect. While some simulators try to be extensible by being modularized, the issues are not fully resolved.
Objective. In this thesis, the new Slingshot PCM simulator is introduced and developed which tries to improve extensibility by applying the Event-Driven Architecture (EDA). EDAs have the ability to be very extensible. Furthermore, a prototypical implementation of simulation behaviors will be given. The extensibility of the system is evaluated and compared against the most used simulators.
Method. First, the concept of the event-driven simulator is developed and implemented. The simulation behaviors are directly implemented afterwards. Finally, the extensibility of the Slingshot simulator is evaluated by using related software quality metrics as well as comparing the change in lines of code that is needed when implementing such simulation behaviors.
Result. The results show that the event-driven architecture does improve extensibility, as long as the system including the new extension is modularized. The full implementation for the simulation behaviors, however, are still under construction. Nevertheless, the core Slingshot system is now developed.
Conclusion. Because the Slingshot system is extensible, various features can be implemented into the system independently with less or no change of the existing system at all.

Kurzfassung

Kontext. Oft modellieren Softwareentwickler und -architekten ihre komponenten-basierten Softwaresysteme, ohne sich dessen bewusst zu sein, welche Folgen ihre Architekturentscheidungen für das resultierende System haben könnten. Aus diesem Grund wurde das Palladio Component Model entwickelt, das es erlaubt, neben dem eigentlichen Modellieren des Systems gleichzeitig auch die Qualitätseigenschaften vorherzusagen. Die Qualitätseigenschaften werden durch die Simulatoren für PCM berechnet.

Problem. In den letzten Jahren wurde PCM jedoch stetig weiterentwickelt und mit neuen Metamodellen, Semantiken, Werkzeugen, usw. ausgestattet. Simulatoren müssen, um die neuen Weiterentwicklungen zu unterstützen, ihre bereits bestehenden Module umprogrammieren, was häufig zu einem Ripple Effect führt. Während zwar manche Simulatoren versuchen ihre Erweiterbarkeit durch Modularisierung zu verstärken, können häufig eine Reihe von verschiedenen Features ohne Änderung des Codebases nicht implementiert werden.

Ziel. In dieser Arbeit wird ein neuer Simulator konzeptioniert und entwickelt, der auf die ereignisgesteuerte Architektur basiert. Ereignisgesteuerte Architekturen haben die Eigenschaft stark erweiterbar zu sein. Außerdem werden neben dem Basissystem prototypische Implementierungen der Simulationen gegeben. Die Erweiterbarkeit wird schließlich mit den anderen Simulatoren evaluiert.

Methode. Zunächst wird ein Konzept des ereignisgesteuerten Simulators entwickelt und schließlich implementiert. Gleichzeitig werden auch direkt die Simulationen implementiert. Am Ende wird die Erweiterbarkeit evaluiert, indem bestimmte Softwaremetriken verglichen werden. Zusätzlich werden die Änderungen erwähnt, die nötig sind, um eine Simulation zu entwickeln.

Ergebnis. Die Werte zeigen, dass ein ereignisgesteuerter Simulator tatsächlich die Erweiterbarkeit verbessert, sofern das System inklusive der Erweiterungen modularisiert bleibt. Nichtsdestotrotz sind die Entwicklungen der Slingshot-Simulationen noch nicht abgeschlossen.

Fazit. Dadurch, dass das System erweiterbar ist, können verschiedene Erweiterungen unabhängig voneinander entwickelt werden, sodass nun weniger oder sogar keine Änderungen am existierenden System nötig sind.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

Ca Afferent Coupling. 7

CBO Coupling between Object Classes. 7

CBSE Component-Based Software Engineering. 1

CC Component Cohesion. 43, 46, 48

EDA Event-driven Architecture. 1

EMSA Extensibility Metric for Software Architectures. 41

EPC Event-driven Process Chain. ix, 10, 30

EXTCBO External Coupling between Object Classes. 43, 46, 48

FCFS First-Come, First-Served. 20

GQM Goal-Question-Metric. 42

IEE Integrability and Extensibility Evaluation. 43

LCOM Lack of Cohesion in Methods. 8

LOC Lines of Code. 43, 46

OSGi Open Service Gateway Initiative. 12

PCM Palladio Component Model. ix, xi, 1

PU Processing Units. 20

QADA Quality-driven Architecture and quality Analysis. 43

QoS Quality of Services. 1

RDSEFF Resource Demanding Service-Effect Specification. 18

RFC Response for a class. 8

SEFF Service-Effect Specification. 18, 61

WMC Weighted Methods per Class. 8

XSD XML Schema Definition. 14

1 Introduction

In practice, there are many different approaches when creating software systems. One approach is the Component-Based Software Engineering (CBSE) [KB98], in which the software is divided into multiple components and later composed together into one system. This approach is beneficial insofar that the components can be developed independently from each other, and hence having the benefit to be reused by others later on. However, software architects face the challenge of applying the non-functional quality requirements to the system, because it is sometimes hard to know what impact architectural decisions have in terms of quality in the resulting system.

The Palladio Component Model (PCM) [BKR09] provides the ability to predict certain Quality of Services (QoS) of component-based softwares. Types of QoS are typically performance, scalability, reliability, etc. The PCM defines a set of meta-models that can be used to model the system and at the same time get quality predictions to the system in advance. With this, the architect can then decide to change its model in order to meet the quality criteria.

The quality predictions are done by simulators such as the SimuLizar [Mey11], EventSim [MH11], or one of the first simulators like SimuCom by [Bec08]. Each simulator is adhering to a certain simulation world view and approach that impact the performance of the simulator itself. Furthermore, all these simulators are typically performance simulators that predict the performance of a PCM system. Newer simulators or extensions to existing ones are either providing more quality prediction to other types of QoS, or were created to be more scalable and have an improved performance compared to older simulators. More often, extensions have to be made in order to support new features of PCM models.

Problem Statement

However, the current simulators lack a great deal of extensibility when it comes to create these extensions. Quite often it is required to make certain changes to the simulators when a new feature arrives. This is due to the fact that some simulators are not modularized enough, or do not provide the corresponding extension mechanism to include these extensions. In that case, this could result in high development efforts in order to create those extensions, and such efforts are often expensive. Furthermore, changing the inner structures of systems can lead to interface breaks which other software systems rely or depend on and thus the so-called ripple effect.

In order to create a highly extensible system, the right architecture for the simulator must be used. One such architecture is the Event-driven Architecture (EDA) that combines multiple software modules and services and lets them exchange and publish events. These systems exhibit extremely loose coupling as one module does not depend on the actual formal methods of other modules. On the contrary, EDA based services talk to each other with publishing events to which other modules listen. In this way the software modules are decoupled and can allow for changes without the risk of the ripple effect.

Because the simulators are written in Java, annotations can further ease the process of developing the system. Due to the fact that annotations only provide more information of the code but do not invoke any methods at runtime, they can be used to further decouple the system and achieve more extensibility.

Purpose of the Thesis

The goal in this thesis is to create a prototypical PCM simulator called **Slingshot**[1] which is based on EDA and uses Java annotations. By this, Slingshot aims to be more extensible than the current simulators. With this in mind, the thesis tries to answer the following research question:

> **Research Question**
> Does an event-driven architecture coupled with annotation definitions and processing improve the extensibility of a PCM simulator?

The method to answer this question is to look at the different software quality metrics that heavily impact extensibility and compare them to the current established simulators. Moreover, a small case study is conducted in which certain simulation behaviors are implemented. Also, different literatures are used to gain more insights in other EDA based systems. The change that is needed to implement the behavior will be also compared. The less change is required, the more extensible the system can be considered.

Contribution

The contribution of this thesis is hence a new concept for EDA based PCM simulators, providing the way of handling events in such systems, including the implementation of the Slingshot simulator. Other PCM developers then have the ability to simply extend the Slingshot simulator with their behavior implementations to support a variety of feature types in PCM. Furthermore, the first main behavior implementations needed to simulate PCM are presented.

Thesis Structure

The thesis is divided in the following five parts:

[1]The name was chosen as this refers to the nature of events. The simulator considers and sees only events. Also, it lets "throw" events, and they often happen suddenly to others, like an ordinary slingshot throws a stone.

Chapter 2 – Foundations and Related Work: In this chapter, the foundation for this work is laid down upon which Slingshot is built. It discusses the extensibility definition and its connection to EDA systems.

Chapter 3 – Slingshot Simulator: Afterwards, the first conceptual and prototypical implementation of Slingshot is presented. This will also discuss the implementation of EDA and different security measures, and explains the extension point that it defines used to extend the system.

Chapter 4 – Simulation Implementation: Next, the prototypical implementation of different simulation types is provided as an example to extend the system. This includes the simulation of usage, system and resource models of PCM.

Chapter 5 – Evaluation: Finally, the extensibility of Slingshot is evaluated and compared to the other simulators here using the quality and change metrics, and it also discusses the threats to validity that could occur.

Chapter 6 – Conclusion: At the end, this chapter concludes the thesis, including a summary of the contributions and results that were found.

2 Foundations and Related Work

In the introduction, a clear goal was given that Slingshot needs to achieve. This chapter provides the foundation of this work to achieve the goal. It defines the notion of extensibility and its relationships with other quality attributes, and additionally will briefly discuss the EDA that is going to be implemented in Slingshot. This chapter will already answer the question as to why EDA is considered extensible and hence is fitting to the aim of Slingshot. Furthermore, because Slingshot bases on Eclipse and Java Annotation Processing, both tools will be presented as well. Lastly, the Palladio Component Model itself is introduced as this work will also consist of a simulator implementation, and some related works will be presented.

2.1 Extensibility

As introduced in the introduction, the problem of existing simulators is that it is hard to extend them without modifying existing code. The consequence to mitigate this problem is to make the simulator *extensible*. In order to do so, a clear and concise definition of extensibility is needed. In this section, the notion of extensibility and its relations to other software quality metrics are presented. These metrics will later help designing the study and evaluation of this work.

2.1.1 Definition

Definition 2.1.1 (Extensibility [JL09])
A software system is considered to be extensible if it is able to be extended with new functionality without or with minimal change and effects to its internal structures.

A software system that is not extensible requires developers to change its internal structures that may lead to API breaks and harder maintenance. Consequently to the definition for this case, a new module can easily be added and plugged in without risking of breaking the system. Also, it means that data is only added to the system and does not have to be somehow referenced by the core. As Slingshot is implemented in Java which is an object-oriented language, a refinement of the definition is needed, but in such a way that it does not contradict Definition 2.1.1.

Definition 2.1.2 (Extensibility (refined))
Let M_{new} be a software module that will be added to a software system S which in turn is a set of modules. S is considered to be extensible if for each existing module $M \in S$ there is no dependency from M to M_{new} with M_{new} still adding behavior to the system that has not existed before.

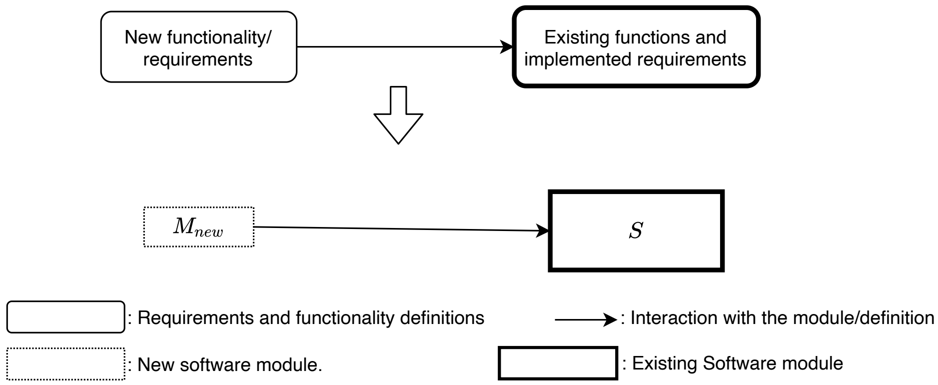


Figure 2.1: General example of an extensible system: Adding a Module M_{new} to the software system S so that the result will include new functionalities and meets requirements without changing S (or any module therein)

This definition still applies the general extensibility definition: As M_{new} does not have to be referenced by any other module M or is dependent by it, no change is required in the internal structure. Typically, the internal structure of a system is a special module `CORE` $\in S$ onto which every extension module can depend. However, `CORE` does not depend on any extension.

Figure 2.1 shows an example of the extensibility definition. In this scenario, a new functionality is required by some stakeholder of the software system and should be included in the overall functionality. This should be done without or with minimal change in the existing system S. The new module M_{new} adds meets those new requirements and interacts in some way with S. This interaction can be seen as inclusion, that is, M_{new} is now part of S. The interaction, however, goes only in one direction: S itself does not have to know the existence of M_{new}. Also, S and other module $M \in S$ will not know the origin of data coming from M_{new}, but can safely use this data.

If a software architecture can be built with the conceptual pattern as in Figure 2.1, then that system is considered extensible. Ultimately, not only is S extensible, but it is also modular. This fact can be easily derived from the following definition by Booch:

Definition 2.1.3 (Modularity [BME+07])
Modularity is the property of a system that has been decomposed into a set of ***loosely coupled*** *(changes should not affect other modules) and* ***cohesive*** *(contains only functions that logically belong together) modules.*

2.1.2 Software Quality Metrics for Extensibility

As presented, one key aspect of extensibility is the modularization of software. It can therefore be assumed that modularization correlates with extensibility if implemented correctly [JL09]. It can, however, also lead to the complete opposite direction for software projects where there is a high amount of inter-module dependencies [ADA15]. To counteract this phenomenon, the system should achieve *highly* or *extremely* loose coupling within its module. Additionally, high cohesion is very important to not fall into the trap, as slicing the system into low cohesive modules often results in excessive inter-module dependencies. For this reason, the definition of modularity given in Definition 2.1.3 on the preceding page has been chosen in such a way to not include the case of a "modular" but coupled and low cohesive systems.

Evaluating extensibility hence includes evaluating cohesion, coupling and complexity. The following part will introduce some notations and formal definitions that will be used in the evaluation. The definitions are mostly based on the [SQuaRe11] ISO standard. Furthermore, the definitions presumes the following notations:

- For $i \in \mathbb{N}$, a (Java) class is denoted as C_i.
- The set of methods in C_i is denoted as $\mathbf{Methods}(C_i)$.
- The set of methods of a class C_j that are called by any of the methods in C_i is denoted as $\mathbf{Response}(C_j, C_i)$.

Definition 2.1.4 (Dependence of Classes and Modules [ADA15])
A class C_i depends on a class C_j, $i \neq j$ (denoted as $C_i \rhd C_j$) if at least one of the following statements applies:

- *C_i calls a method in C_j, that is $\mathbf{Response}(C_i, C_j) \neq \emptyset$.*
- *C_i accesses a field of C_j.*
- *C_i inherits from C_j.*
- *C_i has a method that has a parameter of type C_j.*
- *C_i has a method whose return type is C_j.*
- *C_i has a method that can throw an exception of type C_j.*

Ultimately, if C_i is contained in a module M_i (denoted as $C_i \in M_i$) and $C_j \in M_j$ with $C_i \rhd C_j$, then M_i is dependent on M_j and the dependency $C_i \rhd C_j$ is called inter-module dependency. If, however, both $C_i, C_j \in M_i$, then the dependency is called intra-module dependency.

The definition of a class dependency above is important as many of the needed software quality metrics rely on the dependency factor. In fact, coupling of a class C_i can be loosely translated to the degree of dependencies in C_i. This can hurt extensibility especially if there are ample amount of inter-module dependencies. High degree of dependencies should therefore result in a tight coupling of a class:

Definition 2.1.5 (Coupling between Object Classes (CBO))
The Coupling between Object Classes (CBO) of a class C_i is defined to be the number of classes C_j that depend on C_i. This is synonymous to Afferent Coupling (Ca).

$$\mathbf{CBO}(C_i) = \left|\left\{C_j \middle| C_j \rhd C_i\right\}\right|$$

Definition 2.1.5 gives a hint of the so-called **ripple effect** that can occur when C_i is in some way changed. This means that any change in C_i can cause breaks in any class $C_j \rhd C_i$. Keeping the **CBO** low for every class $C_i \in M_i$ will help improving extensibility; however, this is not the only factor that needs attention. Moreover, it is important to know how many methods are invoked by C_i when it receives a message. This can determine the complexity and understandability of a class which in turn can also affect extendibility.

Definition 2.1.6 (Response for a class (RFC))
The Response for a class (RFC) is the number of methods in C_i and all methods in C_j that are called by C_i.

$$\mathbf{RFC}(C_i) = \left| \mathbf{Methods}(C_i) \cup \bigcup_{C_j; C_i \triangleright C_j} \mathbf{Response}(C_j, C_i) \right|$$

High RFC means high complexity of the class, from which then follows low maintainability. For instance, in such a case it is difficult to trace an error back to the origin. It therefore measures coupling between objects as well, as it highlights the communication between objects [CAIS14; CK94].

As noted previously, coherent methods and classes should be put together into one module, otherwise high inter-module dependencies could occur. For this reason, the notion of cohesion is needed. [SQuaRe11] defines it in terms of the lack of cohesion.

Definition 2.1.7 (Lack of Cohesion in Methods (LCOM))
The Lack of Cohesion in Methods (LCOM) is the number of methods that are not related to each other through the class fields. Two methods m_i and m_j are related (denoted as $\mathbf{Rel}(m_i, m_j)$) if there is a class field in C_i that both methods access.

$$\mathbf{LCOM}(C_i) = \left| \left\{ \{m_i, m_j\} \middle| \neg Rel(m_i, m_j) \right\} \right| - \left| \left\{ \{m_i, m_j\} \middle| Rel(m_i, m_j) \right\} \right|$$

The lower LCOM is, the more cohesive the class C_i is, which in turn can improve extensibility. Nevertheless, coupling and cohesion alone are not sufficient enough to measure modularity. Another important metric is the complexity of a class. A complexity of a method $C(m)$ where m is a method in C_i is typically defined as the *cyclomatic complexity* [McC76].

Definition 2.1.8 (Weighted Methods per Classes (WMC))
The Weighted Methods per Class (WMC) is the sum of (cyclomatic) complexities $C(m)$ for each method m in C_i.

$$\mathbf{WMC}(C_i) = \sum_{m \in \mathbf{Methods}(C_i)} C(m)$$

2.1.3 Modular Metrics

While the given metrics defined in the last sections can approximate the extensibility of a software, they can be misused in modular software systems. They measure quality attributes of classes and objects. A class can therefore be tightly coupled to other classes; however, this does not mean that the module is tightly coupled to other modules. Indeed, if all the tightly coupled classes are placed into a single module, then the module itself can be considered highly cohesive. If the module further defines *explicit extension points* which will be explained in Section 2.3 on page 12, it can encapsulate all the classes from the system.

In order to quantitively measure extensibility of a module to the outside world, it is rather necessary to consider inter-module dependencies and external coupling. If a module is too tightly coupled to other modules, or if the module has high afferent coupling, then the ripple effect can occur on change.

For this work, two quality attributes are used when it comes to modular softwares:

Definition 2.1.9 (External CBO [VGS+00])
An external CBO of a class C_i inside a module M is defined to be the number of classes coupled to C_i that are not placed within the same module M.

$$\mathbf{EXTCBO}(C_i) = \left| \left\{ C_j \middle| C_j \rhd C_i \land C_j \notin M \right\} \right|$$

The external CBO of a module M, however, is defined to be the sum of all external CBOs of its classes.

$$\mathbf{EXTCBO}(M) = \sum_{C_i \in M} \mathbf{EXTCBO}(C_i)$$

The external CBO measures the coupling from other modules to this module. It ignores the intra-module dependencies and inner couplings of classes. If another module M' extends the module M through one of its extension points, then $M' \rhd M$. It is therefore necessary that the classes in the module are as cohesive as possible. Unlike LCOM which gives cohesion of methods in one class, the module cohesion is required that considers the cohesion between classes.

Definition 2.1.10 (Component Cohesion [VGS+00])
The component or module cohesion of a module M measures the number of internal classes that are coupled. Here, it is normalized with the number of possible coupling relationships among the classes.

$$\mathbf{CC}(M) = \sum_{C_i \in M} \sum_{\substack{C_j \in M \setminus \{C_i\} \\ \mathbf{coupled}(C_i, C_j)}} \frac{1}{|M| \cdot (|M| - 1)}$$

where $\mathbf{coupled}(C_i, C_j)$ *is true if and only if* C_i *and* C_j *are in any way coupled, that is* $C_i \rhd C_j$ *or* $C_j \rhd C_i$.

2.2 Event-Driven Architecture

Systems built on EDAs are generally considered to be extremely loosely coupled [Mic06]. For high cohesive modules, EDA-based systems can therefore become highly extensible as presented in the previous section (Section 2.1 on page 5).

Figure 2.2 on the next page shows a general architecture of EDA. EDA-based systems consist of *Event Generators*, *Event Channels*, *Event Processor* and *Subscribers*. The Event Generator generates the events with necessary data. This will then be queued in the Event Channel. The Event Processor then fetches a single event from the queue and does some work based on that, i.e., printing debug information, informing that the event is about to be published, transforming the format of

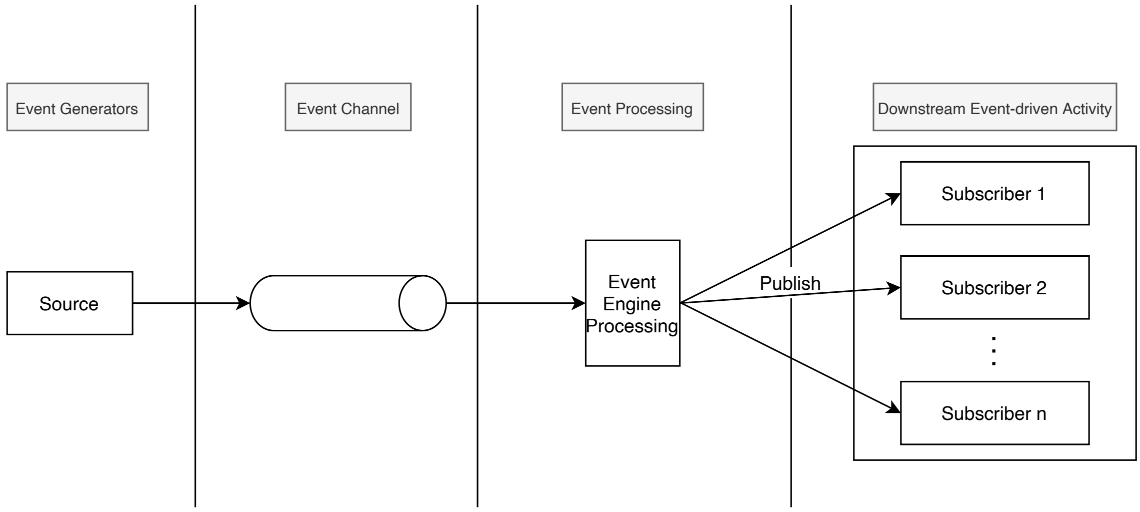


Figure 2.2: General architecture of EPC [Mic06]

event to another format understood by the subscribers etc. After all the processing is done, the processor publishes the events to the subscribers, and the subscribers then react to those events. A reaction is typically a generation of new events. A formal definition of event is given as follows:

Definition 2.2.1 (Event [BD10])
*An **event** can be everything that is happening or of which is expected to happen (i.e., an activity, a process, a decision). Generally, an event refers to a change of state, typically of an entity.*

As EDA communicates through events, the *call stack* found in popular request-response-schemes (for example, a method invocation in Java) is omitted [Hoh06]. This is due to the fact that there is no continuation and coordination of calls, as multiple subscribers just listen to a certain event and react to it, but generally do not know where this event originates from, and thus there is no direct invocation of the subscriber handlers from the originators. This leads to the omission of the call stack, which provides the *context* of the call. But without the context, the called methods cannot know on which values of variables or parameters they need to calculate. Therefore in EDA, the context must be given in the event itself. In this case, the event contains an entity that can serve as a context holder for the subscribers [Hoh06]. The disadvantage is therefore that the programmer must implement its own context holder or somehow simulate the call stack, which in fact weakens understandability of EDA-based software architectures.

Despite this disadvantage, EDA can enable high modularization of systems. Typically, there is a module `EventProcessor` $\in S$ that is responsible for processing the events. All other modules $M \in S$ are then publisher and subscriber at the same time: they generate and publish events to the processor, but also listen to other events. However, a “full” modularization cannot be given. In order for a new module M_{new} to work with S, it must somehow interact with it. While other modules do not have to have knowledge about M_{new} and its event definitions, M_{new} must listen to events happening in S, and must therefore depend on the existing modules. The best case is that the dependency graph is acyclic.

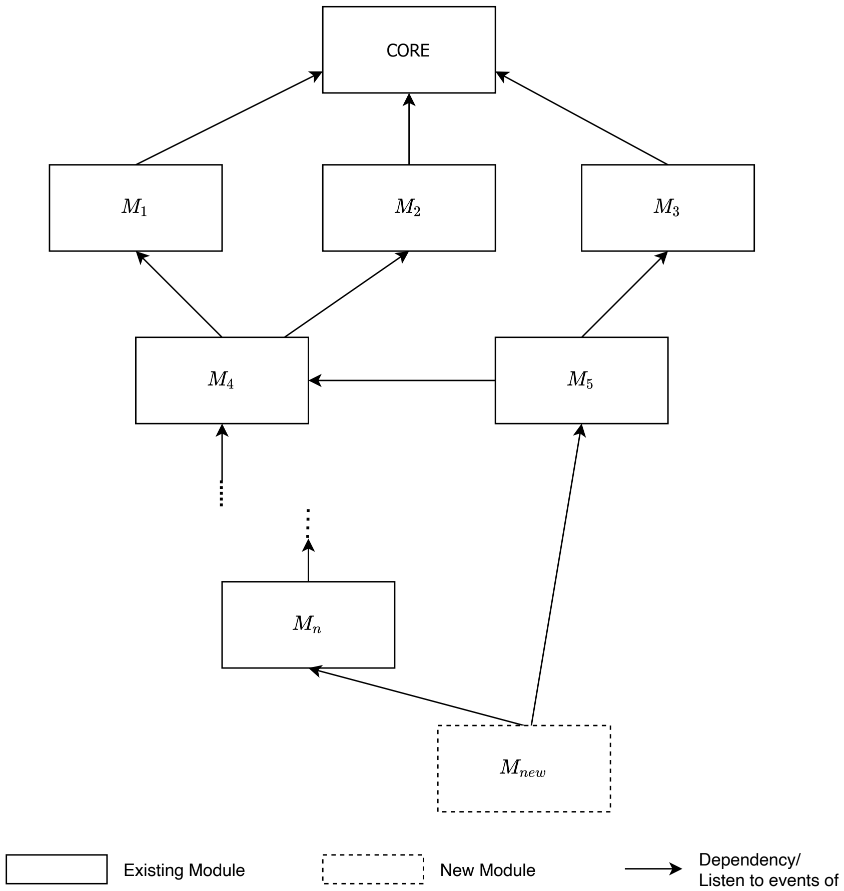


Figure 2.3: Example dependency graph of an EDA-based system S when adding a new module.

Figure 2.3 depicts a typical dependency graph of a general modular software system. This dependency graph also applies to EDA-based systems. As described above, each module must listen to the events that are already defined in the system in order to interact with it. The `CORE` $\in S$ hence defines pre-existing events, and the modules M_1, M_2, M_3 listen to at least one of these events. They further can define custom events, and M_4 and M_5 listen to them. At the end, M_{new} listens to the events defined in M_n and M_5, and may contribute further events to the system. In such a way, an extensible system can be built by using EDA.

In practice, there are different kinds and approaches of implementing EDA. One common and most used way is the **Publish/Subscribe** group communication paradigm [EFGK03; Par12]. It simplifies the architecture in such a way that there are only publisher and subscribers (to events). A module M subscribes to events when it listens to those events. On the other hand, the same module M can publish events at runtime, causing the invocation of the subscriber services.

In the case of Slingshot, EDA is going to be implemented not in a distributed system, but in an offline modular system. There is no real difference as to the communication of events, and such scenarios are already in use, i.e. in JavaFX. EDA in offline systems still exhibit an extensible and modular behavior [KPW16].

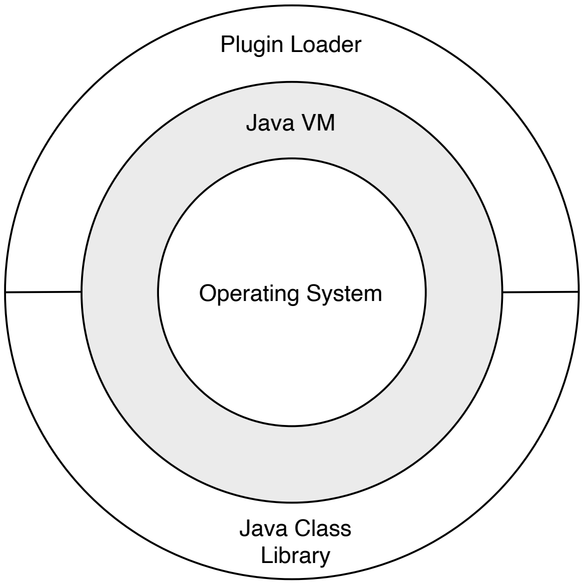


Figure 2.4: The general plugin architecture of eclipse [CR06]

2.3 Eclipse PDE & OSGi

Simulators for the PCM are most always Eclipse plugins. Eclipse in itself is based on the so-called *Plugin Architecture*, an extensible architecture that allows including new modules into the system. Because Eclipse is open-source, it is very typical for simulators to be built as an Eclipse plugin. Slingshot will also be built as an Eclipse plugin, and therefore it is necessary to know about its workings. In this section, the architecture of Eclipse as well as information and the importance of OSGi are presented.

2.3.1 Eclipse's Architecture

Generally, the Eclipse software is not monolithically built, but is actually just a kernel called a *Plugin Loader* as seen in Figure 2.4 [CR06]. The Plugin Loader resides with the Java Class Library or Class Loader. While the class loader is responsible for loading classes at runtime, the plugin loader imports all plugins and keeps track of so called **extension points** which define an interface. Plugins that want to extend Eclipse actually extend another plugins. The plugin defines its extension points and interfaces to which an extending plugin has to conform.

2.3.2 OSGi and Dependency Management

The modularity of Eclipse is given by a framework called Open Service Gateway Initiative (OSGi). The goal of OSGi is to make Java systems modular and supports the implementation of component-based, service-oriented Java applications. It achieves this by creating *bundles* that are generally JAR-files with cohesive classes and packages. The bundle definition has to follow the modularity definition given in Definition 2.1.3 on page 6 [OSGiMod]. In the case of Eclipse, OSGi bundles are also ordinary Eclipse plugin projects that will be later compiled to individual JAR files.

Listing 2.1 Sample `MANIFEST.MF` file for a plugin.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: somename
...
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime,
  ...
Export-Package: my.package,
 my.package.subpackge,
  ...
```

To achieve good modularization, each bundle must follow the encapsulation principle and hence hide as many class objects as possible. In OSGi the project must include a manifest file `META-INF/MANIFEST.MF` that contains necessary meta-data of the JAR file [OraJar]. Encapsulation can be achieved by exporting only the necessary classes to the outside which in turn is specified in the manifest file. Hence, OSGi defines headers of the specification which describe the dependencies and public packages of the bundle (definition are from [OSG18]):

- **`Require-Bundle`**: This imports all exported and public packages of the referenced bundle. This results in the direct dependency between two modules, and no information is given as to what packages exactly are needed.
- **`Import-Package`**: Unlike `Require-Bundle`, this header defines what packages are needed in order for bundle to work. It does not depend on a certain bundle anymore; instead, it can now be chosen from different bundles that export the packages, meaning that it can be chosen and substituted later.
- **`Export-Package`**: A list of defined packages within the bundle that can be accessed to the public. This does not include sub-packages; these must be given explicitly.

Listing 2.1 gives an example of a `MANIFEST.MF` file in a OSGi bundle. Aside from the common meta-data, it asserts that it requires the `org.eclipse.ui` and the `org.eclipse.core.runtime` bundles, meaning that every package these bundles export are automatically imported. It furthermore states that the bundle provides the packages `my.package` and `my.package.subpackage` to the public.

For OSGi to work, there must be a runtime implementation of it. Eclipse provides its own implementation of OSGi called *Equinox* [EquE]. Each Eclipse plugin is also an OSGi bundle, defining which other plugins and libraries are required in order to be used. In the realm of OSGi, bundles then need to be installed and started. Each bundle has therefore a state, as seen in Figure 2.5 on the following page. In Equinox, a plugin will be activated as soon as a class is loaded from that plugin, assuming the plugin is correctly installed before.

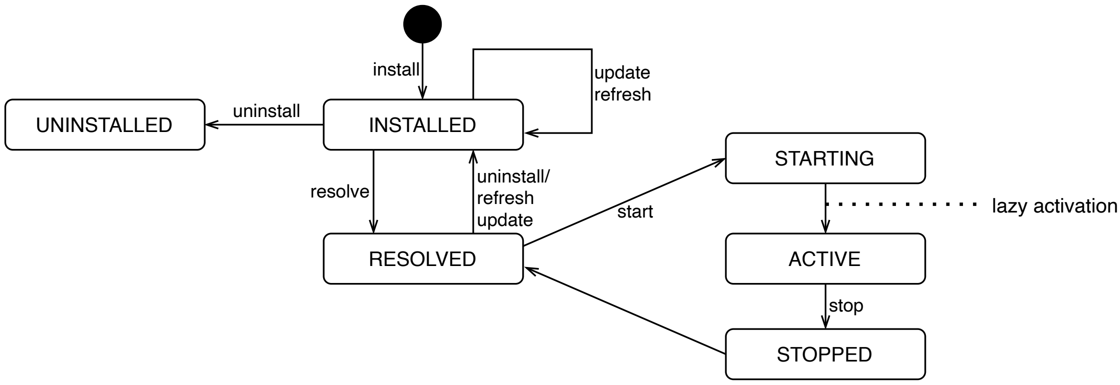


Figure 2.5: State Diagram of a OSGi Bundle [OSG18]

2.3.3 Plugins

While OSGi defines the modular dependencies of each bundle, this alone is not sufficient to create a pluggable architecture. Eclipse closes this gap by further providing *explicit extension points* onto which other modules can plug-in their services. Explicit extensions points are formal description or interfaces that the bundle defines and have the possibility to test them without the whole software in place [KK08]. Therefore, aside from describing the dependencies in manifest files, extension points can be provided by the `plugin.xml` file.

In the `plugin.xml` file, the provided extension point can be described by the `<extension-point />` element. This points to a custom XML Schema Definition (XSD) definition which are typically saved under the `schema/` root folder of the project. These files formally describe how the extension element of an extension bundle must conform to. The file can also describe elements stating that a Java class must be provided that implements a certain Java interface. As soon as such an `<extension-point />` element is given in the `plugin.xml` file, the bundle becomes extendable, and are therefore called *extendable bundles* in this thesis.

A bundle that wants to extend another bundle has to provide an `<extension />` element in its `plugin.xml` file with an attribute `point` containing the id of the extension point. The extension bundle must, of course, be dependent on the extendable bundle. These bundles are called *extension bundles* or simply *extensions* in this thesis. The inner XML content of `<extension />` must then conform to the XSD schema defined in the extension point.

An example is given in Listing 2.2 on the next page, where the plugin plugs itself on the behavior extension point of the Slingshot Behavior Module. It points to a base class implementing a Java interface that the Behavior Module needs. As soon as the Behavior Module plugin is activated, Equinox will search for every plugin that provides a extension definition at start.

Beside plugins, it is possible to define complete features in Eclipse. Features are logically a set of plugins, and a client can then install these features. Whole products can further require features instead of requiring plugins directly. Technically, features are independent Eclipse projects or plugins that does not provide Java code, but instead another `feature.xml` file describing the plugins needed. Nevertheless, features are out of scope of this work.

Listing 2.2 Sample `plugin.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
   <extension point="org.palladiosimulator.analyzer.slingshot.extensionpoint.behavior">
      <behavior-provider baseClass="org.palladiosimulator.analyzer.slingshot.behavior.
systemsimulation.SystemSimulationBehavior" />
      <behavior-provider baseClass="org.palladiosimulator.analyzer.slingshot.behavior.
systemsimulation.SeffSimulationBehavior" />
   </extension>
</plugin>
```

Listing 2.3 An example Java annotation definition with a method [RV11]

```
@Target(ElementType.TYPE) // can only be annotated on classes, interfaces, enums, etc.
@Retention(RetentionPolicy.RUNTIME) // retain annotation in the .class file, but will be
omitted at runtime.
public @interface Author {
    String name() default "[unassigned]";
}
```

2.4 Java Annotations and Processing

Slingshot will utilize Java annotations to further describe the implemented code. These are formal meta-data of the existing Java code providing further meaning without adding functionality to the code itself [GJSB05; RV11]. This section will introduce annotations, how they are defined and can be processed.

2.4.1 Definition

The language already provides pre-defined annotations such as `@Override` or `@Deprecated`, but in Java, custom annotations can further be defined by creating an `@interface` element. These elements are also annotated (meta-meta-information) that describes which Elements can be annotated by it (`Target`) and how long the annotation should be retained (`Retention`).

The example in Listing 2.3 describes a definition of annotation `@Author` that can only be annotated on type elements (such as classes, interfaces, enums, etc.) and will be retained at runtime. Annotations do not have fields, but only consists of parameterless methods. The return type of the methods must be either a primitive type or `String`, `Class<T>` or an array of one of the previous types, but must not be `void`. In this example, if the value is not explicitly given, then `name()` will return `"[unassigned]"`. A Java annotation can then be used like `@Author(name = "Joe")`.

As of now, there are three possible retention policies that could be used:

- **`SOURCE`**. The annotation is only retained at source level. After compilation, the annotation will be omitted in the `.class` files.

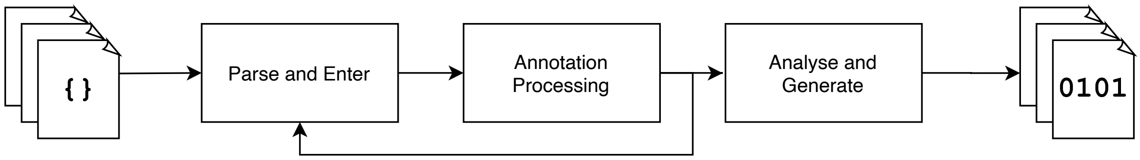


Figure 2.6: The compilation process with annotation processors [Ora20]

- `CLASS`**.** The annotation is present in the `.class` file, but will be ignored at runtime. This means that it is not possible to find the annotation using reflection.
- `RUNTIME`**.** The annotation is present at runtime and can be accessed using reflection.

2.4.2 Annotation Processors

While Java annotations are typically used at runtime using reflection, it is also possible to process Java annotations at compile-time. This is done by creating *Java Annotation Processors*, which are extensions to the Java compiler [Ann11]. This processor has access to the **source code** through the Model API [MirrorAPI][1] at the locations where the annotations are present and has the ability to give compiler errors or warnings. This ability is especially beneficial for IDEs, as the processor can give information about what and where the error lies so that the IDE can highlight the exact line of the error or warning. Another powerful ability of Annotation Processors is the generation of custom source files that in turn can be processed again.

Figure 2.6 depicts the flow of compilation where annotation processors are in place. The source code is first parsed and entered, the latter meaning that definitions are entered into the symbol tables, and outputs a `Tree` object representing the syntax tree of the source file. Next, the annotation processor is invoked. It can access the Tree through the Model API; however, the processor is not able to look into method implementations, as it is not possible to annotate elements within the method body. Each processing consists of multiple rounds. If the processor generates files, then this whole compilation process is done again. As soon as there are no rounds left, the compiler will finally compile the source trees into Java ByteCode if no error occurred before.

To create an annotation processor, it is necessary to define a new Java project. A processor is a class that implements the `Processor`[2] interface which is part of the [Ann11] library. However, it is advised to instead inherit from `AbstractProcessor` class, as certain implementation and fields are already defined there. First, it must be specified which annotations the processor supports, as well as the source version number, by overriding the `getSupportedAnnotationTypes()` and `getSupportedSourceVersion()`. The class then must implement the `boolean process(Set annotatedElements, RoundEnvironment round )` method. The compiler will call this method and provide the necessary parameters. Using the `Messager` instance, it is possible to throw errors, warnings or infos together with the exact location of occurrence. This instance can be retrieved by overriding the `init(ProcessingEnvironment processingEnv)` method, where the parameter provides the instance. After the logic has been executed,

[1] In earlier versions, this was called *Mirror API* instead.

[2] `javax.annotation.processing.Processor`

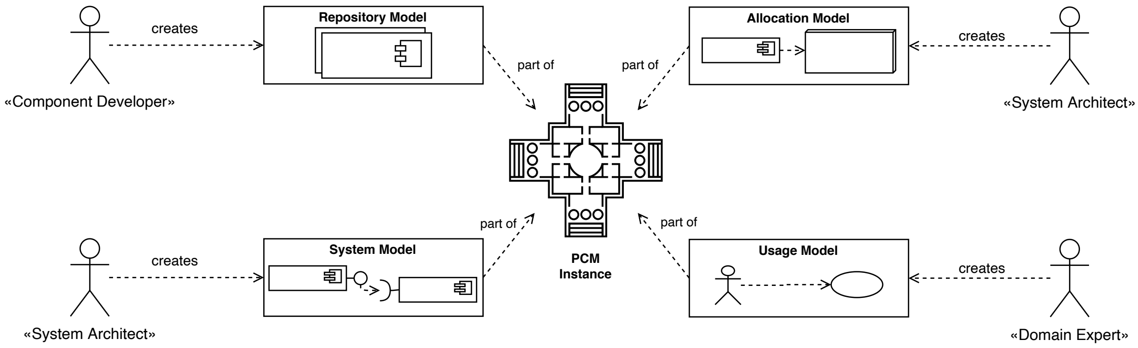


Figure 2.7: General Process of PCM [BKR09]

`process` then returns either `true` or `false` indicating whether the annotations are claimed. If they are claimed, they cannot be processed by other processors anymore. As already stated, if new source files where generated during the process, then a new processing round is started.

After the necessary class has been defined, it must be registered as a service for the compiler to find. This is done by creating a `META-INF/services` directory at the root of the project. In this directory, the file `javax.annotation.processing.Processor` file is created. Each line of this file is a fully qualified name of the defined processor.

2.4.3 Limitations

While annotations can provide further information of the code, it is not possible to annotate elements within a method body. Consequently, annotation processors are not able to look inside method bodies.

2.5 Palladio Component Model

The PCM is a model for component-based software systems and deals with predicting quality attributes such as performance and reliability [BKR09]. PCM itself is a meta-model providing different views and view points of the component-based software system. The meta-model therefore describes the process of CBSE by [KB98]. In CBSE there are four different developer roles for the development which PCM uses: The *Component Developer*, the *System Architect*, the *System Deployer* and the *Domain Expert* [KH06]. The general process is depicted in Figure 2.7. This section briefly discusses the process of CBSE that PCM assumes, and the different view points are presented.

2.5.1 Component Developer / Repository and Service-Effect Specifications

The component developer creates the components of the systems, including the requiring and providing interfaces. The components are modeled in the **Repository Model** and are UML-like component models with explicit interface definitions; however, these do not include relationships between the components yet. Each component is associated with certain roles connecting it to the interfaces (`RequiredRole` and `ProvidedRole`). An interface consists of multiple Java-like method definitions; however, these are not directly implemented.

Instead, the component developer specifies so-called **Service-Effect Specifications** (SEFF), specifically the Resource Demanding Service-Effect Specification (RDSEFF), that represent the "grey box view of the component" [BKR09], describing the externally visible behavior of a interface method provided by the component, for each component. They are comparable to UML Activity Diagrams and have the ability to simulate loops, branches, internal computations in the sense of resource demands, and external method calls from other interface. RDSEFFs are explicitly defined in components, as each component exhibit different behaviors and implementations for a certain interface. Therefore, RDSEFFs are used for performance predictions specifically.

2.5.2 System Architect / System Model

The system architect receives the distinct components from the component developer and architects them together into one system. This is done by connecting the corresponding requiring and providing roles of each component. The system itself is also an entity that can provide and require certain interfaces. In fact, the provided interface of the system itself is typically used as the entry to the system and thus to start the application. At the end, the system model is akin to the UML component model.

Nevertheless, the system model does not define further interfaces. Instead, when the system provides an interface, it delegates it from one of the inner components. To the outside world, it is not visible which component actually provides the interface.

2.5.3 System Deployer / Resource Model and Allocation

The system deployer is response for creating the allocation context of the system. In the resource model, the deployer first defines different resource contexts which can be connected together. A connection between two resource contexts semantically means that both can communicate with each other in some way. In PCM only ethernet connections can be used as of now. A resource context further specifies its capabilities and hardware attributes, such as CPU frequency, hard-drive size, etc. Resource models are hence related to UML Deployment Models.

After such resource model is created, the system deployer then assigns each component to a resource context. A call to a provided service of a component is then simulated using the hardware specifications of the corresponding resource context. The assignment defines the allocation context.

2.5.4 Domain Expert / Usage Model

The domain expert typically is not a technical expert. They do not have to understand the inner workings of the system, which highly resembles the reality of users. However, the domain expert knows about the possible usage of the system and has the users in mind. They create a so-called usage model that are like RDSEFFs. The difference, however, is that the usage model can only call services provided by the whole system itself; that is called `EntryLevelSystemCall`. But nonetheless, branches, loops and sequences can be used as in RDSEFFs.

Aside from the usage itself, the domain expert also has knowledge about the users. They therefore provide the workload of the usage model resembling the users. The `ClosedWorkload` means that there are a fixed number of users with a certain think time using the system. The think time specifies the time they need before starting the usage flow. The `OpenWorkload` on the other hand specifies an inter-arrival time which describes the time a new user appears using the system. In this case, the (fixed) number of users is not known in advance.

2.5.5 Parametrization and Stochastic Expressions

Many elements in the models are parameterized, such as methods, workloads, resource specifications, etc. For the simulation, their values are always stochastic expressions (`StoEx`) matching random variables as known in the stochastic theory. PCM provides a grammar for these expressions, but are theoretically specified by using Probability Density Functions for continuous random variables or Probability Mass Functions for discrete random variables. A simulator of PCM will evaluate those expressions at runtime and randomly.

The `StoEx` can use variables and parameters, such as parameters of a method or the input values of the user. The call to a function can be seen as an ordinary call in Java. In order to use the parameters and to refer to the correct variable, PCM mimics the behavior of a call stack. Each user get its own stack, and in these stacks so-called stack frames are saved which contain the current and correct values of the variables for the specific user. Hence, the call stack is needed to distinguish the values of each user.

2.6 Simulation World Views

Simulators are typically built on the notion of events that happen in the model. The events in this case have to be distinguished to the events in EDA introduced in Section 2.2 on page 9: Here, events are model elements that correspond to an event in the real-world system. But, as in EDA, these often indicate a change of state in the system itself or of an entity. This section is based on the work on [Car93].

In the case of PCM, the simulators are based on *discrete-event simulation models*. Discrete corresponds to the advancement of time within simulation: Each event in the system is on a discrete time line and therefore advances to fixed times. The simulator moves the current time to the next event occurring in the model. As of now, there are two different approaches of advancing times in the discrete-event simulation that the modeler adheres to and are called world-views: The *process-interaction world view* and the *event-oriented world view*. The distinction between them is necessary as other works have shown that there is a clear performance trade-off when adhering the process-interaction world view instead of the event-oriented world view [BCNN10; Mer11].

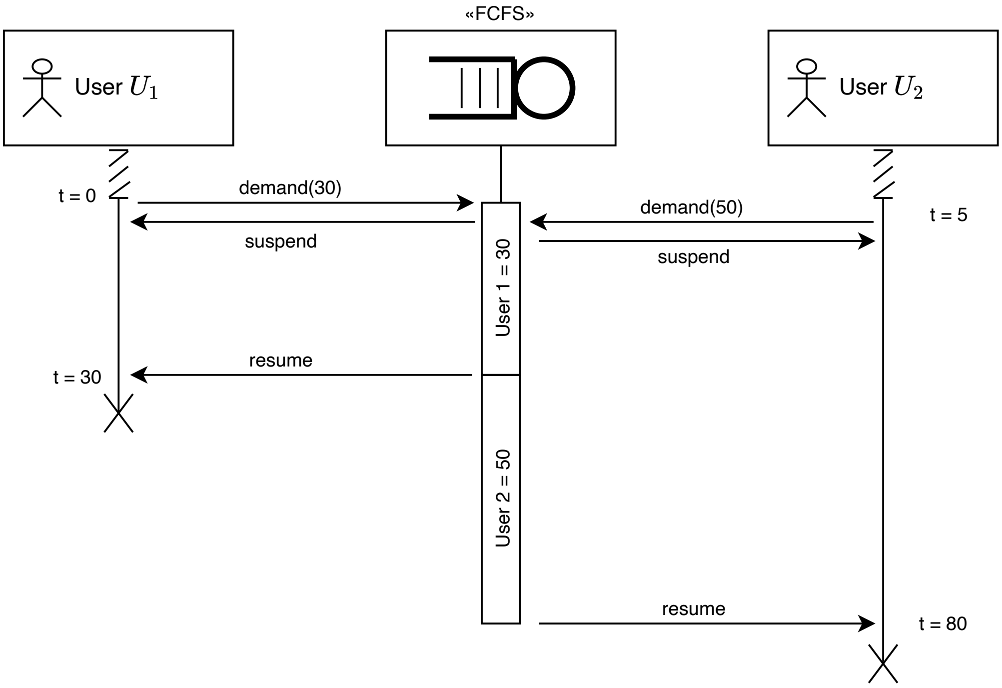


Figure 2.8: Example simulation adhering to Process-interaction world view. Taken from [Mer11]

2.6.1 Process-Interaction World View

When simulating a model by adhering to the process-interaction world view, the simulator bases the model on the dynamic entities, their actions and their overall behavior. These processes describe the interaction with other entities. The processes are therefore modeled using designated process threads, which are executed at runtime. When a simulation time advance is required, the threads are suspended until the simulation time has passed.

The example in Figure 2.8 by [Mer11] gives a clear view of process-interaction world views. In here, two users request a demand a certain resource in a First-Come, First-Served (FCFS) manner. The user U_1 demands 30 Processing Units (PU) which in the example correspond to 30 simulation time units. The execution of U_1 is then suspended until the resource demand has been served. At the meantime, U_2 demands 50 PU. However, because the the resource hasn't been fully served yet to U_1, U_2 must first wait until its serving time-slot is given. Nevertheless, U_2 is suspended as soon as the request arrives, but must wait 75 simulation time units in total before it can continue execution.

Because each process must be modeled and executed distinguishably, they must somehow be run concurrently. In Java, this would be most intuitively be done using Java threads. This means that every process is mapped to a corresponding Java thread. As already said above, this can lead to performance trade-offs, especially when large systems with many processes and entities are simulated [BCNN10].

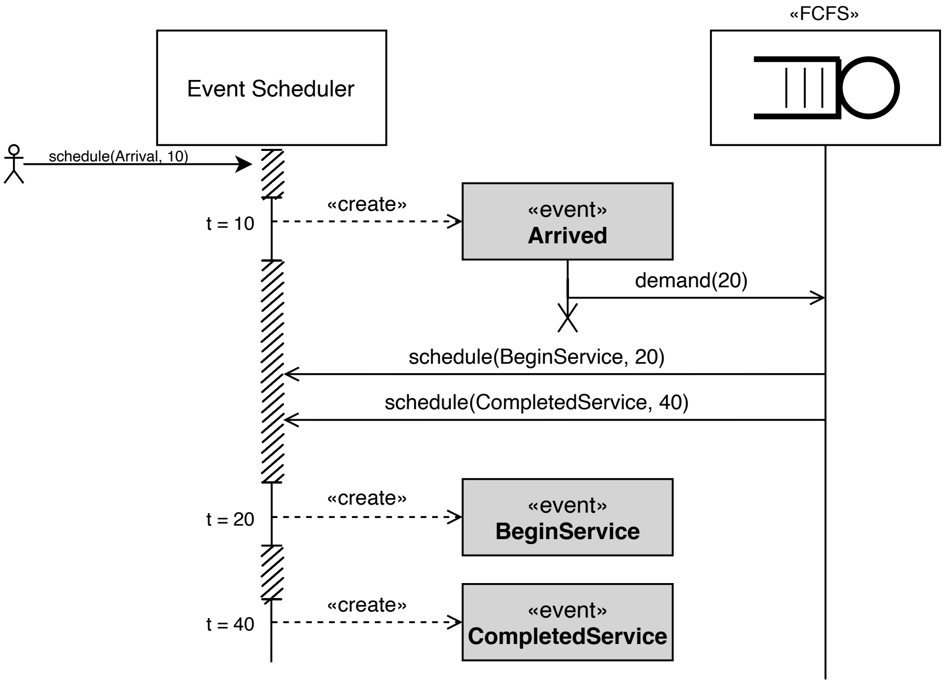


Figure 2.9: Example simulation adhering to the Event-Oriented World view. Taken from [Mer11]

2.6.2 Event-Oriented World View

In the event-oriented world view, the simulator is interesting in the events that happen in the system rather than the processes. The simulation time advances directly to the next event that is scheduled in the system. The modeler therefore focuses on the events that occur in the model, but does not model the interaction between the entities. Such an event can be the change of state of an entity or the system as a whole. Because only events are considered, there is no need for multiple threads.

The example in Figure 2.9 again by [Mer11] shows the difference that happens when using the event-oriented world view. Here, a user arrives to the system at time $t = 10$. This will be scheduled as an event `Arrived`, which in turn requires a demand of 20 PU. The handler schedules two events: `BeginService` at the time $t = 20$ and the event `CompletedService` at $t = 40$, being 20 simulation time units apart. This emphasizes that no process is here suspended or run 20 time units, but that really only events are being put on the time line.

2.7 Related Work

As of now, there are some simulators already present for the PCM. As of now, none of the current simulators are event-driven, but are typically monolithic and modular. This work focuses on the two currently most used simulators that will also be compared against in terms of extensibility.

2.7.1 SimuLizar

The SimuLizar by [Mey11] is a PCM simulator that focuses on simulating self-adaptive systems and is based on model-driven scaleability analysis. The modeler can model the system and conditional reconfiguration strategies. SimuLizar then analyses the models at runtime, and finally returns the results of performance, efficiency and scalability of the system. SimuLizar extends the PCM by introducing further model elements for these reconfiguration strategies.

SimuLizar is an *interpretive* simulator, meaning that the models are accessed at runtime. The contrast would be the *generative* approach where the models are first transformed into a simulation code using Model-2-Text transformations. Furthermore, SimuLizar adheres to the *process-interaction* world view and extends and substitutes certain functionalities of the SimuCom simulator created by [Bec08].

2.7.2 EventSim

The EventSim by [Mer11] is an *event-oriented* simulator for PCM. The system implementation itself is, however, still monolithic, but tries to be extensible [SMRL13]. As given in the examples of Figure 2.9 on the preceding page and Figure 2.8 on page 20, they try to highlight the performance difference of the simulation strategy between the event-oriented and process-interaction world view, and prove that the event-oriented world view typically runs better and is also more scalable. The goal of this simulator is therefore to be modular and scalable.

The modularity of EventSim is achieved by exhaustively defining as many extension points as possible to the system. One example is that there is an extension point for the RDSeff interpretation that can use different traversal strategies, which seem to affect the simulation results [SMRL13].

As already mentioned, in this work the notion of *event-driven* and *event-oriented* must be differentiated. Event-driven refers to the architecture that is used, namely EDA. Event-oriented is the world view that the model and simulator adheres to. EventSim is event-oriented, but not event-driven.

3 Slingshot Simulator

Now that the foundations are given, the Slingshot simulator can be built. The main goal of Slingshot is to be extensible by adhering to the Event-Driven Architecture and using annotations to give further information about the extensions. The system is therefore designed to be modular from the ground up. In retrospect to related works, the key difference between Slingshot and the other simulators are given in Table 3.1.

Property	SimuLizar	EventSim	Slingshot
Approach	Interpretive	Interpretive	Interpretive
World View	Process Interaction	Event Oriented	Event Oriented
Architecture	Monolithic	Monolithic	**Event-Driven**

Table 3.1: Table showing the differences between the most used simulators for Palladio Component Model.

3.1 Core Architecture

The Slingshot Simulator consists of different modules that are implemented as the `CORE`-Bundle in Eclipse. This extendible bundle defines the core behavior of the system, such as the event handling mechanism, and provides the extension point for behavior extensions.

Figure 3.1 on the next page depicts the core architecture. The main driver of is the `SlingshotDriver` component that acts as a middleware for the events. It loads the behavior extensions through the `BehaviorSystem`-component and forwards all published events to the `SimulationEngine`. This in turn is responsible for the simulation of events. It serves as an interface translating the Slingshot definition of events to the simulation definition of events by [Car93], and hence depends on a library that can simulated event-oriented world views. In this case, Slingshot will be based on *SSJ*, a framework for stochastic simulation in Java by [LMV02]. The `SimulationEngine` then forwards the events to the `EventSystem`-component which is responsible for internal dispatching of Slingshot events. This again, uses a library that implements an EventBus with an appropriate order and publish-subscribe system. In this case, Slingshot uses the *Google Guava* library that includes such EventBus [GooGua].

As seen in Figure 3.1 on the following page, the `BehaviorSystem`-component is extensible. It defines the extension point `behavior`[1]. As described in Section 2.3 on page 12 each extension to the system will provide a class that implements a certain interface, in this case the `SimulationBehaviorExtension`

[1] The actual name is a fully qualified name, in this case `org.palladiosimulator.analyzer.slingshot.extensionpoints.behavior`

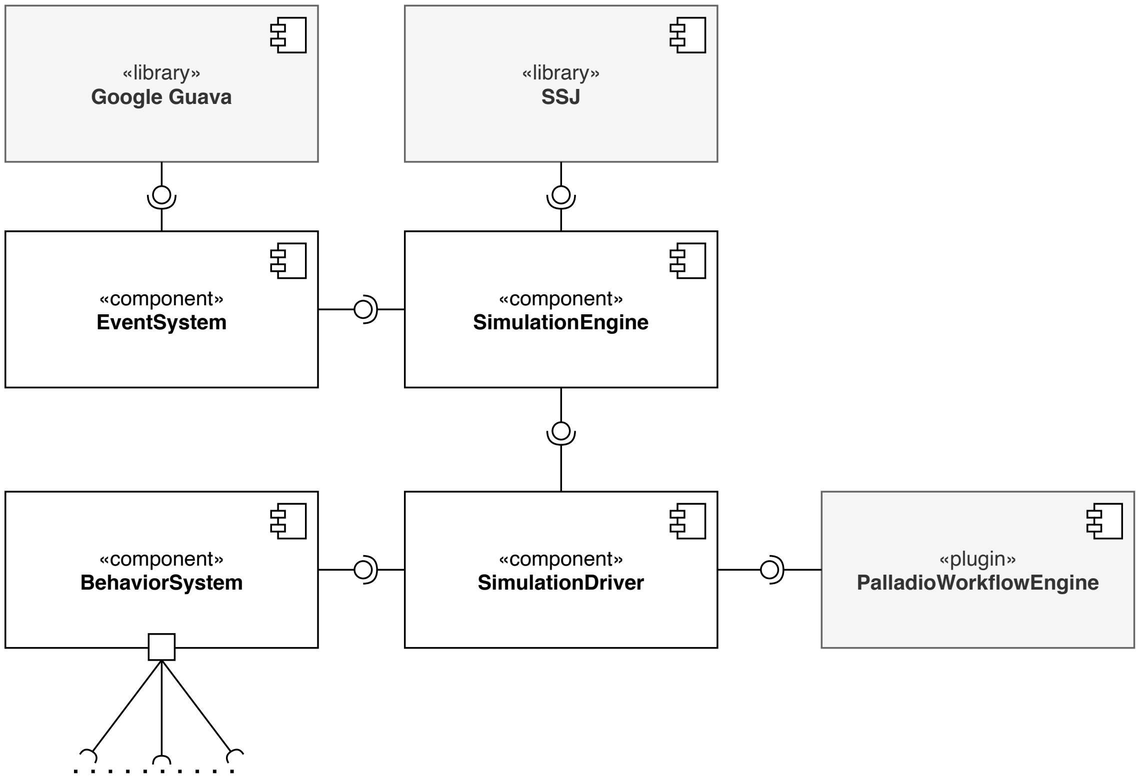


Figure 3.1: Logical view of Slingshot's core architecture in a UML component diagram. Gray boxes are re-used libraries.

interface. The `BehaviorSystem` then loads all extensions using the Eclipse plugin system, and forwards all the classes to the `SimuationDriver`. Hence, its main function is to provide the extension point and an extension loader.

3.2 Event-Driven-Architecture and Events in Slingshot

A *simulation event* is an event as defined in Definition 2.2.1 on page 10 that also provides meta-data necessary for the simulator. One such meta-data is the *delay* in time units. The delay is used in order to increment the overall simulated time. This section will discuss the event definition as well as contract definition.

3.2.1 Event Definition

Slingshot hence requires an interface `DESEvent`[2] that each defined event has to comply. The current required meta-data are, as stated, the *delay*, the *id*, but also the current *simulation time*. Listing 3.1 on the next page describes the necessary Java interface.

[2]This stands for *discrete event-based simulation event*.

Listing 3.1 The `DESEvent`-Interface for defining custom events.

```
interface DESEvent {
    String getId();
    Double getDelay();
    Double getSimulationTime();
}
```

Listing 3.2 The method signature of event handlers in Slingshot.

```
@OnEvent(when = SampleEventB.class, then = SampleEventA.class, cardinality = SINGLE)
public class ExtensionBehavior implements SimulationBehaviorExtension {
    @Subscribe
    @EventHandler // only needed if method name does not start with "on"
    public ResultEvent<SampleEventA> onSampleEventB(final SampleEventB sampleEventB) {
        // ...
    }
}
```

Events generally describe a (state) change of an entity. The event carries a reference to the entity and can also specify which information has changed. Such entities could be, for example, an interpretation context holding information needed for interpreting PCM model elements [BBM13]. Especially context holder is needed as, by using EDA, it is not possible anymore to rely on the call stack, and must therefore be simulated by such context holders.

Each extension in Slingshot act as the publisher and subscriber at the same time. They listen to single events, process them and respond with new events by providing so-called *event handlers*. An event handler is an operation with the event as the single parameter and a return type of a collection of events that should be published. The class that holds these instance methods are called *event handler classes*. An example of such event handler is given in Listing 3.2. Here, the event method listens to the `SampleEventB` event. Each method must return a custom defined collection called `ResultEvent<T>`, where `T` is a (sub-) type of `DESEvent`. It holds an ordered set of events of type `T` and provides further operations. Because Slingshot uses the Google Guava library which is further described in Section 3.1 on page 23, event handlers must be annotated with Guava's `@Subscribe` annotation. Also, event handler names should always start with `on` followed the event name. If the name does not start with `on`, it can instead be annotated with `@EventHandler` which is needed by the Slingshot system itself.

3.2.2 Contracts

The event handler classes define and document certain contracts for the handlers. Contracts describe which events will be published by another event. They therefore draw the event graph as seen in Figure 3.5 on page 30 without depicting the handlers. Each event that is published must be given. If an event is published without mentioning it in the contract, the simulation will be immediately

Listing 3.3 An event contract describing that the `SampleEventA` can only be published three times in total in the whole runtime, as well as that only event handlers handling `SampleEventB` and its subtypes are allowed to publish this event.

```
@EventContract(maximumPublishing = 3, causedBy = {SampleEventB.class})
public final class SampleEventA extends AbstractEvent {...}
```

stopped. Conversely, if there is an event handler for a certain event, which is not mentioned in the contract, a compiler error will be thrown. In order for the contract to be checked at compile time, an Annotation Processor is defined as described in Section 2.4 on page 15.

The contracts are described at class-level with the annotation `@OnEvent(when, then, cardinality)`. The `when` argument requires a class reference to the event onto which is being listened by one or more event handlers. The `then` argument should provide an array of class references to events that **might** be published by these event handlers. The cardinality then provides whether a set with a `SINGLE` event is returned, or whether `MANY` events are returned. The `SINGLE` cardinality is more strict and will throw an exception when a set with more than one event is returned. The `MANY` cardinality subsumes single sets and will hence never throw an exception.

Another contract type that Slingshot offers is `@EventContract`. Unlike `@OnEvent`, this annotation describes certain limitations on the events themselves that can be published, and are furthermore optional. An example is given in Listing 3.3. The `maximumPublishing` parameter is an non-negative integer describing how often the event can be published. If this parameter is set to `EventContract.PUBLISH_INFINITY` or 0 (default), then there is no limitation. The `causedBy` parameter is an array of types that describes which event handlers are allowed to publish this event. If the default value `DESEvent.class` is set, then all event handlers can publish this event. If it is an empty array, then no event handler is allowed to publish this event, with the exception of the internal handlers in the core.

3.2.3 Pre-defined Events

As of now, Slingshot defines three pre-defined events that cannot be published by any custom event handler and are allowed to be published only once.

`SimulationStarted`. This is the first event that will be published upon simulation start. In order for an extension to work, there must be a path in the event flow from `SimulationStarted` to the events that are handled in the extension. This event is therefore the root of the graph.

`SimulationFinished`. Upon ending, when no event is published anymore from any handler, this event will be published lastly. It forces the system to not listen to any other event anymore. Event handlers can listen to this event for finishing up some work, such as closing files or flushing buffers.

`SimulationInterrupted`. If there is any error in the system, such as a contract is violated or the user requests an interruption, then this event will be published. It forces the system to not listen to any further events. This is a subtype of `SimulationFinished` and will therefore fire up those event handlers, too. The difference here is that this event is published indicating an error, while `SimulationFinished` indicates a success.

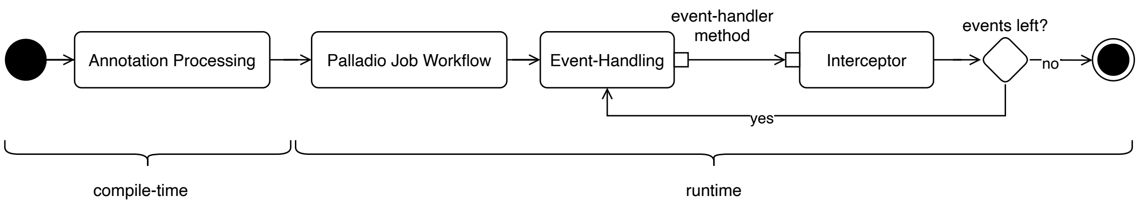


Figure 3.2: UML Activity Diagram presenting the logical workflow of Slingshot.

3.3 Workflow

The Slingshot system underlies a certain order of execution. The starting point is defined in the `Workflow`-module which, according to Section 3.1 on page 23, is dependent on the workflow plugin provided by Palladio. Figure 3.2 shows a general view of how Slingshot works. It also includes the compilation step of a extension module. At compile-time, the annotation processor is started and checked that the contract and annotations are valid. After compilation and at runtime, certain Palladio jobs are performed. Then, for each event handler, the events are being monitored, and the handler are intercepted with further functionality that is required for contract checking. This process will be repeated until no events are published anymore, and the simulation stops.

In this section, each of these steps will be further discussed and presented.

3.3.1 Annotation Processing

When a extension module is being developed and event handler classes are being defined, the annotations describing the contract of the event flow can be checked at compile-time. With the features and limitations described in Section 2.4 on page 15, a single Eclipse plugin were developed as `org.palladiosimulator.analyzer.slingshot.annotationprocessors` which defines following Annotation Processor Services:

`org.palladiosimulator.analyzer.slingshot.annotationprocessors.OnEventProcessor`

This service processes `@OnEvent` annotations at compile-time. It checks whether the following conditions are fulfilled. If one of them is broken, it will throw a compiler error and stop compilation, or will print a compiler warning and continue compilation.

Error: `WhenIsNotDESEvent`. Each `@OnEvent` contract definition must not restrict output events on `DESEvent` which is the most general event. This is due to the fact that an event handler should listen to the most concrete event if possible. A contract's `when` definition does not describe its subtypes; it has to be the exact same type as the event handled by one or more event handlers in that class.

Warning: `ThenIsNotDESEvent`. A contract definition should restrict to the events that will be published by an event handler. Setting the possible output events to `DESEvent` will include every defined event which is not the purpose of the definition. This will, however, not result in a compilation error but in a compiler warning, as on development the events might still not be known, or because it is simply the purpose of the developer to allow every event as a result.

Error: `ContractDefinitionNotDefined`. When a class contains event handlers and at least one event contract, the processor will check that for each event handler there is a appropriate contract. The contract relates to an event handler if the contract's `when` value and the parameter type of the handler are equivalent. If no contract definition is found, the compiler will result in an error.

Warning: `EventHandlerNotDefined`. When there is a contract definition with no corresponding event handler, then a compiler warning will be printed. However, there should not be any effect at runtime as the event defined in the contract is simply never handled.

Error: `ThenTypesNotCompatible`. Each event handler returns a `ResultEvent<T>`, which is a generic class containing an ordered set of type `T`. If there is a type defined in the `then`-clause that is not compatible with `T`, then the compilation will stop, as that event will never occur in the set. This also includes the case if `T` is not compatible with any of the event types defined in `then`-clause.

`org.palladiosimulator.analyzer.slingshot.annotationprocessors.EventContractProcessor`

This service checks the `@EventContract` annotations which define event-level contracts. As in the previous service, it can stop compilation or simply print a warning.

Error: `AnnotationNotOnEventType`. This error will be thrown at compile-time when the annotation is placed on a class (or generally a type) that is not an event type. This is done by checking whether the annotated class or interface is a subtype of `DESEvent`.

Error: `MaximumPublishingIsNegative`. The `maximumPublishing` parameter of the annotation must be a positive number or 0. If this is not the case, then the compilation will stop.

Limitations

Both services will not generate new files, and therefore a new round is not needed. If no error is thrown and the compilation is not stopped, the annotation processor can successfully end. It is important that due to the limitations of annotation processors not all checks can be done at compile-time. As the processor is not able to see the implementation of methods, it won't be able to get the return value and check that the right events are returned. This check can only be done at runtime.

3.3.2 Jobs

The whole system itself is an Eclipse plugin as described in Section 2.3 on page 12. In Figure 3.1 on page 24, this is depicted as the `EclipsePluginSystem`-component, which is not part of Slingshot itself, but is provided by the existing Palladio Workflow Engine of [PWE14]. It solely depends on `SimulationDriver`. The workflow system defines different kind of jobs that the new PCM plugin can provide which the workflow system then executes. As seen in Figure 3.3 on the facing page, there are two kinds of jobs currently defined: The `LoadModelsJob` that loads the defined models into the system, and the `StartSimulationJob` that starts the simulation with the provided models. These jobs are run sequentially.

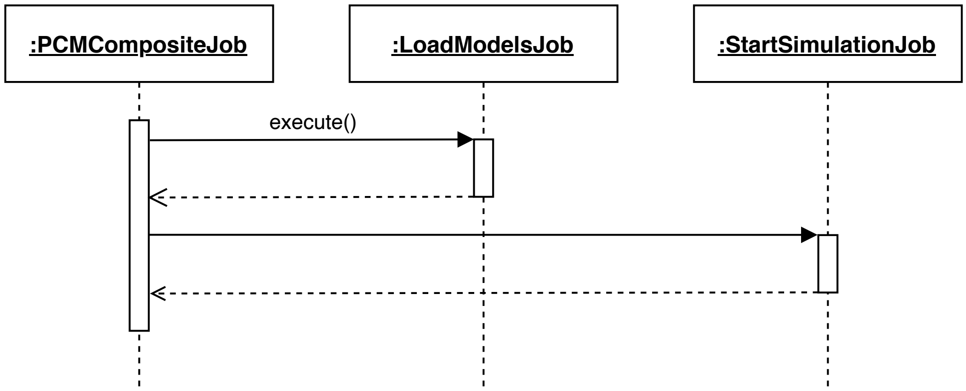


Figure 3.3: UML Sequence Diagram of PCM Workflow Jobs in the case of Slingshot.

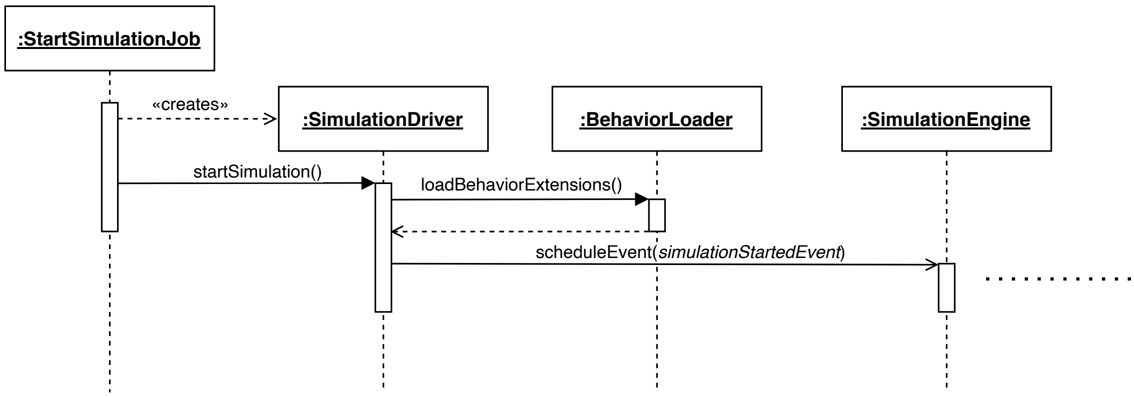


Figure 3.4: UML Sequence Diagram of the `StartSimulationJob` performing the start of simulation.

`StartSimulationJob` creates a new instance of the `SimulationDriver` and starts the simulation. By starting the simulation, the driver first loads all the extensions. Afterwards, the extensions will be modified in a certain way such that they can listen to events and can be monitored. Each extension will be instantiated and registered in the event system as subscribers. Lastly, the driver calls the engine and schedules a special event `SimulationStarted`. Each extension can listen to this event to perform certain actions right at the beginning. When no action is done after all events are consumed, the driver will once again schedule an event `SimulationStopped` so that the extensions can perform last operations. This whole process is depicted in Figure 3.4.

3.3.3 Event-flow

The important aspect of Slingshot is the event flow. As soon as the `SimulationStarted` event is published, all corresponding event handlers are activated sequentially. The event handlers, however, can subsequently publish further events. Nonetheless, the published events will be handled after all event handlers listening to `SimulationStarted` are first finished. The event bus is therefore a FIFO-queue. It only contains information about the events, but does not track which event methods

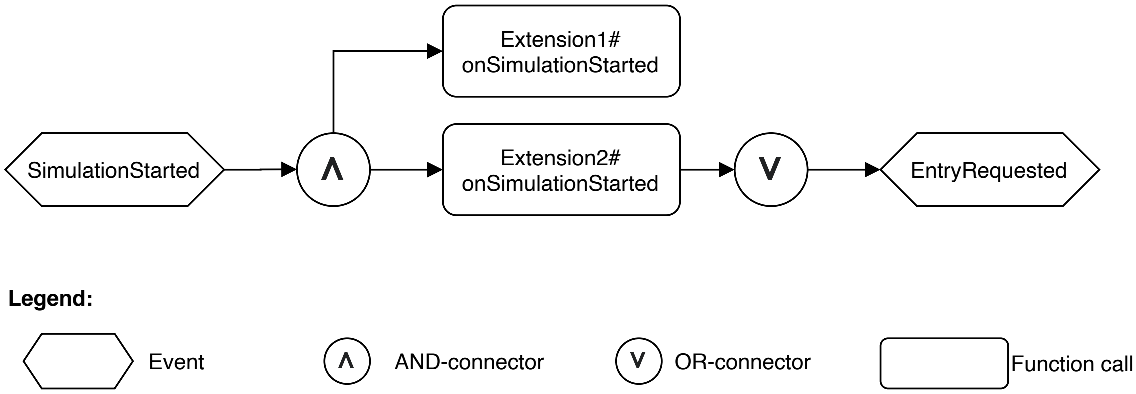


Figure 3.5: An example EPC depict the flow of events and interaction with the handlers as in [Men08]

are handled. This means that the order of event handlers within a same event is unknown. However, this is part of the EDA's nature. Especially because the extensions do not communicate within a call stack or depend directly on each other, the order of method calling must not be considered.

The event flow can be depicted using Event-driven Process Chain (EPC). Figure 3.5 gives an example of a EPC flow after the `SimulationStarted` event has been published. EPCs depict the interaction between processes or methods and events, which are connected with logical connectors. Here, the logical AND-connector specifies that the input event will call all output functions. In the case of Slingshot, it is important to note that the functions are not called concurrently, unlike AND-connectors in the specification of EPC. The OR-connector says that the input function will call one or more of the output events. The EPC graphs used in this work comply to the specification and descriptions by [Men08].

In case of Slingshot, the current implementation only allows for a single input in both logical connectors. Also, the input of AND-connectors are always events, the output are always event handlers; and for OR-connectors this is true for the other way around. Thus, event handlers can only listen to one event, and subsequently, all events returned by those handlers are published.

The event flow, however, highly depends on the interceptors which can manipulate the flow, although in a limited way, or monitor the events. The interceptors will be discussed next.

3.3.4 Interceptors

The interceptors are a part of the `SimulationDriver` module that intercepts each event handler method of the event handler classes. An interception can be done before or after method invocation. Technically, this is done by extending the class and overriding the event methods. All pre-interception method are then done first, then the super method is called (that is the actual method invocation), and lastly the post-interception logic is executed. This process is done at runtime. Because the class has to be extended, the event handler methods as well as the event handler class must not be final.

As of now, Slingshot defines four interceptors by implementing an interface `Interceptor` which in turn defines two methods `preIntercept()` and `postIntercept()`. The interception mechanism is not extendable for security reasons, and is therefore included in the driver and not separated as an independent component.

ExtensionLoggingInterceptor. This interceptor is purely for logging purposes and will inform which extension and which event handler is called.

SimulationExtensionOnEventContractEnforcementInterceptor. This will enforce the contract at runtime for each event handler. Unlike the compile-time check of the contract, this can check that the returned events are adhering to the contract of each event, that is, the returned events are listed in the `then`-parameter, and the cardinality fits to the cardinality of the returned set. If this is not the case, a `SimulationInterrupted` event will be published, and no further events can be published afterwards.

EventMonitoringInterceptor. This tracks the event flow and draws an event graph similar to EPC. It post-intercepts each event handler method and looks at the parameter value of the method as well as the return value. The event graph is saved on memory and will be exported after the simulation is done. The generated event graph can also be used for debugging purposes.

SchedulingInterceptor. This interceptor runs after all other interceptors are done, and schedules and publishes the returned events of the event handlers.

4 Simulation Implementation

Aside from building a extendable foundation, the current work of Slingshot already includes a prototypical implementation of the simulation behavior. PCM defines different view points of the modeling software, especially the behavioral or dynamic view point (*UsageModel*) as well as a static view point (*System*, *ResourceModel*, *AllocationModel*) [BKR09; RBB+11]. In this work, the following extensions were implemented:

- **`UsageSimulation`.** Interpreting the usage simulation. This is generally the start of the whole simulation.
- **`SystemSimulation`.** Interpreting and accessing information of the RepositoryModel and interpreting each RDSEFFs.
- **`ResourceSimulation`.** Accessing the ResourceModel and Allocation Contexts and simulating the resource containers as well as demands.
- **`GeneralSystemConfigurationBehavior`.** Configuring simulation instances like initializing the StoEx-Cache.

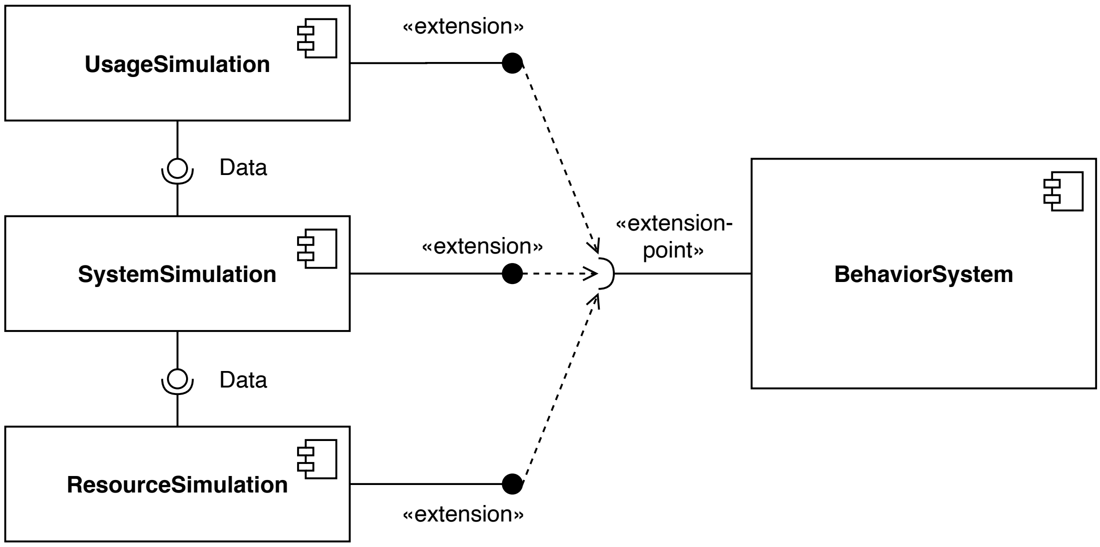


Figure 4.1: UML Component diagram depicting the composition of the behavior modules. The black dot distinguishes that it is an extension-point, meaning that the `BehaviorModule` does not know of the components at compile-time.

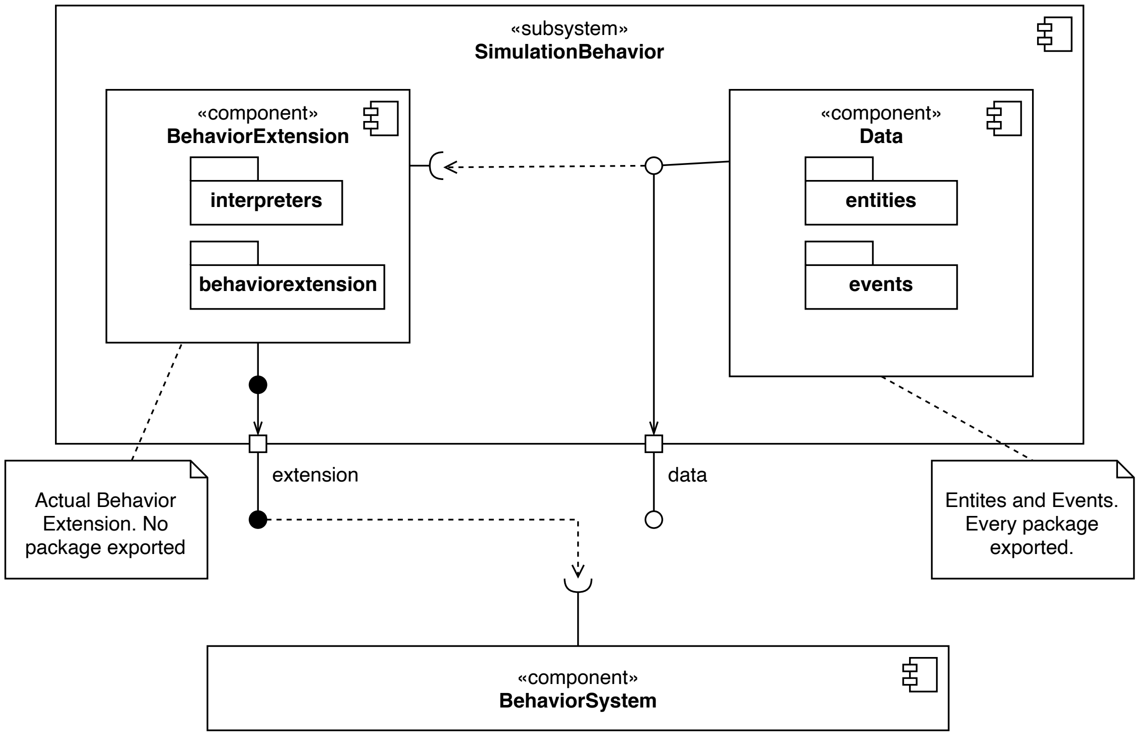


Figure 4.2: UML Component Diagram showing a general pattern of extensions used to implement simulation behaviors.

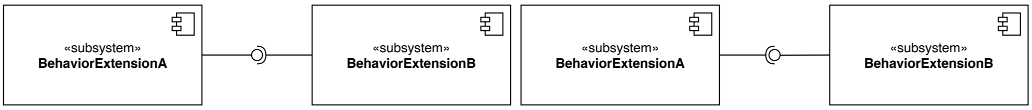
In Figure 4.1 on the previous page the general composition of the simulation behavior modules is depicted. It describes that all the simulation behavior modules extend the Slingshot system by using the extension point `behavior` of the `BehaviorModule`. However, it should be noticed that the `SystemSimulation` depends on the `UsageSimulation`, and `ResourceSimulation` on the other hand depends on `SystemSimulation`. This begs the question as to why this decision was made.

In the following sections, first the general implementation for such a simulation will be presented, as every of the given simulation behaviors follow the same pattern. This will also discuss the design decisions that were made. Furthermore, each module will be briefly presented and discussed. At the end of this chapter, the limitations of the current implementation will be shown.

4.1 General Pattern of Simulation Behavior Extensions

The extensions that were made in this work are *simulation behavior extensions*. They all extend to the extension point `behavior` given in Section 3.1 on page 23. This means that every extension defines their own sets of entities and events, and provide different event handlers. Both events and entities should be visible to the whole system to support full extensibility, permitting other extensions listening to certain events.

In Figure 4.2, the whole simulation module is split into two bundles: The actual simulation behavior that has all the handlers, interpreters, etc. connecting to the extension point of the Slingshot system; and a data bundle where all the entities, events, data structures and holders are defined. The behavior



(a) Provision Pattern **(b)** Requiring Pattern

Figure 4.3: Dependency Patterns for the implementation.

module does not need to export any package or make any class visible to the outside module. The reason for this is that other extensions should not extend the handlers by using inheritance, composition, etc. but rather should only talk using events. For this same reason, all the events and entities should be visible the outside world. When all events are exported, only in this case can other extension access the events. In this way, an extendible and modular system can be built over time and independently.

Furthermore, splitting the modules into behaviors/handlers and data modules can prevent a cyclic dependency. Sometimes, two systems must listen to each others events. Listening to an event defined in another module results in depending on that module. This, in turn, results in a cyclic dependency, because they both then depend on each other. If, on the other hand, the handlers only depend on the corresponding data modules of the other system, then the handler modules do not depend on each other as a whole, and therefore there is no dependency cycle. This only works if the data modules are leaf nodes in the inter-system dependency graph.

Dependency Patterns

In this work, every module that contributes to the simulation follows this pattern. Nevertheless, there is still one problem that remains: Having a too high inter-module dependency can hurt extensibility. The question is, which module should have a dependency when one requiring module "requires" a functionaliy from a providing module? It is important to note that "requires" here means that the handler should publish an event that fires a certain behavior. This means that this problem is not obvious to solve as it would be in monolithic systems. Internally this can happen in two ways:

- **Provision Pattern.** In this dependency pattern, the providing module depends on the requiring module. It listens to the event defined in the requiring module which requests a functionality not given in the module. In this way, the providing module provides the functionality for the requiring module directly. The advantage is that the requiring module can live on its own, and the user can deside which implementation should be used for the required functionality. The downside, however, is that the providing module only provides its functionality to the requiring module and is not for everyone to use.
- **Requiring Pattern.** In this dependency pattern, the requiring module depends on the providing module. The requiring module knows which event to fire up in order to call the required functionality. The advantage is that the functionality can live on its own and every module can use it like a library. Unfortunately, this results in the requiring module being dependent on the providing module.

In Figure 4.3 on the preceding page the different patterns are depicted. In here, the component `BehaviorExtensionA` requires a certain functionality that `BehaviorExtensionB` provides. While in the provision pattern, `BehaviorExtensionA` provides (as in the sense of the UML component diagram) the events that will be fired when the functionality should be called. Hence, `BehaviorExtensionB` requires these events in order to provide its functionality. On the other hand in the requiring pattern, `BehaviorExtensionA` directly requires `BehaviorExtensionB`'s functionality. As always in EDA, the call must happen through an event, and therefore `BehaviorExtensionB` provides its event definition to `BehaviorExtensionA`.

The requiring pattern is the pattern that is most natural to programmers, as this is the typical scenario in monolithic applications. The library provides a functionality, and therefore the system should depend on it. There are some solutions to further decouple them, but an indirect dependency always remains. Here, in EDA, when using the provision pattern, the module could also make an hybrid approach an define a different entry point to the functionality for others to use, meaning it can also be used in the requiring pattern. However, the dependency remains.

As a general approach, the provision pattern should be used when the module does not necessarily need the calculation in order for it to work. In that way, the user can later decide the actual module which can make the calculations. But it is not necessary for the system as a whole to work. On the other hand, when the module really needs that calculation and its results for it to continue, then the requiring pattern should be used instead.

As seen in Figure 4.1 on page 33, this work mainly used the provision pattern. The reason is that the flow of interpretation that is used in PCM. First, the usage simulation starts. When the user wants to make an entry call to the system, the system should therefore provide its simulation behavior to the usage simulation. This results in the `SystemSimulation` depending on the `UsageSimulation`. However, it would be perfectly fine to use the requiring pattern instead. If there are any drawbacks or gains for using one pattern over the other is out of scope of this work.

4.2 GeneralSystemConfigurationBehavior Module

This module is a small behavior extension that provides certain pre-configurations before the simulation can start. This module should not depend on other behavior modules and is thus independent.

One example for such configuration would be configuring the `StoEx`-Cache which is used for evaluating the stochastic expressions in PCM. This is done as soon as the simulation has started, which is implemented by listening to the `SimulationStarted` event. Another configuration is the generation of a random seed at runtime that is also needed when a stochastic expression is evaluated.

4.3 UsageSimulation Module

The UsageSimulation module handles the part for interpreting the PCM's usage models. It does this by creating an interpreter UsageScenarioSwitch as in SimuLizar [Mey11], but for each element, the a corresponding event is published. The interpretation of the whole PCM system starts here, hence this module listens to the SimulationStarted event.

The PCM's usage model consists of a certain kind of workload and a usage scenario. As of now, there exists an entity User that represents a single user. Furthermore, depending on the workload, there is a ClosedWorkloadUserInterpretationContext and OpenWorkloadUserIntrepretationContext. In the ClosedWorkloadUserInterpretationContext, there is a think time specified which is the time the user needs to re-enter the system. In the OpenWorkloadUserInterperetationContext, the number of users is not known in advance, but will be evaluated at runtime.

Each time when a user starts, the UserStarted event will be published. The corresponding end when the user has finished its walkthrough is the UserFinished event. Both are used to create the stack contexts that is used for parametrization. Each model element has furthermore a corresponding event for logging purposes and further functionalities. For instance, the loop has a UserLoopEntered event which is used to track whether a user is in a loop scenario or not, especially because usage scenarios are also defined inside loops. The same applies for branches, but are as of now not fully implemented yet.

The most important element is the UserEntryRequested event. This event is fired whenever a EntryLevelSystemCall model element occurs. In that element, the method of the providing role of the System model is specified. Using the provision dependency pattern, a simulator that is responsible for simulating systems and repositories listens to this event and starts.

4.4 SystemSimulation Module

The SystemSimulation module further extends the UsageSimulation by listening to the UserEntryRequested event. This module consists of two simulations: The simulation of the Repository as well as the System model, and the simulation of RDSEFFs.

Figure 4.4 on the following page depicts the working of the SystemSimulation module. It shows the distinction of the behavior handlers and interpreters. The SystemSimulationBehavior handles the events corresponding to the system, and also listens to the UserEntryRequested event. As in the UsageSimulation, each model element has a corresponding event.

The entry request by the user is a method invocation, which needs the interpretation for its corresponding RDSEFF. For interpreting them, there exists another dependent extension SeffSimulationBehavior. This also exists within the SystemSimulation module, because they are more or less tightly coupled in a logical way. The simulation of can be started by publishing the SeffInterpretationRequested event. Each corresponding element in a RDSEFF then has their own set of events.

When calling an external method, namely using the ExternalCallAction element, the same event is published as if an entry request was done. The InternalCallAction typically states the resources needed for the call, and therefore fires the ResourceDemandRequested.

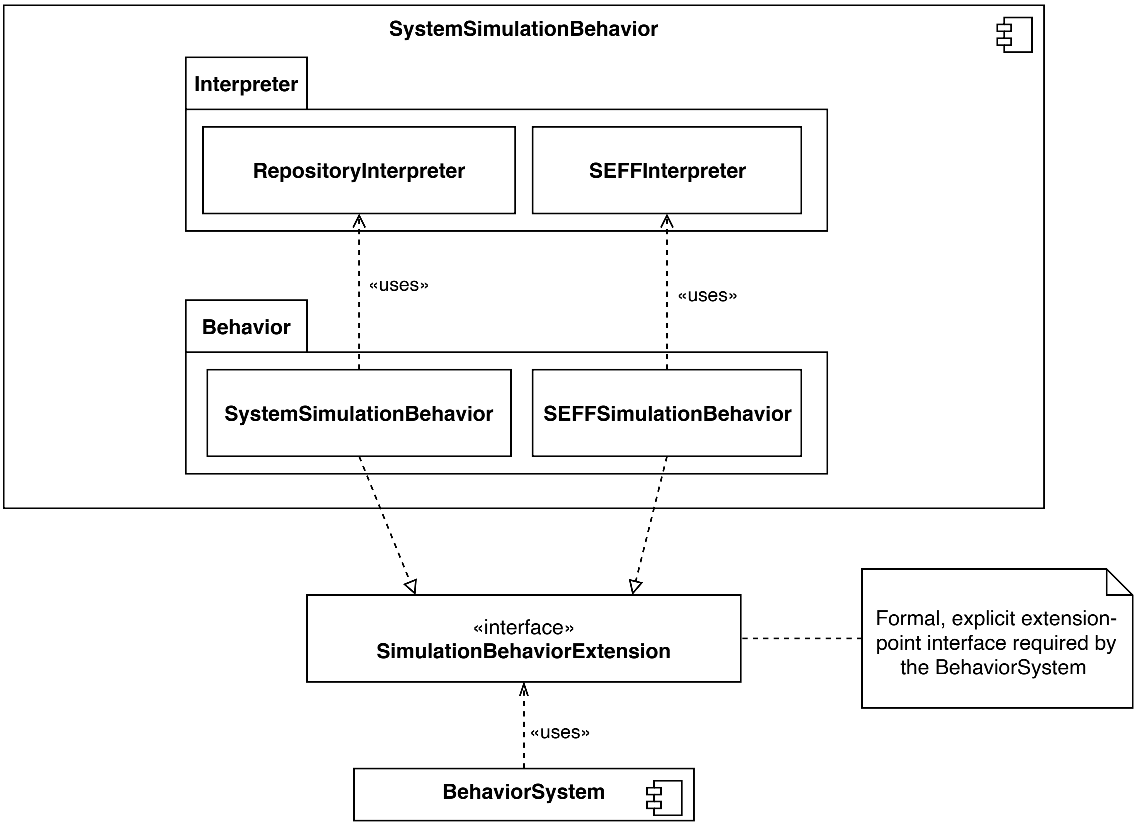


Figure 4.4: UML Component Diagram of the `SystemSimulation` Module

4.5 ResourceSimulation Module

This module depends on the `SystemSimulation` and listens to the `ResourceDemandRequested` event to simulate the resources. In PCM, there exists a `ResourceModel` containing `ResourceContainers`. When a RDSEFF, or more specifically the user, demands a certain resource, it looks up whether the on which container the component in which the RDSEFF is stored lies. The container has specified its hardware and resource metrics. Depending on that, the simulator publishes the `JobStarted` event where the resource is in used. In the background, the actual demand and simulation time of the resource is calculated, and the `JobFinished` event is fired with a delay of that calculated demand. In that way, the simulation time advances.

In PCM, there is a distinction between active resources such as CPU or HDD, and passive resources such as Database Transactions, the latter being more logical resources. Passive resources are often implemented as semaphores to the system. Nonetheless, passive resources are not fully implemented as of now.

Feature	`UsageSimulation`	`SystemSimulation`	`ResourceSimulation`
Entry call to the next simulation layer	✓	✓	-
Loops	✓	✗	-
Branches	~	✗	-
Delays	✓	✓	✓
Open Workload	~	-	-
Closed Workload	~	-	-
Active Resources	-	-	✓
Passive Resources	-	-	✗

Table 4.1: Table showing the features that are supported in the current system. Legend: ✓Fully supported – ✗not supported – ~ partially supported.

4.6 Result and supported Features

Currently, not every functionality is supported yet. Table 4.1 specifies which features are fully, partially or not supported. Nevertheless, the current prototype allows a simulation from top to bottom, meaning that it is possible to fully simulate PCM models when excluding certain elements.

5 Evaluation

This chapter evaluates the Slingshot system in terms of extensibility. Most metrics that are related to extensibility or evidently improve it were already presented in Section 2.1 on page 5. By using the Goal Question Metric [BCR94], the metrics are classified and later interpreted. To answer the hypothesis, however, there has to be a comparison between Slingshot and the currently most used PCM simulators such as EventSim [Mer11] and SimuLizar [Mey11]. Moreover, to answer the question whether EDA improves extensibility in this case, another metric called Extensibility Metric for Software Architectures (EMSA) is used.

EMSA [KKAL18] is a task-based metric for software architectures that uses information theory combining four different task metrics into a single metric. The different task metrics are:

- `Understandability.` How complex is the (core) architecture of the system?
- `Designability.` How complex is the new architecture (that is, the new modules extending the system)?
- `Implementability.` How many modules have to be implemented?
- `Adaptability.` How coupled are the modules together?

EMSA allows for predicting extensibility of the software architecture. The resulting number is given in person hours like in COCOMO II. The lower EMSA is, the more extensible the system can be considered.

The chapter is ordered as follows: In the first section, the study design is presented and discussed. Later, the results of the metrics are presented. Afterwards, the results are interpreted and discussed, including answering the hypothesis. Lastly, some threads to validity are presented.

5.1 Study Design

The study was designed to do a quantitive analysis of the Slingshot software as well as the EventSim and SimuLizar software. Aside from the metrics, some evidence were used from other studies showing that generally EDA improves extensibility. Therefore, the following steps were made within this study:

1. **Literature Research.** In this task, some literatures on EDA and extensibility were found to substantiate the hypothesis. However, literatures and studies in this area are very scarce. The ones that were found and are used describe the EDA architecture and explain why it is extensible. Furthermore, the literature can provide a more sophisticated study design and metrics for extensibility.

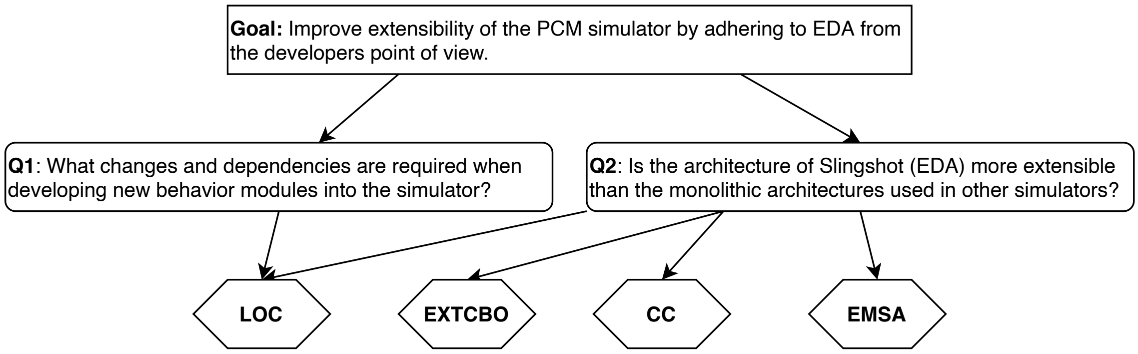


Figure 5.1: The GQM tree discovering what metrics are needed for the study.

2. **Architecture Analysis.** Afterwards, the extension points and architectures of each project are analyzed.
3. **Scenario and Metric Analysis.** After the literatures were found, as they are pointing out that extensibility improves on certain software quality metrics, these metrics of the PCM simulators are collected and analyzed. Studies such as [KPW16] and [KKAL18] analyzed extensibility by trying to implement a new module to the system. Because Slingshot allows an encapsulated behavior extension, all the metrics are focused solely of behavior implementations of each simulator. This work then tries to implement the `SystemSimulation` behavior into the given system. This is done by doing the following steps:
    a) Isolation of the software architecture excluding the behavior that interprets the system model.
    b) Calculating the metrics of that isolated software architecture and implementation.
    c) Finding the metrics of the new `SystemSimulation` behavior module that should be included into the system.
4. **Interpretation.** Lastly, all the results from the literature research and metric analysis are interpreted.

This work focuses on the PCM simulators EventSim and SimuLizar for comparison, as they are the most common used simulators as of now. The code that is being analysed is from [Pal20a] and [Pal20b] GitHub projects. That tool that is used collecting the metrics is the *o3measures* tool by [Aze20] and the CodeMR tool.

In regards to the metrics that needs to be used, the Goal-Question-Metric (GQM) approach by [BCR94] was used. In Figure 5.1, the tree shows the goal that needs to be achieved, the question that needs to be answered for achieving the goals, and the quality metrics that are used to answer the questions. These metrics will then be used in the Scenario and Metric Analysis step.

5.2 Results

This section presents the results of each step. The interpretation of the results is done in the next section, namely Section 5.3 on page 47.

	Klusman et al	Henttonen et al	Kim et al
Metrics	LOC[1], CBO/EXTCBO, LCOM/CC	Like Klusman	EMSA
Improved by EDA?	Yes, but depending on the modules	-	No
Modularity importance	Important	Very Important	Important

Table 5.1: Properties that are important to or are (directly) improving extensibility according to the aforementioned studies.

5.2.1 Literature Research

As already pointed out, there are not many studies about verifying extensibility. This is due to the fact that extensibility itself is not a number that suggests extensibility, but instead other metrics are often used to predict extensibility.

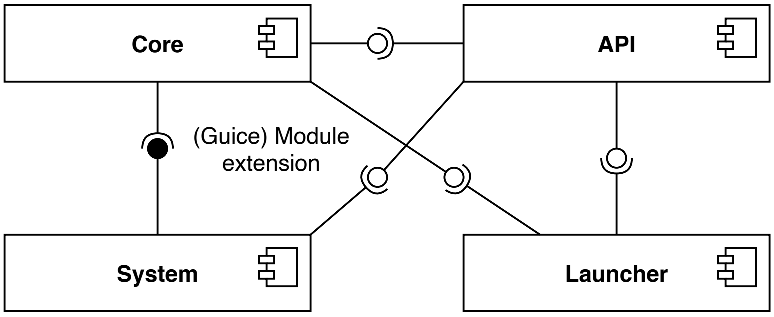
Event-Driven Architecture in Software Development Projects by Klusman et al [KPW16]. This study reveals that extensibility especially in EDA based softwares increases depending on the change that is made. Extensibility suffers when changes has to be made on the current system, that is, no new module has to be plugged in, but rather changes in code has to be made. However, EDA based systems can have the property that module or component coupling is extremely loose. Therefore, an EDA based system with high modularity which is achieved through high cohesion and extreme loose coupling will be extensible. Actually, EDA is already extremely loosely coupled by nature [Mic06].

Integrability and Extensibility from Software Architectural Models by Henttonen et al [HMNK07]. This case study tries to achieve high extensible softwares by following the so-called Integrability and Extensibility Evaluation (IEE) method in Quality-driven Architecture and quality Analysis (QADA) methodology [MND02]. In their case study, they suggest that extensibility suffers when not enough extension points are defined. However, for a single extension point, if the architectural style is extensible, then the extension point itself is extensible, but only for its purpose.

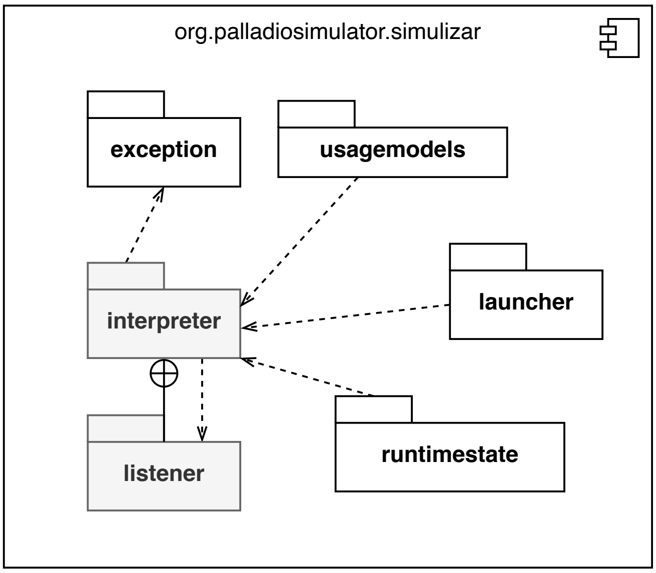
EMSA: Extensibility Metric for Software Architectures by Kim et al [KKAL18]. This paper offers a metric for extensibility for software architectures. The metric itself is derived from other works and combined together into one single metric. The validation of this metric is done by multiple case studies. They suggest that extensibility is also connected to understandability and can therefore suffer if the overall architecture is (too) complex.

Table 5.1 shows the results of the studies. They all agree on that the metrics which have been discussed in Section 2.1 on page 5 need to be considered for extensibility. Furthermore, modularity is more or less a must when trying to make extensible software according to them. Interestingly, there seems to be a contradiction to EDA for extensibility. [KPW16] suggest that extensibility can be improved by EDA depending on the modules. In their case study, however, extensibility only improved slightly. [KKAL18] also conducted the case study with an distributed EDA system and in their system, the monolithic but modular systems had better extensibility than the EDA system.

[1] Especially code line changes.



(a) Architecture of the EventSim [Pal20a]



(b) Architecture of SimuLizar [Pal20b]. The gray packages are the required changes.

Figure 5.2: The architectures of each simulator. They depict only the required part of the architecture interesting for extending the behavior.

5.2.2 Architectural Analysis

In this section, the architectures are presented from the analysis of each project. Slingshot's architecture is already presented in this work in Chapter 4 on page 33 and Chapter 3 on page 23. The architectures of the other systems is not more refined for the analysis in this section.

EventSim

EventSim has a very modular system. The results show that these three bundles are the most important ones when trying to extend the system: `API`, `CORE` and `LAUNCH`. When creating an extension to the system, one has to create a Guice module and implement certain strategies, listeners and simulation event, and then implement the extension point with the created Guice module and all the strategies. The extension point is given by the `CORE` module. A Guice module maps instances to certain types that can will be injected into parameters and attributes using the `@Inject` annotation [Goo20].

In Figure 5.2a on the preceding page the components of the systems are depicted. In the project, all the system simulations happen in the `SYSTEM` module with all the strategies. The metrics are given in Table 5.2 on the following page in the next section. Importantly, EventSim still handles the simulation modules as an integral part of the system and therefore requires a change in the `API` and `LAUNCH` modules. The `API` module contains only (Java) interfaces that define methods for handling simulations and hence will be required by the `LAUNCH` and `CORE` modules. To implement the `SYSTEM` module, the interface `ISystem` must be implemented. Also, the `LAUNCH` module depends on the `API` needed for to start the simulation.

In Figure 5.2a on the preceding page, the "provided" relationship by the `API` here means that the corresponding modules need the interface definition. However, `SYSTEM` actually implements the interface `ISystem`.

SimuLizar

SimuLizar has a more monolithic and complex approach. Every interpretation and simulation happens in the `CORE` bundle. In fact, nearly all core functionality to PCM is implemented there. It is important to note, however, that SimuLizar focuses more and reconfigurations and self-adaption of PCM models, which are implemented elsewhere.

Interestingly, implementing the `SYSTEM` simulation functionality requires only two classes: `RDSeffSwitch` and `RepositorySwitch`. Like Slingshot, the interpretation is done by using the Switch classes which bases on the Visitor pattern [PJ98]. These classes must be included in the `CORE` bundle of the simulator, specifically in the `interpretation` package. Furthermore, the `UsageModelSwitch` is tightly coupled to the `RepositorySwitch` and this in turn is tightly coupled to the `RDSeffSwitch`. This is due to the flow of interpretation: As the simulation starts with the usage model, the `UsageModelSwitch` is first called, and this then needs to call the `RepositorySwitch` for interpreting the repository and system models, which in turn will call the `RDSeffSwitch` as soon as a modeled service is called.

In Figure 5.2b on the facing page, the depiction shows a single bundle containing all the packages that are used or dependent to other packages. The `listener` package is inside the `interpreter` package.

5.2.3 Scenario and Metric Analysis

In this section, the results of the scenario "Implementing System-Interpretation behaviors to the System" are given. The results focus on the implementations given by [Pal20a] and [Pal20b]. The focus then lies further to the implementation regarding the system interpretation including interpreting the SEFF at runtime. Table 5.2 on the next page shows the results of each metric and general information for each simulator. The extensibility result is categorized as follows:

- **`High.`** The system is considered highly extensible if the system in mind does not require relatively a lot of change in the modules, and if the architecture does not require a lot of effort to extend the system.
- **`Low.`** The system is considered to be little extensible if the system requires a lot of change in its existing modules or if the architecture requires a lot of effort to extend the system.

Property	Simulators		
	Slingshot	EventSim	SimuLizar
Project Data			
Architecture	Event-Driven	Monolithic	Monolithic
# of Bundles	27	23	49
# of Extension Points	1	5	11
Changes Metrics			
# of New Bundles	2	1	0
# of Changed Bundles	0	**1**	**1**
# of Classes	21	40	4
LOC of new bundles	1260	1709	-
LOC of changes	-	356	**1130**
Module Metrics			
# of inter-module dependencies	2	4	3
EXTCBO	30	55	318
CC	40%	9.3%	3.5%
EMSA			
Architectural complexity	2.635	4.969	8.634
Implementability & Adaptability	1292	2173	2070
EMSA	225.762	370.934	355.124
Extensibility	**High**	**High**	**Low**

Table 5.2: Results of the Scenario and Metrics Analysis.

For the results shown in Table 5.2, two main points are important to note: First, the module metrics relate to the module or modules that needed to be changed in order to include the system interpretation functionality. Secondly, the definition of a module varies between the projects. Sometimes, an ordinary Java package can be seen as a module. Because SimuLizar has its PCM functionalities in a single Eclipse bundle, the module metrics relate to the package metrics.

Some of the results were acquired using tools like o3smeasures or CodeMR. However, none of these modules have the ability to calculate modular metrics. For this reason, the modular metrics as well as EMSA were calculated by hand.

5.3 Discussion

First of all, the literatures highly suggest that offline EDA-based systems are indeed highly extensible. Their case studies all suggest, however, that extensibility is only reached when the modules are independent enough. Furthermore, the notion of extensibility highly depends on the goal that is tried to be achieved. In Slingshot, extensibility is defined in terms of behavior. It is not possible to extend the system in terms of annotations or interceptors. This can hinder extensibility as a whole. This is also shown in the presented case studies. EDA based systems have furthermore the disadvantage to be complex or complicated to the novice programmer to which extensibility can suffer, as extensibility is often defined to be correlated with understandability. However, understandability is often a subjective experience rather than an objective conclusion.

According to the results given in Table 5.2 on the preceding page, it clearly shows that Slingshot is highly extensible at least in terms of behavior. This is due to the fact that Slingshot requires no changes at all for implementing a certain behavior of the system. This explains why the EMSA metric seems to be the lowest of all three. EventSim also exhibits a highly behavioral extensibility, because the changes that need to be done only depends on the implementation and goal of the extension. However, EventSim has some assumptions about how the behavior and simulation has to be done, as it assumes the standard of PCM. The EMSA, however, is the highest of all three, which is a consequence of the number of LOC that affect implementability and adaptability. Lastly, extensibility in terms of behavior is mostly not given in SimuLizar. It is important to note that SimuLizar does define extension points, but these are tightly coupled to certain behaviors to its system. It requires a lot of effort and change when implementing a new behavior that is not yet defined in PCM.

This section also answers the following research question:

Research Question
Does an event-driven architecture coupled with annotation definitions and processing improve the extensibility of a PCM simulator?

The results state that extensibility improves when adhering to EDA. Due to the extremely loosely coupled system that EDA and hence Slingshot has, nearly no change was required when implementing the system simulation. This also shows that Slingshot is at least slightly more extensible than EventSim; however, the factor of improvement is not known. Furthermore, as the literature and the analysis suggest, methods in Slingshot do not depend on each other as much as it does in the other two simulators. This comes from the nature of EDA as there is no traditional call stack anymore. In turn, this results in a more independent programming and therefore strengthens extensibility.

The disadvantage of Slingshot's implementation of EDA, however, is the stamp coupling. The defined events act more or less as an interface between modules. Stamp coupling refers to a type of coupling where only complex data is exchanged and has no communication between methods. Because they act like interfaces, a change in such a data structure can too lead to breaks. The stamp coupling can be seen in the results in EMSA's adaptability metric which considers the inter-module dependencies. Furthermore, understandability of EDA based systems are generally higher because of the lack of method call stack which leads to a high difficulty of tracing calls.

The results do not directly show what impact annotations have on extensibility. While class coupling has decreased using annotations (because the contract checking behavior does not have to be called by the classes or have to programmatically be specified), it is hard to isolate annotations for extensibility analysis, as the other simulators do not use such annotations. A comparison is therefore not possible as of now.

5.4 Threats to Validity

The data and results where collected using free tools for Eclipse. Also, because EXTCBO and CC aren't implemented in most metric tools, the numbers had to be calculated manually. In order to counteract errors, they were calculated multiple times and compared. However, there is still a chance that some calculations error occurred, and furthermore there might be errors made by the tools.

Furthermore, EMSA itself states that it more or less only works for small projects. Moreover, the architectures given here might not represent the architectures the developers intended. Due to time limitations, the developers could not have been contacted. Also, the system simulation were already implemented in EventSim and SimuLizar at the time of writing, which could distort the metrics for EMSA, as it only predicts systems and considers architectures, but not the actual implementation.

It is important to note that the data presented here represents the changes needed to be done in order to implement `SystemSimulation`. While the data suggest that the system is extensible, it only does so by looking into the implementation of `SystemSimulation`. Depending on the extension that needs to be implemented, the results can vary and the system as a whole including the extension can become hardly extensible. Because of the time constraints in this work, there was no possible way to prevent this threat.

6 Conclusion

This section concludes and summarizes the thesis including the results. It highlights the advantages and disadvantages, as well as further work that needs to be done in order to either further improve the extensibility or gain more evidence for it.

6.1 Summary

The objective of this thesis was to create an extensible PCM simulator by using the Event-driven Architecture. Reason for this was that current simulators lack the ability to fully support new features without the need to change its core architectures. Because literature suggests that EDA can create highly extensible systems, this was used to improve extensibility of PCM simulators. The main hypothesis of this work was that EDA with annotations can improve extensibility.

The foundation (Chapter 2 on page 5) discussed the notion of extensibility and the metrics that affects it. Furthermore, the EDA was also presented and connected to the extensibility factors. With this foundation, the Slingshot system was designed.

This work created a prototype of such simulator that has the possibility to create and plug in certain behaviors to the system. Additional to the core extensible system, the first prototypical implementation of simulation behaviors was also created, namely the simulation of usage, system, and resource models.

By using the created prototypes, the extensibility of the system was evaluated by comparing the simulators on changeability and the metrics introduced in the foundation. The results suggest that extensibility was achieved by using EDA; however, a connection to the annotations could not have been made. Therefore, the hypothesis is partially answered, but the question as to whether annotations had an impact on extensibility is still up to debate.

6.2 Benefits

Even though extensibility by annotations could not have been proven, the usage of a EDA based system did already improve extensibility. The system is already intact and it allows the development of further extensions independently to each other. This means that there can be multiple teams that work parallelly on different modules. If the system extension in mind is designed well in the way that the modules are not too tightly coupled, then even communication between the teams can be reduced, which could reduce the effort for implementation. Another great benefit of this system is that the end-user can have multiple choices as to what module to use in order to simulate its system. The system allows for a competitive environment with different simulation modules for the same functionality, and the user can choose the better one according to their standards.

6.3 Limitations

The system itself can only be extended through its behavioral extension point. Other extensions such as extending or substituting the interceptors or event libraries is not possible as of now due to security reasons.

Furthermore, the prototypical simulation implementation does not cover all the PCM elements for a full simulation. The prototype can therefore not be used in practice as of now.

Additionally, the method to prove the improvement of extensibility is limited to the `SystemSimulation` module. This method only allowed to show the extensibility of the modules that are designed as the `SystemSimulation`, and is hence highly depending on the extension itself whether the system can be extended. Due to the time constraints to this work, a isolated case study could not have been conducted in order to prove extensibility with other systems. Furthermore, the study could not find information about whether annotations can actually improve extensibility in EDA-based systems.

6.4 Lessons Learned

Because EDA systems are typically very complex to programmers new to this environment, the planning and development of such systems need more time than expected. This can lead to an underestimation of the project and time needed to fully implement the system. Moreover, the Eclipse IDE and its plugin system has a difficult learning curve for new programmers to this system. The lesson here is that EDA based systems in Eclipse need more work than typical Java implementations, which shouldn't be underestimated.

However, as soon as the EDA systems and the Eclipse's plugin system has been fully grasped, new opportunities for such systems can be found, which allows for implementing extensible and needed plugins. The more such plugins are developed, the richer the IDE experience can become. The understanding of such systems also allows a deeper understanding of the Java softwares which are typically hidden by current programmers when using Maven based projects.

6.5 Future Work

Because a case study could not be conducted, this is one of the important future works. Such case study can highlight further extension points and direct the developers to the way where extensibility can be further improved. This would also highlight the failures and hindrances that extension developers for Slingshot might face.

Another great feature of EDA based system is its ability to be extremely loosely coupled, which allows for a distributed systems. The Slingshot system can be implemented in such a way that the modules can be placed on the server. This allows the heavy calculations to be done on high-performance servers where the typical end-users computer might fail. In that way, services to Slingshot and PCM can be provided online without the need to install the modules offline.

Lastly, the implementation of the simulation behaviors needs to be further developed to fully support the PCM. Besides, these implementations will work as a reference for other developers to create new extensions.

Bibliography

[ADA15] A. B. AL-Badareen, J.-M. Desharnais, A. Abran. "A Suite of Rules for Developing and Evaluating Software Quality Models". In: *Software Measurement*. Ed. by A. Kobyliński, B. Czarnacka-Chrobot, J. Świerczek. Cham: Springer International Publishing, 2015, pp. 1–13. ISBN: 978-3-319-24285-9. DOI: 10.1007/978-3-319-24285-9_1 (cit. on pp. 6, 7).

[Ann11] Oracle. *Java Annotation Processor: Processor Interface and Specification*. URL: https://docs.oracle.com/en/java/javase/11/docs/api/java.compiler/javax/annotation/processing/Processor.html (cit. on p. 16).

[Aze20] M. Azevedo. *o3measures Tool: Plugin for Eclipse measuring software qualitiy in Java projects*. Feb. 9, 2020. URL: https://github.com/mariazevedo88/o3smeasures-tool (cit. on p. 42).

[BBM13] M. Becker, S. Becker, J. Meyer. "SimuLizar: Desing-Time Modeling and Performance Analysis of Self-Adaptive Systems". In: *Software Engineering 2013*. Ed. by S. Kowalewski, B. Rumpe. Bonn: Gesellschaft für Informatik e.V., 2013, pp. 71–84. ISBN: 978-3-88579-607-7 (cit. on p. 25).

[BCNN10] J. Banks, J. Carson, B. Nelson, D. Nicol. *Discrete-event system simulation*. English. 5th ed. Upper Saddle River, N.J. Singapore: Prentice Hall, 2010. ISBN: 0136062121 (cit. on pp. 19, 20).

[BCR94] V. R. Basili, G. Caldiera, H. D. Rombach. "The goal question metric approach". In: *Encyclopedia of software engineering* (1994), pp. 528–532 (cit. on pp. 41, 42).

[BD10] R. Bruns, J. Dunkel. *Event-Driven Architecture*. Springer, 2010. ISBN: 978-3-642-02438-2. DOI: 10.1007/978-3-642-02439-9 (cit. on p. 10).

[Bec08] S. Becker. "Coupled model transformations for QoS enabled component-based software design". PhD thesis. 2008. 297 pp. ISBN: 9783866442719. DOI: 10.5445/KSP/1000009095 (cit. on pp. 1, 22).

[BKR09] S. Becker, H. Koziolek, R. Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (2009). Special Issue: Software Performance - Modeling and Analysis, pp. 3–22. ISSN: 0164-1212. DOI: 10.1016/j.jss.2008.03.066. URL: http://www.sciencedirect.com/science/article/pii/S0164121208001015 (cit. on pp. 1, 17, 18, 33).

[BME+07] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, K. Houston. *Object-Oriented Analysis and Design with Applications, Third Edition*. Third. Addison-Wesley Professional, 2007. ISBN: 9780201895513 (cit. on p. 6).

[CAIS14] L. Cheikhi, R. Al-Qutaish, A. Idri, A. Sellami. "Chidamber and Kemerer Object-Oriented Measures: Analysis of their Design from the Metrology Perspective". In: *International Journal of Software Engineering and its Applications* 8.2 (Mar. 2014), pp. 359–374. ISSN: 1738-9984. DOI: 10.14257/ijseia.2014.8.2.35 (cit. on p. 8).

[Car93] J. S. Carson. "Modeling and Simulation Worldviews". In: *Proceedings of 1993 Winter Simulation Conference - (WSC '93)*. 1993. DOI: 10.1109/WSC.1993.718024 (cit. on pp. 19, 23).

[CK94] S. R. Chidamber, C. F. Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493. ISSN: 1939-3520. DOI: 10.1109/32.295895 (cit. on p. 8).

[CR06] E. Clayberg, D. Rubel. *Eclipse: building commercial-quality plug-ins*. Pearson Education, 2006 (cit. on p. 12).

[EFGK03] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec. "The Many Faces of Publish/Subscribe". In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078 (cit. on p. 11).

[EquE] Eclipse Foundation. *Equinox*. URL: http://www.eclipse.org/equinox (cit. on p. 13).

[GJSB05] J. Gosling, B. Joy, G. Steele, G. Bracha. *The Java ™Language Specification*. 2005. ISBN: 0-321-24678-0 (cit. on p. 15).

[Goo20] Google. *Guice: Dependency Injection Framework*. May 19, 2020. URL: https://github.com/google/guice (cit. on p. 44).

[GooGua] Google. *Guava: Google Core Libraries for Java*. URL: https://github.com/google/guava (cit. on p. 23).

[HMNK07] K. Henttonen, M. Matinlassi, E. Niemela, T. Kanstren. "Integrability and Extensibility Evaluation from Software Architectural Models - A Case Study". In: *The Open Software Engineering Journal* 1.1 (Dec. 2007), pp. 1–20. DOI: 10.2174/1874107x00701010001 (cit. on p. 43).

[Hoh06] G. Hohpe. "Programming without a call stack – event-driven architectures". In: *Objekt Spektrum* (2006). URL: https://www.enterpriseintegrationpatterns.com/docs/EDA.pdf (cit. on p. 10).

[JL09] N. Johansson, A. Löfgren. "Designing for Extensibility: An action research study of maximizing extensibility by means of design principles". B.S. thesis. Gothenburg, Sweden: Göteborgs Universitet, Department of Applied Information Technology, June 30, 2009. URL: http://hdl.handle.net/2077/20561 (cit. on pp. 5, 6).

[KB98] W. Kozaczynski, G. Booch. "Component-Based Software Engineering". In: *IEEE Software* 15.5 (Sept. 1998), pp. 34–36. ISSN: 1937-4194. DOI: 10.1109/MS.1998.714621 (cit. on pp. 1, 17).

[KH06] H. Koziolek, J. Happe. "A QoS Driven Development Process Model for Component-Based Software Systems". In: *Component-Based Software Engineering*. Springer Berlin Heidelberg, 2006, pp. 336–343. DOI: 10.1007/11783565_25 (cit. on p. 17).

[KK08] B. Klatt, K. Krogmann. "Software Extension Mechanisms". In: *Proceedings of the Thirteenth International Workshop on Component-Oriented Programming (WCOP'08), Karlsruhe, Germany*. Ed. by R. Reussner, C. Szyperski, W. Weck. Interner Bereich Universität Karlsruhe (TH) 2008-12. 2008, pp. 11–18. URL: http://sdqweb.ipd.uka.de/publications/pdfs/klatt2008a.pdf (cit. on p. 14).

[KKAL18] J. Kim, S. Kang, J. Ahn, S. Lee. "EMSA: Extensibility Metric for Software Architecture". In: *International Journal of Software Engineering and Knowledge Engineering* 28 (Mar. 2018), pp. 371–405. DOI: 10.1142/S0218194018500134 (cit. on pp. 41–43).

[KPW16] M. Klusman, R. Plasmeijer, R. Wolter. "Event-Driven Architecture in software development projects". Nijmegen. MA thesis. Radboud University, 2016. URL: https://www.ru.nl/publish/pages/769526/maxime_klusman.pdf (cit. on pp. 11, 42, 43).

[LMV02] P. L'Ecuyer, L. Meliani, J. Vaucher. "SSJ: A Framework for Stochastic Simulation in Java". In: *Proceedings of the 2002 Winter Simulation Conference*. Ed. by E. Yücesan, C.-H. Chen, J. L. Snowdon, J. M. Charnes. Available at http://simul.iro.umontreal.ca/ssj/indexe.html. IEEE Press, 2002. DOI: 10.1109/WSC.2002.1172890 (cit. on p. 23).

[McC76] T. J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320. ISSN: 1939-3520. DOI: 10.1109/TSE.1976.2333837 (cit. on p. 8).

[Men08] J. Mendling. "Event-Driven Process Chains (EPC)". In: *Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness*. Springer Berlin Heidelberg, 2008. Chap. 2, pp. 17–57. ISBN: 978-3-540-89224-3. DOI: 10.1007/978-3-540-89224-3_2 (cit. on p. 30).

[Mer11] P. Merkle. "Comparing Process- and Event-oriented Software Performance Simulation". MA thesis. Karlsruhe Institute of Technology (KIT), Germany, 2011. URL: http://sdqweb.ipd.kit.edu/publications/pdfs/merkle2011a.pdf (cit. on pp. 19–22, 41).

[Mey11] J. Meyer. "Modellgetriebene Skalierbarkeit von selbst-adaptiven komponentenbasierten Softwaresystemen in der Cloud". MA thesis. Paderborn: Universität Paderborn, Fakultät für Elektrotechnik, Informatik und Mathematik, Nov. 2011 (cit. on pp. 1, 22, 37, 41).

[MH11] P. Merkle, J. Henß. "EventSim - An Event-driven Palladio Software Architecture Simulator". English. In: *Palladio Days 2011. Proceedings, 17-18 November 2011, FZI Forschungszentrum Informatik, Karlsruhe, Germany. Ed.: S. Becker*. Vol. 2011. Karlsruhe Reports in Informatics (früher: Interner Bericht. Fakultät für Informatik, Karlsruher Institut für Technologie) 32. 2011, pp. 15–22. DOI: 10.5445/IR/1000025188 (cit. on p. 1).

[Mic06] B. M. Michelson. "Event-driven architecture overview". In: *Patricia Seybold Group* 2.12 (2006), pp. 10–1571. URL: http://elementallinks.com/el-reports/EventDrivenArchitectureOverview_ElementalLinks_Feb2011.pdf (cit. on pp. 9, 10, 43).

[MirrorAPI] Oracle. *Model of the Java Programming Language: API*. URL: https://docs.oracle.com/javase/8/docs/api/index.html (visited on 11/20/2020) (cit. on p. 16).

[MND02] M. Matinlassi, E. Niemelä, L. Dobrica. *Quality-driven architecture design and quality analysis method: A revolutionary initiation approach to a product line architecture*. English. VTT Publications 456. Project code: E1SU00054. VTT Technical Research Centre of Finland, 2002. ISBN: 951-38-5967-3 (cit. on p. 43).

[Ora20] Oracle Corporation. *Compilation Overview*. 2020. URL: https://openjdk.java.net/groups/compiler/doc/compilation-overview/index.html (visited on 12/16/2020) (cit. on p. 16).

[OraJar] Oracle. *JAR File Specification: JAR Manifest*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html#JARManifest (visited on 11/27/2020) (cit. on p. 13).

[OSG18] OSGi Alliance. *OSGi Core Specification, Release 7*. http://www.osgi.org/developer/specifications. Apr. 2018 (cit. on pp. 13, 14).

[OSGiMod] OSGi Alliance. *Why is Modularity so Important?* URL: https://www.osgi.org/developer/modularity/ (visited on 12/15/2020) (cit. on p. 12).

[Pal20a] Palladio. *Palladio-Analyzer-EventSim GitHub Project*. Release 4.3.0. June 18, 2020. URL: https://github.com/PalladioSimulator/Palladio-Analyzer-EventSim/tree/1a222313f971ab8f84c4fa6c54eac942f11ef36f (cit. on pp. 42, 44, 45).

[Pal20b] Palladio. *Palladio-Analyzer-SimuLizar GitHub Project*. Release 4.3.0. May 28, 2020. URL: https://github.com/PalladioSimulator/Palladio-Analyzer-SimuLizar/tree/b6b69b4f1cdf162d17ca6c2e19b696adcf1cd92c (visited on 11/30/2020) (cit. on pp. 42, 44, 45).

[Par12] H. Parzjegla. "Engineering Publish/Subscribe Systems and Event-Driven Applications". PhD thesis. Universität Rostock, Fakultät für Informatik und Elektrotechnik, Dec. 10, 2012. URL: http://rosdok.uni-rostock.de/file/rosdok_disshab_0000001097/rosdok_derivate_0000005220/Dissertation_Parzyjegla_2014.pdf (cit. on p. 11).

[PJ98] J. Palsberg, C. B. Jay. "The essence of the Visitor pattern". In: *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac '98) (Cat. No. 98CB 36241)*. IEEE, 1998, pp. 9–15. ISBN: 0-8186-8585-9. DOI: 10.1109/CMPSAC.1998.716629 (cit. on p. 45).

[PWE14] B. Klatt, M. Hauck. *Palladio Workflow Engine*. https://sdqweb.ipd.kit.edu/wiki/Palladio_Workflow_Engine/Documentation/Job_Types. 2014 (cit. on p. 28).

[RBB+11] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, M. Kuperberg. *The Palladio Component Model*. Tech. rep. 14. Karlsruher Institut für Technologie (KIT), 2011. 193 pp. DOI: 10.5445/IR/1000022503 (cit. on p. 33).

[RV11] H. Rocha, M. Valente. "How Annotations are Used in Java: An Empirical Study". In: Jan. 2011, pp. 426–431 (cit. on p. 15).

[SMRL13] M. Strittmatter, P. Merkle, A. Rentschler, M. Langhammer. "Towards a Modular Palladio Component Model". In: vol. 1083. Nov. 2013, pp. 49–58. URL: https://www.researchgate.net/publication/261797648_Towards_a_Modular_Palladio_Component_Model (cit. on p. 22).

[SQuaRe11] ISO Central Secretary. *Systems and software engineering — Systems and software Quality*. Standard ISO/IEC 25010:2011. Geneva, CH: International Organization for Standardization, Mar. 2011. URL: https://www.iso.org/standard/35733.html (cit. on pp. 7, 8).

[VGS+00] T. Vernezza, G. Granatella, G. Succi, L. Benedicenti, M. Mintchev. "Defining Metrics for Software Components". In: *Proceedings of. World Multi-conference on Systematics, Cubernetics and Informatics* 11 (Jan. 2000), pp. 16–23. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.205.8052&rep=rep1&type=pdf (cit. on p. 9).

All links were last followed on March January 7, 2021.

A Events in the Simulator

This chapter contains the current list of events in each simulation behavior from Chapter 4 on page 33. All the important events and the typical event flow are depicted in Figure A.1 on the following page. Furthermore, the description of each event is also given.

`SimulationStarted`.

Description Tells the system that the simulation has officially started.

Publishable By internal system only.

Owner Internal System.

Entity Change None.

`UserStarted`.

Description This will be publishes as soon as a single simulated user has started the interpretation of the UsageModel flow.

Publishable By Everyone.

Owner UsageSimulation.

Entity Change `User`, `UserInterpretationContext`

`UserInterpretationEntryLevelCallPassed`.

Description Tells the system that the interpretation of the usage model has just passed the `EntryLevelSystemCall` model element.

Publishable By UsageSimulation.

Owner UsageSimulation.

Entity Change `UserInterpretationContext`

`UserSlept`.

Description Is published when the user has started “sleeping”, e.g., when a delay (`Delay` Usage-Model element) or think time is requested.

Publishable By UsageSimulation.

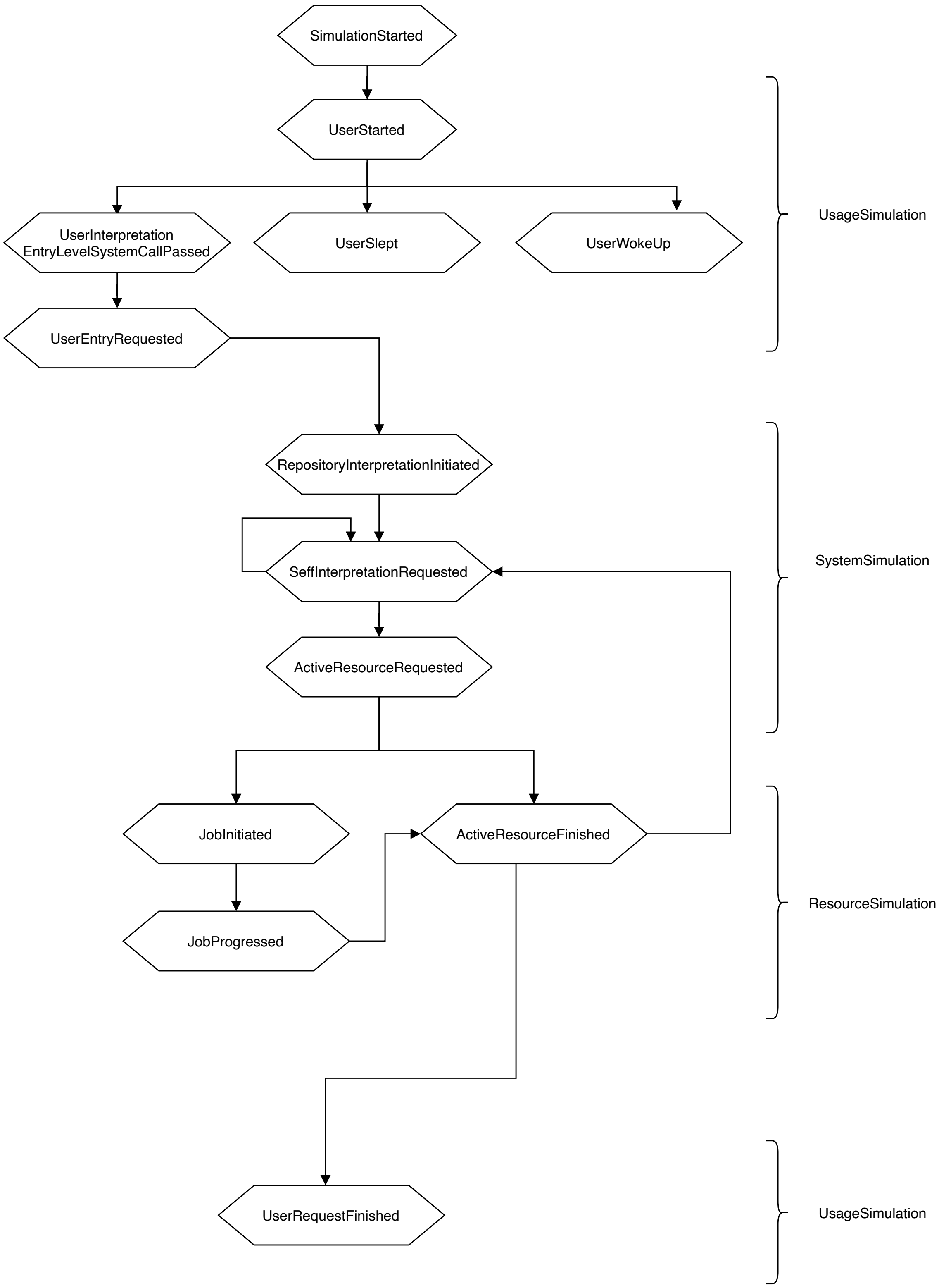


Figure A.1: The typical and important event flow as used in the Slingshot simulator right now. The event handlers are not depicted and between every event there is a hidden OR-gate.

Owner UsageSimulation.

Entity Change `UserInterpretationContext`

`UserSlept`.

Description Is published when the user has woken up from the delay or think time

Publishable By UsageSimulation.

Owner UsageSimulation.

Entity Change `UserInterpretationContext`

`UserEntryRequested`.

Description Will be published as soon as the request to interpret the Service-Effect Specification (SEFF) provided by the PCM system is initialized.

Publishable By UsageSimulation.

Owner UsageSimulation.

Entity Change `UserInterpretationContext`, `UserEntryRequest`

`UserRequestFinished`.

Description Should be published as soon as the entry request of the user has been successfully processed, including the interpretation of the entry call's SEFF.

Publishable By Everyone.

Owner UsageSimulation.

Entity Change `UserInterpretationContext`, `UserEntryRequest`

`RepositoryInterpretationInitiated`.

Description Is published when the interpretation of a repository, such as the system model or the repository model itself, has been initiated and is ready to start interpretation.

Publishable By SystemSimulation.

Owner SystemSimulation.

Entity Change `RepositoryInterpretationContext`

`SeffInterpretationRequested`.

Description Is published when the interpretation of a SEFF has been requested.

Publishable By SystemSimulation.

Owner SystemSimulation.

Entity Change `SEFFInterpretationContext`

`ActiveResourceRequested`.

Description Is published when the RDSEFF is requiring an active resource demand.

Publishable By SystemSimulation.

Owner SystemSimulation.

Entity Change `ActiveResourceRequest`

`JobInitiated`.

Description Notifies that a certain job for a resource has been initiated and started.

Publishable By ResourceSimulation.

Owner ResourceSimulation.

Entity Change `Job`

`JobProgressed`.

Description Notifies that a certain job has progressed but not finished yet.

Publishable By ResourceSimulation.

Owner ResourceSimulation.

Entity Change `Job`

`ActiveResourceFinished`.

Description Notifies the system simulator that the resource simulation has successfully finished the resource demand and the job is done.

Publishable By Everyone.

Owner SystemSimulation.

Entity Change `ActiveResourceRequest`

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature