A Pattern Language for Manual Analysis of Runtime Events Using Design Models

Michael Szvetits
Software Engineering Group
University of Applied Sciences Wiener Neustadt
Wiener Neustadt, Austria
michael.szvetits@fhwn.ac.at

Uwe Zdun
Software Architecture Research Group
University of Vienna
Vienna, Austria
uwe.zdun@univie.ac.at

ABSTRACT

Modeling is an important activity in the software development process whose output are design artefacts that describe the resulting software from a high-level perspective. Recent research investigates the role of models at runtime and the results indicate that analysts perform better at observing the behaviour of a running system if they can utilize models during the analysis. However, setting up a system which allows the analysis of its behaviour at runtime using models involves many challenges regarding the modeling environment, the introspection infrastructure, the traceability management and the analysis integration. This paper summarizes design alternatives for implementing systems with manual analysis support by investigating recurring concepts like patterns, modeling habits, languages, middlewares and development techniques found in approaches that utilize models at runtime. We organize the gained knowledge as patterns in a pattern language which captures various issues and their solution alternatives, including their benefits and liabilities. The pattern language consists of modeling patterns for setting up the models and the environment for the analyst, introspection patterns for extracting data from the running system, traceability patterns for relating the extracted data with the models, and analysis patterns for processing the extracted data using the models. We demonstrate the application of the pattern language based on the implementation of a robot system.

CCS CONCEPTS

• **Software and its engineering → Patterns**; *Designing software*;

KEYWORDS

analysis, events, model, pattern, runtime

ACM Reference Format:
Michael Szvetits and Uwe Zdun. 2018. A Pattern Language for Manual Analysis of Runtime Events Using Design Models. In *23rd European Conference on Pattern Languages of Programs (EuroPLoP '18), July 4–8, 2018, Irsee, Germany*. ACM, New York, NY, USA, Article 4, 24 pages. https://doi.org/10.1145/3282308.3282324

1 INTRODUCTION

The level of abstraction in the software development discipline is constantly increasing. The implementation of software systems is nowadays more focussed on the fulfillment of customer requirements than overcoming technical challenges that are grounded in the underlying execution environment. The shift from solving technical challenges to solving customer-oriented problems can especially be observed when looking at the changes that the software engineering discipline took over the time: Programming languages incorporate more and more mechanisms to define abstractions, while the emergence of model-driven techniques indicate that former informal design artefacts make the jump to become important prescriptive artefacts which guide the software development process from a problem space perspective [10].

Although informal models are still very common in industry [54], model-driven approaches using formal models promise to improve productivity and maintainability when developing software systems [42]. Requirements and expectations on a software system change over time and demand an increased level of flexibility. As a consequence, the boundary between development time and runtime blurs and requires new development and deployment strageties [29]. One of them is the models at runtime paradigm [8, 10, 21, 50] where models are causally connected to the running system, meaning that changes to the models cause corresponding changes within the running system, and vice versa. This paradigm has been evaluated exhaustively in the area of automatic adaptation decisions, especially with performance implications in mind [58]. However, there are scenarios where automatic decisions are limited or too complex, but models can still help in the process of manually analyzing (and in further consequence, adapting) a running system:

- **Confirmation of actions.** Models can provide situation-specific information about a process that needs attention by a human decision maker. Examples are financial transactions or legally binding actions.
- **Control of simulations.** Models provide a condensed view of the system, can highlight simulation errors and allow to adapt testing conditions.
- **Reacting to edge cases.** Example of such cases are violations of service level agreements or hardware faults. Models help to visualize and detect erroneous system parts.
- **Reacting to violations.** Models can highlight process activities which violate rules that arise from non-technical requirements, like mandatory reporting steps that were not detected during execution.
- **Manual assessments.** Examples are qualitative and quantitative runtime data, bottlenecks, or trends. Models allow an

analyst (i.e., the person who diagnoses the running system) to monitor and control the running system from a perspective which is closer to the problem space.

A recent controlled experiment indicated that linking models with the running system improves the comprehension of system behaviour when analysts manually observe the runtime events that were recorded by the running system [57]. Using models for runtime analysis in such a manner requires that events are recorded and traced back to the model elements of interest where the analyst is able to define aggregation operations on them. However, the experiment only applied a small subset of the possible modeling, introspection, traceability and analysis techniques that can be used to realize a software system with manual analysis support. In the general case, software architects and developers are confronted with many decisions when designing a system that supports the analysis of its behaviour with the help of models. Examples of problems that require such decisions are:

- Which models are suited for performing manual analysis?
- Which techniques exist to efficiently capture events at runtime to enable manual analysis?
- How can runtime events be correlated with model elements they belong to?
- Which techniques exist for the analyst to define aggregation operations on a stream of events?

The goal of this paper is to address the challenges software architects and developers face when implementing support for manual analysis using design models for a software system. We divide the necessary design decisions into four categories of related patterns and describe their benefits and liabilities. With the help of those patterns, software architects and developers can deliberately decide which solution strategy fits their problem context best and weigh their advantages and disadvantages. The patterns capture design decisions of varying granularity and level of abstraction and are organized as pattern language, meaning that also the interconnections of patterns are discussed.

Regarding the source of the patterns, we revisited the approaches that were identified by our recent comprehensive systematic literature review [58] of the objectives, techniques, kinds, and architectures of models at runtime as well as our paper on reusable event types [56]. We extracted recurring architecture and design concepts (e.g., patterns, modeling habits, languages, middlewares and development techniques) that are used in those approaches and brought them into the context of manual analysis of running systems using design models. While some of the identified concepts are indeed known patterns, many of them are not explicitly named as such, but are implicitly recurring concepts which influence the way how the analysis is performed. This paper streamlines all the identified concepts into the common notion of a pattern, meaning the organization of their properties into a context description, a problem statement, a solution description and their relationships to the underlying forces that shape the problem. The consequences of choosing a specific pattern (i.e., the benefits and liabilities) are also part of the pattern descriptions.

This paper is structured as follows: Section 2 describes the background of manual analysis of running systems. Section 3 gives a motivating example in the context of a robot system. In Section 4, we present the pattern language. In Section 5, we apply the presented pattern language to the motivating example of Section 3 to demonstrate its applicability. We conclude in Section 6.

2 BACKGROUND: MANUAL ANALYSIS OF RUNTIME EVENTS USING DESIGN MODELS

When speaking about manual analysis of running systems using design models, we assume that an analyst is interested in some runtime characteristic that belongs to an element found in one of the models that were used during the design of the observed system. An example of such a runtime characteristic would be the average runtime of an action found in a UML activity diagram. However, many other runtime characteristics are imaginable, like the average response time of a communication path between two components in a component diagram or the overall runtime that is spent within a component.

Performing an analysis task on the model level can be split into several subtasks: The models of the system must be created or derived from its implementation, the necessary runtime events that belong to a model element of interest must be recoded by the running system (e.g., the execution of a modelled behaviour), the events must be traced back to the model elements they belong to, and analysis operations must be specified based on the recorded events (e.g., the average or maximum runtime of such execution events). According to those subtasks, the model elements and the monitored runtime events (and thus, the running system itself) can be seen as logically connected, as depicted in Figure 1.

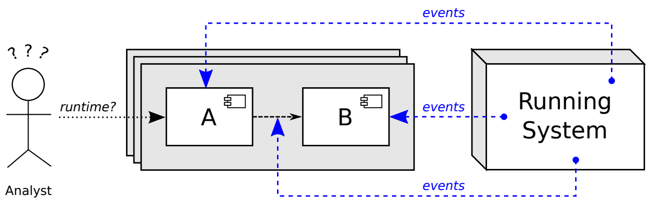


Figure 1: Abstract view of relating runtime events with model elements

In a setting with a perfect separation of concerns, only the last subtask would be of concern to the analyst who wants to analyze the behaviour of the running system. All other subtasks would be the concern of the software architects and developers who must prepare the system to allow such analyses. However, in a traditional development environment (i.e., without models that are connected to the captured runtime events via navigable traceability links), this separation cannot be applied since the analyst must have profound technical knowledge to identify the implementation parts that belong to the model element of interest, analyze the existing code, write the monitoring logic that yields the necessary events and deploy the written code to the running system. After analyzing the events, the analyst may need to repeat this human-in-the-loop approach iteratively to narrow down the analysis, thus constantly switching between various levels of abstraction. We presented a code generation and analysis approach [68] based on a reusable set of event types [56] which lifts some of the burden from the analyst's shoulders.

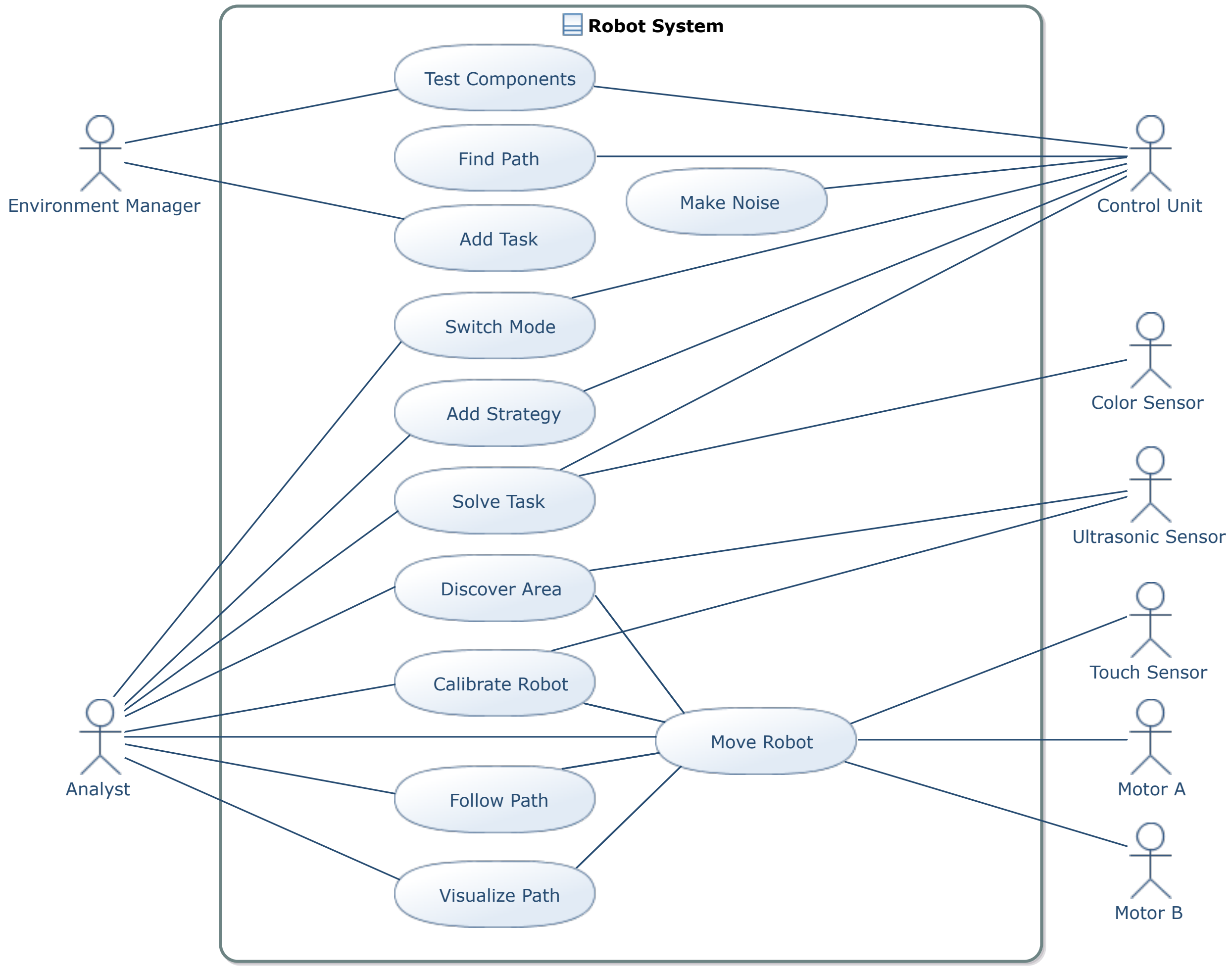


Figure 2: Uses cases of the robot system that will be used as motivating example

However, although there exist approaches that ease the work for the analyst, software architects and developers are still unsupported when designing the system with manual analysis in the first place. They are confronted with several design decisions concerning the modeling, introspection, traceability and analysis aspects of such systems, and chances are high that possible solution alternatives to design issues are assessed unaware of the consequences if appropriate guidance does not exist. As a consequence, this paper derives a pattern language from architecture and design concepts found in the literature which helps architects and developers to balance the trade-offs between the various solution alternatives. The pattern language also highlights interrelationships between the captured patterns so that software architects and developers are able to assess the impact of their design decisions onto other areas of manual analysis.

3 MOTIVATING EXAMPLE

Consider a scenario where a robot system is developed with various control and self-management capabilities. An overview of the desired features is shown in the use case diagram in Figure 2. The actors on the left side of the figure represent the roles that interact with the system. The actors on the right show the hardware units and sensors that participate in the modelled use cases. The robot consists of a main control unit, a color sensor to identify the color of encountered surfaces, an ultrasonic sensor to measure the distance to forward-facing objects, a touch sensor to detect collisions and two motors to drive and rotate the robot.

The main responsibilities of the robot are to follow definable paths, to drive in orthogonal directions according to external manual input, and to discover the environment on its own to create a digital grid-based map. To make the discovery process of the environment more interesting, tasks may appear spontaneously while driving through the terrain, simulated by the detection of specific colours on the ground. Such tasks must be solved either by manual user input on the control unit or by using an autonomous solution strategy known to the robot. Adding such tasks to the robot system and testing the hardware setup is done by the environment manager, while the rest of the operations (e.g., solving a task manually or adding a solution strategy to occurring tasks) are performed by the analyst.

The challenge is now to enable the analyst to observe the behaviour of the robot since it is hard to recognize what the robot is actually doing just by looking at it (e.g., when the robot is driving, it is not immediately clear if it is on an autonomic trip or follows a predefined path). Furthermore, the analyst should be able to perform self-defined queries on recorded events, for example to determine the overall time the robot spends on backtracking during the discovery process to get an insight into the efficiency of the underlying discovery algorithm. Moreover, the analyst should not be concerned with the implementation details of the robot, but instead perform the analysis tasks based on the abstract representations (i.e., the models) of its structure and behaviour, which hide a great deal of the technical realization.

Setting up a system that enables the analyst to perform such analyses leads to various design decisions for the architects and developers who build the system. These decisions are not limited to the implementation of the manual analysis environment, but must also cover architectural considerations regarding constrained resources, network outages and platform limitations. In the next section we present the pattern language which captures such decisions in the form of patterns, but we will come back to the robot scenario when applying the pattern language to it in Section 5.

4 PATTERN LANGUAGE FOR MANUAL ANALYSIS OF RUNTIME EVENTS

4.1 Pattern Language Overview

The pattern language is based on existing literature that documents the usage of important concepts in the context of using models at runtime, like patterns, modeling habits, languages, middlewares and development techniques. More concretely, we inspected the content and references of our comprehensive systematic literature review of the objectives, techniques, kinds and architectures of models at runtime [58] and our paper on reusable event types [56].

We divided the pattern language into four categories based on the way how the manual analysis of running system using design models is performed, as described in Section 2: The models of the system must be created and managed by the modeling environment, the running system must be inspected, the recorded events must be related to the model elements and the analyst must assess the recorded information by applying aggregation operations to the events. This results in four categories of patterns: Modeling, introspection, traceability and analysis patterns. Their interrelationship is shown in Figure 3.

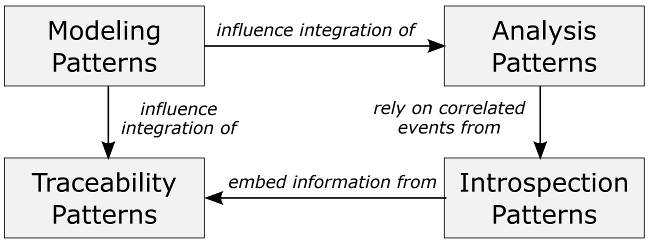


Figure 3: Overview of the pattern language for manual analysis of runtime events

- **Modeling patterns** are concerned with the issues of implementing and using the modeling environment the analyst interacts with, choosing a model syntax for describing the system under observation, and choosing the kinds of models that are linked to runtime events and ultimately used for the analysis. Since these patterns mainly influence how the analyst interacts with the modeling environment (and thus, the running system), the main force that drives the selection of specific patterns is usability, but analyzability and extensibility are important in this category too.
- **Introspection patterns** address the issues of recording events, storing the recorded events and exchanging them between the running system and the modeling environment where the analysis is performed. Patterns of this category have different consequences regarding the integration of monitoring instructions into the implementation of the running system, the ability of performing post-mortem analyses of the system and the up-to-dateness of analysis results in the modeling environment.
- **Traceability patterns** address the issue of implementing the traceability mechanism in the monitoring code that yields the runtime events. The traceability patterns are utilized by a subset of introspection patterns to embed additional information into the recorded runtime events so the modeling environment can correlate runtime events with the model elements they originate from.
- **Analysis patterns** address the issues of processing events in the modeling environment by formulating aggregation operations on the recorded events. In addition, patterns of this category address the integration of analysis tasks into the models that are used by the analyst.

The following subsections describe the patterns that belong to these four categories in detail. Every category is organized into the aforementioned issues so related patterns can be discussed together regarding the forces that shape the respective issues. Pattern discussions contain references to related papers that were identified in the literature [56, 58] and adopt the respective pattern.

4.2 Modeling Patterns

Choosing between the various modeling patterns mainly affects how the modeling environment of the analyst influences the overall architecture of a system. We divided the topic into the modeling environment issue, the model syntax issue and the model kind issue. Figure 4 shows the relationships of modeling-related patterns discussed in this section between themselves and to patterns of other categories which will be discussed later on.

4.2.1 Modeling Environment. The issue regarding the modeling environment addresses the kind of connection between the modeling environment and the running system. Decisions regarding this issue mainly influence performance characteristics, flexibility and the degree of coupling between the components of the resulting system. The decision of how to handle the modeling environment issue ultimately breaks down to the question of how much workload should be shifted to either the modeling environment or the running system. Decisions about this issue are subject to three forces:

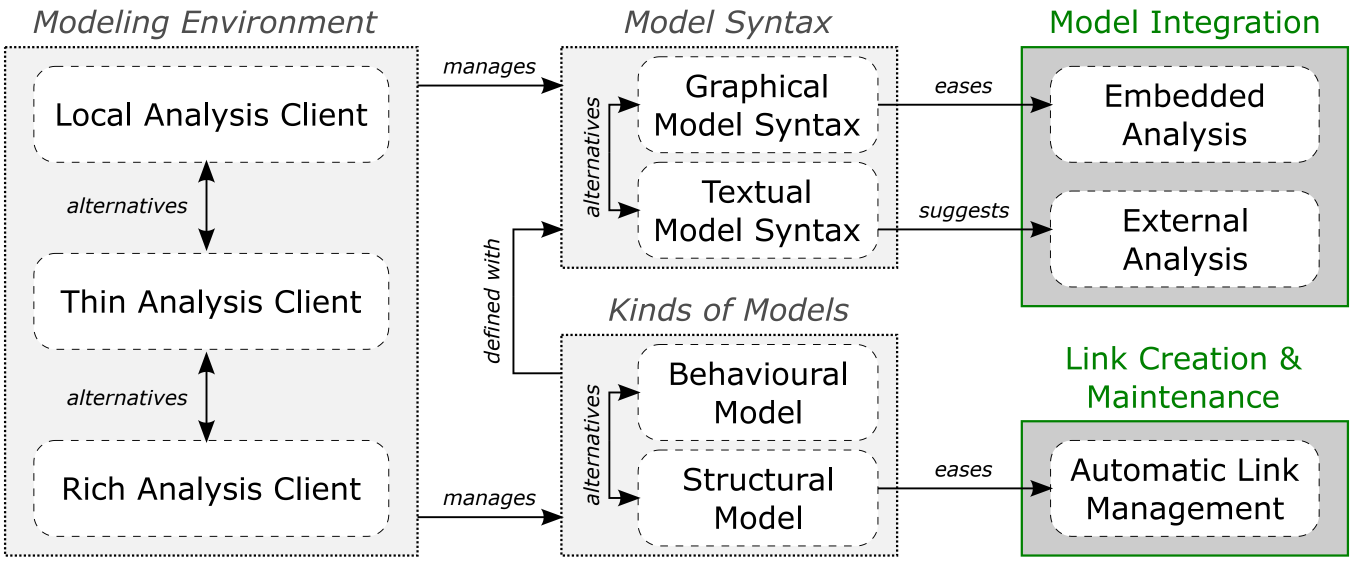


Figure 4: Overview of important relationships between modeling-related patterns and patterns of other categories

Forces for Issue: Modeling Environment

- *Dependability:* The observed system should have minimal extra dependencies.
- *Extensibility:* The modeling environment should be extensible in such a way that analysis tasks can be integrated easily while reusing existing models.
- *Scalability:* Even with a large number of analysts and captured events, the system and analysis performance should hardly be affected.

A very simple setup is the deployment of both the modeling environment and the running system on the same machine. In this setup the analyst utilizes a LOCAL ANALYSIS CLIENT to interact with the running system, which means that communication mechanisms do not need to be applied. The analyst is able to directly operate with the models in the modeling tool that was used during the development of the running system, for example the Eclipse Modeling Framework.

Pattern: Local Analysis Client

Context: The analyst wants to analyze the behaviour of a running system on the model level. The models of the system already exist and were created using a well-established modeling tool.

Problem: You want to connect the modeling environment to the running system and reuse the models, i.e. you do not want to modify the existing models just for the sake of performing the analysis. You want to keep the concerns related to modeling and analysis separated, meaning that required extensions to the modeling environment do not affect the observed system.

Solution: Use a local client, meaning that the modeling environment (i.e., the application that the analyst interacts with) and the system under observation are ultimately running on the same machine, possibly even in the same machine process. The modeling environment can directly access and adapt the running system according to the current monitoring requirements and consume the runtime events yielded by the running system. This setup also allows the analyst to use modeling frameworks which are originally not designed to interact with running systems (e.g., the Eclipse Modeling Framework).

Consequences: Benefits & Liabilities

+ *Dependability:* The observed system has no additional dependencies to modeling frameworks and tools.
+ *Extensibility:* Reuse of existing modeling frameworks is possible. In addition, adapting the running system to changed monitoring requirements is easy.
+ *Scalability:* It is a performance-friendly solution without much overhead.
– *Scalability:* Multiple analysts cannot observe the running system at the same time. Furthermore, the analyst needs both the running system and a full-fledged modeling environment, which hinders a realistic deployment.

Known Uses

- Garzon and Cebulla [27] use state machine models in a local modeling environment to perform simulations for user interaction analyses.
- Simulink is a commercial modeling tool providing a modeling environment which simulates systems locally before moving to hardware deployments.

- LabVIEW is a commercial tool for creating data flow models of embedded systems and performing local simulations using those models.

The other extreme is the pattern of using a THIN ANALYSIS CLIENT, meaning that the analyst does not operate within a full-fledged modeling environment, but is only provided with minimal interaction capabilities (e.g., a browser-based interface) which present the models and display the analysis results. In this setup, the models the analyst interacts with must be hosted somewhere, either by a separate server or the running system itself, which adds complexity in terms of the overall software architecture. The latter case may be problematic if the system under observation operates in an environment with reduced footprint or limited execution capabilities, like in the area of embedded systems.

Pattern: Thin Analysis Client

Context: Analysts want to analyze the behaviour of a running system on the model level. The models of the system already exist and are stored centralized or directly at the observed system. Multiple analysts want to observe the running system at the same time.
Problem: You want to enable the analysts to display the models and connect them to the running system while not being tightly coupled to it. You want the analysts to use modeling environments without the need of local installation routines and minimal dependencies to ease the maintenance and reduce the technical complexity encountered by the analysts.
Solution: Use a thin client where the analysts are only provided with minimal interaction capabilities (e.g., a browser-based interface) which present the models and display the analysis results. The models and analysis results can directly be assembled by the system under observation.

Consequences: Benefits & Liabilities

+ *Scalability:* Multiple analysts can observe the running system at the same time. Analysts have minimal setup requirements and maintenance effort.
– *Dependability:* The running system needs additional dependencies like modeling frameworks and web server libraries.
– *Extensibility:* Reusing or extending existing modeling frameworks is not possible (or at least limited). Two modeling environments must be maintained: the full-fledged environment for creating models at design time, and a reduced one for (remotely) displaying the models and analysis results at runtime.

Known Uses

- Kevoree is a modeling tool for creating arbitrary models of distributed system nodes [22]. It has a web-based version without setup efforts for the analyst.
- GenMyModel is a web-based, collaborative modeling platform for multiple meta-models [17]. It has no setup efforts for the analyst, but lacks appropriate visualizations of analysis results.
- Signavio is a commercial web-based modeling tool for managing business processes. It has no setup efforts for the analyst and also supports simulations.
- Wegmann and Wirz [64] implemented a browser-based interface supporting animation of states in domain diagrams. This solves the problem that deployed applications cannot be analyzed using the domain models (which were created in a local Eclipse-based modeling environment) any more after generating source code from them.

A certain middle ground between the local and the thin client solutions is to use a RICH ANALYSIS CLIENT. In this pattern, the analyst is able to utilize a local modeling environment, but the system under observation is not restricted to run on the same machine. As a consequence, this setup does not require the running system to pull in additional dependencies, but is still required to offer interfaces for receiving new monitoring instructions (i.e., adaptation instructions) and sending recorded events. However, the analysis of the recorded runtime events (e.g., the application of aggregation operations) is now performed by the modeling environment instead of the running system, which means that some of the performance concerns are shifted from the running system to the modeling environment.

Pattern: Rich Analysis Client

Context: Analysts want to analyze the behaviour of a running system on the model level. The analysts utilize local modeling environments and want to observe and aggregate the events of a remote running system using already existing models. Multiple analysts want to observe the running system at the same time.
Problem: You want to enable the analysts to connect the models to the running system and perform aggregation operations on a possibly large number of events. However, at the same time you want that neither the performance, nor the amount of external dependencies of the observed system are negatively affected by the analyses performed by the analysts.
Solution: The analysts utilize local modeling environments, but the system under observation is not restricted to run on the same machine. The analyses of the recorded runtime events (e.g., the application of aggregation operations) are performed by the modeling environments of the analysts instead of the running system.

Consequences: Benefits & Liabilities

+ *Dependability:* The observed system has no additional dependencies to modeling frameworks.
+ *Extensibility:* Reuse of existing modeling frameworks is possible, i.e. the effort of implementing a separate model editor does not exist. In addition, adapting the running system to changed monitoring requirements is easy.

+ *Scalability:* It is a performance-friendly solution allowing multiple analysts to observe the system. Aggregating events is done by the modeling environment, which increases the performance of the observed system.
– *Dependability:* The running system needs additional dependencies like web server libraries to allow remote access to the captured events. This ultimately leads to security considerations too.
– *Scalability:* Performance concerns are shifted from the running system to the modeling environment and especially their connection since all events must be transferred to the modeling environment for analysis.

Known Uses

- Alférez and Pelechano [1] use architecture models hosted in Eclipse. The models are evolved by a context monitor which observes the remotely running system.
- Almorsy et al. [2] enforce security specifications at runtime through aspect-oriented techniques according to a security model in a local modeling environment.
- Fouquet et al. [22] discuss a graphical simulator which is pluggable on applications as debugger/monitor.

4.2.2 *Model Syntax.*

While the used model syntax does not directly influence the overall architecture of the system, it has an impact on the analysis capabilities that are offered to the user. The decision about the model syntax is mainly driven by the usability requirements of the analyst and involves the selection of the right modeling tools which offer the necessary degree of customization.

Forces for Issue: Model Syntax

- *Accessibility:* The models used for the analysis should be easily editable, usable and readable for analysts with variable preferences and assistive needs. Textual models can easily be processed by machines, but usability may be improved when using graphical models for analyzing the running system.
- *Understandability:* The models used for the analysis should assist the analyst in understand the behaviour of the running system.
- *Usability:* The models should enable the analyst to directly interact with the running system and modify the analysis properties. Graphical models are intuitive abstractions for analyzing the running system, but hinder the applicability of accessibility-related editor features.

There are two types of model syntaxes: A GRAPHICAL MODEL SYNTAX describes models with the help of customized nodes, edges and their properties. The Unified Modeling Language (UML) diagram types are prominent examples which fall into this category. The other type is TEXTUAL MODEL SYNTAX where models are described with textual definitions that follow a formal grammar. Prominent examples are architecture description languages, which sometimes provide both types of syntax. Choosing a model syntax mainly depends on the usability requirements of the analyst and the customizability of the used modeling tools: editors for graphical models can often be adapted easily to fit the needs of the analyst, while textual models can easily be edited even without a specialized editor. Prominent tools for creating and editing models which also integrate with a wide range of other model-driven development tools are Eclipse Sirius (graphical) and Xtext (textual).

Pattern: GRAPHICAL MODEL SYNTAX

Context: The analyst wants to analyze the behaviour of a running system on the model level using a coherent interface for both the models and analysis results.
Problem: You want to enable analysts to perform their analysis tasks (e.g., the formulation of aggregation operations on captured events) directly in the model editor. You want a system where the analyst is not required to leave the modeling environment to perform the analysis.
Solution: Use a graphical modeling environment (e.g., Eclipse Sirius) that is customizable in a way such that additional information can be toggled on or off in separate layers containing custom model elements. This allows the analyst to examine recorded runtime events and formulate analysis expressions in the models without leaving the graphical environment.

Consequences: Benefits & Liabilities

+ *Understandability:* Graphical models can offer a quick overview of the system part under observation and integrate customized symbols and visualizations, e.g. to highlight faulty model elements that were detected during analysis.
+ *Usability:* Usability is enhanced, since analysis can be done directly in the models. There is no need for an external viewer for analysis results.
– *Accessibility:* Assistive tools like screen readers do not work. More generally, modeling frameworks which offer the required customizability are scarce.

Known Uses

- Fouquet et al. [22] discuss a graphical simulator which is pluggable on applications as debugger/monitor.
- Georgas et al. [28] use architectural models to manage runtime adaptations. Models are graphical and are enriched with architectural performance metrics.
- Simulink is a commercial modeling tool which enhances graphical models during simulation with additional runtime information.
- LabVIEW is a commercial tool for designing embedded systems. It highlights execution paths in graphical data flow models when starting a simulation.
- Signavio is a commercial web-based modeling tool for creating graphical BPMN diagrams. It highlights business process model activities with simulation data.

Pattern: Textual Model Syntax

Context: The analyst wants to analyze the behaviour of a running system on the model level. The modeling environment for the analysis of runtime events hosts the models and can be extended with custom views and controls.
Problem: You want to enable the analyst to display the analysis results in the modeling environment, but also want the used models to remain easy to edit and process.
Solution: Use a textual model editor to display models describing the system under observation, and use external views (i.e., separate visual areas in the modelling environment) for showing the analysis results. The results must have a reference to the model elements (e.g., their names) they belong to in order to make sense to the analyst. This solution ensures that models can be created and maintained with established text processing tools.

Consequences: Benefits & Liabilities

+ *Accessibility:* Text can easily be interpreted by various tools (screen readers, version management tools, etc.). There exists a vast amount of tools for building textual editors to edit the model of the observed system.
– *Understandability:* Integration of customized symbols and visualizations (and thus, analysis results) is more difficult than in graphical environments.
– *Usability:* External views lead to a fragmented user experience for analysts.

Known Uses

- PlantUML is an alternative to the graphical syntax used in UML diagrams. PlantUML also allows to convert diagram definitions to images (not graphical models).
- yUML is a textual language for defining UML diagrams that can easily be shared online. Diagrams are defined in an URL format.
- QuickDBD is a commercial online tool for drawing database diagrams using a textual syntax. QuickDBD also allows to convert diagram definitions to images (not graphical models).

4.2.3 Kinds of Models.

Similar to the model syntax, the kinds of models do not directly influence the overall architecture of the system, but they have certain control over the information that can be monitored from the running system and the model-driven techniques that can assist in relating runtime events with their corresponding model elements. The decision about the kinds of models to use for manual analysis is mainly driven by the amount of traceability links that can be generated by supplemental model-driven techniques.

Forces for Issue: Kinds of Models

- *Analyzability:* The models used by the analyst should enable a wide range of analysis possibilities for the running system.
- *Traceability:* The models used by the analyst should be traceable to their implementation counterparts to support the formulation of monitoring code. Structural models can easily be traced to their implementation, but the analyzability of modelled behaviours is mainly constituted by using behavioural models.

Structural models describe the static structure of the running system. Structural information is easy to process by model-driven development tools, especially in code generation approaches because entities describing structure can often be transformed directly into equivalent concepts of the target language the observed system is implemented in. As a result, the effort for writing monitoring code and the actual business logic can be reduced.

Pattern: Structural Model

Context: An analyst is expected to analyze the behaviour of a specific system on the model level in the future. The observed system is currently in the development phase and you decide which models to use for future analyses while implementing the system.
Problem: You want to keep the effort of implementing the manual analysis capabilities as small as possible. You want these capabilities to almost come for free when implementing the actual business logic of the observed system.
Solution: Use models that describe the structure of the system. Model-driven techniques like model transformation can generate the corresponding implementation and monitoring code for those models rather easily. It is very often the case that structural high-level model elements can be identified one way or another (e.g., via naming conventions) in the underlying implementation.

Consequences: Benefits & Liabilities

+ *Traceability:* Utilizing automatic traceability approaches like the ones from model transformation languages is easy. The resulting traceability links can be used to automatically generate monitoring code which yields the necessary events for the model element of interest.
– *Analyzability:* Many interesting runtime events are linked to behavioural model elements (e.g., the execution of modelled actions), not structural ones.

Known Uses

- Bauer et al. [7] transform UML deployment and class diagrams into a security monitor. Violations detected by the monitor are fixed using aspect-oriented techniques.
- In the approach of Gjerlufsen et al. [30], events recorded from a Nokia phone are related to graphical, structural models. Special overlays are directly embedded into the models which allows the analyst to inspect the events.
- Hamann et al. [39] present USE, a tool for monitoring programs with the help of class and object diagrams. Object diagrams are enriched with values extracted from the running systems via events.

Applying model-driven techniques to generate code is also possible for BEHAVIOURAL MODELS. However, generating business logic and monitoring code which adequately reflects the data and control flows described in the models is slightly more complex. This is especially true for models that are close to the problem space, i.e. where the conceptual gap to the actual implementation is large. Prominent examples of behavioural models for manual analyses are activity and state machine models.

Pattern: Behavioural Model

Context: An analyst is expected to analyze the behaviour of a specific system on the model level in the future. The observed system is currently in the development phase and you decide which models to use for future analyses while implementing the system.
Problem: You want to enable the analyst to perform sophisticated analyses on the model level, e.g. by annotating specific control flow paths with aggregation operations of corresponding events.
Solution: Use models that describe the behaviour of the system. They enable the analyst to track the execution states of the system under observation and pinpoint the location of problems in case of failures. This can be combined with graphical modeling environments to highlight control flow paths with symbols and additional information to gain insight into the behaviour of the running system.

Consequences: Benefits & Liabilities

+ *Analyzability:* Many interesting runtime events are linked to behavioural model elements (e.g., the execution of modelled actions).
– *Traceability:* Utilizing automatic traceability approaches like the ones from model transformation languages is hard because the generation of fully executable code from behaviour models is complex.

Known Uses

- Bodenstaff et al. [11] use activity models to check consistency constraints in organization cooperations. Consistency is checked by tracing events that are written to multiple logs.
- Garzon and Cebulla [27] use highlighted state machine models for user interaction analyses.
- Fouquet et al. [22] discuss a graphical simulator which is pluggable on applications as debugger/monitor. In the paper, they provide an example of simulating and debugging using state machines.
- Hamann et al. [39] present USE, a tool for monitoring programs with the help of class and object diagrams. Additional state machine and sequence diagrams are used to monitor sequence of operations [38].

For both STRUCTURAL MODELS and BEHAVIOURAL MODELS, many model transformation languages provide traceability information between the model elements and the generated output automatically [20, 46, 52, 62], thus simplifying the feedback of runtime events to their corresponding model elements. However, the same is not quite as easy for other kinds of models which do not directly reflect the structure or behaviour of the system. Examples of such models are role-based access control (RBAC) models, architectural decision models and goal models [31, 63, 65, 67]. For those kinds of models, finding suitable runtime events and corresponding counterparts in the implementation is much harder and can hardly be performed automatically. The more the models describe concepts that are different from the structure and behaviour of the system, the more handiwork is needed to relate the model elements with their corresponding implementation counterparts.

4.3 Introspection Patterns

Choosing between the various introspection patterns mainly affects how runtime events are extracted from the running system, stored and exchanged with the modeling environment. Consequently, we divided the topic into the event recording issue, the event storage issue and the event exchange issue. Figure 5 shows the relationships of introspection-related patterns discussed in this section between themselves and to patterns of other categories which will be discussed later on.

4.3.1 Event Recording. The event recording issue is concerned with the monitoring of runtime events in the most unobtrusive way as possible, meaning that the formulation of monitoring logic should be as independent as possible from the actual business logic of the running system. This independence ensures that the system can dynamically be adapted with new versions of monitoring code while it is up and running. The decision of how to record events is mainly driven by the desired degree of flexibility and the offered services of the used execution environment and middleware technologies.

Forces for Issue: Event Recording

- *Adaptability:* The monitoring code should be changeable at runtime according to changing monitoring requirements. Event recording strategies with high orthogonality are usually also easily adaptable since the business logic needs no changes.
- *Flexibility:* The recording should not be restricted to certain types of events.
- *Modularity:* The monitoring code should be implemented in separate modules. High modularity coincides with high orthogonality since monitoring instructions can be formulated separately from the business logic.
- *Orthogonality:* The recording logic should not interfere with the business logic. High orthogonality ensures high adaptability since changes to the recording logic are isolated from the business logic and can be managed separately at runtime.

One common technique to perform event recording in a crosscutting manner is to use ASPECT-ORIENTED PROGRAMMING (AOP [47]). The monitoring logic is implemented in separate modules named aspects and woven into the actual business logic by a special compiler. Most of the AOP implementations only support the application of monitoring aspects during compile time without the possibility of weaving newly introduced aspects into the running

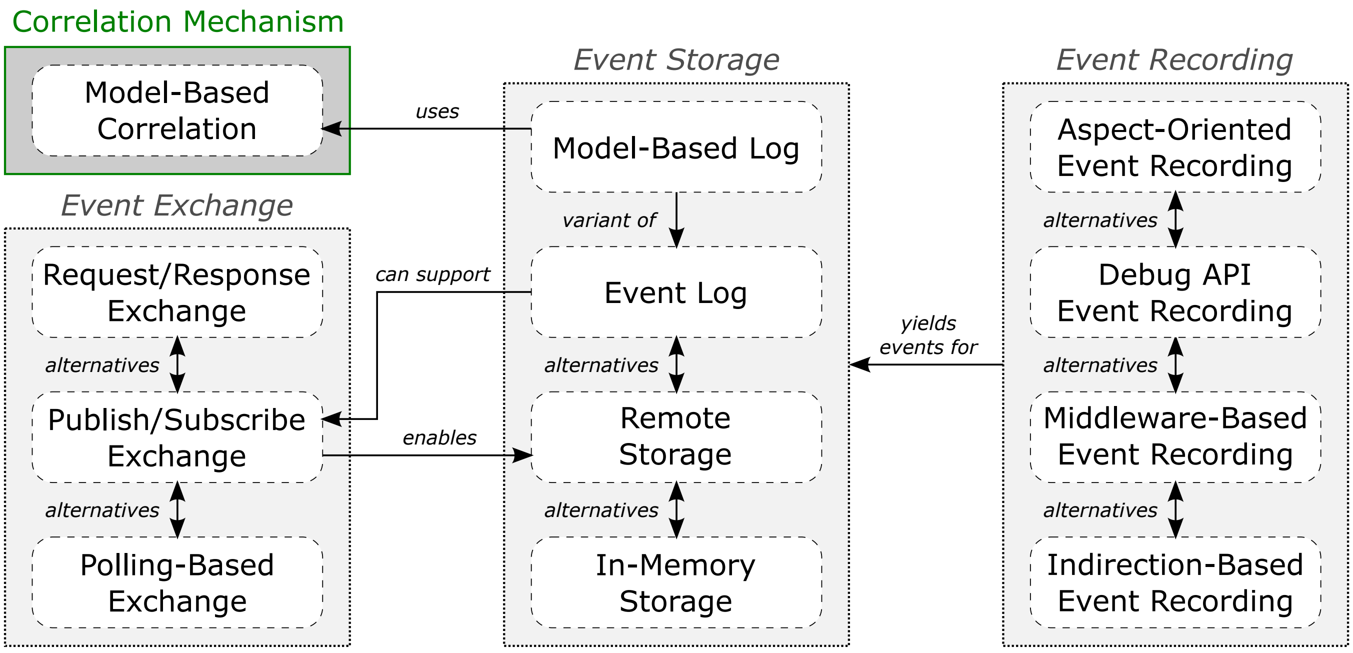


Figure 5: Overview of important relationships between introspection-related patterns and patterns of other categories

system. One possibility of applying aspects at runtime is to weave a generic aspect at system startup which intercepts all possible events and then checks if an intercepted event conforms to the monitoring criteria which can be exchanged at runtime. This solution is flexible but subpar when it comes to performance since every possible event must be intercepted and checked at runtime. Another option is to use an aspect weaver which natively supports the adaptation of aspects at runtime [23].

Pattern: Aspect-Oriented Event Recording

Context: The analyst wants to analyze the behaviour of a running system on the model level. You decide how events should be recorded while the observed system is running. The system under observation is implemented using object-oriented principles.
Problem: Changes to the actual business logic are not possible or forbidden. Nevertheless, you want the monitoring logic to record essential parts of the business logic without violating the separation of concerns principle.
Solution: With aspect-oriented programming (AOP), monitorig instructions are written in separate components called aspects without changing the actual implementation of the system. The final system is assembled by a special compiler which weaves the monitoring instructions into the actual implementation. Dynamic aspect weavers natively support the adaptation of aspects at runtime.

Consequences: Benefits & Liabilities

+ *Flexibility:* Arbitrary monitoring code can be executed at predefined points of the execution.
+ *Modularity:* Aspects allow modular programming of monitoring code.
+ *Orthogonality:* The recording logic does not interfere with the business logic.
– *Adaptability:* Although possible, performance overhead must be considered if aspects are changed at runtime.

Known Uses

- Amoui et al. [3, 4] use aspect-oriented techniques to probe the environment of the observed system and propagate state changes to a runtime model. Adaptations are based on constraints analyzed on this runtime model.
- Arcaini et al. [5] use AspectJ to record events of the observed system and trace them to model elements of a state machine for conformance monitoring.
- The approach of Krüger et al. [48] relies on generating AspectJ monitors from sequence models. The monitored events are used to check if interactions sequences in a distributed system conform to the modelled behaviour.
- Kieker is a framework for continuous monitoring of software services [61]. It relies on AspectJ to record events during execution (here: messages that are exchanged between components), which are then grouped to derive complete execution traces. Such traces can be analyzed using generated sequence diagrams.

Alternatives to aspect-oriented event recording are debug API and middleware-based event recording which both address the utilization of interfaces that are provided by the execution environment or communication middlewares used by the system under observation. Examples of such interfaces are the debug interface of the Java Virtual Machine (JDI) and the message interceptors of the Common Object Request Broker Architecture (CORBA). While those interfaces provide an unobtrusive access to runtime events (i.e., the system implementation needs no changes) and allow to adapt the monitoring conditions at runtime, the operations are rather low-level or limited to the event types offered by the respective middleware.

Pattern: Debug API Event Recording

Context: The analyst wants to analyze the behaviour of a running system on the model level. You decide how events should be recorded while the observed system is running. The system under observation is implemented using a virtual machine.
Problem: Changes to the actual business logic are not possible or forbidden. Nevertheless, you want the monitoring logic to record essential parts of the business logic without violating the separation of concerns principle. You do not want no introduce additional dependencies by utilizing features of the virtual machine.
Solution: Utilize the debug interfaces that are provided by the execution environment used by the system under observation to extract events. This interface provides an unobtrusive access to runtime events (i.e., the system implementation needs no changes) and allows to adapt the monitoring conditions at runtime.

Consequences: Benefits & Liabilities

+ *Adaptability:* Monitoring is adaptable without the need of pre-processing like weaving.
+ *Orthogonality:* The recording logic does not interfere with the business logic.
– *Flexibility:* The captured event types are restricted and rather low-level.
– *Modularity:* Monitoring code structure is arbitrary, modularity is optional.

Known Uses

- Various approaches exist which make use of the Joint Test Action Group (JTAG) debugging interface to capture events [33, 34, 55, 69]. The recorded events are traced back to highlighted state machine models where the system can be analyzed in a graphical manner.
- Hamann et al. [37] utilize the Java Debug Interface (JDI) to relate events to their corresponding UML behavioural models. The recorded events are used in combination with OCL constraints for conformance analysis.
- Keil μVision is a commercial debugger using the JTAG interface to trace events of embedded programs.

Pattern: Middleware-Based Event Recording

Context: The analyst wants to analyze the behaviour of a running system on the model level. You decide how events should be recorded while the observed system is running. The system under observation is implemented using a communication middleware.
Problem: You want the monitoring logic to record essential parts of the business logic without directly interfering it. Although the system operates in a distributed setting, you want to comply with the separation of concerns principle and introduce no additional dependencies for recording the events.
Solution: Utilize the interfaces that are provided by the middleware used by the observed system to extract events. Examples of such an interface are the message interceptors provided by the CORBA architecture.

Consequences: Benefits & Liabilities

+ *Adaptability:* Monitoring is adaptable without the need of pre-processing like weaving.
+ *Orthogonality:* The recording logic does not interfere with the business logic.
– *Flexibility:* Middleware interceptors often operate on a low-level and make high-level analyses harder. Captured event types are restricted to the ones that are observable through the middleware interceptors.
– *Modularity:* Monitoring capabilities are tightly coupled to the used communication middleware. A change in the used technologies for the business logic might invalidate existing monitoring code.

Known Uses

- Bertolino et al. [9] present GLIMPSE, a middleware to capture events according to a custom event meta-model. Events are combined to metrics for quantifiable observations of the system.
- Gamez et al. [24] present FamiWare, a middleware to record and distribute events in a publish/subscribe manner. Recorded events are traced to architecture and feature models for further analyses and reconfiguration actions.
- ReMMoC is a middleware for mobile client interoperability using CORBA-based events [32].

Another possibility of recording events in a modular way is to make deliberate use of indirection patterns when implementing the system under observation [25]. Examples of such patterns are the proxy pattern and the decorator pattern, which can both be used to dynamically attach or detach monitoring instruction to the actual business logic, thus modifying the event recording conditions at runtime. This indirection-based event recording pattern ensures flexibility in an early stage of development, but is not simple to set up because it is hard to decide beforehand which parts of the system to design for the sake of adaptability of the monitoring instructions. Since the monitoring requirements of an analyst cannot easily be foreseen during the development phase, the extreme case is to model every single component and operation with the help of indirection patterns, which ultimately leads to a system whose implementation is very flexible but very hard to understand.

Pattern: Indirection-Based Event Recording

Context: The analyst wants to analyze the behaviour of a running system on the model level. You decide how events should be recorded while the observed system is running. The system under observation is implemented using object-oriented principles.

Problem: You want the monitoring logic to record runtime events of the business logic, but at the same time you want the monitoring logic to be adaptable without the use of additional libraries or pre-execution steps.

Solution: Cope with dynamic event recording conditions at runtime by making a deliberate use of indirection patterns when implementing the observed system. Indirection patterns like the proxy and the decorator pattern allow to change the behaviour (and thus, the monitoring instructions) of components and operations while the system is up and running.

Consequences: Benefits & Liabilities

+ *Adaptability:* Monitoring is adaptable with native language features.
+ *Flexibility:* Flexibility regarding dynamically changing monitoring requirements is ensured in an early stage of the software development phase.
+ *Modularity:* Modularity is an inherent goal of established design patterns.
– *Flexibility:* The monitoring requirements of an analyst cannot easily be foreseen during the development phase.
– *Orthogonality:* A speculative overuse of indirection patterns can lead to an implementation that is harder to understand.

Known Uses

- In the approach of Bertolino et al. [9], events are recorded using either code injection or the proxy design pattern.
- Log4j is a prominent, de-facto standard framework for logging runtime events using the abstract factory pattern. It can also be combined with other event yielding and reporting backends using the popular Java library named Metrics.
- Hibernate is an often used persistence framework which provides runtime events.

4.3.2 Event Storage. The event storage issue is concerned with the temporal availability and storage formats of recorded events. The decision drivers for this issue are the performance of the overall system, the up-to-dateness of the analysis results and the reliability of the system in case of failures:

Forces for Issue: Event Storage

- *Analyzability:* The event storage should ensure the up-to-dateness of analysis results and enable further processing of recorded events.
- *Availability:* The event storage should be accessible by the modeling environment and allow retrospective analysis of recorded events. Availability may contradict analyzability since storing events for retrospective analysis adds complexity for keeping analysis results up-to-date.
- *Efficiency:* The distribution of runtime events should be performance-friendly for both the modeling environment and the observed system. Efficiency may contradict availability since writing events to a persistent log (needed for retrospective analysis) with a high frequency introduces noticeable I/O overhead.

One pattern to solve this issue is to read and write an event log which stores the recorded runtime events. Such a persistent solution allows that the recorded events may outlive the uptime of the system under observation, meaning that the recorded events can be analyzed even after the running system was shut down. The concurrent access of reading (by the modeling environment) and writing (by the running system) the event log must be managed and the up-to-dateness of the analysis results must be ensured by polling the event log or using notification mechanisms. Log files are an important source of information when analyzing the behaviour of a running system [45].

Pattern: Event Log

Context: Runtime events are recorded by the observed system. The modeling environment needs those events to perform aggregations and obtain analysis results.

Problem: You not only want to perform analyses on the recorded events, but also introduce a temporal decoupling between the event recording at runtime and the analyses performed by analysts. You want give the analysts the freedom to perform analyses at a later stage with tools and model environments of their choice, even after the observed system was halted.

Solution: Read and write an event log which holds the recorded runtime information. Such a persistent solution allows that the recorded events may outlive the uptime of the system under observation, meaning that the recorded events can be analyzed even after the running system crashed or was shut down on purpose.

Consequences: Benefits & Liabilities

+ *Analyzability:* Depending on the log format, the events can be further processed by many tools.
+ *Availability:* Post-mortem analysis of the observed system is possible.
– *Analyzability:* Concurrent access to the event log (writing by the observed system, reading by the modeling environment) must be managed. As a consequence, getting a live view of the events introduces additional complexity.
– *Efficiency:* Writing and reading events with a high frequency is problematic.

Known Uses

- Holmes et al. [41] utilize business process execution engine logs as basis for analyses to identify compliance violations. Violations are detected by tracing the events to process models and associated compliance rules.
- Bodenstaff et al. [11] use activity models to check consistency constraints in organization cooperations. Consistency is checked by tracing events that are written to multiple logs.
- Garzon and Cebulla [27] analyze user interactions by logging XML-based events to trace execution states.
- Related: Splunk is a commercial tool for synthesizing event logs from multiple sources and generating reports to help analysts in assessing the system health.
- Simulink is a commercial modeling tool for performing simulaions. Events captured during simulations are lost after ending the simulation.
- LabVIEW is a commercial tool for designing embedded systems and performing simulations. Events captured during simulations are lost after ending the simulation.

An alternative to the persistent solution of using an EVENT LOG is to order the system under observation to keep the recorded events in an IN-MEMORY STORAGE only. This solution is superior regarding performance, but prone to crashes and shutdowns since the recorded events are essentially lost if the system shuts down unexpectedly.

Pattern: In-Memory Storage

Context: Runtime events are recorded by the observed system. The modeling environment needs those events to perform aggregations and obtain analysis results.
Problem: You want a high-performance solution where recorded events are transferred to and processed by the modeling environment as requested. You not only want to perform analyses on the recorded events, but also introduce a temporal decoupling between the event recording at runtime and the analyses performed by analysts.
Solution: Instruct the system under observation to keep the recorded events in memory only. The retrieval of the recorded events must be performed on an on-demand basis initiated by the modeling environment, which means that the running system must be equipped with a suitable interface.

Consequences: Benefits & Liabilities

+ *Efficiency:* The performance is better than in persistent approaches.
– *Analyzability:* The complexity of managing the concurrent access to events is shifted to the observed system. Furthermore, existing offline tools (version management, search, etc.) cannot be applied to the events.
– *Availability:* Post-mortem analysis of the observed system is not possible.
– *Efficiency:* The observed system must store events even if no modeling environment is connected. This introduces additional complexity, e.g. through the need of a round-robin storage mechanism for memory management.

Known Uses

- See Section 5 for a possible manifestation of this pattern.

Another possibility is to watch the recorded events in a REMOTE STORAGE located in the modeling environment. The observed system installs the necessary monitoring logic (see the aforementioned patterns on event recording) for connected modeling environments and publishes the requested events when they arise. This strategy can be combined with the aforementioned IN-MEMORY STORAGE pattern, meaning that the running system keeps the recorded events in memory as long as no observer is connected, but switches to live mode when being observed by an external modeling environment.

Pattern: Remote Storage

Context: Runtime events are recorded by the observed system. The modeling environment needs those events to perform aggregations and obtain analysis results.
Problem: You want a high-performance solution where recorded events are transferred to and processed by the modeling environment as quickly as possible. In addition, it is desired that the observed system experiences no overhead if no modeling environment is currently connected.
Solution: Instruct the running system to not record any events as long as no modeling environment is connected. When a modeling environment connects to the system under observation and transmits its monitoring requirements, the system installs the necessary monitoring logic and publishes the requested events immediately to the modeling environment when they arise.

Consequences: Benefits & Liabilities

+ *Efficiency:* The performance is better than in persistent approaches. No overhead if no modeling environment is connected to the running system.
+ *Availability:* Analysis results can be kept up-to-date without the need of constantly polling the system.
– *Analyzability:* The complexity of managing the concurrent access to events is shifted to the observed system. Furthermore, existing offline tools (version management, search, etc.) cannot be applied to the events.
– *Availability:* The support of post-mortem analysis must be managed individually by each modeling environment.

Known Uses

- See Section 5 for a possible manifestation of this pattern.

A special case of using an EVENT LOG is to store events in a MODEL-BASED LOG. The model-based event log itself can then be further processed with model-driven techniques like model transformation to gain additional insights into the course of events of the running system.

Pattern: Model-Based Log

Context: Runtime events are recorded by the observed system. The modeling environment needs those events to perform aggregations and obtain analysis results.
Problem: You not only want to perform analyses on the recorded events, but also introduce a temporal decoupling between the event recording at runtime and the analyses performed by analysts. You want give the analysts the freedom to perform analyses at a later stage and also enable additional analyses of the recorded events with existing model-driven development tools.
Solution: Store events in a model which conforms to a separate event log meta-model. Such a model can have references to model element instances of other meta-models, which means that the traceability links between runtime events and the model elements of interest can directly be encoded into the event log.

Consequences: Benefits & Liabilities

+ *Analyzability:* Events can be processed by model-driven development tools.
+ *Availability:* Post-mortem analysis of the observed system is possible.
– *Analyzability:* Concurrent access to the event log (written by the observed system, read by the modeling environment) must be managed. As a consequence, getting a live view of the events introduces additional complexity.
– *Efficiency:* Writing and reading events with a high frequency is problematic.

Known Uses

- Mayerhofer et al. [49] present an approach where the behaviour of a system is tracked using events conforming to a separate event meta-model. Recorded events are directly related to nodes of a UML activity diagram.
- Dongen and van der Aalst [18] propose a meta-model for event logs using a custom XML-based format. Recorded events are used for process mining to extract knowledge and models from the system for further analyses.
- Tax et al. [60] present a similar approach for process mining using the XES event log meta-model.

4.3.3 *Event Exchange.*

The issue of exchanging events between the modeling environment and the running system is closely related to the techniques of storing events. Related patterns are concerned with the point of time when the event source is checked for new runtime events. If new runtime events are available, the modeling environment has to reapply the filter and aggregation instructions defined by the analyst to refresh the displayed analysis results.

Forces for Issue: Event Exchange

- *Efficiency:* The distribution of runtime events should be performance-friendly for both the modeling environment and the observed system.
- *Responsiveness:* The event exchange strategy should allow the analyst to keep the analysis results up-to-date. High responsiveness demands high efficiency when a large amount of events is distributed and processed by the modeling environment.
- *Scalability:* Even with a large number of observing analysts and captured runtime events, the event exchange performance should hardly be affected. High scalability demands high efficiency when a large amount of events is distributed and processed by the connected clients.

Events can be exchanged in a REQUEST/RESPONSE EXCHANGE style, which means that the modeling environment receives the recorded events of the running system on an on-demand basis. This strategy is compatible with the IN-MEMORY STORAGE pattern discussed above.

Pattern: Request/Response Exchange

Context: Runtime events are recorded by the observed system. The observed system is ready to transmit those events to the modeling environment to enable the analysis of its behaviour.
Problem: You want a simple solution where the observed system offers the events to connected modeling environments without introducing an overhead for the observed system with respect to keeping track of connected modeling environments.
Solution: Exchange events between the observed system and the modeling environment in a request/response style, meaning that the modeling environment receives recorded events of the running system on an on-demand basis.

Consequences: Benefits & Liabilities

+ *Efficiency:* Easy to implement in a synchronous manner, for both the modeling environment and the running system.
– *Responsiveness:* Up-to-dateness is not guaranteed by single requests.
– *Scalability:* Performance and storage considerations are necessary if the amount of captured runtime events per request is large.

Known Uses

- See distributed system approaches [59].

Another possibility is a POLLING-BASED EXCHANGE of events, which means that that a periodic request for data takes place. This can be applied in combination with EVENT LOGS and MODEL-BASED LOGS by regularly requesting their stored runtime event entries.

Pattern: Polling-Based Exchange

Context: Runtime events are recorded by the observed system. The observed system is ready to transmit those events to the modeling environment to enable the analysis of its behaviour.

Problem: You want the observed system to offer the events to connected modeling environments without keeping track of connected environments. On the other hand, you want the analysis results to be updated regularly in the modeling environment to track the system behaviour.
Solution: Exchange events between the observed system and the modeling environment by regularly requesting the stored runtime events. This strategy can be applied in combination with the event log and model-based log patterns.

Consequences: Benefits & Liabilities

+ *Efficiency:* Easy to implement in a synchronous manner, for both the modeling environment and the running system.
+ *Responsiveness:* The analysis results can be kept up-to-date.
– *Responsiveness:* The polling interval decides the up-to-dateness of the analysis results, which is domain-dependent.
– *Scalability:* Polling is concerned with performance considerations.

Known Uses

- See enterprise integration patterns [40].

Another alternative is the PUBLISH/SUBSCRIBE EXCHANGE pattern where the modeling environment registers itself at the source of events and is immediately notified when new runtime events arrive [40, 59]. This strategy can be applied in combination with the REMOTE STORAGE pattern, but also together with EVENT LOGS if they provide subscription functionality (e.g., offered by databases).

Pattern: PUBLISH/SUBSCRIBE EXCHANGE

Context: Runtime events are recorded by the observed system. The observed system is ready to transmit those events to the modeling environment to enable the analysis of its behaviour.
Problem: You want the observed system to distribute recorded runtime events to the connected modeling environments as quickly as possible. You want to ensure that all those environments always have up-to-date analysis results, thus providing a live view of the system behaviour through a constant stream of events.
Solution: Instruct the modeling environments to register themselves at the source of events (i.e., the running system or an event log) so they are immediately notified when new runtime events arrive. In this setup the running system is not enforced to store the events.

Consequences: Benefits & Liabilities

+ *Efficiency:* Performance- and memory-friendly solution since the observed system is not required to store events.
+ *Responsiveness:* Analysis results are always up-to-date, and stream-based processing of events is possible.
+ *Scalability:* Transmitting single events to multiple observers is fast.
– *Scalability:* The observed system has an overhead since it must manage multiple connected modeling environments (e.g., send events to all environments, manage timeouts and unexpected disconnects, memorize which events are still pending for which modeling environment, etc.).

Known Uses

- Various approaches utilize the publish/subscribe exchange functionalities of middlewares to distribute recorded runtime events [14, 15, 24, 26].
- Caporuscio et al. [14, 15] present an approach where events are routed from publishers to architectural and performance models of subscribers to enable model-based analysis and trigger system reconfigurations.
- See the Publish-Subscribe Channel messaging pattern used in enterprise integration [40]. There are also concrete examples on the accompanied web site using RabbitMQ[1] and Apache Kafka[2].

4.4 Traceability Patterns

Choosing between the various introspection patterns mainly affects how to relate runtime events with their corresponding model elements of the modeling environment. We divided the topic into traceability link creation and maintenance issue and the correlation mechanism issue. Figure 6 shows the relationships of traceability-related patterns discussed in this section between themselves and to patterns of other categories.

4.4.1 Link Creation & Maintenance. The link creation issue is concerned with the establishment of traceability links between recorded runtime events and their corresponding model elements. Link maintenance deals with the prevention of the connected model elements and runtime events drifting apart. The solution alternatives are similar to the traceability link creation strategies. Inconsistencies between the model elements and runtime events are mainly introduced by changes to the model without making the appropriate changes to the monitoring code.

Forces for Issue: Link Creation & Maintenance

- *Productivity:* Traceability links should be created and maintained easily by a developer without noticeable increase in effort.
- *Traceability:* The established traceability links should enable a broad range of analysis possibilities for the analyst. A high degree of traceability decreases productivity if many traceability links must be created and maintained manually.

[1] enterpriseintegrationpatterns.com/patterns/messaging/Filter.html
[2] enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html

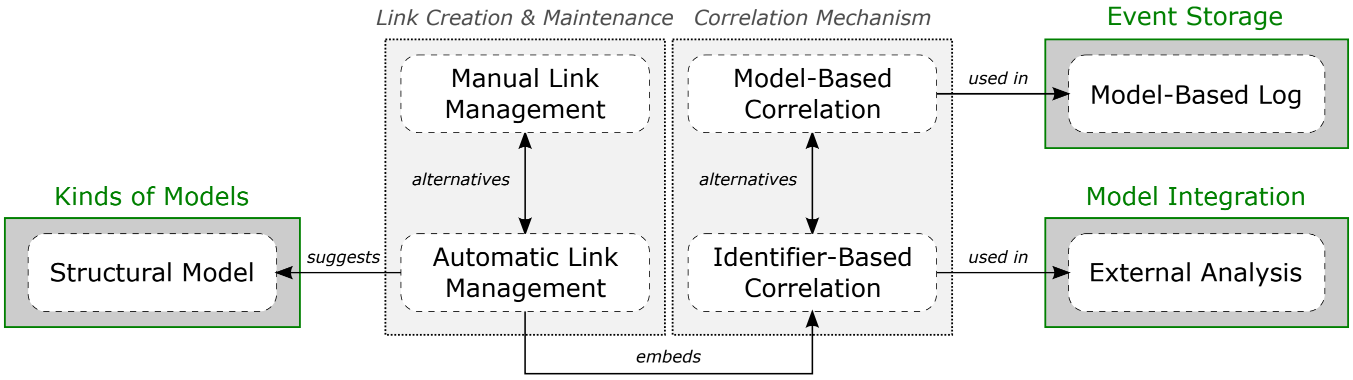


Figure 6: Overview of important relationships between traceability-related patterns and patterns of other categories

One simple strategy is to perform MANUAL LINK MANAGEMENT, i.e. to write the monitoring logic manually so it yields the appropriate events that are relevant for the model elements of interest: The developer must follow existing traceability links (if available), analyze the corresponding implementation and write the necessary monitoring code. For link maintenance, the procedure must be repeated, the monitoring code updated by hand and the system redeployed.

Pattern: Manual Link Management

Context: The analyst wants to analyze the behaviour of a running system on the model level. You decide how to relate recorded runtime events with the models they originate from while implementing the observed system without the help of model-driven techniques.
Problem: You want to relate events and their associated model elements, but the observed system is implemented without the help of model-driven development techniques, which means that the models and the corresponding code live in isolation.
Solution: Write the monitoring logic manually so it yields the appropriate events that are relevant for the model elements of interest. This is a flexible strategy for systems where the relevant parts of the implementation code can quickly be found and changed, if necessary.

Consequences: Benefits & Liabilities

+ *Traceability:* Flexible since the monitoring code can be written carefully for the current analysis task at hand.
– *Productivity:* Manual effort is large, so manual link management is only feasible for small systems.

Known Uses

- Every traceability approach where automatic link management is not performed.

The contrast to the manual strategy, traceability links between runtime events and model elements can also be generated automatically by using model transformation [20, 46, 52, 62]. In this AUTOMATIC LINK MANAGEMENT pattern, a code generator takes the model elements of interest as input and inject references to those model elements into the generated monitoring code. When the generated monitoring code is executed, the runtime events are instantiated with the model references as arguments and transmitted to the modeling environment which can close the feedback loop by relating the events with the model elements they originate from. For maintaining the traceability links in case of model updates, new versions of the monitoring code must be re-generated from the updated models and deployed while the system is up and running. This action also incorporates the undeployment of the old monitoring code.

Pattern: Automatic Link Management

Context: The analyst wants to analyze the behaviour of a running system on the model level. You decide how to relate recorded runtime events with the models they originate from while implementing the observed system with the help of model-driven techniques.
Problem: You want to relate events and their associated model elements. You want to keep the effort of implementing the traceability mechanism as small as possible.
Solution: Integrate a code generator into the existing model-driven development workflow to generate the monitoring code from the model elements of interest. The generator can inject references to those model elements into the generated monitoring code. When the monitoring code is executed, the runtime events are instantiated with the model element references as arguments and transmitted to the modeling environment to close the feedback loop.

Consequences: Benefits & Liabilities

+ *Productivity:* Effort reduction is achieved through code generation.
+ *Traceability:* Many existing model transformation languages have integrated traceability support.
– *Productivity:* Higher initial effort for writing the code generator in the first place. Long-term effort reduction can only be achieved if the generator is written in a generic, reusable way. Code generators are often written for a specific source meta-model, and thus limited in their reuse.

Known Uses

- Jakumeit et al. [44] present a comprehensive comparison of model transformation tools and languages. Many of the presented model transformation languages have built-in traceability support with specialized metamodels (e.g., ATL, Epsilon, QVT, Viatra).
- Amoui et al. [3, 4] present an approach where traceability links are generated from code annotations [3, 4].
- Bai et al. [6] present an approach where sensors are generated from workflow and service models. Recorded events from these sensors are used to analyze the behavior of services.

A middle ground between the two patterns is to establish the traceability links in a semi-automated manner, which means that some traceability links originate from the generated code and some links are established in manually written monitoring code. This is necessary if some models used for the analysis describe concepts that are not directly related to the structure and behaviour of the system.

4.4.2 Correlation Mechanism. While the establishment of traceability links can be done either manually, automatically or semi-automatically, the identities of the corresponding model elements must be somehow encoded into the runtime event entries.

Forces for Issue: Correlation Mechanism

- *Correctness:* Correlated events and model elements should stay correlated even if models are evolved.
- *Interoperability:* The correlation information should be stored in a way such that it can easily be processed by other tools.
- *Traceability:* Recorded runtime events and their associated model elements should be linked directly. Model-driven tools can ensure correctness by updating the links between runtime events and their model elements in case the model changes.

One common pattern to do this is IDENTIFIER-BASED CORRELATION where an artificial identifier is introduced which is unique for every model element. This identifier is encapsulated in the properties of an event and can be used by the modeling environment to reconstruct the references to the related model elements. We used the identifier-based approach in our previous work on manual analysis with reusable event types [56], but it is also a well-known pattern in enteprise integration [40] and distributed systems in the form of tokens [13].

Pattern: IDENTIFIER-BASED CORRELATION

Context: The implemented monitoring code detects an execution state of interest and yields an event. You decide a strategy of how to relate the event to the model elements of interest in the modeling environment of the analyst.
Problem: You want to encode the identities of the corresponding model elements into the recorded runtime events so that the modeling environment can close the feedback loop between events and model elements, but the identities are required to be unique for each model element so the feedback is unambiguous.
Solution: Introduce an artificial identifier which is unique for every model element. This identifier is passed to the instantiation routine of runtime events (i.e., in the generated or manually written monitoring code) and can be used by the modeling environment to reconstruct the references to the related model elements. The artificial identifier can be as simple as a hash value of the model element name or the fully-qualified name of the model element within the model.

Consequences: Benefits & Liabilities

+ *Traceability:* The implementation of a unique identifier is simple and already performed internally by some modeling frameworks.
– *Correctness:* The validity of captured events is prone to changes to the referenced model elements. Changes to the model elements (e.g., renaming) entail that the model elements and the recorded runtime events drift apart, which means that events cannot be associated with their model elements anymore.

Known Uses

- In the already mentioned approach of Amoui et al. [3], events are traced to activity diagrams using the fully-qualified name of methods for disambiguation.
- Haberl et al. [36] present an approach where monitoring code is generated from component language (COLA) models. The generated monitoring code yields special names in binary recorded events for disambiguation.
- In the approach of Holmes et al. [41], standardized Universally Unique Identifiers (UUIDs) are assigned to business process models. These UUIDs are embedded in monitoring codes generated from process and compliance models.

Another possibility to relate model elements and runtime events is to use MODEL-BASED CORRELATION with the help of an EVENT LOG, as described in Section 4.3.2.

Pattern: MODEL-BASED CORRELATION

Context: The implemented monitoring code detects an execution state of interest and yields an event. You decide a strategy of how to relate the event to the model elements of interest in the modeling environment of the analyst.
Problem: You want to encode the identities of the corresponding model elements into the recorded runtime events so that the modeling environment can close the feedback loop, but changes to models must not invalidate already recorded correlated events.
Solution: Use a model-based event log where log entries reference the corresponding model elements of the model

directly without the need of artificial identifiers. Modeling frameworks can automatically update such cross-references between models in case a model changes.

Consequences: Benefits & Liabilities

+ *Correctness:* The approach is robust against small-scale model changes.
+ *Interoperability:* Existing modeling tools and libraries can be applied to the model-based event log.
+ *Traceability:* Links between model elements and captured events are directly stored in the model-based log.
– *Correctness:* Profound changes to the model elements like deletion still corrupt already existing traceability links. Furthermore, the approach requires a separate runtime event meta-model which must be created and maintained.

Known Uses

- Mayerhofer et al. [49] present an approach where the behaviour of a system is tracked using events conforming to a separate event meta-model. Recorded events are directly related to nodes of a UML activity diagram.
- Dongen and van der Aalst [18] propose a meta-model for event logs using a custom XML-based format. Recorded events are used for process mining to extract knowledge and models from the system for further analyses.
- Tax et al. [60] present a similar approach for process mining using the XES event log meta-model.

4.5 Analysis Patterns

Choosing between the various analysis patterns mainly affects how to further process recorded events and integrate the results into the models. Consequently, we divided the topic into the event processing issue and the model integration issue. Figure 7 shows the relationships of analysis-related patterns discussed in this section between themselves and to patterns of other categories.

4.5.1 Event Processing. Since the amount of recorded runtime events can be very large, filter and aggregation mechanisms are required so that the analyst is able to obtain manageable insights into the behaviour of the running system. The decision of how the further process events is mainly driven by the expressiveness of the filter and aggregation operations.

Forces for Issue: Event Processing

- *Analyzability:* The language for defining aggregation instructions should be tailored to process streams of runtime events.
- *Expressiveness:* The language for defining aggregation instructions should enable a broad range of analyses in a concise manner.
- *Usability:* From the analyst's point of view, the formulation of aggregation instructions should be ease to accomplish.

A widely used pattern for filtering and aggregating data is to store the data in a database and perform DATABASE-BASED PROCESSING with the help of specialized query languages. The most prominent language for querying databases is the Structured Query Language (SQL).

Pattern: Database-Based Processing

Context: Events were recorded by the running system and transmitted to the storage location. You want to enable the analyst to specify aggregation operations on these events to gain meaningful insights into the behaviour of the running system.
Problem: You want the analyst to be able to apply aggregation operations without the need of learning a new language. Furthermore, you do not want to spend effort on implementing a special editor for the analysts so they can write their queries.
Solution: Use a database for storing runtime events. Databases provide the widely used language SQL for filtering and aggregating data. There exists a vast amount of SQL tools and editors which support the formulation of correct SQL queries.

Consequences: Benefits & Liabilities

+ *Expressiveness:* SQL is a language that is widely used and well-understood by many analysts. It has a wide range of predefined aggregation operations.
– *Analyzability:* SQL is not optimal for analyzing time series of events. SQL-like event processing languages are better suited for this task.
– *Usability:* Existing SQL editors are likely external editors which are not integrated into the modeling environment that is operated by the analyst.

Known Uses

- Widespread use, namely whenever an event of a running system is stored in a database.

An alternative is the usage of COMPLEX EVENT PROCESSING (CEP) systems which offer expressive mechanisms to handle time series of events and data. There are three broad classes of systems to process complex events with a varying degree of performance and scalability [16].

Pattern: Complex Event Processing

Context: Events were recorded by the running system and transmitted to the storage location. You want to enable the analyst to specify aggregation operations on these events to gain meaningful insights into the behaviour of the running system.
Problem: You have high performance demands and want to enable the analyst to apply aggregation operations on a constant stream of events.
Solution: Use complex event processing (CEP) systems which offer expressive mechanisms to handle time series of events and data. There are three broad classes of systems to

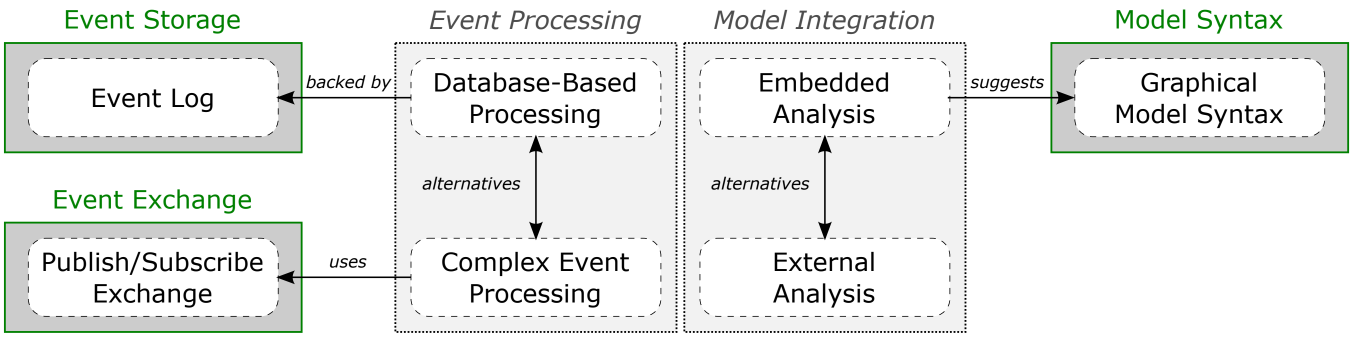


Figure 7: Overview of important relationships between analysis-related patterns and patterns of other categories

process complex events. Publish/subscribe-based CEP systems in particular are characterized by high performance and high scalability.

Consequences: Benefits & Liabilities

+ *Analyzability:* SQL-like event processing languages are well suited for analyzing streams of events. CEP systems promise to deliver high performance and scalability for a large amount of events.
+ *Expressiveness:* CEP utilizes well-known aggregation operations from SQL.
– *Usability:* Existing CEP editors are likely external editors which are not integrated into the modeling environment that is operated by the analyst. Tool support for CEP systems is not as mature as, for example, the support for SQL tools and editors. Although many CEP systems are based on SQL, CEP-specific dialects must be learned by the analyst.

Known Uses

- Fabret et al. [19] present a high-performance publish/-subscribe system with limited query abilities. Observed systems can utilize this approach to record events and filter them with predicates defined in the subscription.
- Various approaches utilize SQL-inspired stream databases. Demers et al. [12, 16] present Cayuga, a general purpose event monitoring system which uses SQL as query language, extended with special expressions to filter, fold and temporally relate events. A similar system is proposed by Motwani et al. [51] where SQL-like queries are continuously applied to event streams instead of using one-time queries of traditional databases.
- A similar approach is followed by Wu et al. [66], using an SQL-like pattern matching mechanism with limited semantics to perform analyses on event streams.

A third alternative is to develop a custom language which allows to analyze the stream of events. We did this in our previous work on reusable event types [56] by introducing a stream-based language that is inspired by functional reactive programming [53]. The language allows to filter and aggregate streams of events in a concise way that can directly be integrated into graphical modeling environments. Although a custom language can be tailored to event streams by providing event-specific and chronological notations, implementing a custom language is accompanied with an initial effort. For better usability, the custom language should be integrated into existing modeling tools, which results in additional effort. Another example is AQL, which is a SQL-based language that can be used to perform queries on architectural models [43].

4.5.2 *Model Integration.*

The issue of how to integrate the analysis results into the models is closely related to the issue of which model syntax to use. The decision about the integration of analysis results into models is mainly driven by the usability requirements of the analyst and involves the selection of the right modeling tools and frameworks which offer the necessary degree of customization.

Forces for Issue: Model Integration

- *Customizability:* The analyst should be supported by customized symbols and decorations to model elements according to current analysis results.
- *Usability:* The analyst should have a seamless user experience when interacting with both the models and the analysis aspects. High usability can be ensured with modeling tools which offer a high customizability.

Many graphical modeling environments are customizable in a way such that existing model elements can be altered in their appearance or additional information can be toggled on or off in separate layers, which means that analysis results can be integrated directly into the models to enable an EMBEDDED ANALYSIS. This allows the analyst to examine analysis results and formulate analysis expressions directly in the models. We followed this strategy in our previous work on reusable event types [56].

Pattern: EMBEDDED ANALYSIS

Context: Events were recorded by the running system and transmitted to the storage location. You want to enable the analyst to specify aggregation operations on these events and decide for a strategy to integrate these operations into the modeling environment.

Problem: You want the analyst to be able to formulate aggregation operations in a very direct way. You want a coherent user experience without requiring the analyst to leave the model environment.
Solution: Use a graphical modeling environment that is customizable in a way such that existing model elements can be altered in their appearance or additional information can be toggled on or off in separate layers, which means that analysis results can be embedded directly into the models. This allows the analyst to examine analysis results and formulate analysis expressions directly in the models without leaving the graphical environment.

Consequences: Benefits & Liabilities

+ *Customizability:* Alternate symbols and visualizations are possible based on the current analysis results.
+ *Usability:* Usability is enhanced since analysis is done directly in the models.
– *Customizability:* Modeling frameworks which offer the required customizability are scarce, especially for textual model editors.

Known Uses

- Georgas et al. [28] use architectural models to manage runtime adaptations. Architectural performance metrics are embedded into those graphical models for further analysis [28].
- In the approach of Gjerlufsen et al. [30], events recorded from a Nokia phone are related to graphical, structural models. Special overlays are directly embedded into the models which allows the analyst to inspect the events.
- Various approaches exist which make use of the Joint Test Action Group (JTAG) debugging interface to capture events and trace them back to related graphical model elements [33, 34, 55, 69]. Analysis results are embedded directly into the models, e.g. by highlighting special execution paths in a state machine model.
- Haberl et al. [36] propose model-level debugging of embedded real-time systems. Similar to the approaches above, analysis results are directly embedded into the models via highlighting of model elements.

The same functionality may be harder to integrate into textual editors, meaning that additional EXTERNAL ANALYSIS views and editors may be necessary to display analysis results and formulate analysis tasks.

Pattern: External Analysis

Context: Events were recorded by the running system and transmitted to the storage location. You want to enable the analyst to specify aggregation operations on these events and decide for a strategy to integrate these operations into the modeling environment.
Problem: You want the analyst to be able to formulate aggregation operations with the help of specialized editors. These editors possibly already exist, so you want them to be integrated into the overall analysis experience.
Solution: Separate the model editor from the query editor used for the formulation of aggregation operations. This strategy allows to utilize a modeling environment and a query editor arbitrarily that fits the abstract description of the observed system and the analysis task best. However, the events that are analyzed in the query editor must still somehow be connected to the model elements in the model editor, e.g. via unique identifiers as described in the identifier-based correlation pattern.

Consequences: Benefits & Liabilities

+ *Customizability:* The syntax and properties of both the query language and the modeling language can be chosen freely.
– *Usability:* Separating the model editor from the query editor can lead to a fragmented user experience for analysts. The analyst is burdened with the mental task of correlating query results from the query editor with model elements of the model editor.

Known Uses

- WindView is a visualization tool which is able to import events gained by instrumenting a running system [35]. Events (and thus, execution traces) are recorded using a XML-based format, the Instrumentation Interface Format (IIF).
- In the aforementioned approach by Haberl et al. [36], runtime events are traced to state machine models, but can also be observed in an external tabular view.
- The Event Viewer of the Windows operating system is often used as external analysis tool to perform post-mortem analyses of systems. The external viewer is primarily accessed by C/C++ programs.

5 APPLICATION TO THE MOTIVATING EXAMPLE

In this section, we apply our pattern language to the robot system example from Section 3. We will focus on a single model to demonstrate the event aggregation performed by the analyst. However, the same procedure can also be applied to other models of the observed system as well. Note that the robot system has actually been implemented using the selected patterns mentioned in this section during a software architecture course on the University of Applied Sciences Wiener Neustadt.

Figure 8 shows the deployment of the resulting solution and the connection between the modeling environment and the robot system. The figure shows that the pattern RICH ANALYSIS CLIENT was chosen as interface for the analyst. This has various reasons: The system was implemented in a model-driven manner, which means that parts of the implementation were directly generated from the models of the system with the help of the Eclipse modeling environment. Reusing those models (and the modeling environment) prevented the effort of implementing a separate model editor

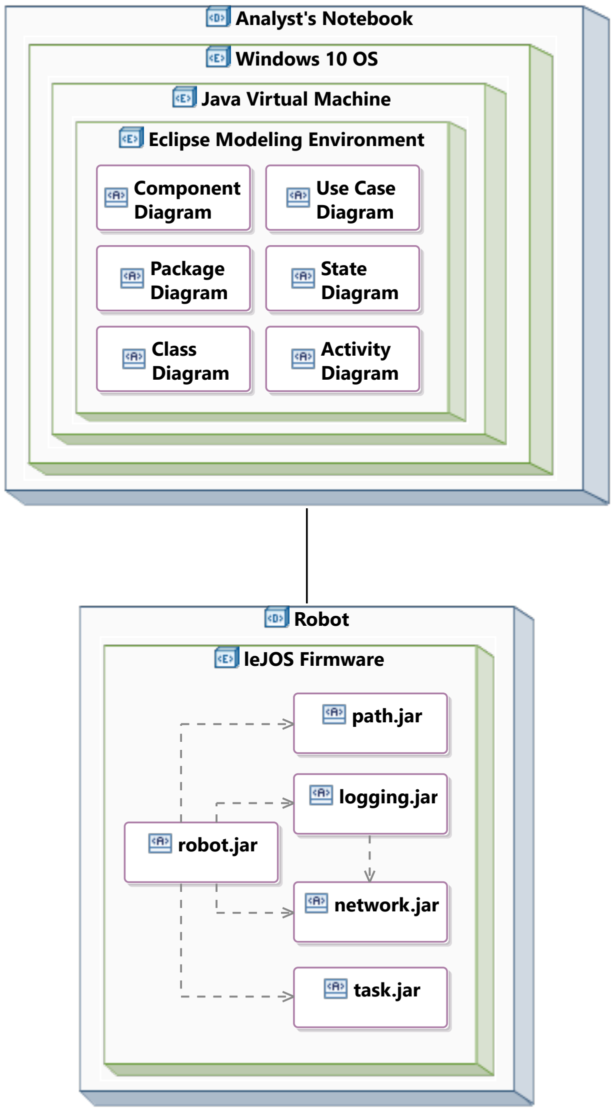


Figure 8: Deployment diagram showing the connection between the modeling environment and the robot system

for the analyst to remotely analyze the robot system. Furthermore, events were aggregated in a separate Eclipse plugin in the modeling environment, which rendered additional performance considerations about the robot and its reduced footprint unnecessary, even for a large number of events. The figure also shows that the patterns GRAPHICAL MODEL SYNTAX (UML), STRUCTURAL MODEL and BEHAVIOURAL MODEL were applied.

Regarding introspection, events were recorded by the robot using the ASPECT-ORIENTED EVENT RECORDING PATTERN with the help of AspectJ[3], an efficient and feature-rich Java-based aspect-oriented programming framework. This was possible due to the fact that the underlying robot firmware leJOS[4] is a tiny Java virtual machine that is able to execute arbitrary Java code. Recorded events were exchanged with the modeling environment using a combination of the REMOTE STORAGE and PUBLISH/SUBSCRIBE EXCHANGE patterns, meaning that the robot performs no event logging until the modeling environment subscribes to the robot. From this point on, events were immediately sent to the modeling environment, which could be done efficiently using the internal Java-based (de-)serialization mechanisms of Eclipse and leJOS. This strategy enabled the robot to operate at full performance and at minimum memory requirements while not being observed by a modeling environment. Furthermore, the application of the PUBLISH/SUBSCRIBE EXCHANGE pattern allowed the analysis results to be kept up-to-date in the observing modeling environment of the analyst.

Traceability was achieved through the AUTOMATIC LINK MANAGEMENT pattern for the structural models and the MANUAL LINK MANAGEMENT pattern for the behavioural models. Regarding the correlation mechanism, the pattern IDENTIFIER-BASED CORRELATION was applied in combination with the automatic link management: AspectJ-based monitoring code was directly generated from the models (where possible) with the fully-qualified names of the participating model elements encoded into the monitoring code. When the monitoring code is executed, events are transmitted immediately to the modeling environment to close the feedback loop by using the fully-qualified name found in the event properties.

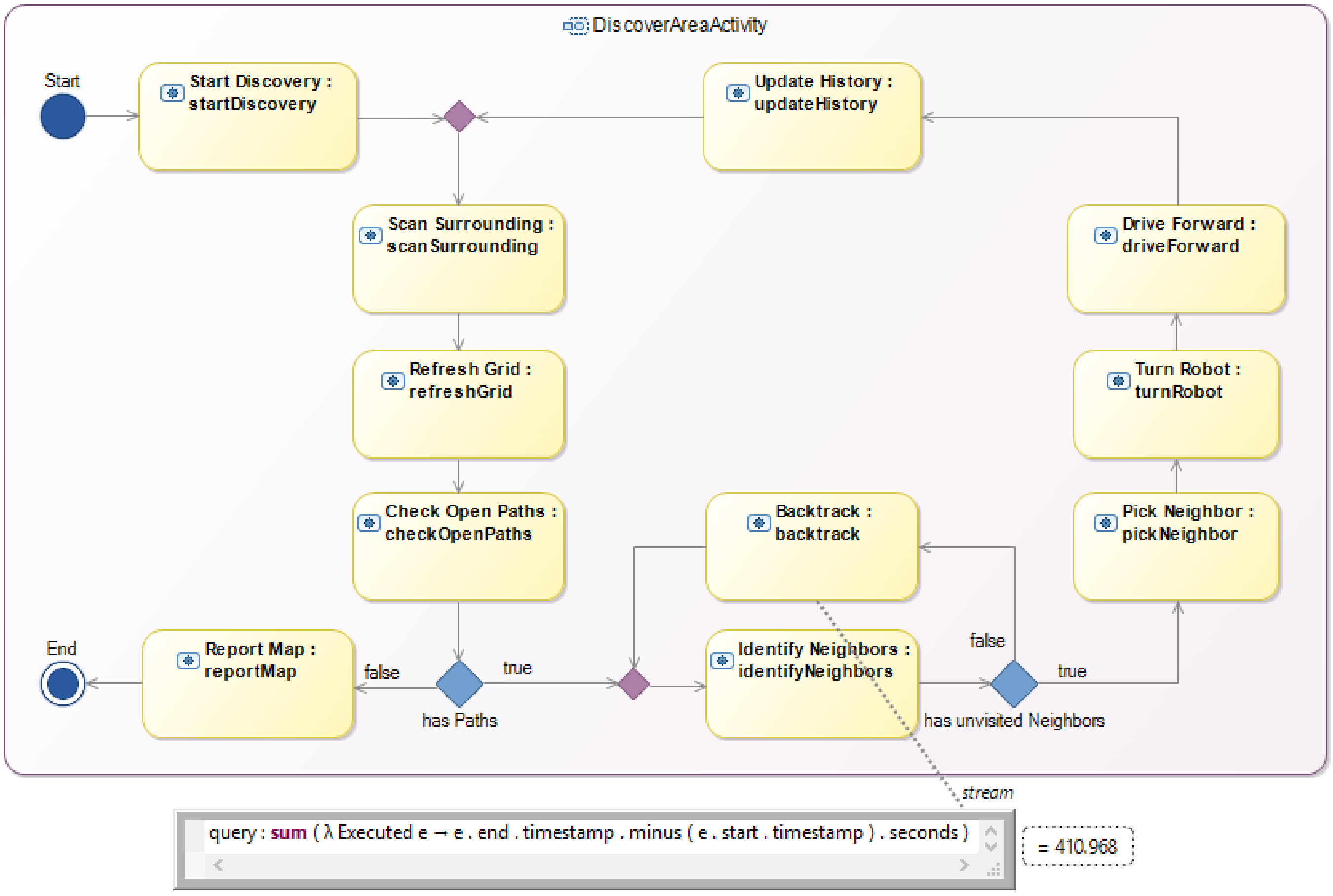
For the analysis, we decided to use the EMBEDDED ANALYSIS pattern and directly integrate the analysis results into the UML diagrams of the robot system. As an example, Figure 9 shows the activity diagram which describes the autonomous discovery feature of the robot. On the bottom of the figure, the backtrack action is annotated with an event aggregation statement to determine the overall time the robot spends on backtracking during the discovery process. As can be seen, the result of the operation is directly shown in the model. We neither used DATABASE-BASED PROCESSING nor COMPLEX EVENT PROCESSING for aggregating events, but instead a custom aggregation language which allows the analyst to filter streams of events (in this case, events of type *Executed* which has some properties regarding the start and end of an operation) in a concise manner without the need of a database. However, this requires that the analyst has a certain level of technical knowledge to formulate the queries in the provided language. This problem could be mitigated by tailoring the language to the analyzed domain and/or to the knowledge of the analyst. More details of the used aggregation language can be found in our previous paper on reusable event types [56].

Table 1 summarizes the patterns that were selected for realizing the robot system and describes the reasons why the patterns were selected. Note that the reasons largely comply with the context and problem descriptions of the patterns presented in this paper.

6 CONCLUDING REMARKS

Models are software engineering artefacts which try to cope with the increasing need of abstraction and flexibility through the use of model-driven development techniques. An increase in the level of abstraction is usually accompanied with an increasing interest in the analyses which can be performed closer to the problem space. This phenomenon can especially be observed in modern

[3]see https://www.eclipse.org/aspectj/
[4]see http://www.lejos.org/


DiscoverAreaActivity
Start
Start Discovery : startDiscovery
Update History : updateHistory
Scan Surrounding : scanSurrounding
Drive Forward : driveForward
Refresh Grid : refreshGrid
Turn Robot : turnRobot
Check Open Paths : checkOpenPaths
Backtrack : backtrack
Pick Neighbor : pickNeighbor
End
Report Map : reportMap
false
true
has Paths
Identify Neighbors : identifyNeighbors
false
true
has unvisited Neighbors
stream
= 410.968

query : sum (λ Executed e → e . end . timestamp . minus (e . start . timestamp) . seconds)

Figure 9: Demonstration of the embedded analysis pattern applied on a graphical model of the robot system

Table 1: Summary of the Selected Patterns and the Reasons for Choosing the Respective Patterns

Category	Selected Pattern	Reasons
Modeling	Rich Analysis Client	Remote analysis, relieve robot from load because of its reduced footprint.
Modeling	Graphical Model Syntax	Enable analysis with existing models, no effort for additional models.
Modeling	Structural Model	Transformations could easily be extended to generate monitoring code.
Modeling	Behavioural Model	Interesting robot properties are captured in the modelled control flows.
Introspection	Aspect-Oriented Event Recording	Monitoring logic is independent from business logic. High performance.
Introspection	Remote Storage	The robot must not have an overhead if the models are not connected.
Introspection	Publish/Subscribe Exchange	Analysis results must be up-to-date while the robot drives around.
Traceability	Automatic Link Management	Low effort of integrating traceability in existing model transformations.
Traceability	Manual Link Management	Monitoring code could not be fully generated from behavioural models.
Traceability	Identifier-Based Correlation	Straightforward implementation using naming conventions of EMF.
Analysis	Embedded Analysis	Analysis directly in the existing models. No external viewers required.

high-level programming languages, where programs are usually analyzed and debugged using the statements and expressions of the language the program is actually written in, not the intermediate language or machine instructions they are eventually compiled to. However, while many modeling languages exist, the design and implementation of their corresponding analysis infrastructure is accompanied with several design decisions regarding the modeling environment, the used introspection and traceability mechanisms and the analysis interface the analyst is interacting with.

There is a great amount of recurring patterns in the literature which tackle the various issues when designing a system with support for manual analysis using design models. Unfortunately, these patterns are seldomly labelled as such and discussed rather implicitly in the presented approaches. As a consequence, although the current literature contains the required knowledge about manual analysis, it is inconvenient, time-consuming and hardly usable in its current form for software architects and developers when designing and implementing a system with manual analysis support. Furthermore, the indexed scientific literature cannot be searched efficiently for names, properties and consequences of recurring patterns as well as their interrelationships.

The contribution of this paper is the extraction of these recurring patterns from the literature and the explicit description of their properties and trade-offs. Furthermore, the extracted patterns are organized in a pattern language so that software architects and developers are able to understand the interrelationships of patterns and the impacts of certain decisions on other parts of the system. For better usability, the pattern language presented in this paper is divided into four categories that provide a comprehensive coverage of the involved design decisions: Modeling, introspection, traceability and analysis. Within each category, related patterns are grouped according to the issues they are trying to solve to allow a better comparison between the patterns with respect to the underlying forces that must be balanced accordingly.

Even though the presented pattern language covers the most important aspects of manual analysis using design models, it could also be extended by including recurring patterns that can be found on a more fine-grained level when implementing such systems. Examples of such patterns are architectural and design patterns for realizing the model editor, the running system itself or patterns for deciding where to log which events. However, we argue that design decision on this level of granularity are covered adequately by the existing literature about architectural patterns [13, 40] and design patterns [25] which are also mentioned throughout this paper.

REFERENCES

[1] Germán H. Alférez and Vicente Pelechano. 2012. Dynamic evolution of context-aware systems with models at runtime. In *Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems (MODELS'12)*. Springer-Verlag, Berlin, Heidelberg, 70–86. https://doi.org/10.1007/978-3-642-33666-9_6

[2] Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. 2012. MDSE@R: model-driven security engineering at runtime. In *Proceedings of the 4th international conference on Cyberspace Safety and Security (CSS'12)*. Springer-Verlag, Berlin, Heidelberg, 279–295. https://doi.org/10.1007/978-3-642-35362-8_22

[3] Mehdi Amoui, Mahdi Derakhshanmanesh, Jürgen Ebert, and Ladan Tahvildari. 2011. Software Evolution towards Model-Centric Runtime Adaptivity. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering (CSMR '11)*. IEEE Computer Society, Washington, DC, USA, 89–92. https://doi.org/10.1109/CSMR.2011.14

[4] Mehdi Amoui, Mahdi Derakhshanmanesh, Jürgen Ebert, and Ladan Tahvildari. 2012. Achieving dynamic adaptation via management and interpretation of runtime models. *J. Syst. Softw.* 85, 12 (Dec. 2012), 2720–2737. https://doi.org/10.1016/j.jss.2012.05.033

[5] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2012. CoMA: conformance monitoring of java programs by abstract state machines. In *Proceedings of the Second international conference on Runtime verification (RV'11)*. Springer-Verlag, Berlin, Heidelberg, 223–238. https://doi.org/10.1007/978-3-642-29860-8_17

[6] Xiaoying Bai, Yongli Liu, Lijun Wang, Wei-Tek Tsai, and Peide Zhong. 2009. Model-Based Monitoring and Policy Enforcement of Services. In *Proceedings of the 2009 Congress on Services - I (SERVICES '09)*. IEEE Computer Society, Washington, DC, USA, 789–796. https://doi.org/10.1109/SERVICES-I.2009.103

[7] Andreas Bauer, Jan Jürjens, and Yijun Yu. 2011. Run-Time Security Traceability for Evolving Systems&#8224;. *Comput. J.* 54, 1 (Jan. 2011), 58–87. https://doi.org/10.1093/comjnl/bxq042

[8] Nelly Bencomo. 2009. On the use of software models during software execution. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering (MISE '09)*. Vancouver, Canada, 62–67. https://doi.org/10.1109/MISE.2009.5069899

[9] Antonia Bertolino, Antonello Calabrò, Francesca Lonetti, Antinisca Di Marco, and Antonino Sabetta. 2011. Towards a model-driven infrastructure for runtime monitoring. In *Proceedings of the Third international conference on Software engineering for resilient systems (SERENE'11)*. Springer-Verlag, Berlin, Heidelberg, 130–144. http://dl.acm.org/citation.cfm?id=2045537.2045557

[10] Gordon Blair, Nelly Bencomo, and Robert B. France. 2009. Models@ run.time. *Computer* 42, 10 (Oct. 2009), 22–27. https://doi.org/10.1109/MC.2009.326

[11] Lianne Bodenstaff, Andreas Wombacher, Manfred Reichert, and Roel Wieringa. 2010. MaDe4IC: an abstract method for managing model dependencies in inter-organizational cooperations. *Serv. Oriented Comput. Appl.* 4, 3 (Sept. 2010), 203–228. https://doi.org/10.1007/s11761-010-0062-7

[12] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. 2007. Cayuga: A High-performance Event Processing Engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*. ACM, New York, NY, USA, 1100–1102. https://doi.org/10.1145/1247480.1247620

[13] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. 2007. *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. Wiley, Chichester, UK. https://www.safaribooksonline.com/library/view/pattern-oriented-software-architecture/9780470059029/

[14] Mauro Caporuscio, Antinisca Di Marco, and Paola Inverardi. 2007. Model-based system reconfiguration for dynamic performance management. *J. Syst. Softw.* 80, 4 (April 2007), 455–473. https://doi.org/10.1016/j.jss.2006.07.039

[15] Mauro Caporuscio, Antinisca Di Marco, and Paola Inverardi. 2005. Run-time performance management of the Siena publish/subscribe middleware. In *Proceedings of the 5th international workshop on Software and performance (WOSP '05)*. ACM, New York, NY, USA, 65–74. https://doi.org/10.1145/1071021.1071028

[16] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. 2007. Cayuga: A General Purpose Event Monitoring System. In *CIDR'07*. 412–422.

[17] Michel Dirix. 2013. Awareness in Computer-Supported Collaborative Modelling. Application to GenMyModel. *ECOOP Doctoral Symposium* (2013). https://hal.archives-ouvertes.fr/hal-01251412

[18] B. F. Van Dongen. 2005. A Meta Model for Process Mining Data. In *In Proceedings of the CAiSE WORKSHOPS*. 309–320.

[19] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. 2001. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. *SIGMOD Rec.* 30, 2 (May 2001), 115–126. https://doi.org/10.1145/376284.375677

[20] Jean-Rémy Falleri, Marianne Huchard, and Clémentine Nebut. 2006. Towards a Traceability Framework for Model Transformations in Kermeta. In *ECMDA-TW'06: ECMDA Traceability Workshop*, J. Oldevik J. Aagedal, T. Neple (Ed.). Sintef ICT, Norway, Bilbao (Spain), 31–40. https://hal-lirmm.ccsd.cnrs.fr/lirmm-00102855

[21] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. 2006. Using Architecture Models for Runtime Adaptability. *IEEE Softw.* 23, 2 (March 2006), 62–70. https://doi.org/10.1109/MS.2006.61

[22] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. 2012. An eclipse modelling framework alternative to meet the models@runtime requirements. In *Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems (MODELS'12)*. Springer-Verlag, Berlin, Heidelberg, 87–101. https://doi.org/10.1007/978-3-642-33666-9_7

[23] Lidia Fuentes and Pablo Sánchez. 2009. Transactions on Aspect-Oriented Software Development VI. Springer-Verlag, Berlin, Heidelberg, Chapter Dynamic Weaving of Aspect-Oriented Executable UML Models, 1–38. https://doi.org/10.1007/978-3-642-03764-1_1

[24] Nadia Gamez, Lidia Fuentes, and Miguel A. Aragüez. 2011. Autonomic computing driven by feature models and architecture in FamiWare. In *Proceedings of the 5th European conference on Software architecture (ECSA'11)*. Springer-Verlag, Berlin, Heidelberg, 164–179. http://dl.acm.org/citation.cfm?id=2041790.2041811

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[26] D Garlan, B Schmerl, and J Chang. 2001. Using Gauges for Architecture-Based Monitoring and Adaptation. In *Proc. of the Working Conference on Complex and Dynamic Systems Architecture*. Brisbane, Australia.

[27] Sandro Rodriguez Garzon and Michael Cebulla. 2010. Model-Based Personalization within an Adaptable Human-Machine Interface Environment that is Capable of Learning from User Interactions. In *Proceedings of the 2010 Third International Conference on Advances in Computer-Human Interactions (ACHI '10)*. IEEE Computer Society, Washington, DC, USA, 191–198. https://doi.org/10.1109/ACHI.2010.12

[28] John C. Georgas, André van der Hoek, and Richard N. Taylor. 2009. Using Architectural Models to Manage and Visualize Runtime Adaptation. *Computer*

42, 10 (Oct. 2009), 52–60. https://doi.org/10.1109/MC.2009.335

[29] C. Ghezzi. 2011. The Fading Boundary between Development Time and Run Time. In *Web Services (ECOWS), 2011 Ninth IEEE European Conference on.* Lugano, Switzerland, 11–11. https://doi.org/10.1109/ECOWS.2011.33

[30] Tony Gjerlufsen, Mads Ingstrup, Jesper Wolff, and Olsen Olsen. 2009. Mirrors of Meaning: Supporting Inspectable Runtime Models. *Computer* 42, 10 (Oct. 2009), 61–68. https://doi.org/10.1109/MC.2009.325

[31] H.J. Goldsby, P. Sawyer, N. Bencomo, B. H C Cheng, and D. Hughes. 2008. Goal-Based Modeling of Dynamically Adaptive System Requirements. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the.* 36–45. https://doi.org/10.1109/ECBS.2008.22

[32] P. Grace, Gordon S. Blair, and S. Samuel. 2003. ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability. 1170–1187.

[33] P. Graf, M. Hubner, K.D. Müller-Glaser, and J. Becker. 2007. A Graphical Model-Level Debugger for Heterogenous Reconfigurable Architectures. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on.* 722–725. https://doi.org/10.1109/FPL.2007.4380754

[34] Philipp Graf and Klaus D. Müller-Glaser. 2006. Dynamic Mapping of Runtime Information Models for Debugging Embedded Software. In *Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP '06).* IEEE Computer Society, Washington, DC, USA, 3–9. https://doi.org/10.1109/RSP.2006.15

[35] Zonghua Gu, Shige Wang, Sharath Kodase, and Kang G. Shin. 2004. Multi-view modeling and analysis of embedded real-time software with meta-modeling and model transformation. In *Proceedings of the Eighth IEEE international conference on High assurance systems engineering (HASE'04).* IEEE Computer Society, Washington, DC, USA, 32–41. http://dl.acm.org/citation.cfm?id=1890580.1890584

[36] Wolfgang Haberl, Markus Herrmannsdoerfer, Jan Birke, and Uwe Baumgarten. 2010. Model-Level Debugging of Embedded Real-Time Systems. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology (CIT '10).* IEEE Computer Society, Washington, DC, USA, 1887–1894. https://doi.org/10.1109/CIT.2010.323

[37] Lars Hamann, Martin Gogolla, and Mirco Kuhlmann. 2011. OCL-based Runtime Monitoring of JVM hosted Applications. *Electronic Communications of the EASST* 44 (2011).

[38] Lars Hamann, Oliver Hofrichter, and Martin Gogolla. 2012. OCL-based runtime monitoring of applications with protocol state machines. In *Proceedings of the 8th European conference on Modelling Foundations and Applications (ECMFA'12).* Springer-Verlag, Berlin, Heidelberg, 384–399. https://doi.org/10.1007/978-3-642-31491-9_29

[39] L. Hamann, L. Vidacs, M. Gogolla, and M. Kuhlmann. 2012. Abstract Runtime Monitoring with USE. In *16th European Conference on Software Maintenance and Reengineering (CSMR).* Szeged, Hungary, 549–552. https://doi.org/10.1109/CSMR.2012.73

[40] Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[41] Ta'id Holmes, Uwe Zdun, Florian Daniel, and Schahram Dustdar. 2010. Monitoring and analyzing service-based internet systems through a model-aware service environment. In *Proceedings of the 22nd International Conference on Advanced Information Systems Engineering (CAiSE'10).* Hammamet, Tunisia, 98–112. http://dl.acm.org/citation.cfm?id=1883784.1883797

[42] J. Hutchinson, J. Whittle, Mark Rouncefield, and Steinar Kristoffersen. 2011. Empirical assessment of MDE in industry. In *Software Engineering (ICSE), 2011 33rd International Conference on.* 471–480. https://doi.org/10.1145/1985793.1985858

[43] Mads Ingstrup and Klaus Marius Hansen. 2005. A Declarative Approach to Architectural Reflection. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA '05).* IEEE Computer Society, Washington, DC, USA, 149–158. https://doi.org/10.1109/WICSA.2005.6

[44] Edgar Jakumeit, Sebastian Buchwald, Dennis Wagelaar, Li Dan, Ábel Hegedüs, Markus Herrmannsdörfer, Tassilo Horn, Elina Kalnina, Christian Krause, Kevin Lano, Markus Lepper, Arend Rensink, Louis Rose, Sebastian Wätzoldt, and Steffen Mazanek. 2014. A Survey and Comparison of Transformation Tools Based on the Transformation Tool Contest. *Sci. Comput. Program.* 85 (June 2014), 41–99. https://doi.org/10.1016/j.scico.2013.10.009

[45] D. Jayathilake. 2012. Towards structured log analysis. In *Computer Science and Software Engineering (JCSSE), 2012 International Joint Conference on.* 259–264. https://doi.org/10.1109/JCSSE.2012.6261962

[46] Frédéric Jouault. 2005. Loosely Coupled Traceability for ATL. In *European Conference on Model Driven Architecture (ECMDA) Workshop on Traceability.* Germany, 29–37. https://hal.archives-ouvertes.fr/hal-00448118 ISBN=82-14-03813-8.

[47] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. *Aspect-oriented programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242. https://doi.org/10.1007/BFb0053381

[48] Ingolf H. Krüger, Michael Meisinger, and Massimiliano Menarini. 2010. Interaction-based Runtime Verification for Systems of Systems Integration. *J. Log. and Comput.* 20, 3 (June 2010), 725–742. https://doi.org/10.1093/logcom/exn079

[49] Tanja Mayerhofer, Philip Langer, and Gerti Kappel. 2012. A runtime model for fUML. In *Proceedings of the 7th Workshop on Models@run.time (MRT '12).* ACM, New York, NY, USA, 53–58. https://doi.org/10.1145/2422518.2422527

[50] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. 2009. Models@ Run.time to Support Dynamic Adaptation. *Computer* 42, 10 (Oct. 2009), 44–51. https://doi.org/10.1109/MC.2009.327

[51] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. 2002. *Query Processing, Resource Management, and Approximation in a Data Stream Management System.* Technical Report 2002-41. Stanford InfoLab. http://ilpubs.stanford.edu:8090/549/

[52] Gøran K. Olsen and Jon Oldevik. 2007. *Scenarios of Traceability in Model to Text Transformations.* Springer Berlin Heidelberg, Berlin, Heidelberg, 144–156. https://doi.org/10.1007/978-3-540-72901-3_11

[53] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016).* ACM, New York, NY, USA, 33–44. https://doi.org/10.1145/2976002.2976010

[54] Marian Petre. 2013. UML in Practice. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13).* IEEE Press, Piscataway, NJ, USA, 722–731. http://dl.acm.org/citation.cfm?id=2486788.2486883

[55] M. Spieker, A. Noyer, P. Iyenghar, G. Bikker, J. Wuebbelmann, and C. Westerkamp. 2012. Model based debugging and testing of embedded systems without affecting the runtime behaviour. In *Emerging Technologies Factory Automation (ETFA), 2012 IEEE 17th Conference on.* 1–6. https://doi.org/10.1109/ETFA.2012.6489656

[56] Michael Szvetits and Uwe Zdun. 2015. Reusable event types for models at runtime to support the examination of runtime phenomena. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS).* 4–13. https://doi.org/10.1109/MODELS.2015.7338230

[57] Michael Szvetits and Uwe Zdun. 2016. Controlled Experiment on the Comprehension of Runtime Phenomena Using Models Created at Design Time. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS '16).* ACM, New York, NY, USA, 151–161. https://doi.org/10.1145/2976767.2976768

[58] Michael Szvetits and Uwe Zdun. 2016. Systematic Literature Review of the Objectives, Techniques, Kinds, and Architectures of Models at Runtime. *Softw. Syst. Model.* 15, 1 (Feb. 2016), 31–69. https://doi.org/10.1007/s10270-013-0394-9

[59] Andrew S. Tanenbaum and Maarten van Steen. 2006. *Distributed Systems: Principles and Paradigms (2Nd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[60] N. Tax, N. Sidorova, R. Haakma, and W. van der Aalst. 2018. Mining Process Model Descriptions of Daily Life Through Event Abstraction. In *Intelligent Systems and Applications*, Yaxin Bi, Supriya Kapoor, and Rahul Bhatia (Eds.). Springer International Publishing, Cham, 83–104.

[61] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. 2009. *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework.* Research Report. Kiel University. http://eprints.uni-kiel.de/14459/

[62] J. M. Vara, V. A. Bollati, ÃĄ. JimÃľnez, and E. Marcos. 2014. Dealing with Traceability in the MDDof Model Transformations. *IEEE Transactions on Software Engineering* 40, 6 (June 2014), 555–583. https://doi.org/10.1109/TSE.2014.2316132

[63] Mira Vrbaski, Gunter Mussbacher, Dorina Petriu, and Daniel Amyot. 2012. Goal models as run-time entities in context-aware systems. In *Proceedings of the 7th Workshop on Models@run.time (MRT '12).* ACM, New York, NY, USA, 3–8. https://doi.org/10.1145/2422518.2422520

[64] Lukas Wegmann and Dominique Wirz. 2013. Modellgetriebene Visualisierung von Echtzeitsystemen im Browser. https://eprints.hsr.ch/310/

[65] K. Welsh, P. Sawyer, and N. Bencomo. 2011. Towards requirements aware systems: Run-time resolution of design-time assumptions. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on.* 560–563. https://doi.org/10.1109/ASE.2011.6100125

[66] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance Complex Event Processing over Streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06).* ACM, New York, NY, USA, 407–418. https://doi.org/10.1145/1142473.1142520

[67] Eric Yu and John Mylopoulos. 1998. Why goal-oriented requirements engineering. In *Proceedings of the 4th International Workshop on Requirements Engineering: Foundations of Software Quality.* 15–22.

[68] Uwe Zdun and Michael Szvetits. 2017. Automatic Generation of Monitoring Code for Model Based Analysis of Runtime Behaviour. In *24th Asia-Pacific Software Engineering Conference (APSEC 2017).* http://eprints.cs.univie.ac.at/5313/

[69] Kebin Zeng, Yu Guo, and Christo K. Angelov. 2010. Graphical model debugger framework for embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10).* European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 87–92. http://dl.acm.org/citation.cfm?id=1870926.1870949