Reusable Event Types for Models at Runtime to Support the Examination of Runtime Phenomena

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

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

***Abstract*—Today's software is getting more and more complex and harder to understand. Models help to organize knowledge and emphasize the structure of a software at a higher abstraction level. While the usage of model-driven techniques is widely adopted during software construction, it is still an open research topic if models can also be used to make runtime phenomena more comprehensible as well. It is not obvious which models are suitable for manual analysis and which model elements can be related to what type of runtime events. This paper proposes a collection of runtime event types that can be reused for various systems and meta-models. Based on these event types, information can be derived which help human observers to assess the current system state. Our approach is applied in a case study and evaluated regarding generalisability and completeness by relating it to two different meta-models.**

***Index Terms*—events, examination, models, runtime**

I. Introduction

Recent research utilizes models at runtime to equip systems with additional capabilities to reflect on their own structure and to adapt themselves in response to changing requirements and execution environments [1]–[3], [5]–[7], [18], [20]. This is a contrast to conventional software engineering where models are created during software development and have no direct connection to the resulting executable software system. This new paradigm uses models as causally connected self-representations of the associated system, meaning that changes to the models are reflected within the running system, and vice versa. The used models emphasize the properties of the system from a problem space perspective [2].

A prominent approach to ensure a continuous connection between the system and its models is an autonomic control loop. The idea originates from the autonomic computing research community to realize self-management of systems according to desired goals [4], [16]. With models in this control cycle, live information from the system is fed back to these models, and updates to these models are propagated to the system while it is up and running. While automatic adaptation rules are common practice in control loop approaches, scenarios with a need of human interaction have, to the best of our knowledge, hardly been addressed by approaches utilizing models at runtime yet. We argue that an integration of human activities into the model-based control loop has many advantages in scenarios where automatic decisions are limited or too complex, like:

- Confirmation of actions where automatic decisions are forbidden or undesirable (e.g., when performing financial transactions or legally binding actions). Models can provide contextual information about the situation that needs attention by a human observer.
- Control of simulations and handling of unexpected errors which are not known in advance (e.g., when testing software). Models provide a condensed view of the system and enable on-the-fly adaptation of simulation and testing conditions.
- Reacting to events which need individual assessment (e.g., violation of service level agreements or hardware faults). Models can help to visualize the context and tracing history that led to a violation and help to localize faulty communication paths.
- Reacting to violated constraints which originate from non-technical requirements (e.g., violation of reporting rules). Models can highlight processes and activities that miss the required actions.
- Manual assessment of system health by analysing various parameters on the model level (e.g., qualitative and quantitative runtime data, bottlenecks, or trends).

However, combining the models at runtime paradigm with human interaction leads to non-trivial problems in terms of traceability, extraction of situation-specific data, model navigation and generation of the monitoring environment. In this paper, we focus mainly on supporting the extraction of situation-specific data and present a meta-model-agnostic, reusable collection of runtime event types which serves as a basis for applying aggregation mechanisms to enable humans the manual assessment of the system status.

This paper is organized as follows: In Section II we give an overview of our approach to support human interactions at runtime. In Section III we describe the approach details, mainly centering around the idea of model-based runtime event types. Section IV discusses the application of our approach in a case study. In Section V we analyse our proposed event types regarding generalisability and completeness. In Section VI we discuss our results, and in Section VII we compare to related work. We conclude in Section VIII.

II. Approach Overview

Integrating runtime models into the control loop enables human observers to monitor system properties like error occur-

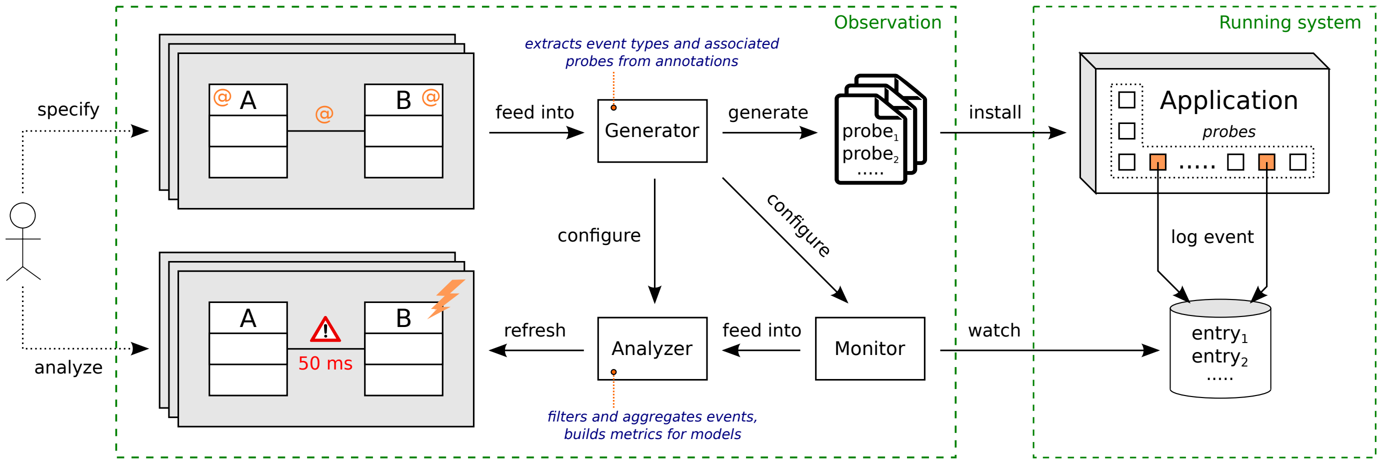

Figure 1. Development steps when performing model-based human interaction

rences, constraint violations, usage statistics or performance characteristics on the model-level. Models serve as both an abstract view of the system and dashboard for aggregating data and controlling the analysis, which makes them better targets for annotations than the actual source code. As an example, a human observer might decide to measure the performance of a software component and thus annotate the desired component in the respective UML component diagram. Performing an equivalent monitoring procedure on the code-level would require to annotate every sub-part of the component by hand, meaning in most cases that many different implementation artefacts must be altered.

However, there is no straight mapping between the recorded events of the running system and the used runtime models, so our approach presents a collection of runtime event types which are independent of concrete meta-models and running systems. These event types serve as a target for building runtime metrics and should guide researchers to implement mappings between runtime events and custom meta-models. To make our approach work, however, we have to make some assumptions about the used (meta-)models and the observed systems:

- Models must exist before the analysis, either by manual creation or by reverse engineering.
- The used meta-model must describe structural and behavioural aspects of the running system. We use UML throughout the paper, but demonstrate the independence of concrete meta-models in Section V.
- The source code of the observed system must be available. While this is not strictly necessary, in our Java-based prototype, we did not cover black-box approaches yet which require techniques like load time weaving [17] to unobtrusively extract runtime data.

A detailed view of the model-based control loop realized in our approach is depicted in Figure 1. Elements of interest are annotated directly in the models of the system under observation. Based on the annotations and their parameters, a generator component automatically generates software artefacts with multiple probes which provide events to extract relevant information from the system. The generation process also configures a monitor and an analysis component of the observing system. The monitoring component is configured to listen for events of the running system while the analysis component is configured to filter and aggregate data according to the model annotations. The benefit of such a decoupling is that the monitoring and analysis components can also be configured by rules that do not result from annotations, but from external data sources.

In case of constraint violations, the analysis component refreshes the user views that contain the associated model elements to highlight the incident. Thus, the monitor and the analysis components are key parts in our control loop because they are responsible for receiving and analysing information gathered at runtime. They correspond to the monitoring and analysis control loop phases, respectively.

In the running system, the generated software artefacts and their probes are installed on-the-fly while the system is executing. This is realized using aspect-oriented techniques as described in Section III-C. Changes might include detaching of previous probes. The probes enable the causal connection between the model annotations and the installed monitoring environment. The newly installed probes then produce runtime information which is stored in a file or a database for further analysis by the monitor that performs the system observation.

The gathered information consists of runtime events which occur if specific conditions are met. Events and their conditions depend on the type of the annotated model element, the annotation itself and the parameters of the annotation. For example, if a user wants to observe the average runtime of a modelled behaviour, the *type* of model element would be *Behaviour*, the annotation type would be an *average annotation* and the *parameter* of the annotation would be an expression which measures the runtime of a single behaviour execution. Figure 2 shows the model annotation for this example to measure the average runtime of a sorting behaviour. The resulting probe triggers an event if the annotated behaviour

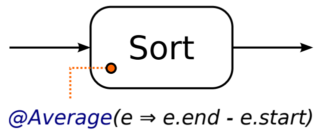

Figure 2. Example model element annotation to measure the average runtime

executes, with the condition that the execution of the behaviour completes without errors. The average runtime is calculated by the analyser by taking the successfully occurred events into account that have been recorded by the monitor. Similar to the average runtime, other metrics can be assigned to model elements whereas every model element provides a set of runtime event types from which metrics can be calculated or on top of which other analyses can be performed.

Overall, our approach utilizes the causal connection of the models at runtime paradigm to interactively analyse and adapt a running system while minimizing human interventions concerning technical details below the model level. To achieve this, it is necessary to identify the types of runtime events that can be captured by generated probes and how they can be related to model elements they originate from. This paper focusses on the identification of these event types, their relations to model elements, and provides a generic approach and corresponding prototype implementation to utilize them for metric calculations and other automatic analyses.

III. Approach Details

A. Relating Meta-Models and Event Types

Figure 3 shows the relationship between models, events and metrics as well as between their respective meta-level descriptions. Our proposed runtime event types can be traced to meta-model elements describing structural and behavioural features of a system. Based on these event types, a user can specify metrics to define measurements of interest. Reusability is supported as our event types are defined independently of concrete meta-models describing the structure and behaviour of a system. Furthermore, metrics and other analyses are based on event types, which in turn are independent of a concrete meta-model, which makes the metrics/analyses meta-model-agnostic as well. The application of our approach to arbitrary meta-models requires a developer to enrich his or her code (or code generator) with events that conform to our proposed event types and contain tracing information to model elements they originate from. As a result, our proposed runtime event types serve as guidance to implement mappings between runtime events and custom meta-models.

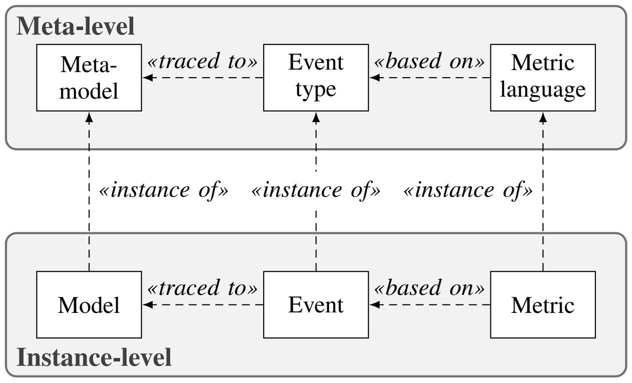

Figure 3. Relationships between models, events and metrics

Please note that we use metric calculation in the remainder of this paper as an exemplary type of analysis; however, other analyses like forecasting or recommendations can be implemented in a similar way.

B. Categorizing Model Elements based on their Traceable Runtime Event Types

A runtime event is the most basic unit which enables the extraction of information from a running system. We define a runtime event as an occurrence at a specific point in time within a specific context while a system is executing. A runtime event thus consists of a time stamp and arbitrary data that represents the context. In object-oriented programming terms, this data is often called *event arguments* and usually contains information about the source of the event and data describing the impact on the running system, for example old and new values of properties when dealing with mutable state. A concrete example would be a change in the balance of a bank account: The account object is the source of the event while the old and new values of the balance field represent the changed state.

Such basic runtime information enables analysis of system activities over time, but it is not directly linked to the model elements which are responsible for the event or its context. To provide a human with adequate runtime information on the model level, in the majority of cases a set of runtime events that can be traced and related to one or more model elements during software execution is needed. Since different model elements comply with different meta-models, our approach aims to categorize model elements according to the runtime events they can possibly be linked to. This allows us to abstract from various meta-models and look at the model elements from a runtime analysis perspective.

In our research, we have first analysed the UML 2.5 meta-model for different kinds of meta-model elements and event types that are typically related to those meta-model elements. We chose UML because it is a complete and de-facto standard for modelling both structural and behavioural features of software systems. In addition, UML provides modelling means for both high-level models (e.g., use case diagrams) and low-level models (e.g., sequence diagrams) which makes it suitable for monitoring runtime behaviour on various levels of abstraction.

Regarding the event types, our starting point of research were prominent approaches for handling cross-cutting concerns like logging and tracing. We analysed aspect-oriented frameworks regarding their abilities to intercept execution points in the implementation code to extract runtime information. Most frameworks allow to set hooks before, after and around operation executions to extract data, add custom logic

Table I
CATEGORIZATION OF MODEL ELEMENTS ACCORDING TO THEIR TRACEABLE RUNTIME EVENT TYPES

Category	Traceable event types	Event arguments
Container	*Derived from children*	*Derived from children*
Part-whole relationship	Change in multiplicity	Added/removed objects
Data exchange	Data sent/received	Sender/receiver, transmitted data
Mutable state	Value changed	Old value, new value
Behaviour	Started, ended, error	Input/output parameters, error message
Physical entity	Environmental change	Old and new environment configuration
Concrete state	State occurred	Participating objects
Testable condition	Condition does (not) hold	Constrained element
Instantiable element	Instance created, instance destroyed	Created instance, destroyed instance

or alter existing behaviour. We combined these abilities with all the elements of the UML meta-model to identify possibly traceable event types for each of them. Examples would be events before entering a state in a state machine diagram or after sending a message in a sequence diagram. The combination of the de-facto standard modelling language UML and the interception abilities of modern logging approaches gave us confidence that our identified event types are both relevant and appropriate.

In the next step, we have abstracted from the concrete meta-model elements and devised the categories and abstract event types discussed in this section, in order to make our approach applicable to other meta-models. In Section V, we evaluate our categorization and show that we are able to apply it to two other existing meta-models as well, which indicates that the categorization shows a high degree of completeness and is generalisable. Table I shows our proposed categories with their typical runtime event types and arguments. In the following we describe the categories and give examples of typical runtime events and event arguments that can be associated with them.

Containers serve as a mechanism to build hierarchies where child elements can propagate runtime events to their parents. These hierarchies allow to cluster and capture multiple runtime events by annotating single model elements. Thus, the set of runtime event types for an element in this category is the union of all runtime event types of its children. Formulae can be computed with the event types of the children, but the annotation needs to specify the concrete event type that is actually captured (e.g., all data received events within a package). Examples of containers in UML are components and packages.

A *part-whole relationship* indicates that one entity is composed of one or more other entities, its parts. It is characterized by multiplicities that indicate how many entities are related in the relationship. As a consequence, a typical runtime event type for such relationships is the change of the number of entities that are participating in the relationship. The associated event arguments usually hold the added and/or removed elements. An example would be a list where elements are added and removed at runtime. In UML, part-whole relationships are composition and aggregation.

A model element that represents a *data exchange* between entities can either be a communication interface or a connection. Both forms can be targets for model-based analysis, typically by taking time, rate and size of exchanged messages into consideration. Associated runtime events are the dispatch and reception of data, whereas the event arguments are the sender, the receiver and the transmitted data. UML-based examples of data exchange elements are ports, pins and connectors.

Model elements that represent *mutable state* typically provide a runtime event if the associated value changes at runtime. The event arguments then contain both the old and the new value. Properties and value lifelines are examples of model elements that represent mutable state in UML.

Model elements that represent *behaviour* are characterized by input parameters and output parameters. An execution of the involved actions may lead to an error and thus to a premature termination of the behaviour. As a result, possible runtime events for behaviours are typically start, end and error. The event arguments are input parameters, output parameters and error messages, respectively. UML-based examples of behaviour are operations, receptions and actions.

Monitoring of environmental changes can be achieved through model elements which reflect *physical entities* like deployment artefacts and network connections. Associated runtime events are usually changes of the environmental set-up, dispatch and reception of data, and errors in case of transmission failures. Event arguments for environmental changes are the old and the new hardware configuration. The arguments for exchanging data are the same as for exchanging local messages. In UML, physical entities are represented by nodes and communication paths.

Modelling *concrete (partial) states* of a system enables to capture specific object collaborations and thus allows to monitor more complex situations. An example of an adequate runtime event is the occurrence of a modelled situation that matches the state of the system under observation. Exemplary event arguments are the concrete instances that are participating in the modelled state. UML-based examples of modelling concrete states are instance specifications, slots and links (instances of associations).

Model elements that represents *conditions* are usually textual and are attached to another element. Monitoring conditions enable the detection of violations if conditions describing system constraints do not hold. Typically, a runtime event occurs if the condition holds or does not hold, and its argument

Table II
APPLICATION OF OUR PROPOSED CLASSIFICATION TO THE UML 2.5 META-MODEL

Category	UML 2.5 counterparts
Container	Component, Node, Class, Interface, Package, Partition, CombinedFragment, Region, UseCase
Part-whole relationship	Composition, Aggregation
Data exchange	*Interface:* Port, ActivityParameterNode, Pin, ExpansionNode, ControlNode *Connection:* Generalization, Connector, Dependency, Import, ActivityEdge, Message, Extend, CommunicationPath
Mutable state	Property, State/Condition Timeline, Value Lifeline, CentralBufferNode, DataStoreNode
Behaviour	Behaviour, Operation, Reception, Action, ExecutionSpecification, InteractionUse, State, Transition
Physical entity	Node, CommunicationPath
Concrete state	InstanceSpecification, Slot, Link (instance of Association)
Testable condition	Constraint, Invariant, Continuation
Instantiable element	Class, Interface

is the constrained element. Examples of UML elements for conditions are constraints, invariants and continuations.

From a monitoring perspective, an important observation is the consumption of system resources, e.g., system memory. Intense memory allocation is caused by instantiating a large amount of objects. As a consequence, a model element that represents an *instantiable entity* can be related to runtime events that occur if the modelled entity is created or destroyed. Event arguments are the created instance or the instance to be destroyed, respectively. UML-based examples of instantiable entities are classes and interfaces. Note that although an interface cannot be instantiated directly, from a runtime analysis point of view it is still desirable to analyse the memory consumption of instances that implement a specific interface.

To demonstrate the adequateness of our categorization, Table II assigns elements of the UML 2.5 meta-model to the associated categories. Note that the *UseCase* element is assigned to the container category since it serves as logical parent for multiple realizing classifiers. Also note that the elements *Generalization, Dependency, Import* and *Extend* appear to have nothing to do with data exchange, since they model static relationships between entities. Nevertheless, from a runtime analysis point of view, these elements also represent communication, namely whenever an object from the client calls functionality from the supplier. *CentralBufferNode* and *DataStoreNode* represent mutable state if the changeable values are defined by their stored elements. *State* and *Transition* fall in the category of behaviour since they have optional behaviour attached to them. Note that *Class* is not only an instantiable element, but also a container, since it can contain other elements like subclasses and interfaces. Runtime events are propagated from such subclasses and interfaces to their parent element.

We presented a meta-model-agnostic categorization of model elements and an associated collection of runtime event types which can typically be triggered for the related model elements by probes that are installed in a running system. These events can be traced back to the model elements they originate from and consist of a time stamp and arbitrary data that describe the context. Thus, it is possible for a human user to analyse runtime phenomena on the model level and compose events (that is, create formulae using multiple event types and metrics) to build more sophisticated runtime metrics to reason about the system state.

C. Implementation Aspects

It is necessary that runtime events can be linked to the annotated model elements they originate from. One way of achieving this is to embed IDs into the probes which correspond to specific model annotations (and thus, to specific model elements). If a probe triggers an event, the IDs are integrated into the event arguments and can be associated with the correct model elements by the analyser component of the observing system (recall Figure 1)

Dynamic attaching and detaching of probes is achieved by extending aspect-oriented programming mechanisms for that purpose. More specifically, we developed a custom extension of the AspectJ[1] parser to extract a so-called pointcut expression from an aspect and transform it into a tree structure which is suitable for runtime interpretation. Such a tree is similar to the abstract syntax tree of the pointcut expression and consists of operators (internal nodes) and atomic sub-pointcuts (leaf nodes) which can be evaluated for a given method execution at runtime. If the whole tree is evaluated to true, the associated advice is executed and performs the logging activities which can be traced back to associated model elements. The extension is needed because there is, to the best of our knowledge, no native way of AspectJ to swap aspects at runtime and load-time weaving does not support the replacement of already woven aspects.

Note that the monitoring aspect generated from a model annotation is completely independent from the used adaptation

[1]https://eclipse.org/aspectj/

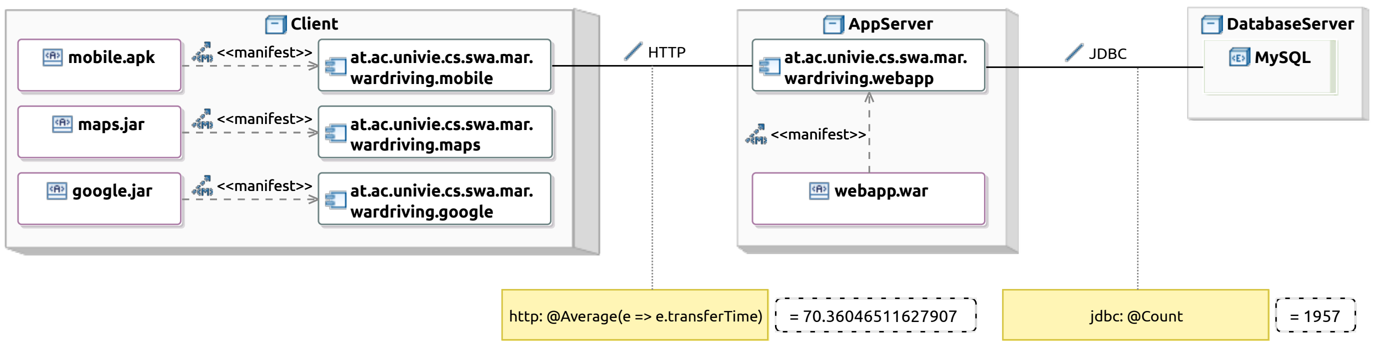
Client
mobile.apk
maps.jar
google.jar
<<manifest>>
<<manifest>>
<<manifest>>
at.ac.univie.cs.swa.mar.wardriving.mobile
at.ac.univie.cs.swa.mar.wardriving.maps
at.ac.univie.cs.swa.mar.wardriving.google
HTTP
AppServer
at.ac.univie.cs.swa.mar.wardriving.webapp
<<manifest>>
webapp.war
JDBC
DatabaseServer
MySQL
http: @Average(e => e.transferTime)
= 70.36046511627907
jdbc: @Count
= 1957

Figure 4. Measurements through model annotations which reveal a high number of database requests. Transfer time is measured in milliseconds.

mechanism, meaning that the same generated code can be used either for compile-time or runtime weaving of pointcuts. However, a mechanism is needed which intercepts all control flow activities and matches them to the currently active aspects. We achieved this by a separate aspect that intercepts all method calls and interprets currently active aspects to check if one of them matches the current execution context. Intercepting all method calls and shifting the pointcut matching logic from compile-time to runtime involves a performance overhead which depends on the number of currently active aspects.

IV. Case Study

To study the applicability and the relevance of the event types, we have applied our approach in a case study which was originally implemented without using model-driven techniques. The case study is on a small system implementing a wireless scanner. We choose this system because of our familiarity with it and to see whether the data provided by our proposed event types helps to locate and improve a number of problems of the initial implementation. The system was realized independently by three developers for half a year and was originally not intended to serve as unit of analysis for this case study. We applied our approach to multiple models extracted from its documentation to find the root causes of existing performance problems.

In the wireless scanner project, mobile devices scan their environment for other devices and transmit their scan results to a central database. Based on the collected data, the system then allows arbitrary clients to query device information by using filter parameters like locations, time stamps, transmission types and device properties. We realized the system by using a three-tier architecture as depicted in Figure 4. The system consists of a mobile Android application for sending and querying scans, a server which acts as interface between the mobile application and the backend, and a MySQL database as persistent data storage. All parts are written entirely in Java, using 10,064 lines of code in total. Modelling the system was done in Eclipse with Obeo UML Designer[2] and Papyrus[3]. We inserted 875,000 test scan entries into the database (approximately the number of households in Vienna) and noticed a disturbing latency when requesting scans within a range of 1.5 kilometres around the University of Vienna.

[2]http://marketplace.obeonetwork.com/module/uml
[3]http://www.eclipse.org/papyrus/

We applied our annotation-based approach as shown in Figure 4 to track down the root cause of the latency problem. The first obvious action is to examine the network connections between the involved subsystems. According to our categorization in Table I, the model elements representing the network connections are data exchange relationships where the associated event type captures data transmission and reception. The annotation:

$$http : @Average(e => e.transferTime)$$

indicates that we want to calculate the average data transfer time from all transmission events that be traced to the annotated communication path. The term *http* is simply a name other metrics can refer to, while *transferTime* is a special keyword that instructs the code generator to capture send and receive events and calculate their differences. As a consequence, the concrete event type of e (“Data sent/received”, recall Table I) can be omitted since it is inferable from the monitoring expression. The term @*Average* indicates that the average of all captured data transfer times should be calculated, although specific time windows can also be specified by utilizing the aforementioned time stamps which are part of every event (they are accessible via a special property of e). In a similar fashion, the annotation named *jdbc* depicted in Figure 4 counts all transmission events between the application and the database server.

The annotations produce aspects that are dynamically interpreted and are responsible for emitting the transmission and reception events of data. The results are directly visible in the models, and the annotation of the backend communication path revealed a surprisingly high number of database queries for a single scan request.

We assumed that there could be a problem of using sub-queries in a loop instead of making a single join over connected database tables. We wanted to assure that the number of sub-queries is indeed responsible for the disturbing latency experienced by the client. We decided to apply our annotation-based approach to a model describing the query of scans in more detail. Figure 5 shows an activity diagram which describes the process of querying scans. The diagram shows

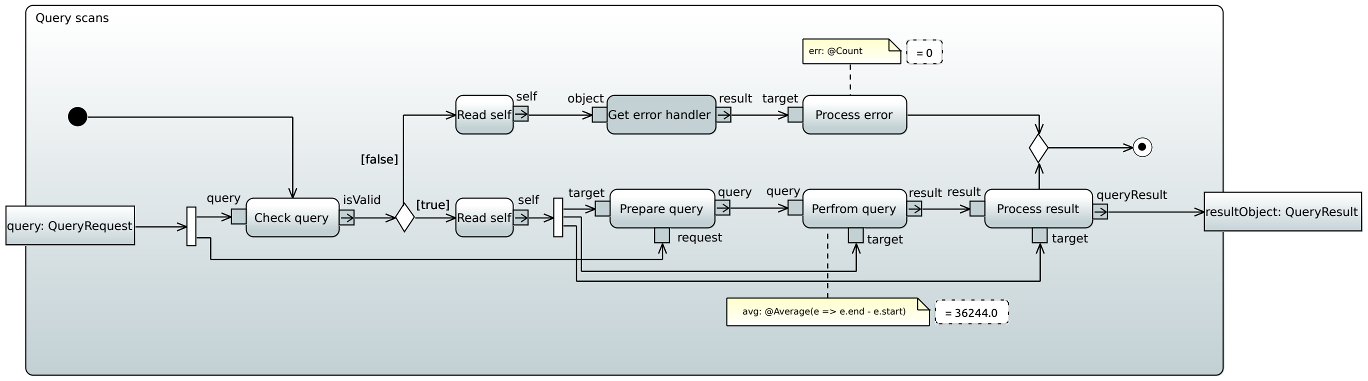

Figure 5. Annotations in the activity diagram describing a query of wireless scans. Average runtime is measured in milliseconds.

that there are two process paths: One for faulty and one for correct requests. We annotated both paths of the activity to check if there are errors and to measure the processing time of performing the database actions of a client request. This could be done since the annotated model elements are behaviours, and according to our categorization in Table I, the associated event type provides information about the start, end and possible errors of the operation. Note that the keywords *start* and *end* indicate that it must be a measurement of a behaviour, so the concrete event type of e can be omitted. Figure 5 shows that the database actions require a processing time of around 36 seconds, which is unacceptable from the client's point of view.

We checked if there are any database requests inside loops, which is easy to do since Eclipse allows to quickly jump to code statements that were woven by the generated monitor aspects. We identified a defective code snippet by navigating through the woven statements where capabilities of a WLAN access point were requested in a sub-query for every access point in the result set of a previous query. After replacing the snippet with a join operation, the result of the count annotation in Figure 4 was reduced from 1957 to three. We came to the decision that even three requests were too much.

Since we have three types of scans (WiFi Infrastructure, WiFi Peer-To-Peer, Bluetooth), we assumed that the system was falsely requesting all scan types instead of the requested one only. We identified a defective code snippet which possibly resulted from a copy-paste-operation from a method that requests all three scan types. However, we were able to locate the error only by analysing the implementation code directly. After fixing all the problems, we were able to reduce the original processing time of around 36 seconds for performing a query by half (while the monitor aspects were still woven). Although this is a significant improvement and we consider the case study as success, further adaptations are required to enhance the client experience, for example by limiting the amount of requested information by introducing paging.

As a summary, our proposed event types provided information that enabled us to successfully narrow down the root cause of a performance problem by stepwise application of model annotations to gain insight into the behaviour of the running system. The combination of model-based metrics and Eclipse-based navigation capabilities (that is, the navigation from aspects to woven code fragments) allowed us to quickly search for defective code snippets. After fixing the sub-query issue, our approach revealed a masked error of requesting excessive data from the database which we were able to fix without the help of navigation support.

V. Evaluation

As described in Section III, we extracted the categorization of model elements according to their traceable runtime event types from the UML 2.5 meta-model. To evaluate the generalisability and completeness of our proposed categorization, we have mapped the categorization to two other existing meta-models. Many meta-models that can describe both structural and behavioural aspects are closely related to UML themselves (e.g., fUML, Ecore, SysML, SoaML), which makes a comparison rather trivial, and thus not too useful for this evaluation. Instead, we decided to compare our categorization against meta-models which are capable of describing system parts, but differ in their pursued modelling goal: BPMN[4] and FAD [22]. We conducted the evaluation by mapping meta-model elements of the BPMN and FAD specifications to our proposed categorization.

By applying our categorization to the BPMN meta-model, we can show that our approach is also able to relate runtime event types to models at a higher abstraction level (that is, business processes). FAD is an analysis and design methodology for systems that follow the functional programming paradigm. By applying our categorization to the FAD meta-model, we can show that event types following our categorization are conceptually independent from the underlying programming paradigm that realizes the actual implementation.

The results of our mappings are shown in Table III. Unsurprisingly, because of its process-oriented nature, BPMN contains various model elements that represent behaviour, container for grouping process participants and data exchange for communication activities between tasks. Since BPMN describes a system at a rather high abstraction level, it follows

[4]http://www.omg.org/spec/BPMN/

Table III
APPLICATION OF OUR PROPOSED CATEGORIZATION TO THE BPMN AND FAD META-MODELS

Category	BPMN model element counterparts	FAD model element counterparts
Behaviour	Activity, Choreography Task, (Sub-)Process, Sub-Choreography, Task, Transaction	Curried Function, Function
Concrete state	*None*	Partial Application, Type (Named Value)
Container	Activity, Group, Lane, Pool, (Sub-)Process, Sub-Choreography, Transaction	Exclusive Signature (Association), File, Module, Permissive Signature (Association), Project, Subsystem
Data exchange	Choreography Task, Event (Cancel, Compensation, Error, Escalation, Link, Multiple, Parallel Multiple, Signal, Timer), Fork, Join, Gateway, Message, Message Flow, Sequence Flow	File Use Relationship, Function Use Relationship, Module Use Relationship, Project Use Relationship, Signature Inheritance Relationship, Subsystem Use Relationship, Type Use Relationship
Instantiable element	Data Object, Event, Message	Function Argument, Function Result, Permissive Signature, Type
Mutable state	Data Object	*None*
Part-whole relationship	*None*	Containment Relationship, Partition Relationship
Physical entity	*None*	File, Module
Testable condition	Branching Point, Decision, Event (Conditional)	Function Use Relationship

quite directly that it contains no model elements for physical entities. Interestingly, we could not find any model element which represents part-whole relationship in a sense that it supports the event type of adding and removing parts at runtime (recall Table I). Furthermore, we could not identify model elements that represent a concrete state, like the actual value of a data object for a given situation. While activities and processes serve as containers, they could also be categorized as instantiable elements since a process or activity can be executed multiple times, usually by an execution engine.

Regarding FAD, the underlying functional programming paradigm avoids mutable state as much as possible. This is also reflected in the meta-model which contains no model element that represents mutable state. On the contrary, the FAD meta-model provides various model elements for composing functions and types, the two central concepts of functional programming. Furthermore, in alignment with the functional programming paradigm, the only two elements that represent behaviour are functions and curried functions.

Although we extracted our proposed categorization of model elements and event types from UML, we were able to apply it to all elements of two other meta-models which are quite different in their modelling goals. This gives us confidence that the categorization has a high degree of relevance, generalisability and completeness and that the proposed event types help to understand what kind of runtime events can be associated with what kind of model elements.

VI. DISCUSSION

In our approach we extracted the necessary information from the UML meta-model. Other meta-models could have led to a different categorization. We argue that, since UML is the de-facto standard for modelling structural and behavioural features of a system, the choice of UML led to the most complete categorization because the meta-model provides a mixture of highly abstract and more low-level models of many different types. Furthermore, we mitigated the risk of creating an incomplete categorization by applying it to all model elements of two other meta-models in Section V. We argue that additional meta-models would not significantly contribute to the completeness of our categorization, since many standardized meta-models are either closely related to UML or not suited for analysing structural and behavioural properties of a running system (e.g., entity-relationship models). Regarding the event types, we argue that a high degree of completeness is given since they are inspired by interception operations of well-established, orthogonally working aspect-oriented programming approaches.

In Section IV, we utilized parts of our proposed event types in a case study and were able to detect the root cause of a performance problem. Regarding measurement accuracy, the performed observations may be flawed due to incorrect aspect generation and environment-specific deviations. We mitigated such risks by explicitly excluding some paths in the control flow to prevent flaws in the measurements. For example, recursive calls must not be measured if the overall processing time of that method is desired. This makes the formulation of advice more complex, error-prone and thus less suited for writing them by hand. Regarding environment-specific deviations, some subcomponents cannot be influenced directly, like network buffers of caching mechanisms of database systems. Furthermore, exact measurements require an integration of distributed clocks which is currently not realized in our prototype.

Regarding applicability, our approach assumes that models of the monitored parts exist, which is rarely the case, but various reverse engineering tools exist to generate initial models which can then be refined. As demonstrated in our case study (see Section IV), modelling the most important parts of a system is an adequate basis to locate various forms of unexpected runtime behaviour. However, to exploit the full potential of our approach, model-based metrics must be complemented with model navigation and tracing capabilities to seamlessly navigate between models and their implementation. Nevertheless, some problems still remain hard to detect, like unexpected runtime behaviour that results from unmodified copy-paste operations. Narrowing down the root cause of a problem needs some practice, since some annotations may lay a false trail (e.g., measuring the average transfer time would not have revealed the excessive query count in Figure 4).

More specifically, our approach requires that traceability links and a fair degree of cohesion are ensured for the modelled software components. Otherwise, our approach is limited in a sense that monitoring results are either inaccurate or completely impossible. A bad example would be a use case that is traceable to its realizing classifiers, but these classifiers are implemented with low cohesion, meaning that they are implemented poorly with respect to the separation of concerns principle. If this is the case, monitoring the realizing classifiers would be flawed since some parts of their control flows do not belong to the use case of interest. Such a problem can only be compensated by an intelligent code generator or by refactoring actions to ensure high cohesion.

Our approach supports models which are close to the problem space as well as models that contain more technical details. For problem space-oriented models, our approach requires tracing information to either models of lower abstraction or directly to associated code fragments. A code generator can exploit this information and ensure that the generated code emits events that can be traced back to the corresponding model elements. The models themselves must be somehow related to structural or behavioural properties of the observed system to narrow down unexpected behaviour and eventually find defective implementation artefacts.

Regarding portability, the event types provide guidance to implement monitoring for other meta-models. An alternative would be a model-to-model transformation which converts a meta-model to another one for which a code generator already exists. This requires additional trace links between the source and the target meta-model for reasoning on the source model.

An alternative to our proposed model-based approach would be dynamic code analysis. We argue, however, that a strict examination of measurements on the code level provides not necessarily enough aggregation mechanisms to make assumptions of whole system parts. An example would be the measurement of a use case (e.g., the average time spent for a specific use case) where a direct counterpart does not exist in the implementation code, hence dynamic code analysis is not enough for such scenarios. However, an integration of dynamic code analysis approaches into our approach may provide more runtime data to reason about.

In its current form, our approach does not provide sophisticated mechanisms to detect deviations between the running system (the probes) and its causally connected models (the annotations). Such a synchronization problem occurs if an annotation provides monitoring code for a component which has already crashed. The attached probes would then simply yield no runtime data instead of communicating a possible error. Another form of deviation is introduced if the system structure changes, which means that new probes must be generated accordingly. Such form of automation is currently not implemented. As a result, a current limitation is consistency, and more runtime checks are needed in the future when attaching new probes to a running system.

VII. Related Work

The tool SM@RT [23], [24] is similar to our approach and maintains the causal connection between the system and its architecture model in a bidirectional way. This is achieved by creating a runtime architecture infrastructure [20], [21] without modifying the system under observation by using QVT model transformations. The used models are application-specific with no reusability in mind, while our approach aims to present a reusable collection of event types and needs no additional models than the ones resulting from the software design phase.

Holmes et al. [12]–[14] analyse monitored information to check compliance to regulations by using models as first-class citizens. Models are stored in a repository and are accessed via Web services at runtime. Multiple versions of models can be stored, and old versions can be used until they are migrated or not referenced any more. Our approach is similar, but supports not only compliance checks but also custom monitoring situations.

Regarding traceability, an interesting approach is proposed by Johanndeiter et al. [15] where business process modelling tools serve as dashboards with tracing capabilities. These dashboards contain business process type models which themselves contain aggregated information about their corresponding process instances, thus serving as high-level views of a process with drill down functionality to specific instances of interest. Our approach has a similar objective, but on a more fine-grained level: The business process models in the approach by Johanndeiter et al. tackle long-living and business level (i.e., high-level) processes, while we focus on models that provide views for short-living activities and operations with more technical detail (like UML). While this seems considerably easier from an abstraction perspective (since it is closer to the solution space), it is challenging to use high-level models alone to detect unexpected runtime behaviour or make fine-grained adaptations to monitoring properties. To this end, our approach provides a language that helps to define monitoring properties directly in structural and behavioural models.

Model-based monitoring can also be achieved by using Triple Graph Grammars (TGG) [8]–[10] to support architectural monitoring. In this approach, a low-level source model is causally connected to one or more high-level target models. Synchronization is declared by TGG rules at the meta-model level for both source and target models. A similar approach is proposed by Cheng et al. [3] and Garlan et al. [6] where a low-level runtime layer observes the system, a model layer interprets recorded data with analysable architecture models, and a task layer determines new requirements. These approaches give only vague indications how a human operator can influence the monitoring properties and what commonalities regarding events can be utilized across multiple applications.

Nordstrom et al. [19] introduce a fault localization mechanism in workflow models to cope with unforeseen error occurrences. In case of an error, a simulation algorithm determines future states of the workflow assuming no external intervention or future faults. Jobs within the workflow model are annotated as desired or undesired using a metric for determining the relative desirability of a partial workflow. Similar to the approach of Johanndeiter et al. [15] above, our approach operates on a more fine-grained level, focussing on models with more technical detail, which is inevitable to realize fine-grained adaptations to monitoring properties.

Regarding model-based human intervention, Hamann et al. [11] present an approach where a detailed platform aligned model is extracted from the source code and subsequently reduced by the user by selecting central classes and associations. While this also requires human interaction, our approach focusses the examination of runtime phenomena by developing a reusable collection of event types from which custom metrics can be derived.

VIII. Conclusions

In this paper we presented a meta-model-agnostic, reusable collection of runtime event types, their mapping to model elements (of UML and other meta-models), and an approach and corresponding prototype implementation for using these concepts for supporting humans in the model-based examination of runtime phenomena. Our approach provides human users with an adequate basis for further system analysis. Our interactive approach allows humans to observe and manipulate monitoring properties on the model level. We applied our approach to a case study where we were able to find the root cause of a performance problem. Our prototype enabled us to specify monitoring properties via model annotations from which a code generator was able to generate code that contained the software probes responsible for emitting the needed events. In addition, we evaluated the relevance, generalisability and completeness of our categorization by applying it to two quite different meta-models, and discussed its limitations in terms of applicability and measurement accuracy.

As future work we plan to extend our approach so that adaptation rules can be defined based on the measured runtime metrics. Furthermore, we plan to conduct an experiment with human participants to analyse if our proposed model-based event types and metrics enable to make more precise statements about the behaviour of a running system.

References

[1] N. Bencomo. On the use of software models during software execution. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, MISE '09, pages 62–67, Vancouver, Canada, 2009.

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

[3] S.-W. Cheng, D. Garlan, B. R. Schmerl, J. a. P. Sousa, B. Spitznagel, P. Steenkiste, and N. Hu. Software architecture-based adaptation for pervasive systems. In *Proceedings of the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing*, ARCS '02, pages 67–82, Karlsruhe, Germany, 2002.

[4] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, Dec. 2006.

[5] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *IEEE Softw.*, 23(2):62–70, Mar. 2006.

[6] D. Garlan, B. Schmerl, and J. Chang. Using gauges for architecture-based monitoring and adaptation. In *Proc. of the Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia, 2001.

[7] C. Ghezzi. The fading boundary between development time and run time. In *Web Services (ECOWS), 2011 Ninth IEEE European Conference on*, pages 11–11, Lugano, Switzerland, 2011.

[8] H. Giese and S. Hildebrandt. Incremental model synchronization for multiple updates. In *Proceedings of the third international workshop on Graph and model transformations*, GRaMoT '08, pages 1–8, Leipzig, Germany, 2008.

[9] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems*, MoDELS'06, pages 543–557, Genova, Italy, 2006.

[10] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software & Systems Modeling*, 8:21–43, 2009.

[11] L. Hamann, L. Vidacs, M. Gogolla, and M. Kuhlmann. Abstract runtime monitoring with use. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 549–552, Szeged, Hungary, March 2012.

[12] T. Holmes. From business application execution to design through model-based reporting. In *Proceedings of the 2012 IEEE 16th International Enterprise Distributed Object Computing Conference*, EDOC '12, pages 143–153, Beijing, China, 2012.

[13] T. Holmes, U. Zdun, F. Daniel, and S. Dustdar. 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, pages 98–112, Hammamet, Tunisia, 2010.

[14] T. Holmes, U. Zdun, and S. Dustdar. Automating the management and versioning of service models at runtime to support service monitoring. In *Proceedings of the 2012 IEEE 16th International Enterprise Distributed Object Computing Conference*, EDOC '12, pages 211–218, Beijing, China, 2012.

[15] T. Johanndeiter, A. Goldstein, and U. Frank. Towards business process models at runtime. In N. Bencomo, R. B. France, S. Götz, and B. Rumpe, editors, *MoDELS@Run.time*, volume 1079 of *CEUR Workshop Proceedings*, pages 13–25, Miami, FL, USA, 2013. CEUR-WS.org.

[16] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[17] T. Keuler and Y. Kornev. A light-weight load-time weaving approach for osgi. In *Proceedings of the 2008 Workshop on Next Generation Aspect Oriented Middleware*, NAOMI '08, pages 6–10, Brussels, Belgium, 2008.

[18] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, Oct. 2009.

[19] S. Nordstrom, A. Dubey, T. Keskinpala, R. Datta, S. Neema, and T. Bapty. Model predictive analysis for autonomicworkflow management in large-scale scientific computing environments. In *Proceedings of the Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems*, EASE '07, pages 37–42, Tucson, AZ, USA, 2007.

[20] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[21] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 177–186, Kyoto, Japan, 1998.

[22] D. Russell. *FAD: A Functional Analysis and Design Methodology*. Phd thesis, Computing Laboratory, University of Kent at Canterbury, January 2001.

[23] H. Song, G. Huang, F. Chauvel, and Y. Sun. Applying MDE Tools at Runtime: Experiments upon Runtime Models. In N. Becomo, G. Blair, and F. Fleurey, editors, *Proceedings of the 5th International Workshop on Models at Run Time*, Oslo, Norway, Oct. 2010.

[24] H. Song, G. Huang, F. Chauvel, Y. Sun, and H. Mei. Sm@rt: Representing run-time system data as mof-compliant models. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 303–304, Cape Town, South Africa, 2010.