Architectural Design Decisions for Systems Supporting Model-Based Analysis of Runtime Events: A Qualitative Multi-Method Study

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

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

***Abstract*—Models are popular artefacts in the software development process which promise to improve stakeholder communication and the overall quality of a software system under construction. Recent research proposes that the usefulness of models is not limited only to the software design phase: Empirical evidence indicates that manual analysis of a running system is improved when models are linked to recorded runtime events. However, software architects are confronted with various design decisions when designing a system that yields the required runtime events and correlates them to the model elements they originate from. The contribution of this paper is a taxonomy of architectural design decisions distilled from a series of qualitative studies following a multi-method research study design: We utilized coding techniques from Grounded Theory to build an initial taxonomy from architectural concepts found in the literature, and verified and extended the taxonomy independently by five novice software architects in a practical scenario. The resulting taxonomy captures essential architectural decisions when implementing a system that supports the analysis of its runtime behaviour using models. We then performed initial steps towards a architectural guidance model by applying the taxonomy to another realistic scenario, following the Design Science Research method, in order to analyze the properties and deepen our own technical understanding of the captured concepts in the taxonomy.**

***Keywords*-analysis; architecture; events; model; runtime;**

I. Introduction

Models are artefacts which are created throughout the software development process to capture properties of a software system from a high-level perspective. The ongoing research about model-based techniques like domain-specific languages and model-driven engineering indicates that there is a strong interest in a shift from technology-centric processes to solutions which are closer to the problem space [1]. This shift requires that models are used not only as informal sketches of the actual implementation, but also as machine-readable input for model transformation and code generation techniques to close the gap between the problem space and the solution space in an automated way. Although informal models are still very common in industry [2], model-driven approaches promise to improve productivity and maintainability when developing software systems [3].

However, while the majority of model-driven approaches close the gap between the problem space and the solution space at design time, relating models and runtime behaviour, e.g., to analyze the running system from a high-level perspective, is often neglected. Using models at runtime allows to apply changes to a running system closer to the problem space without restarting the system [4]–[6], and enables the measurement of runtime characteristics and identification of problems while the system is up and running [7].

There exists a broad range of scientific evaluations regarding approaches using models at runtime [8]. The results of a recent controlled experiment indicate that using models can improve the quality of answers to questions about the running system, and thus the comprehension of its behaviour [9]. Furthermore, there is a recognizable trend from simple adaptation interests to wider application fields and more goal-oriented and user-centric approaches [8], [10].

However, researchers put a lot of effort into the development of monitoring, adaptation and synchronization mechanisms between models and the running system, but so far hardly any approaches address the human factors when analyzing runtime behaviour with the help of models [8]. Systems with model-based analysis support are especially challenging for software architects who are confronted with various architectural issues when designing such systems. Examples of such issues are the selection of adequate techniques to extract information at runtime and exchange the obtained data with the modeling environment, as well as the integration of data analysis techniques into the models that are used by the analyst. Deciding between the various solution alternatives to such issues is challenging since an exhaustive list of solution alternatives is usually not readily available, and the trade-offs between them are not stated explicitly. As a consequence, a structured overview of the available solution alternatives is needed.

In this paper, we identify the architectural challenges and solution alternatives when designing systems with model-based analysis support in a systematic way by performing a series of qualitative studies following a multi-method design [11]. The result is a taxonomy of architectural design

decisions which is based on the concept of architectural decision models [12], [13]. The proposed taxonomy captures essential architectural decisions that must be made when realizing systems whose runtime behaviour can be analyzed with the help of models. Furthermore, the taxonomy lays the groundwork for deriving an architectural guidance model which is obtained by continually integrating the experiences from practical applications of the captured solution alternatives. Our multi-method study consists of three phases:

1) We systematically distilled an initial taxonomy by applying coding concepts from Grounded Theory [14]–[16] to the data sets from a systematic literature review [8] and a paper focussed on reusable event types [17].
2) We instructed five novice software architects to design and implement a system with model-based analysis support and capture their architectural decisions. We merged the decisions into the initial taxonomy to confirm and improve this model, following the directed content analysis method [18].
3) Following the Design Science Research method [19], we applied the improved taxonomy to a realistic existing system to deepen our own technical understanding of the captured concepts and to lay the groundwork for an architectural guidance model which captures essential properties of the captured solution alternatives.

II. Background: Model-based Analysis

Consider a scenario where a user wants to analyze some high-level property of a software system, like the average response time of a communication path between two components or the overall runtime that is spent for the operations of a specific component. The targeted system parts of such high-level analyses can often be found in the software models that are created at design time. For the mentioned analysis tasks, possible model elements of interest could be a dependency relationship between the communicating entities in a deployment diagram or a component in a component diagram, respectively. To perform the actual analysis in a traditional development environment, the analyst must find the corresponding implementation of these model elements, analyze the current implementation, write/deploy the necessary monitoring code which yields the required information, and analyze the information that is produced at runtime.

In this paper, we assume that the produced runtime information can be traced in the form of events, meaning that the information (i.e, the event properties or arguments) is labelled with a time stamp to allow a chronological analysis of the behaviour of the running system [17]. The goal of model-based analysis is to monitor such runtime events that are relevant for the model elements of interest (e.g., the start and end events of a UML behaviour) and to perform the desired analysis based on those events (e.g., calculating the average runtime of the modelled UML behaviour using the time stamps of the start and end events). Therefore, 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. Relating runtime events and model elements in such a manner allows an analyst to perform analysis tasks on the model level by applying aggregation operations to the recorded events [17]. The analyst is not concerned with implementation details of the model elements of interest once the necessary monitoring code is deployed to the running system.

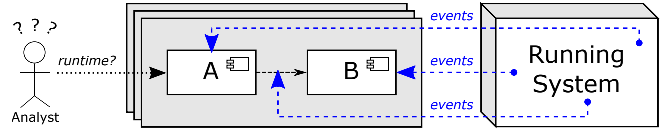


Figure 1. Abstract view of relating runtime events with model elements

While analysts are enabled by such a model-based analysis approach to perform high-level analyses of running systems, software architects are confronted with several design decisions during the creation of such systems, including:

- How are models connected to the running system?
- How are runtime events recorded and published?
- How are runtime events and model elements linked?
- How are aggregation operations incorporated?

Finding answers to such questions is inherently concerned with various decision drivers like flexibility, scalability, traceability and usability which are often assessed by software architects in a subjective manner. As a consequence, architectural guidance is needed to help software architects to balance the trade-offs between the various solution alternatives for recurring design situations when implementing systems with model-based analysis support. In this paper we derive a taxonomy of architectural design decisions following a multi-method approach and provide initial steps towards an architectural guidance model which assists software architects in choosing between solution alternatives.

III. Research Study Design

A. Planning the Multi-Method Study

Our study design had the goal that the resulting taxonomy represents a comprehensive view on model-based analysis in terms of architectural decisions and their respective solution alternatives. Another goal was to evaluate the taxonomy in an unbiased fashion and to perform initial steps towards an architectural guidance model by applying the captured solution alternatives in a realistic scenario with interesting challenges and observing their properties. As a consequence, we followed a strategy that is both exploratory and confirmatory in its nature. More precisely, we pursued a multi-method approach [11] containing a sequence of three qualitative studies as shown in Figure 2 that allowed us to create an initial taxonomy, evaluate it independently and analyze the properties of the captured solutions in a practical manner.

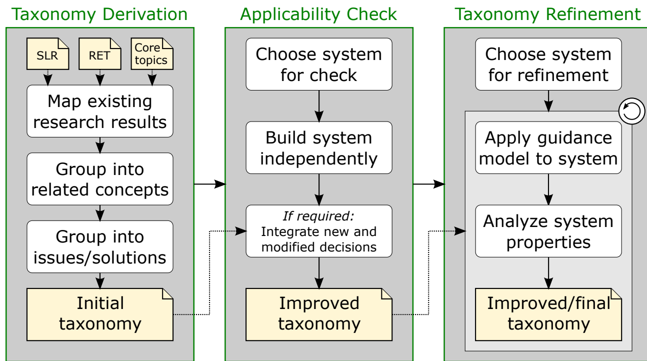


Figure 2. Our sequential, qualitative multi-method design

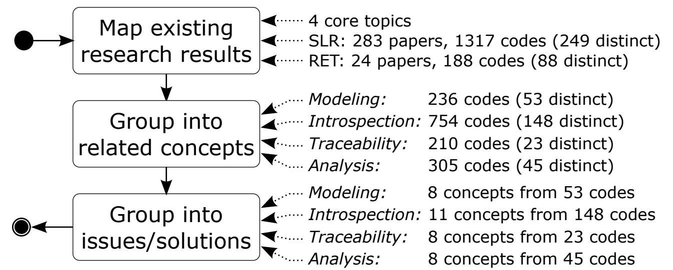


Figure 3. Details of the first phase for deriving the initial taxonomy

B. Taxonomy Derivation Phase

For the derivation of the initial taxonomy, we decided not to rely on the aggregated results of an existing systematic literature review (SLR, [8]) because they do not focus the analysis of runtime events and their levels of detail does not suffice to derive architectural decisions (e.g., they point out the importance of traceability, but give no indications of the architectural options to realize it). Instead, we relied on the coverage of the literature review and re-examined its data set of 283 referenced papers to extract architecture relevant concepts (e.g., patterns, modeling habits, languages, middlewares, development techniques) and gradually identified architectural issues and solutions to obtain the initial taxonomy. We also applied this procedure to a follow-up paper of the literature review on reusable event types (RET, [17]) to compensate the missing focus on runtime events.

For extracting the necessary data, we applied coding techniques from Grounded Theory [14] to translate the textual content of the analyzed papers into a set of architecture relevant concepts. The general coding procedure is to assign *codes* to segments of text (e.g., paragraphs) which are then grouped into *concepts* and *categories* [15]. Codes describe the key concepts found in the analyzed text segment, which in our case are architecture concepts in the form of descriptive codes [16]. We coded the content of the papers by reading every paper once, and at the same time constantly comparing codes with already assigned ones while iterating through the papers to eliminate synonyms and split codes into more fine-grained codes if necessary. Regarding coding and grouping codes, we followed a rigorous, linear process as shown in Figure 3. The process utilizes three activities: Open coding, axial coding and selective coding [15].

1) Open Coding: This is the first activity for making sense of the collected qualitative data by iterating through the collected data and assigning codes to text segments. In our case, this means iterating through the referenced papers of the systematic literate review [8] and the paper specialized on reusable event types [17], extracting architecture relevant codes (e.g., patterns, modeling habits, languages, middlewares, development techniques) while constantly updating and modifying them. It is recommended to use codes that address topics that the consumer (in our case, the architect using the taxonomy) expects and to stick to preassigned coding schemes and process codes [11].

Following these recommendations, we decided to not only extract architecture relevant concepts, but also relate them to preassigned core topics that reflect the process of designing a system with model-based analysis support from the software architect's view, as well as the process of interacting with the system from the analyst's point of view. Four core topics follow quite directly from the way how model-based analysis of running system is performed (recall Figure 1): The models of the system must exist or be created (*modeling*), the running system must be inspected (*introspection*), the recorded events must be related to the model elements (*traceability*) and finally the human user has to assess the recorded information (*analysis*). An initial assignment of extracted codes to these topics prevented us from losing focus on codes that are actually relevant for designing systems with model-based analysis support. The open coding activity paired with the assignment to the core topics is represented by the first step in Figure 3.

The result of the first activity are four buckets full of codes, ready to be grouped into collections of related codes (*concepts* according to the Grounded Theory approach). An insight into the open coding activity is shown in Table I which visualizes the top 10 assigned codes in terms of their frequency during the coding process. Another insight is shown in Table II where the numbers of assigned codes are clustered with respect to the core topics and architectural concepts found in literature. Detailed results of the open coding activity (e.g., the exact codes assigned per paper) and the study in general can be found online[1].

2) Axial Coding: Axial coding follows after open coding and entails the grouping of codes with shared commonalities into concepts, or more generally the process of relating some abstract category to its subcategories [15]. We performed the axial coding activity twice: In the first pass, we grouped codes with respect to conceptual similarities (e.g., *Debugger*, *GDB* and *JTAG* of the introspection topic relate to the

[1] see: http://jarvis.fhwn.ac.at/taxonomy/

Table I
TOP 10 CODES IN TERMS OF THEIR FREQUENCY DURING OPEN CODING

Modeling		Introspection		Traceability		Analysis	
Code	**#**	**Code**	**#**	**Code**	**#**	**Code**	**#**
Components	25	Monitor	47	CodeGen	52	XYDiagram	50
UML	20	MAPE	39	ModelTrafo	47	Vis.Extension	23
MOF	18	Middleware	38	Repository	18	Annotation	22
EMF	15	AOP	29	QVT	17	Trace	21
StateModel	14	Reflection	28	ID	16	OCL	19
BPMN	14	CORBA	25	TGG	12	Highlighting	15
FeatureModel	12	Instrumentat.	24	EventLog	11	GoalModel	13
ArchModel	12	WebService	20	QVT-R	6	Profile	13
GoalModel	11	Interpreter	20	Trace	5	Report	11
ActivityModel	8	Weaving	19	UUID	4	BarChart	11

Table II
NUMBER OF CODES ASSIGNED TO THE FOUR CORE TOPICS MODELING (M), INTROSPECTION (I), TRACEABILITY (T) AND ANALYSIS (A) WITH RESPECT TO CONCEPTS FOUND IN LITERATURE (SLR [8], RET [17])

Source	Architectural Concept	# Papers	# Codes (M/I/T/A)
SLR	Various architectures	63	39 / 200 / 63 / 49
SLR	Kinds of models	53	51 / 117 / 25 / 55
SLR	Autonomic control loop	14	7 / 87 / 6 / 8
SLR	Introspection	15	15 / 43 / 12 / 8
SLR	Model conformance	12	7 / 59 / 5 / 8
SLR	Model comparison	4	4 / 19 / 2 / 4
SLR	Model transformation	11	8 / 29 / 30 / 6
SLR	Model execution	17	18 / 44 / 7 / 16
SLR	Adaptation	23	8 / 21 / 9 / 7
SLR	Monitoring	32	27 / 87 / 42 / 29
SLR	Abstraction	11	4 / 22 / 25 / 14
SLR	Consistency	9	10 / 25 / 3 / 7
SLR	Policy enforcement	17	14 / 34 / 20 / 26
SLR	Error handling	18	15 / 40 / 9 / 39
RET	Event processing	11	14 / 47 / 18 / 21
RET	Model integration	13	11 / 36 / 14 / 19

general concept of a *debug API*) and to abstract concrete frameworks and tools to their underlying technical concepts (e.g., *AspectJ* is a framework for the *aspect-oriented programming* concept). During the first pass, we also dismissed codes that were too broad or turned out to be negligible for architectural decisions (e.g., *HTTP* and *TCP/IP*).

In the second pass, the condensed concepts from the first pass were considered tool-independent solution alternatives which were grouped into actual architectural issues they are trying to solve. For example, the aforementioned concepts *aspect-oriented programming* and *debug API* are two solution alternatives for recording runtime data (events in our case). For the traceability core topic, another example are the concepts *identifier-based correlation* and *model-based correlation* which represent solution alternatives for relating runtime events with the model elements they originate from. The two passes of the axial coding activity are represented by the second and third step in Figure 3, respectively.

3) Selective Coding: The selective coding activity follows the axial coding activity and is concerned with integrating the emerged concepts into an overall structure by describing relationships between them. In our case, relating the concepts to the overall category of model-based analysis was implicitly done by defining the four core topics at the beginning of the coding process. An example is aspect-oriented programming, which is a solution alternative for the issue of recording runtime data, which in turn belongs to the core topic of modeling issues, which in turn is a specific aspect of designing systems with model-based analysis support. As a consequence, the selective coding activity has no counterpart in Figure 3.

The initial taxonomy was obtained by mapping the resulted groupings onto instances of architectural decision meta-model concepts. Architectural decision models capture architectural decision issues, alternatives and outcomes and the interrelationships between them [13]. More precisely, we utilized a subset of architectural decision meta-model concepts introduced by Zimmermann et al. [13] for the mapping: An *issue* represents a recurring architectural issue (cf. concept), *alternatives* describe the solution alternatives for each issue (cf. code) and *topic group* is a supplemental concept to cluster architectural issues into topics which can be nested for additional structuring (cf. category).

C. Applicability Check Phase

Since the first phase was solely focussed on the literature, in the second phase we needed to check the applicability of the taxonomy in practice and integrate newly acquired knowledge, if necessary. For this reason, we instructed five novice architects during a course held at the University of Applied Sciences Wiener Neustadt to design and implement a LEGO robot system with model-based analysis support, meaning that runtime characteristics of an autonomously acting LEGO robot should be measurable on the model level. The novice architects had programming experience between 4 and 7 years, software architecture and industry experience between 1 and 3 years and were able to utilize contemporary tools and processes to develop software solutions for industry partners. The novice architects were given one month to realize the system. We deliberately decided to use novice architects to complement the first phase because they are inherently required to perform their own research about how to design such a system, so they might stumble across modern practical approaches which were not found during our analysis of the scientific literature.

The developed robot should be able to calibrate its sensors, follow user-defined paths, receive step-by-step directions from an external operating software and discover its environment autonomously to build a digital map of its surroundings. While in discovery mode, spontaneous tasks may appear which require either manual user interaction or automatic solving strategies. The external operating software should be able to retrieve a map of the detected environment and introduce new tasks and solving strategies for the robot. Designing the system involves some interesting

challenges and various architectural considerations for both the functionality of the robot and the implementation of the model-based monitoring environment. Since it is an embedded mobile system, the novice architects had to take constrained resources, network outages and platform limitations into consideration. Furthermore, the system requires the architects to care about all facets of model-based analysis, which makes their architectural decisions suitable for cross-checking the initial taxonomy.

The architects were instructed to document their software designs and design decisions in a form of their choice, which then had to be integrated into the existing taxonomy. For this research task we considered three distinct research methods: Conventional, summative and directed content analysis [18]. The conventional approach repeats the coding procedure from the first phase to build a second taxonomy which is then synchronized with the initial one. The summative approach quantifies certain words in the produced artefacts with the purpose of understanding their contextual use and builds relationships to the existing taxonomy. The directed approach begins coding the artefacts with the predetermined codes from the first phase, whereas text segments that cannot be categorized indicate new or adapted codes. The first two approaches are not viable since the application of the conventional approach to a single project does not yield a comparable taxonomy, while word frequencies of the summative approach do not yield meaningful information when applied to artefacts like models or code. Hence, we applied the directed content analysis method by coding the artefacts using the codes from the axial coding activity from the first phase (after removing tool-specific aspects).

Through this triangulation of methods and data sources [20], we gain confidence that the initial version of the taxonomy is indeed a comprehensive view on model-based analysis of running systems. In addition, where the documented issues and solutions of the team are missing in the initial taxonomy, we improved it by integrating them.

Note that the initial taxonomy was not introduced to the architects beforehand, so their design decisions were independent from the ones we distilled from literature. As a consequence, the novice software architects were forced to conduct their own research to identify architectural difficulties and possible solution alternatives.

D. Taxonomy Refinement Phase

In Design Science Research, knowledge and understanding are achieved through the building and application of a designed artefact [19]. We used this method to make initial steps towards an architectural guidance model by applying the captured solution alternatives in practice and observing their properties, but also to deepen our own understanding of the design alternatives: In a complex engineering discipline, insights purely based on the literature are prone to misinterpretation by researchers because of limited understanding of the technical details of the designed artefact – which this phase helped us to eliminate. More precisely, the phase helped us to get a clearer picture of the advantages and disadvantages of the solution alternatives captured in the taxonomy. Since this information is essential for a guidance model [13], in the third phase we applied the improved taxonomy to an existing, realistic software system following the Design Science Research method [19] for the refinement of the captured concepts, especially the qualitative properties like pros and cons of captured solution alternatives. A central idea of Design Science is an iterative cycle of developing and evaluating an innovative artefact [19]. Applied to our case, the innovative artefact is the taxonomy of architectural design decisions which is iteratively refined by applying it to an existing system and observing the process (e.g., the enactment of solution alternatives and the dynamic behaviour of the resulting system).

To further minimize researcher bias, we decided not to implement a system ourselves, but rather retrofit the captured solution alternatives to an existing system. The plan was to apply every solution alternative captured in the taxonomy one after another to the existing system, if possible with reasonable effort. We chose the video game Mars Simulation Project version 3.07, an open source social simulation of future human settlement of Mars, written in Java, as the study object. It has a reasonable size (213.794 lines of code), provides UML diagrams that describe the essential parts of the software and possesses a good software quality with a deliberate use of best practices and design patterns. Similar to the LEGO system used for the second phase, the system poses some interesting questions, like how different methods of recording events affect the performance of the game.

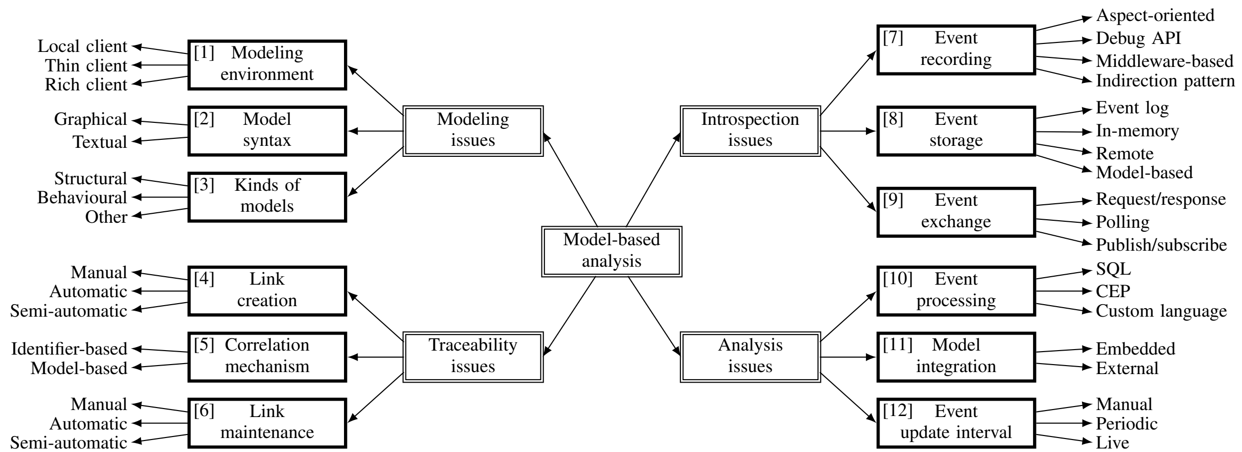
The application of the taxonomy to a realistic scenario is a suitable addition to the preceding phases as their coding processes yielded the architectural issues and solution alternatives, but hardly any properties of the captured approaches.

IV. Study Results

A. Taxonomy Derivation Results

The initial taxonomy that was extracted in the first phase is shown in an overview in Figure 4. We use double-bordered boxes to represent architectural topics, bold boxes for architectural issues and borderless nodes for solution alternatives. Containment relationships between topics, topics and architectural issues and between architectural issues and their solution alternatives are represented by black arrows.

Advanced relationships between architectural topics and architectural issues like refinements, triggers, enforcements and incompatibilities [13] are not shown in the graphical representation of the taxonomy. Descriptions of solution alternatives have no visual representation, we provide them in textual form in Table III. More detailed descriptions can be found in the online resources linked in Section III-B1 where we also provide an interactive version of the taxonomy.



Decision drivers:

[1] Dependability, extensibility, scalability
[2] Accessibility, understandability, usability
[3] Analyzability, traceability
[4] Efficiency, scalability, traceability
[5] Correctness, interoperability, traceability
[6] Efficiency, scalability, traceability
[7] Adaptability, flexibility, modularity, orthogonality
[8] Analyzability, availability, correctness, efficiency, fault tolerance, interoperability, scalability
[9] Efficiency, responsiveness, scalability, up-to-dateness
[10] Analyzability, expressiveness, usability
[11] Customizability, extensibility, usability
[12] Availability, responsiveness, up-to-dateness

Figure 4. Taxonomy which captures architectural topics (double-bordered), issues (bold-bordered) and solution alternatives (borderless)

Table III
DESCRIPTIONS OF THE SOLUTION ALTERNATIVES THAT WERE IDENTIFIED BY THE CODING PROCESS OF THE FIRST PHASE

Topic	Issue	Solution Alternative	Description and Notes
Modeling issues	[1]	Local client	Modeling environment and observed system run on same machine. Allows to utilize existing modeling frameworks like Eclipse EMF.
	[1]	Thin client	Analyst has only a browser for models and analyses. Observed system depends on modeling, remoting and adaptation libraries.
	[1]	Rich client	Analyst has a modeling environment which performs analysis locally. Remote observed system has reduced dependencies/footprint.
	[2]	Graphical	Graphical modeling environments (e.g., Eclipse Sirius) have customizable layers to formulate queries and display analysis results.
	[2]	Textual	Textual models are defined easily, but additional views may be required to formulate queries and display analysis results for them.
	[3]	Structural	Techniques like model transformation can be used to generate implementation and monitoring code for models describing structure.
	[3]	Behavioural	Generation of data and control flows for behavioural models is more complex than for structure, especially for high-level models.
	[3]	Other	For non-structural/non-behavioural models (e.g., goal models), automation of analysis and finding suitable runtime events is harder.
Traceability issues	[4]	Manual	Manually writing monitoring code which yields runtime events is flexible, but only feasible for small and easy-to-change systems.
	[4]	Automatic	A generator takes model elements as input and injects references to them in the generated monitoring code. Bridges abstraction gaps.
	[4]	Semi-automatic	Some traceability links originate from generated code, some from manually written monitoring code (e.g., for non-structural models).
	[5]	Identifier-based	A unique ID of a model element is passed to the instantiation routine of runtime events. Prone to inconsistency if a model evolves.
	[5]	Model-based	An event log is a model itself which is able to correlate model elements and event data. Robust against small-scale model evolution.
	[6]	Manual	Overcome inconsistencies between models and monitoring code by manually adapting the code. Only feasible for small systems.
	[6]	Automatic	Re-generation and re-deployment of monitoring code from a model in case the model changes. Requires undeployment of old code.
	[6]	Semi-automatic	Mixture of re-generating monitoring code for easily traceable system parts and manual adaptation of monitoring code for the rest.
Introspection issues	[7]	Aspect-oriented	Monitoring instructions are written in separate components called aspects. Special compilers weave the aspects into the actual code.
	[7]	Debug API	Utilize platform-specific debug interfaces to record events. Low-level, but like aspects, does not require changes to the business logic.
	[7]	Middleware-based	Utilize middleware-specific interfaces to record events (e.g., CORBA interceptors). Unobtrusive, but limited to predefined event types.
	[7]	Indirection pattern	Design patterns allow to change behaviour (and thus, monitoring) at runtime. Requires deliberate preparation of the observed system.
	[8]	Event log	Persistent log which may outlive the observed system. Enables post-mortem analysis. Requires management of concurrent access.
	[8]	In-memory	Observed systems keeps recorded events in memory. Prone to crashes. On-demand retrieval of events by the modeling environment.
	[8]	Remote	Observed system sends occurring events immediately to connected modeling environments. No overhead if no one is connected.
	[8]	Model-based	Special case of an event log in form of a model. Traceability links to model elements are directly encoded into the model.
	[9]	Request/response	Modeling environment retrieves events of the running system on an on-demand basis. Requires the observed system to store events.
	[9]	Polling	Modeling environment regularly requests runtime events from the running system. Can be combined with (model-based) event logs.
	[9]	Publish/subscribe	Modeling environment subscribes to the running system or the event log (if possible, e.g. through a database). Enables live analysis.
Analysis issues	[10]	SQL	Query language of choice if events are stored in a database. Cumbersome for complex analyses (e.g., chronologically related events).
	[10]	CEP	Complex event processing techniques are similar to SQL, but offer expressive mechanisms to handle time series of events and data.
	[10]	Custom language	Custom language to filter and aggregate streams of events (e.g., using functional programming concepts). Requires customized editor.
	[11]	Embedded	Analysis results (e.g., the average runtime of a modelled behaviour) are shown directly in the model. Often used in graphical models.
	[11]	External	Analysis results are shown in external views, for example a query output window. No integration effort, but usability considerations.
	[12]	Manual	A single read of the event log, request from the running system or read of the event log model. Initiated actively by the analyst.
	[12]	Periodic	Constantly updating the analysis results by regularly requesting stored events of logs, model-based event logs or the running system.
	[12]	Live	Analysis results are always up-to-date. Requires the modeling environment to register at the event source (e.g., the running system).

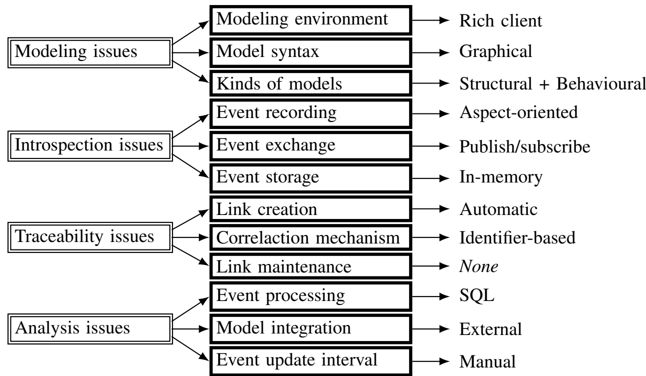


Figure 5. Made decisions using the notion of the initial taxonomy

B. Applicability Check Results

Regarding the independent check of the initial taxonomy using the LEGO robot system, the novice architects decided to realize the system with the help of leJOS, which is a tiny virtual machine capable of executing Java programs on the robot. This allowed the team to utilize object-oriented principles in the design and implementation. The robot software was realized using model-driven techniques by generating Java stubs from 6 UML models (use case, component, package, class, activity and state diagram) and writing corresponding Java code for the generated artefacts.

All models except the use case diagram were used to perform model-based analysis. The team generated not only Java stubs from the UML models, but also corresponding AspectJ monitoring code which captured essential events of the modelled entities (e.g., the start and end of a modelled UML behaviour). Each event contained the fully qualified name of the model element the monitoring code originated from, which was known from the code generation process. For performance reasons, the architects decided not to perform any storage or aggregations of events on the constrained robot system. Instead, recorded events were immediately serialized and sent to the external operating software when connected to the robot, or else discarded.

On the external operating software side, recorded events were inserted into a relational database. The analyst was provided with the UML models for the analysis and a simple web-based SQL interface to query the properties of the recorded events. Runtime characteristics for specific model elements could be calculated by filtering the rows according to the fully qualified name of the model element of interest.

Regarding the integration of decisions into the initial taxonomy, we applied the directed content analysis method to the documentation artefacts produced by the architects. Figure 5 shows how the design decisions of the novice architects are represented by the architectural issues and solution alternatives of the initial taxonomy. We can see that every design decision has an appropriate counterpart in the taxonomy. However, technical details such as concrete selections of tools and frameworks are deliberately not covered by the taxonomy. Note that traceability link maintenance was not covered by the implemented system, meaning that the novice architects planned no corrective actions if the UML models and recorded events drift apart (e.g., renaming model elements would invalidate the stored events in the database, since they rely on the fully qualified model element names). Also note that two solution alternatives for the kinds of models are combined, since both structural and behavioural UML models were used for the analysis.

The mapping of the design decisions onto the initial taxonomy, paired with the comprehensive literature we distilled the taxonomy from, gives us confidence that the taxonomy poses a suitable starting point for creating an architectural guidance model for developing systems with model-based analysis support. The fact that the taxonomy captured a decision which the novice architects were not aware of indicates that the taxonomy can also ensure that essential architectural facets are covered by the system design.

C. Taxonomy Refinement Results

For further refinement of the qualitative properties of solution alternatives and laying the groundwork for an architectural guidance model, we iteratively applied the captured techniques to the Mars Simulation Project according to the Design Science Research method. The techniques were applied in various combinations over the course of 12 iterations and the insights – mainly their pros and cons – recorded. Details of the solution alternatives and their properties can be found in the mentioned online resources.

1) Modeling Issues: Regarding the modeling environment, we decided to use a local client, that is, we started the modeling environment (Eclipse in our case) and the game under observation on the same computer. Furthermore, we relied on the structural and behavioural UML diagrams that were already available from the project documentation and used Eclipse Sirius as graphical modeling environment.

2) Introspection Issues: Regarding the issue of recording events, we decided to use AspectJ since it does not require changes to the targeted source code. Using alternatives like middleware-based or pattern-based recording would require extensive changes to the original implementation.

Our first approach was to simply dump the events in a log which is then read by the modeling environment. This approach turned out to be highly inefficient, depending on the component that yielded the events. For components with a high call count (e.g., the game rendering component), we recorded up to a million events in one minute – too much for constantly writing and reading the log. For the same reason, model-based event storage (e.g., through XMI serialization) was inappropriate, so in the next iteration we chose in-memory storage and thus sacrificed the possibility of storing the events persistently for post-execution analyses.

For the exchange of events between the modeling environment and the running system, a polling strategy combined with the high amount of events led to an excessive memory consumption of the observed system since events had to be stored between subsequent retrievals. We then switched to a publish-subscribe strategy where events were not recorded as long as the modeling environment was not connected.

3) Traceability Issues: While implementing link creation and maintenance in a fully automatic way through model transformation and generation of AspectJ constructs, we realized that this approach is suitable for models which represent the structure and behaviour of the system on a technical level (e.g., a class diagram of the units on Mars), but hard for models which do not (e.g., a use case diagram) since the exact counterpart on the implementation cannot easy be identified in an automated way. For those models we wrote the link management manually, thus following an overall semi-automatic approach. Regarding correlation, we did not apply model-based correlation because of aforementioned performance reasons, but used unique identifiers in the generated code to relate events with the models.

4) Analysis Issues: We implemented an extension of Eclipse Sirius which allows the analyst to write filtering and aggregation expressions in a custom language directly in the graphical models by annotating the model elements of interest. Although the initial development and integration effort of the language was high, it allowed the analyst to process events directly in the models with the possibility of reusing existing views on the architecture. The extension also provided a live update of the analysis results by utilizing the publish-subscribe interface of the running system and showing the expression results directly in the models.

5) Summary: The application of the taxonomy demonstrated that choosing between solution alternatives highly depends on the domain and individual project requirements. In the domain of video games, the case showed that performance considerations have to be weighted against analyzability as well as resource-related concerns and have great impact on architectural issues regarding event storage, event exchange, correlation mechanism and event update interval. Aspect-oriented techniques turned out to be a performance-friendly solution for extracting events without the need of changes to the original code, but further analyses are needed to test if dynamic aspect-oriented programming variants (i.e., changing pointcuts and advices at runtime to meet changing monitoring requirements) can cope with the amount of events in terms of performance because they usually rely on additional runtime checks.

For the analysis of runtime events, using SQL or other existing event processing languages is usually accompanied by maintaining additional resources (e.g., a database server) and additional integration issues. Creating an own analysis language is concerned with an initial effort, but may lead to an overall effort reduction if applied in multiple projects.

V. Discussion

A. Resulting Taxonomy of Architectural Design Decisions

Our multi-method study gives us confidence that the proposed taxonomy is a suitable starting point to provide architectural guidance for designing systems with model-based analysis support. The comprehensive nature of the literature the systematic coding process was based on ensures that the most important techniques and architectural approaches are adequately reflected in the taxonomy. Furthermore, retrofitting the architectural decisions of the novice architects onto the proposed taxonomy demonstrated that the taxonomy can indeed be applied when designing systems with model-based analysis support. We validated our understanding of the technical details of the taxonomy and made initial steps towards an architectural guidance model in a confirmatory Design Science Research project on a realistic case.

However, it is possible that the proposed taxonomy needs to be updated in the future if new issues are raised or new solution alternatives are discovered. In fact, architectural decision models (which our taxonomy is based on) are intended to be refined in an iterative manner by harvesting the insights and lessons learned of project outcomes [12]. As a consequence, applying the taxonomy in ongoing research projects can yield valuable feedback to improve the taxonomy and turn it into a more comprehensive guidance model by integrating the feedback into our observations presented in the online resources. Nevertheless, we are confident that the taxonomy is quite stable because we used the process of performing model-based analysis as foundation for the derivation of the model instead of concrete technologies or trends, which are prone to changes in the future.

Technical details about the modeling environment and the running system can be used to extend the taxonomy, however. In the current version, we neglected supplemental structuring concepts which enable the categorization of architectural topics into conceptual, technology and asset-specific levels to emphasize both technical and strategic concerns [12]. An incorporation of these concepts would allow us to include different tools and frameworks into the taxonomy to provide a more holistic view of model-based system analysis. Furthermore, with a more fine-grained structuring of architectural issues it may be possible to integrate supplemental technologies like model transformation explicitly into the model instead of describing them implicitly in the various solution alternatives.

B. Threats to Validity

We identified the relevant architectural issues and solution alternatives by analyzing the approaches and the related work of a comprehensive systematic literature review [8] and a follow-up paper on reusable event types [17]. However, additional sources like expert opinions and practitioner literature (e.g., blogs) may yield further techniques to be

considered for complementing the taxonomy and create a more comprehensive guidance model. We argue that finding experts and appropriate literature (i.e., is model-based, deals with runtime analysis, clearly states the used techniques, etc.) is much harder than summarizing indexed scientific literature. Furthermore, many practical approaches are concrete implementations of concepts found in scientific literature. We explicitly extracted those concepts by abstracting from concrete tools and frameworks. Another source of design decisions are modern deployment strategies provided by DevOps and micro-service platforms, which are only covered slightly by the literature used in our study.

While extracting the initial taxonomy, we used a preassigned coding scheme with four core topics to help focus on codes that are relevant for the design and implementation of systems with model-based analysis support. Other coding schemes are possible, but they would not necessarily have the architect's view in mind. We argue that performing open coding from the beginning without some predefined scope might lead to codes and concepts that are irrelevant for the software architect's tasks when designing a system with model-based analysis support. The level of detail of the assigned codes inherently depends on the researcher who performs the coding, which means that a certain amount of bias cannot be eliminated completely.

The taxonomy could be evaluated (and consequently, improved) with respect to many other criteria than the ones presented in this paper. The second phase of our multi-method study already demonstrates that the taxonomy is able to indicate and force architects to consider certain architectural decisions. As a consequence, the degree of support the taxonomy can actually provide for architects when designing systems from scratch is an interesting metric to analyze. Additional experiments are necessary to validate the taxonomy with respect to reduction in effort and time when designing a system with model-based analysis support.

In a complex technical domain, there is always the risk of technical misinterpretations by the researchers. We mitigated this risk by applying the techniques in our taxonomy ourselves in a Design Science Research project in order to deepen our own technical understanding.

VI. Related Work

Structured analysis of runtime events, especially in the form of analyzing event logs, is widely adopted to understand the behaviour of running systems [21]. Runtime events are not only used for tracking the behaviour of a running system on a low abstraction level (e.g., by capturing methods calls and state changes), but also for analyzing the runtime behaviour of a system from a high-level perspective like business processes. Rozinat and van der Aalst [22] describe an approach where events related to the start and completion of process activities are checked for conformance against given process models. Results can directly be integrated into the graphical notation of the process models, e.g. to highlight currently active process activities or faulty process activities in case of model evolutions [23]. Our proposed taxonomy covers software models of all kinds, i.e. the presented issues and solution alternatives are independent from the level of abstraction provided by the models used for the analysis. For example, if an architect decides to utilize code generation for creating the monitoring code, the generated artefacts could be as simple as Java statements, but also high-level configuration scripts which instruct some business process execution engine to yield the required runtime events.

Regarding our proposed taxonomy, architectural decision models make architectural decisions explicit and capture architectural knowledge which would otherwise exist solely in the mind of software architects and decision makers. Zimmermann et al. [13] describe the relevant concepts and their interrelationships of architectural decision models. The authors also provide formal definitions of dependency relations, integrity constraints and production rules within architectural decision models which we discussed, if at all, in a rather informal and selective way when deriving the taxonomy. There is no problem in integrating these concepts into the taxonomy, but we decided to keep it simple and prevent overloading of the taxonomy with additional relationships in the graphical representation.

There also exists advancements in architectural decision modeling to incorporate traceability links to artefacts of the software engineering lifecycle. Capilla et al. [24] propose meta-model extensions which allow to trace individual decisions to their affected artefacts via fine-grained traceability links. Such traceability links can also be applied if the decision networks becomes more complex, whereas the software architect has to decide about the granularity of the used traceability links. The proposed extensions lead to an improvement in impact analysis processes since changes in the requirements or the implementation code can be traced to the affected parts of the architecture more easily.

While the concept of architectural decision models was successfully applied and extended to the domains of information technology services [25] and service-oriented architectures [12], the current literature provided no architectural guidance for the implementation of system where runtime events are fed back to their corresponding model elements. The previously conducted systematic literature review [8] accumulates approaches that utilize models at runtime, but it neither provides any architectural guidance for implementing systems with model-based analysis support, nor does it focus the analysis of runtime events. As a consequence, this paper and the previous work do not render each other obsolete.

VII. Conclusions

In this paper we derived a taxonomy of architectural design decisions from existing literature using a qualitative multi-method study. The study entailed a structured

exploratory phase using coding techniques from Grounded Theory to build the taxonomy and two confirmatory phases to improve the taxonomy and prepare its extension to an architectural guidance model. The resulting taxonomy captures architectural decisions when realizing systems whose runtime behaviour can be analyzed with the help of models. We discussed the potential solutions to monitoring, introspection, traceability and analysis issues and described some of their advantages and disadvantages with respect to their associated decision drivers. The validation phases of our multi-method study showed that the taxonomy can indicate design decisions missed by the architects and provided insight into the properties of captured solution alternatives by applying them in a realistic case.

REFERENCES

[1] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22–27, Oct. 2009.

[2] M. Petre, "Uml in practice," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 722–731.

[3] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of mde in industry," in *Software Engineering (ICSE), 2011 33rd International Conference on*, May 2011, pp. 471–480.

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

[5] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg, "Models@ run.time to support dynamic adaptation," *Computer*, vol. 42, no. 10, pp. 44–51, Oct. 2009.

[6] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, "Using architecture models for runtime adaptability," *IEEE Softw.*, vol. 23, no. 2, Mar. 2006.

[7] J. Cito, P. Leitner, H. C. Gall, A. Dadashi, A. Keller, and A. Roth, "Runtime metric meets developer: Building better cloud applications using feedback," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2015. New York, NY, USA: ACM, 2015, pp. 14–27.

[8] M. Szvetits and U. Zdun, "Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime," *Software & Systems Modeling*, pp. 1–39, 2013.

[9] ——, "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*, ser. MODELS '16. New York, NY, USA: ACM, 2016.

[10] N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, and E. Letier, "Requirements reflection: requirements as runtime entities," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 199–202.

[11] J. Creswell, *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications, 2013.

[12] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster, *Reusable Architectural Decision Models for Enterprise Application Development*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 15–32.

[13] O. Zimmermann, J. Koehler, F. Leymann, R. Polley, and N. Schuster, "Managing architectural decision models with dependency relations, integrity constraints, and production rules," *J. Syst. Softw.*, vol. 82, no. 8, Aug. 2009.

[14] B. Glaser and A. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*, ser. Observations (Chicago, Ill.). Aldine Publishing Company, 1967.

[15] A. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 1998.

[16] M. Miles and A. Huberman, *Qualitative Data Analysis: An Expanded Sourcebook*. SAGE Publications, 1994.

[17] M. Szvetits and U. Zdun, "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)*, Sept 2015, pp. 4–13.

[18] H.-F. Hsieh and S. E. Shannon, "Three approaches to qualitative content analysis," *Qualitative Health Research*, vol. 15, no. 9, pp. 1277–1288, 2005, pMID: 16204405.

[19] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Q.*, vol. 28, no. 1, pp. 75–105, Mar. 2004.

[20] M. Q. Patton, "Enhancing the quality and credibility of qualitative analysis." *Health Services Research*, vol. 34, no. 5 Pt 2, pp. 1189–1208, Dec 1999, 10591279[pmid].

[21] D. Jayathilake, "Towards structured log analysis," in *Computer Science and Software Engineering (JCSSE), 2012 International Joint Conference on*, May 2012, pp. 259–264.

[22] A. Rozinat and W. M. P. van der Aalst, "Conformance checking of processes based on monitoring real behavior," *Inf. Syst.*, vol. 33, no. 1, pp. 64–95, Mar. 2008.

[23] S. Rinderle, M. Reichert, and P. Dadam, "Supporting workflow schema evolution by efficient compliance checks," DBIS, Technical Report UIB-2003-02, May 2003.

[24] R. Capilla, O. Zimmermann, U. Zdun, P. Avgeriou, and J. M. Küster, *An Enhanced Architectural Knowledge Metamodel Linking Architectural Design Decisions to other Artifacts in the Software Engineering Lifecycle*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 303–318.

[25] O. Zimmermann, C. Miksovic, and J. M. Küster, "Reference architecture, metamodel, and modeling principles for architectural knowledge management in information technology services," *J. Syst. Softw.*, vol. 85, no. 9, Sep. 2012.