On the Need for Extended Transactional Models@Run.Time

Mahdi Derakhshanmanesh[1], Marvin Grieger[2], and Jürgen Ebert[1]

[1] University of Koblenz-Landau, Institute for Software Technology, Germany
{manesh,ebert}@uni-koblenz.de
[2] University of Paderborn, Department of Computer Science, Germany
marvin.grieger@uni-paderborn.de

Abstract. Models at runtime that are causally connected to the base software play a central role in the software architecture of flexible and self-adaptive software systems. Decisions based on them may be wrong if the models are accessed or modified in a state different than assumed. Although transaction concepts for model repositories have been presented in analogy to well-established transaction concepts for databases, they do not sufficiently address the specific needs for models at runtime. In this paper, we describe some of the extended features expected from a runtime environment for Transactional Models@Run.Time (TMRT). These features are derived from issues identified in the context of the Graph-based Runtime Adaptation Framework (GRAF).

Keywords: Transactions · models at runtime · self-adaptive software

1 Introduction

Software models have traditionally been used during design, implementation and verification phases of software development. However, models can be also used at runtime to realize flexible and adaptive software systems. These models can be either abstractions of an underlying system programmed in code, e.g., as proposed under the term *models@run.time* [8], or they represent stand-alone information, e.g., executable UML activity diagrams [4]. In both cases, the software models are connected to the rest of the software application.

From a high-level perspective, the resulting architectures usually follow a *layered approach* as depicted in Figure 1. This structure is taken from previous research on the *Graph-based Runtime Adaptation Framework* (GRAF) which uses models at runtime to achieve adaptivity [4].

The first two layers, i.e., L_1 and L_2, actually conform to the design pattern of *architectural reflection* [9] if the models are models@run.time representing the software's architecture. In this setup, the models at runtime are *causally connected* [13] to the adaptable software's architecture and state. Any changes in the adaptable software are propagated to the models (*reification*) and changes in the models are propagated back to the adaptable software (*reflection*).

The second and the third layer, i.e., L_2 and L_3, are also connected to each other but in a slightly different way. Changes in the models at runtime are monitored by the adaptation manager (*sensing*) which then analyzes and plans for potential changes (*controlling*) and finally executes the plan (*effecting*) that modifies the models. In *Self-Adaptive Software* (SAS), these steps are also known as the MAPE-K loop [11]. While concurrently executing feedback loops are possible, we start with the easier case of a single feedback loop.

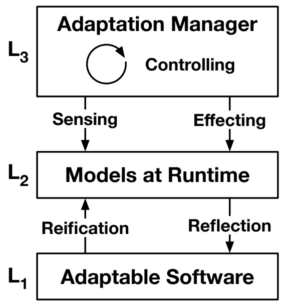

Fig. 1. Architecture of Software Using Models at Runtime (See [4])

Research Problem. Given the central role of models at runtime in a software architecture like GRAF, decisions based on them in the associated layers L_1 and L_3 may be wrong if the models are used in a state different than assumed. Various issues can arise depending on when, where and how the models at runtime are accessed.

Traditional *transaction concepts* [16] for databases tackle these problems and have become key to modern software applications. A transaction is an activity, which can only fail or succeed as a complete unit. It usually consists of a sequence of operations. The term is used in database management systems which provide support for executing a transaction as an atomic, consistency-preserving, and isolated activity with a persistent (durable) effect. Among others, a transaction concept (i) provides reliable units of actions that can be rolled back in case of failure and (ii) provides isolation between concurrently executing actions accessing a central database.

For models, transaction concepts have been developed in analogy to well-established transaction concepts for databases, and implementations such as the *Connected Data Objects* (CDO) *Model Repository* [1] are available. Yet, these infrastructures do not address the particular needs of models at runtime. We observe that software engineers consider related issues on a case-by-case basis when constructing adaptive software systems that use models@run.time. A standardized approach is needed.

This paper tackles the following *research problems* and their challenges:

(Q1) What are transaction-related issues to be aware of when using models at runtime (e.g., to build SAS)?

(Q2) What are the specific needs for a transaction concept for models at runtime in the broader sense?

Contributions. In answering Q1, we describe concrete examples for common transaction-related *issues* with querying, transforming and interpreting models at runtime from the context of GRAF. Furthermore, in an initial attempt to answering Q2, we elicit *desired features* of a transaction concept specific to models@run.time. All in all, we aim to *raise awareness* for this topic in the research community and hope to initiate a fruitful discussion.

2 Running Example: Adaptive OpenJSIP

One of the case studies carried out to show the feasibility of evolving existing software to self-adaptive software using GRAF is the *OpenJSIP study* [4]. OpenJSIP [3] is an open-source stand-alone Java Session Initiation Protocol (SIP) server for voice-over-IP (VoIP) calls. The evolved OpenJSIP shall adapt its behavior to maintain its reliability under extreme system loads. Using adaptation rules, new registrations are automatically rejected if the system load is high.

System load is measured in the number of current server transactions from within OpenJSIP and is stored in an integer variable named `currentSrvTrans`. Moreover, there is a Boolean variable `blockRequests` that is checked at certain points in OpenJSIP depending on which calls are serviced or not. The runtime model contains reifications of these two variables. While `currentSrvTrans` is only updated from within OpenJSIP (reification), `blockRequests` can be also modified by the *Adaptation Manager* (AM) (effecting) and is used in OpenJSIP (reflection). There is also a behavioral part in the runtime model represented by activity diagrams, which – on startup of OpenJSIP – represent the phone call processing behavior as programmed in the adaptable software. Hence, a `defaultBehavior` flag is set to true. A rule engine in the AM loaded with adaptation rules for servicing and rejecting calls observes this runtime model.

GRAF provided the conceptual framework and an implementation for designing and developing this *adaptive OpenJSIP* variant. Subsequently, we relate to this example in order to motivate the need for a *transaction concept* that is specifically suited for models@run.time.

3 Issues

The following *issues* are derived from (i) transactions in databases and (ii) the application domain of SAS. We especially discuss the case of SAS with models@run.time, since this assumption allows to identify even more relevant issues. Please note that we do not claim completeness of the issue list. We addressed each of the identified issues in the GRAF case study. The goal is to point at common issues to be aware of when using models at runtime and to motivate the need for a *generic and unified approach* in contrast to ad-hoc solutions.

Lost Model Update. The runtime model, as sketched in the high-level architecture in Figure 1, abstracts from the adaptable software that reifies its state into the runtime model. This, in return, is observed by the adaptation manager which runs in a separate thread. The runtime model and the adaptable software read and write to the runtime model concurrently.

Without transactions, manipulations of the runtime model by the adaptation manager or the adaptable software may be overwritten (*lost model update*).

For example, in the context of the OpenJSIP case study, the adaptation manager reads the Boolean synchronized state variable[3] `blockRequests` in its

[3] In GRAF, a synchronized state variable (or `SyncStateVar`) is a value in the runtime model that is target of reification and source of reflection.

sensing step. Assuming that it was false before and the controlling step dictates to set it to true in order to block new incoming call requests to the SIP server, `blockRequests` is set to true during the effecting step. Now, the adaptable software might run into the issue that it reifies the old value (false) again to the runtime model and, hence, overwrites the adaptation.

This issue can be tackled by introducing *locking mechanisms* on the runtime model. While the runtime model is locked and transformed, it may be allowed to still query it (shared lock) or not (exclusive lock).

Unrepeatable Model Read. Even though single values or coherent structures in the runtime model may be often sufficient, aggregated values and structures also occur. Examples are sums of integers or associated behavioral models such as one activity diagram invoking another (sub) activity diagram.

In this setup, it is possible to read values or query structures from the runtime model that are only partially correct (*unrepeatable model read)*.

For example, in the context of the OpenJSIP case study, the model interpreter for executing the activity diagrams can run into the issue that it starts executing a behavioral model which is being transformed concurrently by the adaptation manager. Possibly, the action to register calls is still part of the behavior and is exchanged with an action to block calls right after the interpreter executed it.

This issue can be tackled by isolating the different activities on the runtime model such that they are processed in a certain order (serialization).

Dirty Model Read. Given that the runtime model is an appropriate abstraction, not all technically possible changes to the runtime model are allowed. In other words, the runtime model must always conform to a metamodel and its optional constraints.

In an architectural setup in which different parties access the runtime model concurrently, a transformed but not yet validated runtime model might be accessed. Therefore, even if the changes to the runtime model are reverted after validation, there is a time frame in which it is possible to query invalid – or "dirty" – values or structures (*dirty model read*).

For example, in the context of the OpenJSIP case study, an administrator may have provided a new and faulty adaptation rule that transforms a part of the behavioral model and adds an action to block calls but does not remove the action that registers calls. Even though a constraint in the runtime model's metamodel captures this case, the issue that can occur is that the transformed runtime model is used by the adaptable software before model validation detects this flaw and reverts the runtime model to its old state.

This issue can be tackled by isolating the different activities on a runtime model such that they are processed in a certain order (serialization).

Conflicting Model Update. In the context of building SAS with models at runtime, the adaptation manager manipulates the model (effecting) and the adaptable software manipulates it as well (reification).

In this context, independent modifications can conflict with each other such that the second action fails (*conflicting model update*).

For example, in the context of the OpenJSIP case study, an adaptation manager may detect that a reified state variable `currentSrvCpuLoad` is never used by its available adaptation rules and, thus, decides to delete it from the runtime model. Thereafter, whenever the adaptable software attempts to update the runtime model, it can run into the issue that there is no target element in the runtime model anymore. Sensing fails, at least partially.

This issue can be tackled by introducing an application-specific *conflict resolution strategy*. In the example, the deleted state variable can be (re-)created dynamically when it is accessed.

Unrepeatable Adaptation. SAS is often built using a rule-based adaptation manager. In GRAF, adaptation rules are expressed in terms of *Event-Condition-Action* (ECA) rules. The event is a (typed) notification, the condition is a query on the runtime model and the action is a transformation of the runtime model.

Assuming that the runtime model is also updated by the adaptable software during the reification step, the data used in the event, condition and action parts of an adaptation rule can change during execution such that the effecting step is not necessarily directly related to the state of the runtime model as the change event was received (*unrepeatable adaptation*).

For example, in the context of the OpenJSIP case study, the adaptable software reifies a new low value for `currentSrvTrans` that measures the system load. In reaction to this sensed change, the adaptation manager selects a specific ECA-Rule with the matching event type. While this adaptation rule is being processed with the goal to allow calls, the issue can occur that the `currentSrvTrans` is updated by the adaptable software with a high value for `currentSrvTrans`, resulting in the selection of another ECA rule for blocking calls. The condition that is now evaluated for the first ECA rule queries the new high value. Hence, the runtime model is transformed such that incoming calls are blocked. The same adaptation is then redundantly performed by the second adaptation rule.

This issue can be tackled by executing the whole rule as an atomic action (hierarchy/scope) during which no manipulation of the runtime model is allowed.

Outdated Adaptation. Using models at runtime is a common way to build software systems that adapt themselves to changing environments. Thereby, a purpose of the model at runtime can be to represent the environment itself.

If the environment changes fast, the adaptation manager might perform the MAPE loop based on outdated knowledge (*outdated adaptation*).

For example, in the context of the OpenJSIP case study, the system load is permanently reified from the adaptable software to a state variable named `currentSrvTrans` in the runtime model. The adaptation manager checks for an update of specific parts of the runtime model according to a fixed or variable strategy. If changes are sensed, the MAPE loop gets executed. Here, the adaptation manager can run into the issue that the model is updated during the execution of the MAPE loop. Depending on the change of the model, i.e., the change of the environment, the effecting that results from the MAPE loop can be disadvantageous.

This issue can be tackled by providing a capability to cancel an ongoing MAPE loop if the part of the model based on which the loop has been triggered, changes. This encompasses reverting all changes made by the MAPE loop.

Overeager Adaptation. Assuming the same scenario as in the previous section, i.e., a fast changing environment, it may occur that the adaptation manager starts over and over again without effecting (*overeager adaptation*).

For example, in the context of the OpenJSIP case study, the system load as indicated by the `currentSrvTrans` variable changes frequently. In result, the adaptation manager might run into the issue that it cancels and restarts the MAPE loop over and over again. In an extreme case, the effecting step would be rarely, or even never, executed.

This issue can be tackled by *defining tolerable derivations* in changes of the runtime model, e.g., using bounds for the sensed values and a change frequency, that still allows a proper adaptation.

Missed Adaptation. The reflection step in SAS can be realized in different ways, e.g., (i) by manipulating code in the adaptable software depending on the runtime model or (ii) by executing a part of the runtime model that represents an alternative behavior, thereby skipping the default behavior programmed in the adaptable software.

In either case, the decision to modify the adaptable software during the reflection step may be based on not yet validated data in the runtime model. In result, at the time of reflection, the state of the runtime model may suggest that the adaptable software is (still) in the conforming state whereas in reality it needs to be changed (*missed adaptation*).

For example, in the context of the OpenJSIP case study, a behavior description (e.g., for receiving calls) in the runtime model may be already transformed. Thus, the `defaultBehavior` flag in the runtime model is set to false and the runtime model is interpreted at certain points in the control flow of the adaptable software. Assume that the adaptation manager resets the behavior, i.e., the `defaultBehavior` flag in the runtime model is set to true. While the runtime model is being validated, the issue can occur that it is concurrently interpreted. The programmed default behavior is executed, even though validation of the model at runtime fails and, in consequence, the runtime model is reverted to its previous non-default state.

This issue can be tackled by *restricting operations* to only verified parts of the model at runtime which, in addition, must always conform to its metamodel, including additional constraints.

4 Related Work

We can only hint at an excerpt of *related work* here, namely from the research areas of (i) *models@run.time*, (ii) *programming languages* and (iii) *databases*.

Blair et al. [8] proposed the term *models@run.time* to describe a category of reflective models used at runtime that abstract from the base system and are causally connected to it. One popular application area for models@run.time is self-adaptive software. *Aßmann et al.* [6] describe a reference architecture for models@run.time systems and hint at the *synchronization problem*, i.e., to propagate data from the managed system to the interfaced models@run.time system. *Vogel and Giese* [15] discuss language and framework requirements for adaptation models as used in feedback loops. *Bennaceur et al.* [7] point out that *maintaining the consistency* of models@run.time in a dynamic environment – potentially across different levels of abstractions – is a challenge. *Cazzola et al.* [10] describe a way for realizing the causal connection between Java code and class/sequence diagrams based on a fixed set of *operators* on the models as well as by a *formalized mapping* between code and models.

Smith [14] describes the ability of a running program to access and manipulate its own state as *reflection*. The program state may also be represented on a higher level of abstraction, too. In this regard, models@run.time are strongly related to more traditional research on reflection. *Kiczales et al.* [12] describe the vocabulary for accessing and manipulating the structure and behavior of objects under the term *Meta Object Protocol* (MOP).

Andrews [5] describes general techniques for dealing with *concurrency* from a programmer's perspective, like the use of semaphores for mutual exclusion. *Weikum and Vossen* [16] intensively describe the theory, algorithms and practice of concurrency control and recovery for transactional information systems that rely on databases. *Zhang et al.* [17] comprehensively describe the state of the art and core technologies for in-memory systems which, similar to models@run.time, primarily live in main memory for fast processing and real-time analysis of mission-critical data. The *Connected Data Objects Model Repository* (CDO) [1] is a database-like repository developed to transfer techniques from traditional databases to the modeling domain. The *EMF Transaction* [2] project provides transaction management for the Eclipse Modeling Framework (EMF).

While related work on relevant subproblems exists in research and practice, to the best of our knowledge the transaction problem in the broader sense has not been discussed for the specific context of models@run.time and SAS.

5 Desired Features

Based on the described issues, we structured an initial model of *desired features* related to transactions for models@run.time. This feature model is depicted in Figure 2. It is structured into three groups, namely (i) *causal connection* features, (ii) *transactions-based* features and (iii) *model-based* features.

Causal Connection Features. By definition [8], and in analogy to traditional efforts in reflection, a model@run.time is an appropriate self-representation of a system to which it is *causally connected*, i.e., the model is always updated with the most recent and relevant data (*synchronization*). In the context of GRAF,

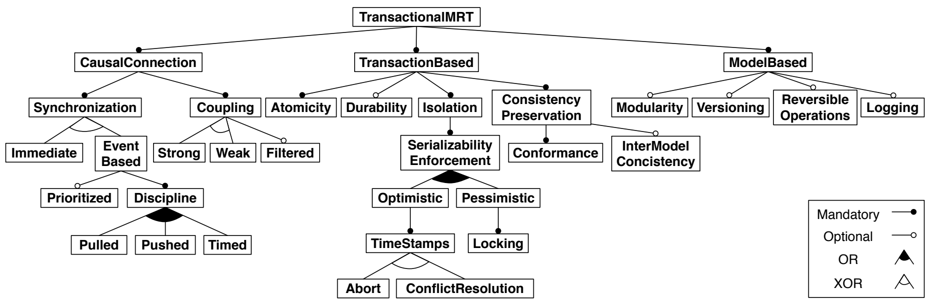

Fig. 2. Features of Transactional Models@Run.Time (Early Draft)

we observe that – in the broader sense – each step between the layers can be seen as a causal connection.

One can distinguish between *immediate* synchronization, i.e., data is propagated immediately (synchronous execution), and *event-based* synchronization, i.e., data is propagated at unforeseen points in time (asynchronous execution). Further desired features are related to the chosen event handling *discipline*. Events may be *pulled* (e.g., the adaptation manager requests updates of the runtime model), *pushed* (e.g. from the adaptable software to the runtime model) and *timed* (e.g., data is propagated according to a defined frequency). Optionally, these events may be *prioritized*, as well.

The chosen *coupling* character of the causal connection has a critical impact on what assumptions a user can make about the state of the model@run.time. While the state of the reified data can be assumed to be accurate with *strong* coupling, *weak* coupling means that data provided via a causal connection is less accurate or lack behind in time. Additionally, data between two causally connected entities can be *filtered*, e.g., aggregated, split or even ignored.

Transaction-Based Features. The *transaction-based* features are concerned with the properties needed to ensure the reliable processing of transactions on a model@run.time. These are namely (i) *atomicity*, (ii) *consistency*, (iii) *isolation* and (iv) *durability* (ACID) [16].

Each transaction shall be *atomic*, e.g., a series of query or transformation operations, on the model@run.time is "all or nothing": if a failure occurs, the state of the model is left unchanged. In contrast to databases, the ability to guarantee atomicity under all circumstances, e.g., power failure and hardware crashes, can be seen as more relaxed.

Consistency preservation is another essential feature, i.e., each operation on the model@run.time shall bring it from one consistent state to another. In other terms: *conformance* between a model@run.time and its metamodel, including constraints, shall be guaranteed. In case that different abstractions or stacks of abstractions are built with models@run.time, it is crucial to ensure *inter-model consistency*, i.e., changes in one part of the model@run.time are consistent with the representation of associated data in another part.

The typical architecture of SAS asks for concurrency because at least one adaptation manager permanently observes and controls the rest of the software. The main goal for handling concurrency in this scenario is to ensure *isolation* of different activities on the model@run.time. The state of the runtime model must be the same when executing activities one after another or concurrently. In analogy to established techniques in concurrency control, *serializability enforcement* can be achieved in different ways. *Optimistic* approaches use *time stamps* in one or the other way, either to *abort* a transaction or to trigger *conflict resolution. Pessimistic* approaches do not allow conflicts to happen by blocking access to used parts of the model@run.time.

In contrast to traditional databases, *durability* is usually not a primary issue for models@run.time that are stored in primary memory and may be even generated during start-up of software. Optionally, the state of a model@run.time – including its history – may be persisted using non-volatile memory, as well.

Model-Based Features. The *model-based* features are concerned with additional capabilities that support transactional models@run.time.

The *modularity* of models can support locking of only parts of a complex model@run.time. Then again, the modularization of models may require that the individual parts are kept consistent with each other. If modularization is also used to combine multiple models@run.time at different levels of abstractions or from different perspectives, then maintaining inter-model consistency is an associated needed property. Modularity of the runtime model can support partial locking to achieve isolation of transactions.

The *versioning* of a model@run.time means to store its past states in an accessible way. This can support predicting future adaptations and recovering from errors that may, for example, occur during reification or due to a wrong adaptation. *Reversible operations* refers to the ability of a model@run.time to undo operations on it and *logging* operations on a model@run.time can support recovery and durability if needed.

6 Conclusions and Future Work

Today, developing case-specific, ad-hoc solutions is the state of the art to tackle issues related to establishing a causal connection between a managed system and its models@run.time. Given the associated effort, this is unacceptable.

Based on an issue analysis from previous research on models@run.time for self-adaptive software, we identified desired features, covering (i) causal connection features, (ii) transaction-based features and (iii) model-based features.

In future work, we plan to propose a full concept for *Transactional Models@Run.Time* (TMRT). This concept shall go beyond low-level transaction issues and provide a pattern-based approach to tackle transaction-related issues in the broader sense when developing software using models at runtime.

Acknowledgements. This work is supported by the Deutsche Forschungsgemeinschaft (DFG) under grants EB 119/11-1 and EN 184/6-1.

References

1. Eclipse CDO, `https://eclipse.org/cdo/` (accessed July 18th, 2015)
2. EMF Transaction, `https://projects.eclipse.org/projects/modeling.emf.transaction/` (accessed September 10th, 2015)
3. OpenJSIP, `https://code.google.com/p/openjsip/` (accessed: July 24th, 2015)
4. Amoui, M., Derakhshanmanesh, M., Ebert, J., Tahvildari, L.: Achieving Dynamic Adaptation via Management and Interpretation of Runtime Models. Journal of Systems and Software 85(12), 2720 – 2737 (2012)
5. Andrews, G.R.: Concurrent Programming: Principles and Practice. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA (1991)
6. Aßmann, U., Götz, S., Jézéquel, J.M., Morin, B., Trapp, M.: A Reference Architecture and Roadmap for Models@run.time Systems. In: Bencomo, N., France, R., Cheng, B.H., Aßmann, U. (eds.) Models@run.time, Lecture Notes in Computer Science, vol. 8378, pp. 1–18. Springer International Publishing (2014)
7. Bennaceur, A., France, R., Tamburrelli, G., Vogel, T., Mosterman, P.J., Cazzola, W., Costa, F.M., Pierantonio, A., Tichy, M., Akit, M., Emmanuelson, P., Gang, H., Georgantas, N., Redlich, D.: Mechanisms for Leveraging Models at Runtime in Self-adaptive Software. In: Bencomo, N., France, R., Cheng, B.H., Aßmann, U. (eds.) Models@run.time, Lecture Notes in Computer Science, vol. 8378, pp. 19–46. Springer International Publishing (2014)
8. Blair, G., Bencomo, N., France, R.B.: Models@run.time. Computer 42(10), 22–27 (2009)
9. Cazzola, W., Savigni, A., Sosio, A., Tisato, F.: Architectural Reflection: Bridging the Gap Between a System and its Architectural Specification. In: In REF98: 6th Reengineering Forum. pp. 8–11. IEEE (1998)
10. Cazzola, W., Rossini, N.A., Bennett, P., Mandalaparty, S.P., France, R.: Fine-Grained Semi-automated Runtime Evolution. In: Bencomo, N., France, R., Cheng, B., Aßmann, U. (eds.) Models@run.time, Lecture Notes in Computer Science, vol. 8378, pp. 237–258. Springer International Publishing (2014)
11. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer 36(1), 41–50 (2003)
12. Kiczales, G., Des Rivieres, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT press (1991)
13. Maes, P.: Concepts and Experiments in Computational Reflection. In: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications. pp. 147–155. OOPSLA '87, ACM, New York, NY, USA (1987)
14. Smith, B.C.: Procedural Reflection in Programming Languages. Ph.D. thesis, Massachusetts Institute of Technology (1982)
15. Vogel, T., Giese, H.: Requirements and Assessment of Languages and Frameworks for Adaptation Models. In: Kienzle, J. (ed.) Models in Software Engineering, Lecture Notes in Computer Science, vol. 7167, pp. 167–182. Springer Berlin Heidelberg (2012)
16. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
17. Zhang, H., Chen, G., Ooi, B.C., Tan, K.L., Zhang, M.: In-Memory Big Data Management and Processing: A Survey. Knowledge and Data Engineering, IEEE Transactions on 27(7), 1920–1948 (July 2015)