On the Use of Software Models during Software Execution

Nelly Bencomo
Computing Department, InfoLab21, Lancaster University, LA1 4WA, UK
nelly@acm.org

Abstract

Increasingly software systems are required to survive variations in their execution environment without or with only little human intervention. Such systems are called "eternal software systems". In contrast to the traditional view of development and execution as separate cycles, these modern software systems should not present such a separation. Research in MDE has been primarily concerned with the use of models during the first cycle or development (i.e. during the design, implementation, and deployment) and has shown excellent results. In this paper the author argues that an eternal software system must have a first-class representation of itself available to enable change. These runtime representations (or runtime models) will depend on the kind of dynamic changes that we want to make available during execution or on the kind of analysis we want the system to support. Hence, different models can be conceived. Self-representation inevitably implies the use of reflection. In this paper the author briefly summarizes research that supports the use of runtime models, and points out different issues and research questions.

1. Introduction

The development and execution of software systems have been considered as separate cycles. During each cycle two entirely different entities are managed. On the one hand, during development, the entity corresponds to the source code that is developed (probably generated from models), debugged and analyzed. On the other hand, during execution the entity corresponds to an enclosed, incomprehensible binary program that just can be executed [19].

However, more and more software is required to survive variations in their execution environment without or with little human intervention. Such systems are called “eternal software-intensive systems” [23]. These modern and more complex systems cannot be shutdown to be changed or updated and restarted again. Such modern software systems need to be conceived and considered differently. In contrast to keeping development and execution as separate cycles, modern complex software systems should not have such a separation [19]. Research in Models-driven Engineering (MDE) has been primarily concerned with the use of models during the first cycle or development (i.e at the design, implementation, and deployment). Such research has proved useful with excellent results in both academia and industry.

The author argues that these eternal modern systems must have a first-class representation of itself (what we call the runtime model) available to enable change. These changes will not only be performed by humans or other systems but also by the system itself. The runtime representations or runtime models will depend on the kind of dynamic changes that we want to make available during execution or on the kind of analysis we want the system to support. Hence, different models can be conceived. The self-representation inevitably implies the use of reflection [*22*]. Furthermore, such self-representation(s) will allow the system to reason about itself and also facilitate the subsequent modification, adaptation, and dynamic evolution of the system. Early work in this emerging MDE area has recently been presented at the MODELS Workshop *Models@run.time* [*4*] that focuses on the topic. As further evidence of the interest on the topic, the IEEE Computer has recently accepted a proposal for a special issue (October 2009).

In this paper, the author discusses early research results that support the use of runtime models. On the basis of these results the author outlines several issues for future research.

2. Background

Although the research topic about the use of models during execution is rather new, it is inspired in seminal research work from the past. Several research projects have already proposed initial ideas of using models during runtime to specifically support adaptability and software evolution [*20*, *13*, *14*, *10*]. The models used in these research projects are architecture-based models. That is not a surprise as the crucial role of software architecture in raising the level of abstraction when

developing software is repeatedly emphasized [*12, 7, 1*]. Some of these research projects are briefly presented below.

Oreizy *et al.* [*20*] were novel in their adoption of an architecture-based approach to runtime software evolution. Their approach emphasizes the role of software connectors in supporting runtime change and system evolution, the explicit representations of software components, their interdependencies, and environmental assumptions. Oreizy *et al*. rationalize about the "openness" needed to allow new application behaviors and adaptation plans to be introduced during runtime. The behavior of a system with an open implementation can be altered through a meta-level interface using reflection [*21*]. That is, the openness described above does not necessarily make the system reflective, but it can be used to offer support for reflective capabilities. As noted in Section 1 and further discussed in [*4*], computational reflection and self-representation of the system are basic principles to support the use of runtime models; hence the work explained in [*20*] is highly relevant for the research topic of this paper.

Another relevant work is the research by Garlan and Schmerl [*14*] on system monitoring, reflection and architectural models. Specifically, Garlan and Schmerl describe their approach to monitor the executing system to translate observed events to events that construct and update an architectural model that reflects the actual running system. The final goal is to compare the dynamically-determined model with the correct architectural model. Garlan and Schmerl argument how inconsistencies found after the comparison can be used to identify implementation errors, or, even possibly, to effect runtime adaptations to correct certain type of faults. Garlan and Schmerl essentially stated the question "*Does the system as implemented have the architecture as designed?"* Certainly, the work by Garlan and Schmerl is highly related to the topic about the use of software models during runtime. The dynamically maintained architectural model is indeed a self-representation of the system from the architectural point of view. The approach proposed in [*14*] is shown in *Figure 1*.

Crucially, Garlan and Schmerl noted that "*different architectural models or views can be chosen depending on the system quality of interest*". They have also emphasized that the details of how models are derived and what to do if something is incorrect (for example to perform an adaptation or a repair) are localized in the external mechanisms (external to the application) and not distributed through the application. Finally, Garlan and Schmerl have stressed the importance of how these models can be used as the basis for reasoning by "*exploiting a large body of existing work on analytical methods*". This is important as it supports the use of runtime models.

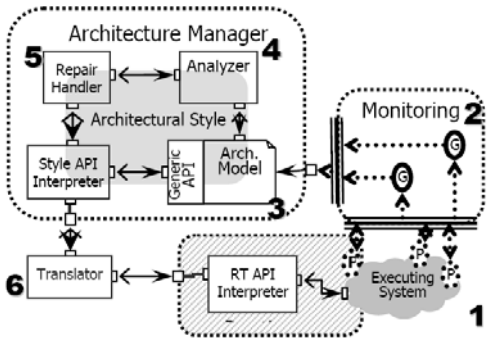

Figure 1. Approach proposed by Garlan and Schmerl (from [14])

Floch *et al.* [*10*] bring the use of architecture models at runtime one step further. Floch *et al.* promote the use of architecture models to support the development of adaptive applications for mobile applications. In contrast to event-action rules, they use goal policies expressed as utility functions leaving to the system the decisions on the actions required to implement the policies. The reconfiguration steps to follow are determined by comparing the actual running system with new architectural variant models based on the utility function.

Another approach is Genie [*2*] that was developed by the author. The implementation is called the Genie tool and is described in detail in [5]. The Genie approach like the case of the projects presented above also uses architectural models, and specifically to support the generation and operation (i.e. during runtime) of component-based adaptive systems. The subjacent reflective component-based technologies will use the artifacts generated by the Genie tool to support adaptation at the architectural level. Two kinds of models can be generated: configuration-based models (i.e. architectural models) and models describing the state transitions that the system can go through. The former models are used to describe the ongoing architecture of the system. The latter models, the transition states models, are used to specify the conditions that represent the dynamic nature of the environment and their impact on the architecture. From the models different artifacts are generated (i.e. configuration files in the form of XML and the even-condition adaptation policies). Such artifacts can be dynamically inserted during execution, as detailed in [5,15], and therefore dynamically changing the behavior of the system. The next section shows further contributions made by the author and her colleagues towards the use of software models during runtime.

3. The Role of Reflection when using Runtime Models

In [18] and partially inspired by the work in [5,15], we demonstrate our approach using runtime models to generate the adaptation logic (i.e. reconfiguration scripts) to reconfigure the system by comparing the current configuration of the running system with a composed model representing the target configuration in a similar fashion as in [10]. However, different from [10], we maintain a runtime model causally connected with the running system (i.e. we use reflection). Figure 2 shows the executing system in the base-level and the runtime model in the meta-level. The causally connected runtime model is used to support reasoning about the system. In that sense, runtime models provide the means to check the impact of architectural changes before applying it to the running system, therefore supporting the analysis for a situation where it would be too complex, expensive, or risky to perform on the real subject. The causal connection can be implemented in different ways. For instance, in [18], it is strongly synchronized from the running system to the runtime model, to ensure the reasoning on an updated version of the model. In contrast, from the runtime model to the running system, the synchronization may present a delay when we want to guarantee the validation of the target configuration before performing changes. As in the cases in the previous section, the models are architectural models. More details are described in [18] and in the research continued by Morin et al. in [17].

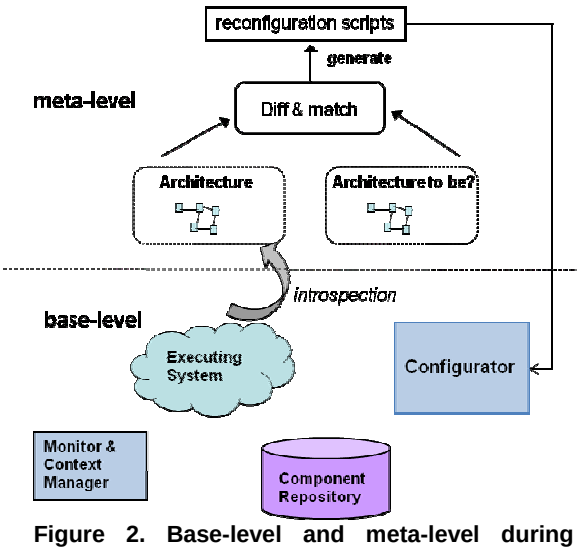

Figure 2. Base-level and meta-level during execution

Our approach can be implemented using the facilities of execution platforms such as Fractal or OpenCOM [5]. The OpenCOM and Fractal platforms provide the system level facilities needed for dynamic evolution [15], such as bringing components into safe state (quiescence management) [16], deleting or replacing component instances, instantiating components, or transferring states.

Also, and different from [10] and [5], where the explosion of the number of configurations is a potential problem, one of our main objectives as explained in [18] is to avoid the enumeration of all possible configurations. A key element is the use of software architecture patterns. These describe generalized configurations of components that are suitable for a set of different environment conditions. Design decisions of patterns are made by domain experts and is further explained in [15].

The application commonalities, i.e. the architecture elements which are part of all configurations, are captured in a "base" model. All the variants are then defined as modules (specified as models) that will be dynamically composed with the base model. The idea is to promote loosely coupled changes [24]. From a particular selection of variants, the corresponding configuration can be built automatically by composing the corresponding variant models (modules) into the base model[1] and according to changes in the environment and context.

We specify constraints on variants that can be used during configuration. For example, the use of a particular functionality (variant model) might require or exclude others. These constraints reduce the total number of configurations by rejecting invalid configurations. More details are shown in [9].

Our research work opens new research questions as for example: what is the correct order of the deletion and incorporation of components during reconfiguration, or what is the impact on the performance of the application. More research in that sense is needed. We already have started tackling the second question. In the specific context of our research, the calculations necessary to obtain the *diff* model to automatically generate the reconfiguration scripts takes more time than the execution of predefined scripts. In some particular cases the consequent delay during runtime may not be acceptable. A possible solution is to pre-generate critical scripts before the system execution. Some early results are already shown in [18] and [17].

1 In [18] we have used the term aspects instead of modules. The author thinks that variant modules helps keeping the reader focused.

4. Discussion, Research Questions

Research on providing support for using runtime models is in its early stages. Therefore, some research questions are open and discussion is needed.

Towards a classification

France and Rumpe [11] describe two broad classes of models, *development models* and *runtime models*. Development models are software models "*at levels of abstraction above the code level*" [11] such as requirements, architectural, and deployment models. Development models are clearly associated with the entities managed during the development cycle described in Section 1 (Introduction). Other authors talk about another category of models called *design-time-models* [10, 3] or *design*-time artifacts [14]. The author is not sure if "development" models mean the same as "*design-time*" models for those authors.

Runtime models provide "*abstractions of runtime phenomena*" [11] and can be used by different stakeholders in different ways. As in the case of traditional models used during design, a runtime model supports reasoning about a system, but can also assist in the automated generation of implementations as was shown in Section 3. System users can also use runtime models to support dynamic state monitoring and control of systems during execution, or to dynamically observe the runtime behavior to understand a behavioral phenomenon [11]. A runtime model can also potentially support semantic integration of heterogeneous software elements at runtime (e.g. dynamically adaptable meta-models)[2] in the domain of systems of systems.

France and Rumpe [11] also picture how adaptation agents (e.g. software maintainers, software-based agents) can use runtime models to determine if an adaptation is needed to consequently perform the changes required. Performing adaptations imply making changes to (runtime) models of the parts to be adapted using the support provided by execution platforms (e.g. a middleware platform with adaptation facilities). Our work presented in [18] makes the envisaged application of runtime models made by France and Rumpe in FOSE (ICSE 2007) slightly closer to reality, however, more research work is needed. In a more visionary approach, the adaptation agents can be envisaged using runtime models to fix design errors or to include new design decisions into a running system [11] to support controlled *ongoing-design*. The mechanisms used to realize this vision are expected to be more complex. Nevertheless, these more complex mechanisms would be able to support unanticipated modifications to some extent [6].

More than architectural models

The research works discussed in this paper present the prevalent use of runtime architectural models. The author believes the vision of *models@run.time* goes further. For instance, requirements reflection [8] is an exciting and promising research topic that studies how the requirements and goals of a software system can be dynamically observed, i.e. during execution [6]. In order to do this, a model of the requirements of the system should be maintained while the system is running. The right associations between such requirements models and the implementation artifacts should also be taken into account to keep requirements information in sync. Future work is needed to examine how technologies may provide the infrastructure to offer support to maintain requirement models that can be consulted at runtime. Furthermore, explicit runtime representations of system goals are crucial for self-aware systems.

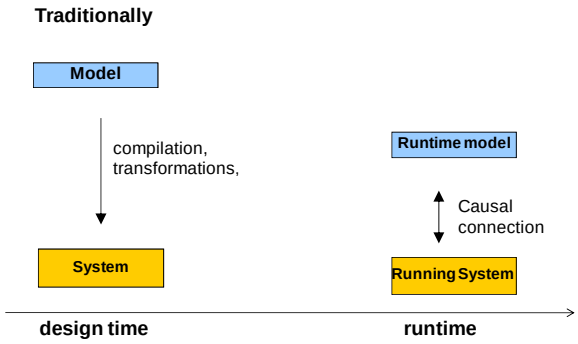

Figure 3. Design models vs. runtime models

Figure 3 shows an initial comparison between traditional software models (used during design-time) and runtime models. MDE research has tended to concentrate on the use of design-time models. Design-time models are transformed or compiled to create a system that eventually will be executed.

The role of reflection and the causal connection between the system and its self-representation (i.e. its runtime model) is specially highlighted in the figure. Take note how on purpose the author has made closer the runtime model and the running system contrasting with the bigger separation between the model used in design and the system to be executed. With that, the author wants to stress that the links between the runtime model and the running system should be light enough to ensure good performance during execution. Any reasoning task and change effects from the

2 As suggested by Bran Selic as a panelist in the 3rd International Workshop on Models@run.time in Toulouse, October, 2008

runtime model to the running system and vice versa should not take long time and put in risk the responsiveness of the system (as discussed in Section 3). In [18] we have shown how runtime models can be used to generate reconfiguration scripts that will dynamically change the running system. We have also shown the potential use of runtime models to support reasoning. Then again, more research efforts are needed to further study the potential use of runtime models.

Final research questions

As stated in [11], the proposed classification may evolve as the research topic of *models@run.time* matures. Runtime models may be used as development models, to dynamically evolve software systems, for instance. Similarly, development models may be used as runtime models to support ongoing design, for example. Therefore, how can the classification proposed be extended or improved, and what would be the relationships between these two categories of models? Specifically, in Figure 3, what are the relationships between the traditional software models and a runtime model? Furthermore, is an *ongoing-design* of the system described by design-time models or by runtime models? or by a combination of both?

In order to reason about the impact of changes, it looks crucial that the history of the system must also be fully accessible and manipulable, therefore, what role should the models play in that task?

Other research questions, partially based on the ideas discussed in this paper and the fruitful discussions during the panel of the third edition of the workshop Models@run.time 2008, are as follows:

-How are the current model synthesis technologies (used during development) different from the more dynamic model synthesis technologies needed when using runtime models during execution? Are the former technologies suitable for dynamic model synthesis?

-What are the methods and standards for specifying semantics suited to automated interpretation (i.e. done during runtime)?

-How can we achieve reversible model transformations, to deal with synchronization issues between the runtime model and the running system, and between the development models and runtime models?

These questions are just few starting points for research in this exciting research topic with potential fruitful results for software engineering.

Acknowledgments: The author gratefully acknowledges Gordon Blair and Robert France for their support on this work. She also acknowledges the support given by her colleagues in the DiVA project, especially Brice Morin and Jean-Marc Jézéquel. This work was partially funded by the DiVA project (EU FP7 STREP).

5. References

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2nd edition, 2003.

[2] Nelly Bencomo. *Supporting the Modelling and Generation of Reflective Middleware Families and Applications using Dynamic Variability*. PhD thesis, 2008.

[3] Nelly Bencomo, Gordon Blair, and Robert France. Models@runt.time workshop in conjunction with MODELS 2006 Conference , October 2006

[4] Nelly Bencomo, Robert France, and Gordon Blair. 2nd international workshop on models@run.time. In Holger Giese, editor, *Workshops and Symposia at MODELS 2007*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[5] Nelly Bencomo, Paul Grace, Carlos Flores, Danny Hughes, and Gordon Blair. Genie: Supporting the model driven development of reflective, component-based adaptive systems. In *ICSE 2008 - Formal Research Demonstrations Track*, 2008.

[6] Betty H.C. Cheng, Holger Giese, Paola Inverardi, Jeff Magee, and Rogerio de Lemos. Software engineering for self-adaptive systems: A research road map, Dagstuhl-seminar on software engineering for self-adaptive systems. 2008.

[7] Paul Clements and Paul Kogut. The software architecture renaissance. *Crosstalk - The Journal of Defense Software Engineering*, 7(11), 1994.

[8] Anthony Finkelstein. Talk “requirements reflection" in schloss Dagstuhl seminar on software engineering for self-adaptive systems. 2008.

[9] Franck Fleurey, Vegard Dehlen, Nelly Bencomo, Brice Morin, and Jean-Marc Jézéquel. Modeling and validating dynamic adaptation. In *Workshops and Symposia at MODELS 2008*, volume 5421M.R.V. Chaudron, 2008.

[10] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *Software IEEE*, 23(2):62–70, 2006.

[11] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In L. Briand and A. Wolf, editors, *Future of Software Engineering*. IEEE-CS Press, 2007.

[12] David Garlan. *Software Architecture: a Roadmap*. ACM Press, 2000.

[13] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[14] David Garlan and Bradley Schmer. Using architectural models at runtime: Research challenges. In *European Workshop on Software Architectures*, St. Andrews, Scotland, 2004.

[15] Paul Grace, Gordon Blair, Carlos Flores, and Nelly Bencomo. Engineering complex adaptations in highly heterogeneous distributed systems. In *2nd International Conference on Autonomic Computing and Communication Systems (Autonomics 2008)*, Turin, Italy, 2008.

[16] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293 —1306, 1990.

[17] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming dynamically adaptive systems using models and aspects. In *International COnference in Software Engineering (ICSE)*, 2009.

[18] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *MODELS'08 Conference*, France, 2008.

[19] Oscar Nierstrasz, Marcus Denker, Tudor Girba, Adrian Lienhard, and David Rothlisberger. *Challenges for Software-Intensive Systems and New Computing Paradigm*, chapter Change-Enabled Software Systems. 2008.

[20] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications*, 14(3):54–62, 1999.

[21] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Comput. Lang. Syst. Struct.*, 34(2-3):46–65, 2008.

[22] B. C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, B. C. Smith. Reflection and Semantics in a Procedural Language. PhD thesis, M.I.T, 1982., 1982.

[23] Martin Wirsing and Matthias Holzl. Report of the beyond the horizon thematic group 6 on software intensive systems. Technical report, 2006.

[24] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.