Model Execution Adaptation?

Eric Cariou
LIUPPA / Université de Pau
B.P. 1155
64013 PAU CEDEX, France
Eric.Cariou@univ-pau.fr

Franck Barbier
LIUPPA / Université de Pau
B.P. 1155
64013 PAU CEDEX, France
Franck.Barbier@univ-pau.fr

Olivier Le Goaer
LIUPPA / Université de Pau
B.P. 1155
64013 PAU CEDEX, France
Olivier.Legoaer@univ-pau.fr

ABSTRACT

One of the main goals of model-driven engineering (MDE) is the manipulation of models as exclusive software artifacts. Model execution is in particular a means to substitute models for code. On another way, MDE is a promising discipline for building adaptable systems thanks to models at runtime. When the model is directly executed, the system becomes the model, then, this is the model that is adapted. In this paper, we investigate the adaptation of the model itself in the context of model execution. We present a first experimentation where we study the constraints on a model to be able to determine if it is consistent (that is, adapted) with an execution environment, possibly including fail-stop modes. Then, we state some perspectives and open issues about model execution adaptation.

Keywords: MDE, models@run.time, model execution, adaptation, state machines

1. INTRODUCTION

Recently, the broader-scope notions of software adaptation and self-adaptive software [13] have gained more and more interest. In [13], the authors list "software engineering" as the primary "supporting discipline" to construct self-adaptive software. Despite a wealth of promising research results [4], authors in [6] claim that the vision of software adaptation, in the wider area of autonomic computing, remains unfulfilled. Mature software development technologies like MDE (Model-Driven Engineering) [9] are natural candidates to create self-adapting applications [7, 8, 16]. This trend is confirmed by the possibility to manipulate models at runtime (*models@run.time*) [1]. Models at runtime are embedded models within the system during its execution. There are potentially several kinds of usage of models@run.time, but the main one is to develop adaptive systems. [1], a reference paper on that topic, even talks only about software adaptation as application of models@run.time.

Specification of a self-adaptive system traditionally requires multiple formalisms (Safran [12]): architecture description languages to describe the structural adaptation of the component system, process algebraic languages to describe the behavior, a set of on-event-do-action rules to describe the adaptation policy, other ones to express constraints on the system like the quiescent states and invariants to check the integrity of the system. With the development of MDE, self-adaptive systems are more convenient to specify because of the unified framework brings out by MDE in general (meta-modeling, model transformation and manipulation). For instance, [11] defines meta-models for expressing the features, context, reasoning and architecture of a system as support for adapting component-based applications. Using MDE techniques, we gain in usability as all required software artifacts are represented under the same form (uniformized models) and also because some design models can directly be used at runtime.

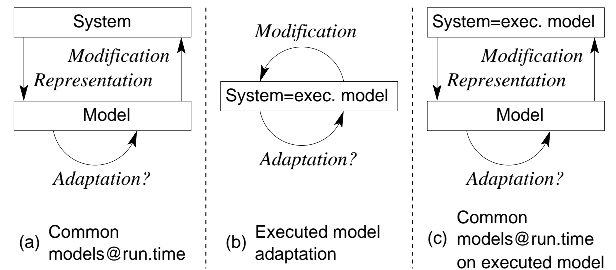

Figure 1: Adaptation loops

Models@run.time aims then at representing the state of the system and to express adaptation policies, conditions and rules on the model instead of on the system directly. The main problem to face is to be able to maintain a consistent and causal connection between the system and the model for the model being a valid representation of the system at runtime [1, 10, 15]. On another way, one of the main goals of MDE is to cope with models as final software artifacts. This can be performed by generating the code from the model or by directly executing the model itself through a dedicated execution engine interpreting the model. In that case, the model is the "code" that is executed. Actually, model execution is a special kind of models@run.time where the model *is* the system that is executed at runtime. Then, there is a trivial causality between the model and the system as the model is the system. In that way, the system adaptation

deals straightforwardly with the direct model adaptation, filling the gap between the system and the model. Figure 1 represents the modification of the adaptation loop in this new context. With "common" models@run.time (part (a)), the model represents the system. The reasoning on the adaptation necessity is made on the model and the required modifications are processed on the system. When adapting an executed model (part (b)), the reasoning is still made on the model but the required modifications are processed directly on the model and without needing a representation of the system.

In this paper, we discuss a first investigation of the adaptation of the model itself in the context of model execution. As far as we know, there are no other works dealing with adaptation of executed models as we do. In section 2, we study a simple example by focusing on the conditions for the model to be adapted to a given execution context. Then, in section 3, based on this example, we state some open issues regarding model execution adaptation.

2. A FIRST EXPERIMENTATION

We have implemented a meta-model for basic state machines and used it, through a train example, for studying the adaptation of a model for an execution environment. Our goal here is not to express how the model has to be adapted, that is modified, during its execution, but to determine if it is suitable for or consistent with a given execution environement.

2.1 A Running Train Example

The example of this paper is freely inspired of a railway system[1]. The behavior of a train is specified through a state machine. The train is stopped or is running at a given speed, depending on the signals along the railway. The environment of execution is then here the signals that control the train speed. Concretely, the different speeds of the train are specified through the states of the state machine whereas the signals are the events associated with the transitions between these states. Within the same state machine, we can specify the behavior of the system (the train) and its interaction with the execution environment (the signals). The verification problem we want to face is in a general way to be able to determine if the behavior of the system is consistent with the execution environment, that is concretely here, if any signal is understandable and correctly processed by the train state machine.

2.1.1 Execution Environments

Three different kinds of environments are considered, for two different countries. Indeed, a train can cross the border of its country and can travel on railways of another country that uses different kinds of signals. The figure 2 shows these three kinds of signals. The country A uses signals with 3 or 4 different color lights aligned within a single column. The country A contains two different kinds of railways: Normal speed sections (up to 130 km/h) and high speed sections (up to 300 km/h). The signal with 3 colors is for normal speed sections while the signal with 4 colors is for high speed ones. The meanings of the colors are the following (only one light is put on at the same time): Red means that the train must stop immediately, amber expresses that the train must not run at more that 40 km/h, green means that the train can run at a normal speed (but not more) and purple that the train can run at a high speed. In the rest of this paper, we will consider the specification of a normal speed train (not able to run at more than 130 km/h) of the country A.

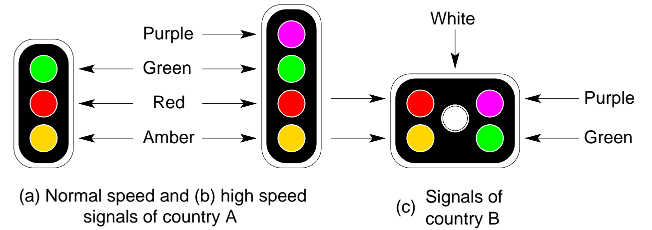

Figure 2: Three different kinds of railway signals

The country B uses for signals a rectangular shape containing 5 color lights: Red, green, amber, purple and white. For our example, we consider that signals of the country B are not known by the train driver but we make the hypothesis that for each country, colors have similar kinds of meaning: Red is for stop, amber for low speed, green for normal speed and purple for high speed. However, the white color has no meaning for the train driver of country A. To be precise, the maximum speeds associated with each color are not exactly the same in country B than in country A. This is why we talk about "similar kinds of meanings" of colors and not about "exactly the same meanings". In country B, the maximum low speed is 30 km/h, the maximum normal speed is 110 km/h and the maximum high speed is 350 km/h.

2.1.2 Basic Running Train Model

Figure 3, part (a), represents the behavior of a non high-speed train of country A. The states define the speeds of the train. For simplifying, the name of a state is the train speed in km/h associated with this state. 0 and 40 are then speeds of 0 km/h and 40 km/h, that is the stop state and the low speed state. When running at a normal speed, the train driver can choose between two speeds, either 100 or 130 km/h. These two states have been put into a composite one representing the normal speeds. Transitions between states are associated with the signal color lights: Red, green and amber. The purple color is not managed here, as the train cannot run at more than 130 km/h, but it can run at 100 or 130 km/h on a high speed section.

There are two particular events: `int_SpeedUp` and `int_SpeedDown`. These events are internal actions of the train, that is, correspond to direct train driver actions. For not confusing them with the external events coming from the execution environment, their name is prefixed by "int_".

The presented state machine seems to be a relevant specification of the train behavior. But the next section shows

[1]The state machines of train behaviors and their associated signals of this paper are not at all intended to be considered as realistic specification of a railway system. On the contrary, we have taken a lot of liberties from real systems and for concision purpose, train behaviors have been made very simple.

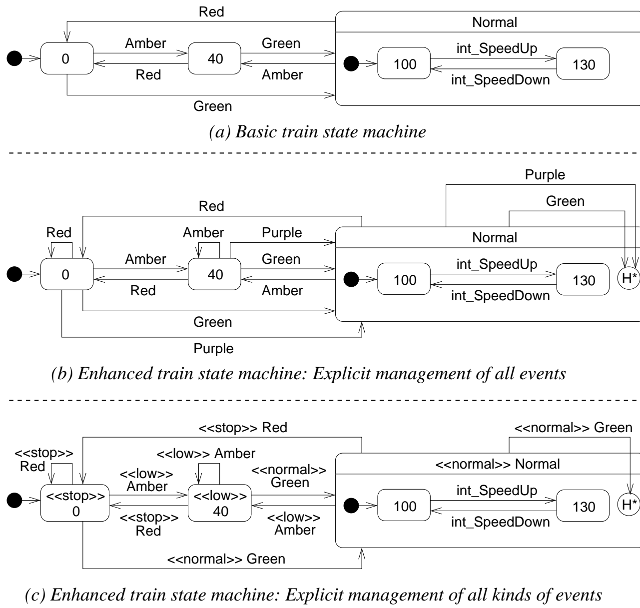

Figure 3: Train state machine variants

that, in order to ensure or check the consistency of the state machine against an execution environment, this is not the case. The main problem is the implicit elements embedded in the model.

2.2 Fitting an Execution Environment

2.2.1 Rigorous Model Definition

The problem of the state machine of the figure 3, part (a), is all the implicit it contains. Indeed, the defined transitions deal only with events that change in effect the state of the model. For instance, if the current active state is the 40 km/h speed (the low speed state), the red event and the green event make the model evolving respectively to the 0 km/h (stop) state and to the normal composite speed state. The associated transitions are then present on the model. If an amber event occurs, as no transition is defined for this event starting from the low speed state, we stay in the low speed state. This is the expected behavior but this is implicitly defined.

According to our problematic, the main problem with an implicit management of events is related to the consistency of the model against a new execution environment. As said, the white event in signals of country B is not known by the train driver. What happens if such a white signal occurs when the train is running on railways of country B? Nothing, it is ignored. Of course, ignoring systematically an unknown signal is not a correct behavior. As a solution, a rigorous definition of the model is required: All expected interactions with the execution environment must be explicitly managed. The goal is to be able to make the difference between an unknown interaction or an expected one even if it does not change currently the state of the model. In term of verification purpose, this has a very interesting consequence: If these interactions compose a finite set (as for the 3, 4 or 5 colors for signals), it is then possible to statically determine if a given model is consistent with or adapted to an execution environment.

Concerning the train model, these interactions are the set of expected signal colors. There are several ways to ensure that a given signal occurrence is an expected one. A first solution is to parameterize the execution engine with this set and to check, before processing a signal (an event occurrence), that it belongs to this set. The problem is that it requires to modify the execution engine to carry out this verification in addition to the model execution. Another solution, avoiding this problem, is that all expected events (except the internal ones) must be associated with a transition starting from each state of the state machine (if a state is embedded in a composite, this transition can start from the composite). The goal is that each event occurrence is systematically and explicitly processed by the state machine. If, for a given state, an event does not make the model evolving, that is the current active state remains the same, the associated transition simply starts from this state and leads to itself[2]. For such self-transitions on composite states, there is a simple solution for keeping the internal current active states. The first thing is to put an history state inside each composite state (including those contained in other composite states). Then the self-transition for a composite state has for target state its history state, making its whole internal active state hierarchy not changing.

Figure 3, part (b), shows the modification of the train state machine following these rules of rigorous modeling. One can notice that for each state, there is a transition associated with each one of the four color events. In term of adaptation, we can know that the new train state machine is consistent with a high speed section of country A as the four colors are processed for each state but not with the railways of country B as the white color is not processed for all states (actually for none, but only one missing transition from one state is enough to conclude that the model is not adapted).

2.2.2 Definition of "kinds" of Elements

Ensuring that the environment interactions are processed by the model is not sufficient. It is necessary in addition to ensure that they are *correctly* processed. In a general way, it is of course not an easy task but with state machines and for the train example, there are some possible and simple verification actions. We can notably check if a given event occurrence processing leads to activate a specific expected state. For instance, if the red event occurs, the state machine must be in the 0 km/h speed (stop) state. This is the case for all the transitions associated with the red event, independently of their source state. Here also, we can statically verify that each transition associated with a given event leads to the correct state.

However, this verification can be too strong. Sometimes, it will be simply required to ensure that a kind of state (here a kind of speed) has been reached. This can be achieved by a composite state embedding several speeds, as for the normal state and its 100 and 130 km/h speed states. But

[2]If business operations are associated with the states, for these particular transitions, a system avoiding the execution of such operations has to be added because these transitions are not business ones but are here for checking purpose.

this solution is not adapted to manage a new environment of execution. If we take the low speed for countries A and B, they are not the same (40 km/h in A and 30 km/h in B). However they are sufficiently close to be considered as substitutable. Defining a composite state containing both 30 and 40 km/h speed states will not work. Indeed, when the amber event occurs, it can lead to only one state (through a direct transition inside the composite or indirectly via its initial or history state): Either the 30 or the 40 km/h. Without structurally modifying the transitions of the state machine, it is not possible to ensure that in A a transition associated with an amber event leads to the internal 40 km/h speed state and in B to the internal 30 km/h speed state. A clever solution is to be able to define that the two states belong to the *kind* of low speed states and to verify that an amber event leads to a kind of low speed state and not anymore to the exact 30 or 40 km/h speed state. Then, defining kinds of states is useful, but defining "kinds" of other elements can also be useful. For instance, we can define a kind of normal speed signal enclosing both green (normal speed) and purple (high speed) events[3].

Figure 3, part (c), is the modification of the train state machine model including kinds of elements. Each state or event is tagged following the << ... >> UML stereotype notation. Now, we can check that each kind of event leads to a kind of state with the same tag (having the same tag between states and events is of course a choice and not mandatory). For instance, each transition leading to the kind of <<low>> speed state (the 40 km/h state) is associated with an event of the kind of <<low>> signal (the amber color). One can notice that transitions associated with the purple event have disappeared. Indeed, they are now embedded within the kind of normal speed events. To be precise, we have a mix of two levels of specification: Either the element (state or event) is present with its exact value (the 40 km/h state or the amber event for instance) or through its kind (kind of normal event for the purple event for instance).

In term of verification, compared to the previous state machine specification (figure 3, part (b)), now we can ensure that in B, the amber event (or any kind of low speed event) is correctly managed by leading to a kind of low speed state, even if is not the exact expected speed of B. Of course, this requires that there exists a common knowledge between the execution environments of A and B. It must be set that a difference of 10 km/h for a low speed is acceptable. In a general way, we can consider that the normal speed train state machine for the country A (the one of figure 3) is a fail-soft version of a reference state machine of the country A high speed trains or of the trains of country B (if we except the white color signal not managed by the state machine).

2.2.3 Multi-Level Classification of Adaptation

Based on the above discussions, we can define a hierarchy of verification. As there are two classifications of knowledge on behavioral elements (the states) and interactional ones (the events), this lead to four combinations defining a hierarchy of consistency of a model against an execution environment. Table 1 details these combinations. The strongest verification is when there is an exact knowledge for behavioral and interactional elements. In this case, the model is considered as a reference model for this execution environment. The weakest is when the verification is based on the kinds for all elements. In this case, the model is a fail-soft behavior for a close execution environment.

2.3 Implementation

For simplified state machines, we have implemented an execution engine in Kermeta[4] and defined adaptation contracts[5]. These adaptation contracts are based on previous works on execution and transformation contracts [2, 3]. Their goal is to ensure that a model is adapted to a given execution environment (through the verification of processing only expected events), following the classification of the section 2.2.3. The verification can be made statically, that is before executing the model. In this case, the contracts are embedded in OCL constraints that are checked on the model. At runtime, the verification is made within the precondition of the Kermeta operation that processes an event.

3. CHARACTERIZATION & DISCUSSION

Based on previous sections, we discuss here the characterization of the adaptation of executable models and associated open issues.

3.1 Executable Adaptable Models

As stated by previous works, such as [2, 5], an executable model has the specificity to contain elements allowing to represent its state during its execution. Concretely, it is a support capable of discrete and traceable evolution. For a state machine, in addition to its static or structural elements (its states and transitions) it is then required to also establish at a given time what are the current active states through dynamic elements defined in the meta-model (these dynamic elements being not defined in the UML 2.0 state machines specification, an extension of the UML meta-model is necessary, as proposed in [2]). The execution engine embeds the operational semantics of execution for a given meta-model (for instance, how to process the transitions for event occurrences). It is also possible to ensure that a given execution semantics is respected, typically with execution contracts [2].

When adapting a model during its execution, this model must basically be an executable model. In addition, it has to respect some others characteristics. As shown before, an adaptable model must be designed for being able to represent simultaneously the behavior of the system and as much as possible the interactions with the execution environment. In other words, it must also include the specification of the execution environment. So, how to represent an execution environment? Concerning our example, it was done in a simple manner through events of the state machine, even if we

[3]The easiest way to generalize the notion of "kinds" for any element is to automatically add on the considered meta-model a single abstract element containing an `elementKind` string attribute, and to make all elements of the meta-model specializing it directly or indirectly. Each element can then be tagged for expressing it belongs to a given kind of element.

[4]`http://www.kermeta.org`

[5]Due to lack of place, we can not put some code excerpt or OCL constraints, but the engine and the contracts are available online:
`http://web.univ-pau.fr/%7Ecariou/adapt-contracts/`

	Exact behavior	Weak behavior
Exact interaction	The exact behavior is valid for this exact environment. This is a reference model for this execution environment.	A fail-soft behavior is valid for a precise execution environment.
Weak interaction	The model behavior is directly valid for a close execution environment compared to the reference execution environment.	A fail-soft behavior is valid for a close environment compared to the reference execution environment.

Table 1: Four adaptation combinations

had to distinguish external events from internal ones. Possibly, the specification of the interaction environment can be specified in another model or be the parameterization of the execution engine but it must be made explicit for being able to differentiate an expected interaction from an unexpected one. As explained in the next section, executable adaptable models can also embed dedicated properties and values for adaptation management, mostly as in common models@run.time.

In common models@run.time, models are designed to exactly and only represent what is required to carry out the system adaptation. The contents of these models are then different from the adaptable executable models that are executable models augmented with a content dedicated to manage the adaptation.

3.2 What means Adapting a Model?

In this paper example, the consistency of a model against an execution environment is discussed. We are able to determine if we face up an unexpected interaction, but of course, an adaptation decision has to be taken in this case. Concretely, what must be done if the unknown white signal is crossed by the train of the country A? Here are three examples of what can be an adaptation decision. First solution, the execution of the model is voluntary stopped: The execution engine does not know how to manage this interaction, so, it stops. The second solution is to determine, thanks to properties associated with this interaction, in which environment we are, and then, to load a reference model for this environment and to execute it. The last possibility is still depending upon properties associated with this interaction and consists in modifying, if possible, the current model to take into account the new discovered interaction.

For the train example and the proposed state machine execution engine, the first solution is applied. The main reason is that we do not yet know what are the kinds of properties that can be attached to the interactions. Intuitively, such a property associated with a color of signal could be the speed (or a range of speeds) the train has to deal with when crossing this signal. Then, concerning the modification of the model, it will be possible to know if there is already a state compatible with the expected speed or if a new state for this speed has to be added to the model. Finally, transitions associated with this new color event can be added outgoing from each state and incoming to this state.

In a more general way, the problem is to define and to integrate within the executable model more generic and detailed properties for both specifying the system and the interaction environment for adaptation purposes. This can of course be based on previous works on adaptation, such as [7] which defines a generic meta-model for specifying properties and associated rules depending on their values. The properties can also deal with QoS parameters and values.

3.3 Levels of Adaptation Policies

Based on the intuitive idea that crossing an unknown signal requires for the train to stop, one can ask why the execution of the model is stopped when an unexpected event occurs instead of stopping the train by activating its "stop" state? The reason is that these two adaptation choices are not at the same level. The engine is dedicated to the execution of state machines based on a given meta-model and its operational semantics. It is not dependent on the business contents of the models, it can execute indifferently any model. However, activating the stop state of a model is dependent of the model contents as it requires to know that there exists such a state. In addition, it also requires to parameterize the engine to force a transition to this state in the case of an unexpected event. This adaptation policy is defined at the model level as it depends on the business contents of the model.

There are also adaptation policies defined at the meta-model level, *e.g.* the engine's stop action or the modification of the model based on properties of interactions. Such adaptations are generic and can be automatically processed on any model. They applied at the domain level, in the sense that a meta-model is dedicated to represent a domain as in the DSL (Domain Specific Language) acronym.

There exists a third level of adaptation policies: Those who apply on the execution engine itself. For instance, when the train of A runs on the railways of B, the verification of expected signals or speeds must be made in a fail-stop mode instead of an exact one. Then, when the train cross the border, the verification policy processes by the engine has to be changed. In the same way that the semantics of adaptation can be changed, we can imagine that the operational semantics processed by the execution engine can also be modified during the execution. Such adaptation or execution policies have to be chosen among a set of already defined ones. Then, this third level of adaptation policies deals with a semantics level.

To sum up, adaptation policies can be expressed at three levels: Business level (depending on the contents of the model), domain level (defined at the meta-model level) and semantics level (applying on the adaptation and operational semantics processed by the execution engine).

3.4 Models@run.time on Executable Models

As seen, models in the context of models@run.time are different in content and goal compared to executable adaptable models. However, the interests of the former are to be explicitly designed for adaptation management. Such models are dedicated to only represent what is required, abstracting away concerns and details not relevant for the adaptation. The drawback is that a causal link between the model and the system has to be established and maintained. This link is not anymore present when directly adapting an executable model but this kind of model could sometimes be too complex or not well structured for easily expressing the reasoning on adaptation. Indeed, it embeds several concerns within a same model: The business content, elements allowing to execute the model and other ones such as those dedicated to the adaptation processing. To manage this complexity, why not representing on a distinct model only the required information for the adaptation? This will lead to apply common models@run.time techniques on a system that will be an executable adaptable model (figure 1, part(c)).

3.5 Self-* properties

In this paper, we discuss adaptation in relation with changing execution environments. There exist other kinds of adaptation, such as the self-* properties (self-healing, self-optimizing, ...). It will be of course interesting to investigate the adaptation of an executable model for ensuring these properties.

4. CONCLUSION

MDE is a promising discipline for building adaptable systems. This comes from the ability of having models at runtime representing the system state (intelligible context) during its execution. When a model is executable, the system becomes the model that is executed, and then, system adaptation becomes model adaptation, fully filling in that way the gap between the system and the model. In this paper, as a first experimentation on the adaptation of model execution, we investigated how to determine that a model is consistent (that is, adapted) with an execution environment, possibly including fail-soft modes. Then, we made a short characterization of these adaptable executable models and stated some open issues on how to realize adaptation of such models.

The perspectives are the definition of techniques for adapting a model (adaptation policies at the business, domain and semantics levels) including self-* properties assurance. Concretely, as in common models@run.time, it is still required to execute a control loop, such as the collect/analyze/decide/act loop of [4], even if this loop could have a different content in this particular context of model execution. In a more general way, it will be interesting to compare model execution adaptation characteristics with other adaptation approaches using for instance the evaluation framework of [14]. This covers to study the limits of directly adapting a model execution versus applying common models@run.time techniques on it. We also need to develop more realistic and complex case studies and to study the adaptation of other kinds of executable models in addition to state machines as we do currently. But, as a conclusion, we believe that the adaptation of model execution will be an useful complementary approach to common models@run.time.

5. REFERENCES

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

[2] E. Cariou, C. Ballagny, A. Feugas, and F. Barbier. Contracts for Model Execution Verification. In *ECMFA '11*, volume 6698 of *LNCS*. Springer, 2011.

[3] E. Cariou, N. Belloir, F. Barbier, and N. Djemam. OCL Contracts for the Verification of Model Transformations. In *OCL Workshop at MoDELS 2009*, volume 24. EC-EASST, 2009.

[4] B. H. C. Cheng, R. De Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, and et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Software Engineering for SelfAdaptive Systems*, 5525(08031):1–26, 2009.

[5] B. Combemale, X. Crégut, P.-L. Garoche, and T. Xavier. Essay on Semantics Definition in MDE – An Instrumented Approach for Model Verification. *Journal of Software*, 4(9), 2009.

[6] S. Dobson, R. Sterritt, P. Nixon, and M. Hinchey. Fulfilling the Vision of Autonomic Computing. *IEEE Computer*, 43(1):35–41, 2010.

[7] F. Fleurey and A. Solberg. A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *MODELS '09*, volume 5795 of *LNCS*. Springer, 2009.

[8] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjørven. Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23(2):62–70, 2006.

[9] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *FOSE '07*. IEEE Computer Society, 2007.

[10] G. Lehmann, M. Blumendorf, F. Trollmann, and S. Albayrak. Meta-Modeling Runtime Models. In *Models@run.time Workshop at MoDELS 2010*, volume 6627 of *LNCS*. Springer, 2010.

[11] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg. Models@Run.time to Support Dynamic Adaptation. *IEEE Computer*, 42(10):44–51, 2009.

[12] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An Architecture-Based Approach to Self-adaptive Software. *IEEE Intelligent Systems*, 1999.

[13] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, 2009.

[14] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas. A framework for evaluating quality-driven self-adaptive software systems. In *SEAMS '11*. ACM, 2011.

[15] T. Vogel and H. Glese. Language and Framework Requirements for Adaptation Models. In *Models@run.time Workshop at MODELS 2011*, 2011.

[16] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06*. ACM, 2006.