A Reference Architecture and Roadmap for Models@run.time Systems

Uwe Aßmann[1], Sebastian Götz[1], Jean-Marc Jézéquel[2], Brice Morin[3], and Mario Trapp[4]

[1] Technische Universität Dresden, Germany
uwe.assmann@tu-dresden.de, sebastian.goetz@acm.org
[2] IRISA, University of Rennes 1, France
jezequel@irisa.fr
[3] SINTEF, Norway
brice.morin@sintef.no
[4] Fraunhofer IESE, Germany
mario.trapp@iese.fraunhofer.de

Abstract. The key property of models@run.time systems is their use and provision of manageable reflection, which is characterized to be tractable and predictable and by this overcomes the limitation of reflective systems working on code, which face the problem of undecidability due to Turing-completeness. To achieve tractability, they abstract from certain aspects of their code, maintaining *runtime models* of themselves, which form the basis for reflection. In these systems, models form abstractions that neglect unnecessary details from the code, details which are not pertinent to the current purpose of reflection. Thus, models@run.time systems are a new class of reflective systems, which are characterized by their tractability, due to abstraction, and their ability to predict certain aspects of their own behavior for the future. This chapter outlines a reference architecture for models@run.time systems with the appropriate abstraction and reflection components and gives a roadmap comprised of short- and long-term research challenges for the area. Additionally, an overview of enabling and enabled technologies is provided. The chapter is concluded with a discussion of several application fields and use cases.

1 Introduction

The term "adaptive software system" is somehow a pleonasm, because software has been first invented to make hardware more flexible and adaptable to varying situations in its environment. Software has then evolved according to different paradigms. Object orientation, combined with design patterns, already provides organized means to customize or even adapt software systems (*e.g.*, Strategy pattern [GHJV95]). Current programming languages (like Java), component-based platforms (like Fractal [BCL$^+$03] or OpenCOM [CBG$^+$08]) or SOA platforms (e.g., OSGi [OSG12]) offer reflection APIs, which enable even more powerful dynamic adaptation (*e.g.*, based on dynamic class loading).

N. Bencomo et al. (Eds.): Models@run.time, LNCS 8378, pp. 1–18, 2014.
© Springer International Publishing Switzerland 2014

Traditionally the development of software systems, adaptive or not, used to be split in distinct steps with a clear distinction between design activities and runtime execution [BBF09][1]. The more critical the system is, the more choices will be made at design-time in order to reduce the reconfiguration options to a set of predictable configurations. For example, safety-critical embedded systems are designed and intensively validated at design-time (e.g., model checking) before they are actually deployed [AG93, ELLSV02]. At runtime, they have a predictable behavior, time and resource consumption, which enables *certification bodies* to approve these systems. There is, however, a growing need for more flexible adaptive systems, able to cope with unanticipated situations, still without jeopardizing safety properties. This is typically the case of Cyber-Physical Systems (CPS) as described in Section 6. Hence, new approaches are needed to enable unanticipated adaptations while ensuring guarantees. This is, in our opinion, the ultimate purpose of models@run.time.

The central advance of models@run.time systems is their use and provision of manageable reflection. In general, a reflective software system is causally connected with its code, i.e., when the code changes, the system changes too. Such a system can inspect its code (introspection), can generate new code (code generation), or even change its code (intercession). Because in most cases, Turing-complete programs are reflected about, the problems to be solved by a reflective system are undecidable and unpredictable, even at a checkpoint at runtime. Models@run.time systems improve on this problematic situation. They abstract from certain aspects of their code, maintaining *runtime models* of themselves. In these systems, models form abstractions that neglect unnecessary details from the code and from the environment, i.e., details which are not pertinent to the current purpose of reflection. In these steps, care has to be taken. In general, several models are formed and maintained at runtime, in order to cope with the information loss of abstraction. Also, it has to be ensured that abstractions work correctly, i.e., are faithful with regard to the real behavior of the software system. However, if these precautions are ensured, a models@run.time system is able to perform tractable reflection, due to the faithful abstractions, and it may predict certain aspects of its own behavior for the future. Therefore, models@run.time systems provide and use manageable reflection, which is characterized to be tractable and predictable and by this overcomes the limitation of classic reflection on code, which faces the problem of undecidability.

Taking this definition into account, models@run.time software systems turn out to be a new class of reflective systems, which are characterized by their tractability and predictability. All application domains utilizing reflective systems benefit from this advancement. In addition, two currently hot application domains, especially benefit from the advancement of models@run.time systems:

- **Cyber-Physical Systems.** Models@run.time systems reflecting upon virtual as well as physical processes in comparison to models@run.time systems, which are meant to reflect on a pure virtual system (i.e., information system).

[1] See in particular the side note by Finkelstein.

– **Safety-Critical Systems.** Systems, which demand for verification and certification (e.g., due to their ability to endanger the safety of human beings) in comparison to systems, which do not have such requirements.

Both application domains face the limitation of current reflective systems: undecidability. Models@run.time enable both domains to abstract all concerns of interest to reflection to the information required for the respective decisions. By this, reflection becomes manageable due to abstraction. Furthermore, both domains demand for predictive reflection, i.e., the ability to reflect upon possible future states of the system in comparison to reflect only upon the current system state (and structure).

To summarize, the key advantage of models@run.time systems over reflective software systems, achieved by modeling and separation of concerns principles, is decidability and tractability. By approaching the capabilities of intelligent thinking, we believe models@run.time is the next step in the evolution of software. Models@run.time allow to "mentally" build several potential models of reality and to mentally evaluate these models by means of what-if scenarios [BBF09][2]: *what* **would** *happen if I* **would** *do this action*? During this mental reasoning, the manipulation of the model does not impact the reality, until an acceptable solution has been found. Then, this solution is actually realized, which has an impact on the real world. In other words, the mental model is re-synchronized with the reality. In the case where a relevant aspect of the reality changes during the reasoning process, the model is updated and the reasoning process should re-build mental models, ideally by updating already existing models. This characterizes predictive reflection based on abstraction, i.e., manageable reflection. Models@run.time enable systems to reason about alternatives to reach their goals and consequences of the particular actions in comparison to classical systems which basically learn and react (*i.e.*, animal-like behavior). This includes that the system is able to justify why it takes a certain decision or not. Models@run.time have, thus, the potential to provide both flexibility and assurance, instead of a mere trade off. It can reconcile users, domain experts, engineers (aware of the obvious need for runtime adaptivity), with certification bodies (which need stringent guarantees).

In the following section we discuss, which technologies form the prerequisite for systems following the models@run.time paradigm and how models@run.time enable the development of modern software systems. Next, in section 3, we present a reference architecture for models@run.time to expose the key advancements of models@run.time over reflective systems. We summarize related work as instantiation of this reference architecture and discuss associated communities in section 4. In section 5, we provide a roadmap for models@run.time by a discussion of central, open research questions on the uses and purposes of models@run.time systems. Finally, we conclude the chapter and present compelling applications in section 6.

[2] See in particular the side note by Selic.

2 Enabled and Enabling Technologies for Models@run.time

In the following, we will first discuss the technologies forming the basis for models@run.time systems and then discuss the technologies enabled by models@run.time in turn.

2.1 Technologies Required by Models@run.time

Systems, according to the models@run.time paradigm, are based on the reflection principles, as defined by Bobrow *et al.*:

> *The ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession. Introspection is the ability of a program to observe and therefore reason about its own state. Intercession is the ability of a program to modify its own execution state or alter its own interpretation or meaning.* [BGW93]

In practice however, reflection is a powerful yet hazardous process (see for example the drawbacks of the Java reflection API, clearly reported by Oracle[3]), since it provides no support to "preview" what will be the result of an adaptation. Basically, erroneous adaptation based on reflection can only be detected *a posteriori*, or even *post mortem* if the rollback mechanisms were not able to put the system back to a safe state. The fundamental idea behind models@run.time is to complement classic reflection with strong modeling foundations as defined by Rothenberg:

> *Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner,* **avoiding the complexity, danger and irreversibility of reality**. [RWLN89]

The key characteristic of a models@run.time system is then its ability to project some aspects of the reality (its context, its behavior, its goals, etc.) to the modeling space in order to enable tractable decisions, in a safe space, to produce decidable plans. This is basically separation of concerns [Dij82] applied in a disciplined way at runtime, and to some extent, how human thinking works.

> *What to my taste is* **characteristic for all intelligent thinking**. *It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that*

[3] `http://docs.oracle.com/javase/tutorial/reflect/`

> *one is occupying oneself only with one of the aspects. It is "the separation of concerns". It does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.* [Dij82]

We perceive reflection, modeling and separation of concerns as the three main pillars to achieve models@run.time and make future software systems able of *intelligent thinking*, i.e., abstract, predictive reflection.

2.2 Technologies Enabled by Models@run.time

Models@run.time as a technology enables various further technologies. Of particular interest is the possibility to realize *safe adaptive systems*. The key problem of such systems is the contradiction between safety and adaptivity. To ensure safety, all variants of the system have to be checked against possible threats, usually at design time. In highly adaptive systems, the number of system variants usually grows exponentially and, thus, prolongs the safety check to an unfeasible degree. The reasoner of models@run.time systems allows for postponing safety checks to the runtime of the system as has been shown in [ST11]. In consequence, only those variants of the system have to be checked, which are reachable from the current variant. This significantly lowers the amount of variants to be considered and, thus, enables the realization of safe adaptive systems.

Furthermore, models@run.time enable the realization of *Cyber-Physical Systems (CPS)*, which are adaptive systems integrating the virtual and physical world. A central requirement for CPS is safety, due to the physical part of the system. This is because the physical actions of a CPS are able to threat human life, the environment or the system itself. In addition, CPS are adaptive systems, because they naturally adjust themselves continuously to the their environment. In consequence, models@run.time is the key enabling technology for CPS, because models@run.time enable the realization of safe adaptive systems.

Besides these two particular application domains, all domains, which already make use of reflection, benefit from the advancements by models@run.time.

3 A Reference Architecture for Models@run.time Systems

The goal of this section is to understand how models@run.time are key enablers for modern software systems, to clarify their typical use cases and fundamental interests, as well as to define a reference architecture for models@run.time. Based on our recent experiences (*e.g.*, in the DiVA project [FS09, MBNJ09] and the MQuAT approach [GWCA11, GWCA12]), we propose the generic reference architecture (RA) depicted in Figure 1, which provides a generic framework for models@run.time, and which is meant to be instantiated for different domains.

According to the reference architecture, a models@run.time system always interfaces with a managed system, which is monitored and controlled by the

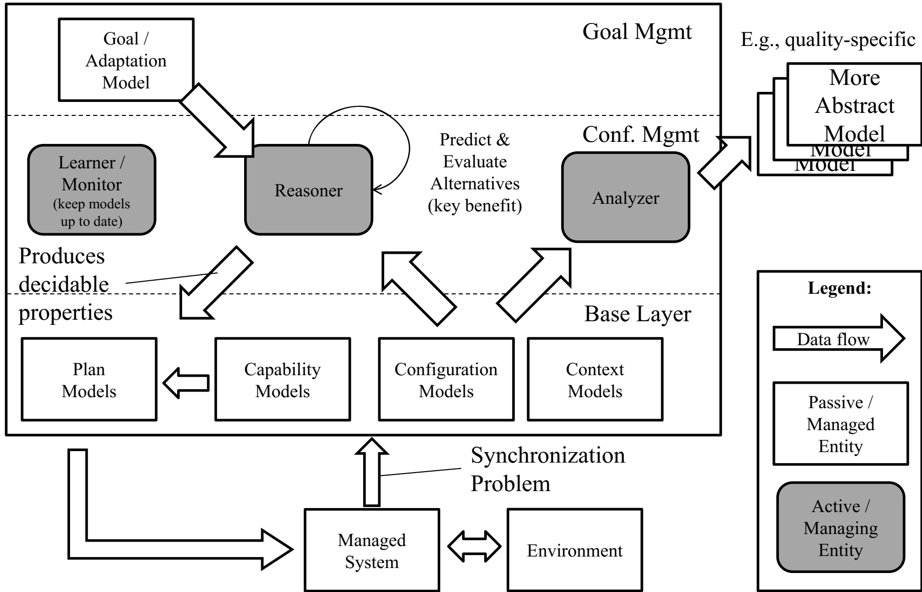

Fig. 1. Reference Architecture for models@run.time Systems

managing models@run.time system. Notably, a models@run.time system is not directly interfaced to the environment. Instead, the managed system's sensors and actuators are utilized for this purpose. The managed system can be any observable and controllable system (e.g., a personal computer, a wireless sensor network, a robot or a managing models@run.time system again). Each models@run.time system comprises three layers, comparable to the layers of Kramer and Magee [KM09]. From bottom to top these are:

- a base layer comprising models of the managed system,
- a configuration management layer comprising active components of the system realizing the feedback loop on the managed system and
- a goal management layer comprising models of the system's goals, realizing an internal feedback loop between the goal management layer as managing element and the configuration management layer as managed element.

3.1 Runtime Models of the Base Layer

The *base layer* comprises four types of models, which are abstractions of specific aspects of the system for a given purpose:

Context Models contain relevant information about the current state of the managed system's environment, *e.g.*, the current temperature, or higher level context information such as an alert information derived by aggregating or interpreting the information of different sensors. The interpretation of context information is context-dependent itself. For example, a temperature of 50 °C

will be interpreted as "hot" as a room temperature, but as "cold" for a furnace. To keep this type of model synchronous to the environment, the sensors of the managed system are utilized. Context models do not cover information about the managed system, but only of the environment's observable state.

Configuration Models express the current configuration of the managed system, i.e., its current state. Current models@run.time approaches usually provide an architectural view on the managed system (i.e., which services are currently deployed and running on which resources). Both, configuration and context models, cover the abstracted runtime state subject to tractable, predictive reflection.

Capability Models describe the features available to influence the managed system (*e.g.*, whether software components can be added/removed and rebound, whether parameters of system components can be adjusted), which actuators are available and how they can affect the environment. Typically this model is rather static and depends on the underlying infrastructure. However, this model can be updated, *e.g.*, after a new actuator has been added in the system.

Plan Models describe a set of actions (according to the capability models) to be performed by the system to realize an adaptation. They represent reconfiguration or action scripts, which describe how the managed system shall be reconfigured and how the actuators of the managed systems shall be used to effect the environment.

3.2 The Configuration Management Layer

The *configuration management layer* contains the active entities of a models@run.time system, which make use of the models of the base layer. This layer typically comprises a reasoner, an analyzer and optionally a learner.

The **reasoner's** evaluates alternative future configurations of the system. This includes (1) to realize the predictive reflection, (2) to identify the best configuration w.r.t. the goals specified on the top layer, and (3) to derive reconfiguration or action plans to establish the envisioned system configuration. To evaluate possible future configurations, the reasoner uses the information provided by the context and configuration models, representing the managed system and its environment's state, and derives possible variations of them, which are reachable in the future, based on the capabilities of the managed system covered by the capability models. To identify the best future configuration w.r.t. the system's goals, the reasoner evaluates each possible future system variant against the goal models of the top layer. To derive reconfiguration plans, the system compares the current system configuration with the envisioned system configuration and deduces a sequence of actions to be taken based on the capability model of the base layer. By these means, the reasoner creates the plan models of the base layer.

The **analyzer** has two tasks. First, the analyzer has to detect whether the whole system (i.e., managed and managing system) should be re-evaluated. To do so, the current system state has to be evaluated against the system's goals. If the current system state deviates from the goals, the analyzer will trigger the rea-

soner, to compute a reconfiguration plan. Second, the analyzer further abstracts the information contained in the models of the base layer. This raises the level of abstraction of the models and, in turn, to lower the complexity of predictive reflection. The analyzer has to abstract the context, configuration and capability models of the base layer to ensure the existence of a capability model on the same level of abstraction for the abstracted context and configuration models. Based on this, models@run.time systems can manage models@run.time system too. The analyzer realizes the bridge between the models@run.time system on lower and higher abstraction level.

The **learner** has two tasks, too. On the one side, the learner is responsible to keep the models of the base layer synchronized with the system. Thus, the learner utilizes the managed systems sensors to capture the environment's state and continuously observes the managed system itself to update the context and configuration model on the base layer. On the other side, the learner can observe the reasoner to detect, whether the decisions of the reasoner are beneficial on the long run or not. Thus, whereas the reasoner evaluates possible future scenarios based on the current system's state, the learner takes into account the system's history to deduce, whether the comparably shortsighted decisions of the reasoner are meaningful and correct on the long run. Based on this, the learner can provide the reasoner with additional (historical) information, to improve the quality of decision making over time.

3.3 The Goal Management Layer

Finally, the *goal management layer* comprises **goal models** of the system, which are used by the reasoner to evaluate the alternative future configurations with respect to the fulfillment of the specified goals. Notably, these goal models can and should be able to change over time, because changes in the context of the system could require adjustments to the goals. This depicts the need for the last feature of the reference architecture: as models@run.time systems are systems themselves, they can be stacked. That is a models@run.time system could monitor and control another models@run.time system. As each models@run.time system realizes a feedback loop, the proposed reference architecture allows for the development of layered feedback loops as has been shown in the Collaborative Research Centre 614 [ADG+09], which focused on self-optimizing systems in mechanical engineering. The proposed architecture is represented as an operator-controller-module (OCM), which realizes three layers of feedback loops. The bottommost layer contains the controller, which directly controls the physical system. On top of the controller, the reflective operator is situated, which is capable of operation scheduling. That is—in contrast to the controller layer—the reflective operator is able to plan the future behavior of the physical system. Finally, the topmost layer comprises the cognitive operator, which is capable of more complex planning methods and utilizes techniques from machine learning. Thus, from bottom to top, the models of the system, utilized by the layers, get more and more abstract, but the applicable techniques get more and more powerful.

In summary, the key advancements over state of the art of models@run.time systems are realized by the reasoner and the analyzer respectively:

1. **Predictive Reflection.** The ability of reasoning about *future* configurations of the system is the advance of models@run.time systems over reflective systems, which are able to reason on the current, but not on future configurations of a system.
2. **Tractability by Abstraction.** The ability of the analyzer to abstract the information used by the reasoner (possibly multiple times) allows for reduction of the reasoning task's complexity and, thus, to get decidability and, finally, tractability of the overall system.

4 Literature Review

4.1 Instantiations of the Reference Architecture

The DiVA project proposes a reference architecture which leverages models@run.time to support dynamic variability [MBNJ09, MBJ+09]. A feature model describes the variability of the system. A reasoner component takes this variability model as input, as well as a model of the context, to compute a set of features well suited to the current context (not necessarily the best). A weaver component then composes these features to produce an architectural model, describing the configuration. This configuration is checked at runtime (since it is not possible to check all possible configuration at design time) and the system is automatically adapted to reflect this architectural model. If the model is invalid, the reasoner computes another configuration.

In [FMS11], the DiVA reference architecture has been instantiated in a different way to fit the need of low-power embedded systems (8-bits, 16MHz, 1Kb RAM). In this setup, the adaptation logic is fully simulated at design-time, so the number of configurations to be addressed by such a small node remains tractable. The adaptation process is compiled into a state machine, which is then merged with the core logic (also expressed as a state machine). The resulting state machine is finally compiled into C code to be deployed on the micro-controller.

The multi-quality auto-tuning (MQuAT) approach [GWCA11, GWCA12] developed in the collaborative research center 912 and preceding projects particularly focuses on self-optimizing systems following economic principles by multi-objective optimization (i.e., the system is optimized w.r.t. the optimal tradeoff between multiple objectives, which represent either cost or utilities). The main constituents of MQuAT are: (1) the cool component model (CCM) and the quality contract language (QCL), which are meta-model defined concepts to be used to specify self-optimizing systems, and (2) the runtime environment THEATRE (THE Auto-Tuning Runtime Environment), which comprises resource managers to monitor and control the target system and control loop managers, which realize the reasoner component by means of an adaptive multi-objective optimizer (i.e., various implementations of the optimizer exist, whereof continuously the best is chosen, based on the current context). A key characteristic of MQuAT

is the application of quality contracts, which cover dependencies between non-functional properties of system components (both software and hardware), to reduce the amount of system configurations, which need to be considered during optimization.

In [SCG+12] the MQuAT reference architecture has been instantiated for multi-tenant applications—applications hosted in the cloud, which are configured by tenants, whose customers use the application. This type of application introduces a further restrictions on possible system configurations, due to tenant constraints like the exclusive use of a single server or the restriction to only use servers within a certain country.

A further reference architecture is ConFract [CCOR06], which particularly focuses on self-healing systems. In this approach functional contracts are used to specify how a valid system is characterized and to initiate self-healing in case of contract violations. The developer is able to explicitly specify resource usage profilers as part of the system. In consequence, the functional contracts, which use data generated by these profilers, can be used as non-functional contracts, so dependencies between non-functional properties can be expressed too.

In [CGK+11], Calinescu et al. present QoSMOS–a generic architecture for adaptive service-based systems (SBS). The central constituents of QoSMOS are formal specifications of QoS requirements (using probabilistic temporal logics) including the specification of dependencies between QoS requirements, model-based QoS evaluation using verification techniques, learning monitoring of QoS properties and reasoning techniques, based on high-level, user-specified goals and multi-objective utility functions. QoSMOS is an instance of our proposed reference architecture, which focuses specifically on SBS comprised of a set of web services under the control of a workflow engine.

4.2 Relevant Communities

Models@run.time are relevant to several research communities as depicted in Fig. 2. Among them two types of communities can be distinguished. First, communities which provide fundamental techniques for models@run.time. Second, communities which benefit from the advancements by models@run.time and provide use cases in turn.

Three communities are relevant to models@run.time in particular: the **self-adaptive systems** (SAS), the **autonomous computing** and **middleware** community. The first two communities investigate systems, which adjust themselves according to changes in their environment. The middleware community covers, among others, the problem how to coordinate multiple independent systems. Whereas the SAS community focuses on a top-down approach w.r.t. the coordination of multiple systems, the autonomous computing community focuses on bottom-up approaches (self-organization), where the coordination originates from each individual system [ST09]. For this purpose, all three communities rely on reflection to observe and adjust the systems they manage or coordinate. Hence, the advanced reflection mechanisms of models@run.time enable

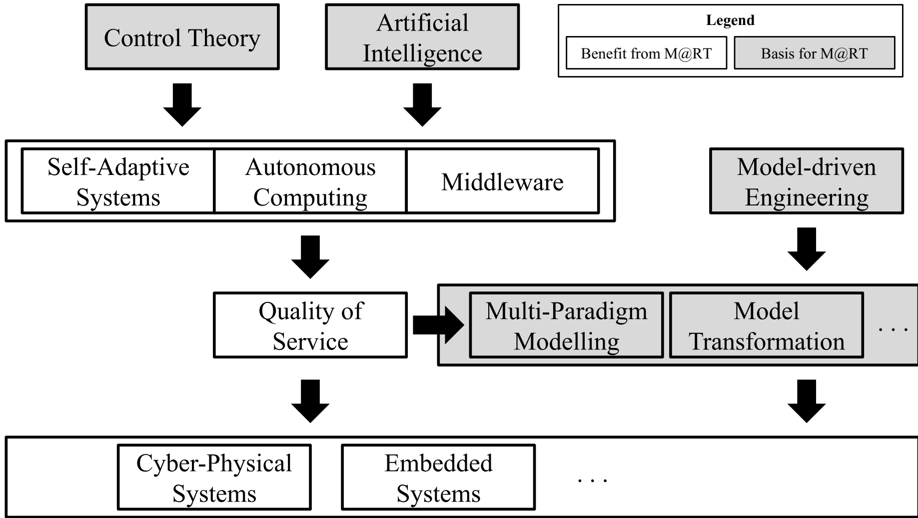

Fig. 2. Research Communities Relevant to Models@run.time

advancements in these communities, which provide use cases to models@run.time in turn.

All three communities rent many concepts from the **control theory** and the **artificial intelligence** community. Whereas control theory covers systems adjusting themselves to external influences in general, artificial intelligence provides, among others, planning and analysis techniques to coordinate autonomous systems.

Another set of communities relevant to models@run.time are those focusing on **quality-of-service** optimization and assurance. This includes various specialized communities which cover particular non-functional properties. For example, *performance*, *fault-tolerance*, *safety*, *physical dynamics* and *energy*. Each community requires means to model the (non-functional) behavior of a system subject to optimization or assurance w.r.t. the specific non-functional property of interest. To realize optimization or assurance, again reflection is used as a basis. Especially, the prediction capabilities of models@run.time are beneficial for these communities. In turn, they provide use cases to models@run.time.

The **model-driven engineering** community provides fundamental techniques to models@run.time. Besides general modeling techniques, solutions to particular problems for models@run.time are addressed by this community. For example, model evolution, model transformation, model synchronization and model-based diagnostics, where each problem usually forms its own community.

To cover multiple aspects of a system's non-functional behavior, modeling techniques from different domains are to be integrated or bridged. This challenge is addressed by the **multi-domain/multi-paradigm** modelling community.

Hence, this community provides fundamental techniques to the models@run.time community that can be useed by the *quality-of-service* community.

Moreover, especially the **cyber-physical systems** community demands for multi-domain modelling (physical dynamics combined with computational models). It is yet another community, which makes use of models@run.time. This is because cyber-physical systems are self-adaptive or autonomous systems, where a particular challenge is the management of multiple non-functional properties from different domains. This includes the **embedded systems** community, which starts to investigate networked embedded systems. Notably, embedded systems are inherently self-adaptive, because they are embedded in an environment and are meant to observe and/or influence it.

5 Short and Long-Term Research Questions - A Roadmap

In the following, open research challenges for models@run.time will be discussed. We examine short-term research topics, followed by long-term research topics.

5.1 Short-Term Research Challenges

As short-term research challenges, we identified the application of MDE techniques, the optimization of reasoning and reconfiguration in terms of efficiency (i.e., optimal tradeoff between cost and utility), the management of uncertainty inherent to models@run.time, synchronization of reflexive models and safety assurance at runtime. In the following, we elaborate on each challenge.

Model-Driven Engineering. Modeling is a central constituent of models@run.time. Hence, in theory, models@run.time could directly benefit from tools and approaches developed by the Model-Driven Engineering (MDE) community: metamodels, editors, simulators, compilers, etc. In practice, however, it is difficult to embedded MDE tools at runtime, since these tools, usually thought for design-time usages in a rather standalone and controlled environment (IDE), come with important memory and performance (time) penalties. Thus, current MDE techniques should be investigated, extended, adjusted and/or directly be applied to models@run.time.

Efficiency @ Runtime. Dynamic adaptation of a software application is a process that might take some time, which–depending on the context–might or might not be an issue. For example doing some reconfiguration to better balance load and energy consumption in a cloud might afford a reconfiguration delay of several seconds, while an interactive system should be able to handle the overall reconfiguration in less than 200 ms. Even worse, if we want to push these system towards safety critical, real-time embedded systems, the constraints might be much harsher.

In our experience, the two main limiting factors in reconfiguring an application are (1) the time and resources taken for the reasoning itself (compute which

configuration is to be chosen) and the (2) the adaptation itself (e.g., stopping and starting components, loading new code, transferring state, etc.).

On the first account, one challenge is of course to leverage reasoning techniques that might be able to make the right tradeoff between time and intelligence. New advances in incremental reasoning, or time constrained reasoners might also be needed. The layering ability of models@run.time (i.e., the ability to manage models@run.time systems by models@run.time systems themselves) allows to adapt or optimize the reasoning and reconfiguration itself. The key challenge to be addressed here are (a) the assessment of reasoners and reconfigurations in terms of their costs (time, energy, etc.) and resulting utility and (b) the determination of cost budgets, which most not be exceeded by reasoning and reconfiguration. For example, if a system can perform a task either in one minute or, if reconfigured, in half a minute, the time budget for reasoning and reconfiguration is less than half a minute. If reasoning and reconfiguration take more than half a minute, the gain of running the task on the reconfigured system is lost.

Additionally, on the adaptation process itself, a few points are subject to possible optimizations, combining system issues (such as maintaining caches for frequently used configurations, efficient code loading, light component models etc.) with optimizations in the reconfiguration planning algorithms taking into account the specificity of the underlying platforms.

Managing Uncertainty. Systems adhering to the models@run.time paradigm have to cope with the uncertainty of the systems they manage or, in other words, uncertainty is inherent to models@run.time systems. This is because the managed systems environment is uncertain by nature. Hence, novel approaches, which enable reasoning in the presence of uncertainty are required.

Handling Reflexive Models of Distributed Systems. Handling reflexive models of distributed systems is a well-known issue in the distributed systems community. Having a centralized reasoner working on a centralized model of a distributed system makes little sense for reliability and robustness reasons, but managing a distributed model implies that the reasoner has to also handle consensus and synchronization issues. Several works already go into that direction [ECBP11], but more is to be done to also deal with performance issues and real-time constraints.

Realizing Safety Assurance at Runtime. Most of the safety-critical business nowadays follows very stringent procedures that are statically checked, most often under strict legal regulation as it is the case in the aerospace domain. Such systems can still be somehow adaptive within well defined boundaries (often called *modes*). There are typically very few modes (such as *normal mode, recovery mode, survival mode, panic mode*, etc.). They are well identified and individually checked for safety. All possible transitions between modes are also checked.

Providing the same level of safety for systems having modes only computed at runtime would imply having the same level of safety checking done on the fly at runtime. While using verification technologies such as *Model Checking* at runtime (using, e.g., the power of the cloud) is no longer considered as science-fiction, it is clear that this is still a challenge, and the proof that models@run.time can be safe and adaptive concurrently even in principle remains to be made.

5.2 Longer-Term Research Challenges

Several challenges of models@run.time, which demand for long-term investigation, can be identified. This includes the handling of quality interferences, the handling of interconnected control loops and the attainment of predictability by top-level feedback loops.

Quality Interferences. As has been pointed out by Salehie and Tahvildari [ST09], most of the current approaches to self-adaptive software exploit only a single quality. The exclusive focus on either reliability or energy or performance or security hides the problem of interferences and general dependencies between qualities. To consider multiple qualities simultaneously, their interdependent behavior needs to be determined and considered in the reasoning approaches. Thus, the monitoring and analysis phase need to be aware of the dependencies between qualities, which leads to a combinatorial explosion of cases to be considered in these two phases (i.e., all situations need to be investigated for all combinations of qualities). In addition, the reasoning approaches for the decision or planning phase need to support multi-objective decision-making, which is known to be an NP-hard problem. Finally, approaches for the act or execute phase need to consider quality interferences too, because their actions might imply a chain of reactions w.r.t. the quality assurance of the system. Thus, all phases of the feedback loop need to be investigated w.r.t. dependencies and interferences between qualities.

Interconnection of Multiple Feedback Loops. The need for multiple, interconnected feedback loops arises from the need for seamless system integration as envisioned by the CPS or Systems-of-Systems community. If multiple models@run.time systems are meant to cooperate, their feedback loops need to be capable of cooperation too. But, the interconnection of multiple feedback loops (e.g., in terms of layers as has been shown in CRC 614 [ADG+09]) opens further research questions, which affect all aspects of self-adaptive systems. The monitoring and analysis phase need to be aware that the system under investigation might not be a (continuous) physical system, but is itself a (discrete) models@run.time system. The same holds for the decision-making or planning and the act or execute phase, which, for example, need to differentiate between continuous and discrete systems. Clear interfaces between models@run.time systems, which are subject to integration, are required. Besides differentiating between continuous and discrete systems, the architecture or architectural style of

a models@run.time system is a key characteristic, which needs to be considered throughout the complete feedback loop of a dependent models@run.time system.

Predictability. Maybe the central challenge of models@run.time system is to reach full predictability of their behavior. The underlying problem is the dichotomy of adaptivity and predictability. Models@run.time systems are highly adaptive systems, but demand for precise predictability to enable more intelligent reasoning approaches. Precise predictability is a key enabler for several technologies as explained in section 2.

6 Conclusion

As highlighted by important roadmaps in research and industry, cyber-physical systems [Lee08] are considered to be the next generation of embedded systems. For example, in the agricultural domain already today it is possible to connect a tractor with another autonomously driving tractor [Fen11]. In the near future, this kind of interconnection of different systems is expected to increase rapidly throughout a broad range of further application domains such as automotive and healthcare. In the former, cars will dynamically connect to each other to implement functionalities like automated cross roads assistants [Con11]. In the latter, medical devices, telecommunication infrastructure and IT-based service systems will build dynamic ecosystems leading to a new generation of health care systems [All11].

All of these examples share the commonality that different devices, machines, and vehicles are integrated at runtime and that they have to adapt to dynamically changing environment contexts. In consequence, neither the structure nor the behavior of the cyber-physical systems can entirely be predicted at design time. This greatly complicates the assurance of important functional and non-functional properties - up to the point of impossibility for some cases. One of these particularly difficult cases is the assurance of safety, which is nevertheless mandatory since many of these cyber-physical systems are inherently safety-critical. As of today, only proprietary approaches are used to ensure safety of strictly predefined machine combinations. This obviously requires an immense effort and strongly limits the desired flexibility. Since traditional approaches are not expected to scale to adequately address cyber-physical systems, safety is a bottle neck preventing the transition from a promising idea to a real business success. Thus, there is an inescapable need for new approaches enabling the development of dynamically adaptive yet safe cyber-physical systems.

Solving this challenge can be a killer application for models@run.time. Regarding the examples mentioned above, it is not any longer the question whether dynamic adaptation is necessary or not. It is the question how important properties such as safety can be assured in the context of open adaptive cyber-physical systems. A general solution approach is to shift parts of the required assurance measures into runtime by means of adequate models@run.time. As opposed to other approaches, models@run.time explicitly define all facets of the dynamic

adaptation behavior of a system. Moreover, models@run.time enable a system to systematically reason at runtime about its current quality state, to predict the impact of possible system modifications on system quality, and, therefore, to select safe adaptation strategies following predictable and traceable rationales.

Particularly safety, as a bottleneck to business success, can be an important factor to create the pressure necessary to introduce a new technology. So the idea of using models@run.time for assuring safety in cyber-physical system can be a door opening killer application for models@run.time. Once the door is opened, the application can easily be extended to any other quality properties.

In summary, models@run.time advance over reflective systems in that they offer abstract, tractable and predictive reflection as shown by the reference architecture presented in section 3. This enables improvements to existing application domains, which already make use of reflection, and–in particular–enables the realization of safety-critical, cyber-physical systems.

References

[ADG+09] Adelt, P., Donoth, J., Gausemeier, J., Geisler, J., Henkler, S., Kahl, S., Klöpper, B., Krupp, A., Münch, E., Oberthür, S., Paiz, C., Podlogar, H., Porrmann, M., Radkowski, R., Romaus, C., Schmidt, A., Schulz, B., Vöcking, H., Witkowski, U., Witting, K., Znamenshchykov, O.: Selbstoptimierende Systeme des Maschinenbaus - Definitionen, Anwendungen, Konzepte. HNI-Verlagsschriftenreihe (2009)

[AG93] Atlee, J.M., Gannon, J.: State-based model checking of event-driven system requirements. IEEE Transactions on Software Engineering 19, 24–40 (1993)

[All11] The Continua Alliance. The continua alliance webpage, `http://www.continuaalliance.org` (visited on December 14, 2011)

[BBF09] Blair, G., Bencomo, N., France, R.B.: Models@run.time. IEEE Computer 42(10), 22–27 (2009)

[BCL+03] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B.: The fractal component model and its support in java. Software Practice and Experience 36(11-12), 1257–1284 (2003)

[BGW93] Bobrow, D., Gabriel, R., White, J.: Clos in context—the shape of the design. In: Paepcke, A. (ed.) Object-Oriented Programming: The CLOS Perspective, pp. 29–61. MIT Press (1993)

[CBG+08] Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., Sivaharan, T.: A generic component model for building systems software. ACM Transactions on Computer Systems 26, 1:1–1:42 (2008)

[CCOR06] Chang, H., Collet, P., Ozanne, A., Rivierre, N.: From Components to Autonomic Elements Using Negotiable Contracts. In: Yang, L.T., Jin, H., Ma, J., Ungerer, T. (eds.) ATC 2006. LNCS, vol. 4158, pp. 78–89. Springer, Heidelberg (2006)

[CGK+11] Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS Management and Optimization in Service-Based Systems. IEEE Transactions on Software Engineering 37(3), 387–409 (2011)

[Con11] Car2Car Communication Consortium. Car2car communication consortium webpage, `http://www.car2car.org` (visited on December 14, 2011)

[Dij82] Dijkstra, E.W.: On the role of scientific thought. In: Selected Writings on Computing: A Personal Perspective, pp. 60–66. Springer (1982)

[ECBP11] Etchevers, X., Coupaye, T., Boyer, F., De Palma, N.: Self-configuration of distributed applications in the cloud. In: IEEE CLOUD, pp. 668–675 (2011)

[ELLSV02] Edwards, S., Lavagno, L., Lee, E.A., Sangiovanni-Vincentelli, A.: Design of embedded systems: formal models, validation, and synthesis. In: Readings in Hardware/software Co-Design, pp. 86–107. Kluwer Academic Publishers, Norwell (2002)

[Fen11] AGCO Fendt. Fendt guideconnect: Two tractors - one driver. Website (2011)

[FMS11] Fleurey, F., Morin, B., Solberg, A.: A model-driven approach to develop adaptive firmwares. In: SEAMS 2011: ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, Waikiki, Honolulu, USA, May 23-24, pp. 168–177 (2011)

[FS09] Fleurey, F., Solberg, A.: A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 606–621. Springer, Heidelberg (2009)

[GHJV95] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of reusable Object-Oriented Software. Addison-Wesley Professional (1995)

[GWCA11] Götz, S., Wilke, C., Cech, S., Aßmann, U.: Runtime variability management for energy-efficient software by contract negotiation. In: Proceedings of the 6th International Workshop Models@run.time, MRT 2011 (2011)

[GWCA12] Götz, S., Wilke, C., Cech, S., Aßmann, U.: Architecture and Mechanisms for Energy Auto Tuning. In: Sustainable ICTs and Management Systems for Green Computing. IGI Global (June 2012)

[KM09] Kramer, J., Magee, J.: A rigorous architectural approach to adaptive software engineering. Journal of Computer Science and Technology 24(2), 183–188 (2009)

[Lee08] Lee, E.A.: Cyber physical systems: Design challenges. In: 2008 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), pp. 363–369 (May 2008)

[MBJ+09] Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F., Solberg, A.: Models@ run.time to support dynamic adaptation. IEEE Computer 42(10), 44–51 (2009)

[MBNJ09] Morin, B., Barais, O., Nain, G., Jézéquel, J.-M.: Taming Dynamically Adaptive Systems Using Models and Aspects. In: International Conference on Software Engineering (ICSE2009). IEEE, Los Alamitos (2009)

[OSG12] OSGi Alliance. Osgi core release 5 (March 2012)

[RWLN89] Rothenberg, J., Widman, L.E., Loparo, K.A., Nielsen, N.R.: The Nature of Modeling. In: Artificial Intelligence, Simulation and Modeling, pp. 75–92. John Wiley & Sons (1989)

[SCG+12] Schroeter, J., Cech, S., Götz, S., Wilke, C., Assmann, U.: Towards modeling a variable architecture for multi-tenant saas-applications. In: Proceedings of Sixth International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS 2012 (2012)

[ST09] Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems 14, 14:1–14:42 (2009)

[ST11] Schneider, D., Trapp, M.: A safety engineering framework for open adaptive systems. In: Proceedings of Fifth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2011), pp. 89–98. IEEE (2011)