Megamodel Consistency Management at Runtime

El Hadji Bassirou Toure(✉), Ibrahima Fall, Alassane Bah, and Mamadou Samba Camara

Institut de Recherche pour le Développement (IRD),
École Supérieure Polytechnique (ESP),
Université Cheikh Anta Diop de Dakar (UCAD), Dakar, Senegal
{bassirou.toure,ibrahima.fall}@esp.sn,
{alassane.bah,mamadousamba.camara}@ucad.edu.sn

Abstract. This paper addresses the problem of ensuring consistency, correctness and other properties in dynamically changing software systems. The approach uses a Megamodel that represents the current state of the system at runtime including some rules. These rules are formulated as Hoare-Triples and allow to check whether modifications to the software system result in a consistent state, otherwise to fix changes that are likely to violate the megamodel integrity.

Keywords: Runtime software evolution · Runtime verification
Megamodeling · Correctness · Axiomatic semantics

1 Introduction

Software architectures increasingly rely on abstractions to describe system components, the way the components interact and the rules governing the composition of components into systems. Very often these descriptions address high-level and complex aspects of the software [1]. And this increased complexity of systems requires raising the level of abstraction. Model-Driven Engineering (MDE) is an approach that advocates software abstraction through an exclusive use of models. In fact, MDE specifically considers software models which are abstractions of static or dynamic properties of a software system [2]. In fact, it is a "*recent*" Software Engineering (SE) field which promotes the exclusive use of *models* in the software system development, maintenance and evolution. A *model* is recursively defined as an artifact which consists of model elements, conforms to a specific *metamodel* and represents a given view of a system. A *metamodel* can be defined as a model of the language used to represent a given model [3]. MDE also distinguishes *prescriptive models* and *descriptive models*. A *prescriptive model* is a model which represents the manner a system has to be created, i.e. the system is built on the basis of the model. While a *descriptive model* provides information about an existing system. This distinction allows to define *model correctness* and *system validity* which are two concepts related to

C. M. F. Kebe et al. (Eds.): InterSol 2017/CNRIA 2017, LNICST 204, pp. 257–266, 2018.
https://doi.org/10.1007/978-3-319-72965-7_24

the causal connection between a model and the system it represents. In fact, a *model* is *correct* if all its statements are true for the represented system. A system is *valid* according to its specifications if no statement in the *model* is false for that system [7].

MDE uses the concept of a *megamodel* as a building block for modeling in the large. For that a megamodel must ignore the internal details of global entities such as models and metamodels by stressing on the "*big picture*", i.e. *system architecture*, *assignment of models as parameters or results for model transformations*. Indeed a *megamodel* is considered as being a model whose elements are models and which considers the interconnections between multiple models in the form of model operations [3]. That is why *megamodels* are often used for representing the components of an architecture and the interactions between them [5].

Software architectures are often made to evolve due to changing user needs and/or to execution environment. Such changes are often done through adaptation mechanisms in which components are often added to or removed from multiple operations in the system. In this context for a system with few changing requirements, it probably suffices to apply these adaptations at the development-time. However in a dynamically evolving system, changes to the execution environment, to the components and/or interconnections, may take place at runtime. Indeed in many application domains there is a need that the system accommodates changes dynamically, without stopping or disturbing the operation of those parts of the system unaffected by the changes [11].

Moreover the identity of the components that a system will utilize may not be known until the execution phase. For that a software architecture is then modeled as a collection of particular MDE models which are called *runtime models*. A *runtime model* provides a view, on a running software system, that is usually used for managing the system to which it is causally connected [4]. The causal connection between models and represented systems means that each time the system changes, then the model is updated, and similarly, if the model changes, it causes the proper system change [6].

In this paper we propose the use of a runtime megamodel to represent the current state of a running system. It consists of related runtime models which represent runtime artifacts. Such runtime artifacts may include *component creation and destruction*, *exceptions/errors*, *operation inputs and output*, *component operations invocation*, *dynamic artifact types*, *dynamic component names*, *and so on*. Modifying part of a system in one model can thus introduce inconsistencies with related parts of the system specified in other models. Thus these models can be inconsistent with each other since they describe the system from different aspects and views. Therefore there is an inherent need to preserve consistency between those models registered in a runtime megamodel even when the software evolves.

To ensure consistency and functional validity of the dynamically changing systems, a runtime verification system is required. *Runtime verification* is a method of checking whether the active execution trace of software adheres to its specifications [13]. Against to other verification techniques such as model

checking which aims at checking all possible execution traces of the software, runtime verification reduces the verification scope to one execution of the software; this increases the accuracy of the verification, especially for dynamic artifacts of the software.

For that we will consider the runtime megamodel as being an execution environment or a program in which instructions are operations that mainly compute with component models by adding or removing them even though these changes have not to violate the correctness of the megamodel $vis - \grave{a} - vis$ to the represented systems. Accordingly we use techniques for proving programs correctness known as *Hoare's axiomatic semantics* and some *inference rules* in order to set up our runtime verification technique which enables us to keep the megamodel consistent at anytime.

Organization of the paper. The remainder of this paper is articulated as follows. In Sect. 2, we begin with the presentation of an illustrative example which will be used throughout the paper. Section 3 Presents the approach by using the running example to illustrate it. Section 4 is reserved for the conclusion.

2 Illustrative Example

In this illustrative example which will be used all around the paper, we suppose that we have a megamodel $\mathbf{M}$ which represents a model of a system $\mathbf{S}$ (see Fig. 1). Given that we have a new tool $\mathbf{T}$ to plug in the system $\mathbf{S}$ which is already deployed and running.

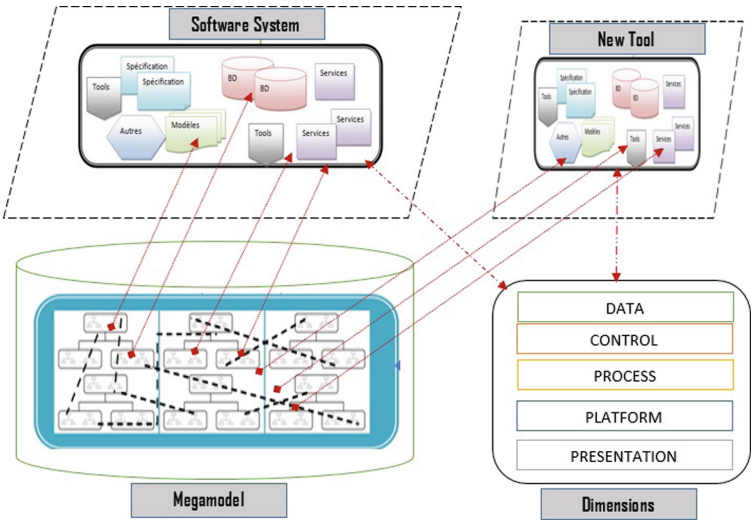

Fig. 1. The running Example

For that we suppose that $\mathbf{T}$ will be added to $\mathbf{S}$ to the *data dimension*.

- Let $\mathbf{M}$ be the megamodel representing the system $\mathbf{S}$.
- Let ms_d be the *component model* representing the *data model* of $\mathbf{S}$.
- Let mt_d be the *component model* representing the *data model* of $\mathbf{T}$.
- Let map_d be the *component model* representing a morphism for ms_d and mt_d.

3 Our Approach

3.1 Overview of the Approach

The presented approach proposes a runtime megamodel to represent a running software system. The approach will use a *megamodel* which represents an execution model of the running system to which it is causally connected. Indeed the megamodel is deployed with the system and used as a basis for *model management* and *changes representation*. Our approach should also provide *inference rules* for reasoning about, specifying, and representing change operations. Thereby checking the *megamodel's consistency* by defining a *formal-safe execution* as well as an *execution semantic* for each operation likely to modify the *megamodel* contents.

3.2 The Runtime Megamodel

To create a composite system, the first step should be to highlight the system components which represent the artifact that will be put in the new system. To facilitate this process, *component models* will be stored in a *megamodel* considered as a registry of components and their interactions. The access and the management of such a *megamodel* should be done at runtime. Each *component model* will have a name to identify it and a type to represent its metamodel. A component model should also specify the information it represents. Such informations can be a functional or nun-functional element, a process element, a data element, etc. A *Component model* can either be used to create a new artifact, in this case it is said a prescriptive model or it can be used to describe an existing artifact and in that case it is called a descriptive model Fig. 2. An extensive study of the semantic of each operation is presented in [5] to which we refer the reader for further information.

Application

In our application, the *megamodel* is represented through an *ecore metamodel* (Fig. 2). It is considered as an environment which is managed using a textual Domain-Specific Language (DSL): the *Mega Operation Management and Execution (MOME)* which is being constructed.

3.3 Changes in the Megamodel

The megamodel consists of *component models* and *global operation models*. A *global operation model* can be considered as a type of a *global operation instance*.

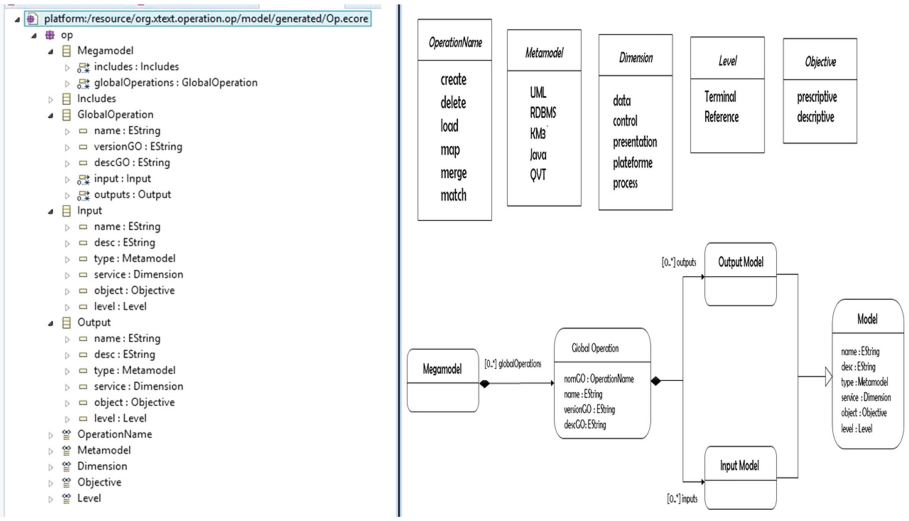

Fig. 2. The megamodel metamodel.

Therefore, it represents a model of future interactions which are *global operation instances*. And those *global operation instances* link some *component instances*. That is, a *global operation model* defines some interaction rules, is instantiated on *component instances* and allows to dynamically set connections between components. A *global operation model* will be applied to component models already registered in the megamodel, and its results are *new component models* that will have to be put in the megamodel.

The management of the megamodel (megamodelling) sounds like programming where the megamodel plays the role of the execution environment (program) [9]. The content of the megamodel is modified by the execution of *global operation execution*. Therefore, the megamodel is proposed to dynamic and frequent changes. Such changes may include the *introduction of new components*; the *recreation of failed components*; the *modifications of component interconnections*; the *change component operating parameters*, etc. A *global operation models* corresponds to a *set of operations* which are executed in response to changes related to the underlying system state changes. Otherwise these changes have to leave the megamodel consistent.

Application

In the megamodel, the *changes representation* are performed through the global Operation execution. For example at Figs. 3 and 4, we show respectively how the *Merge global Operation* is represented in a *MOME program* and a *part of its execution semantics*.

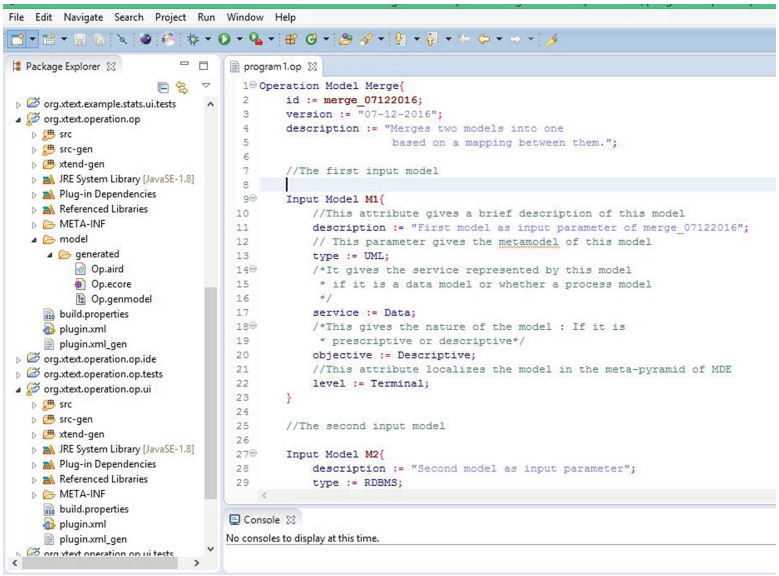

Fig. 3. A *MOME program*

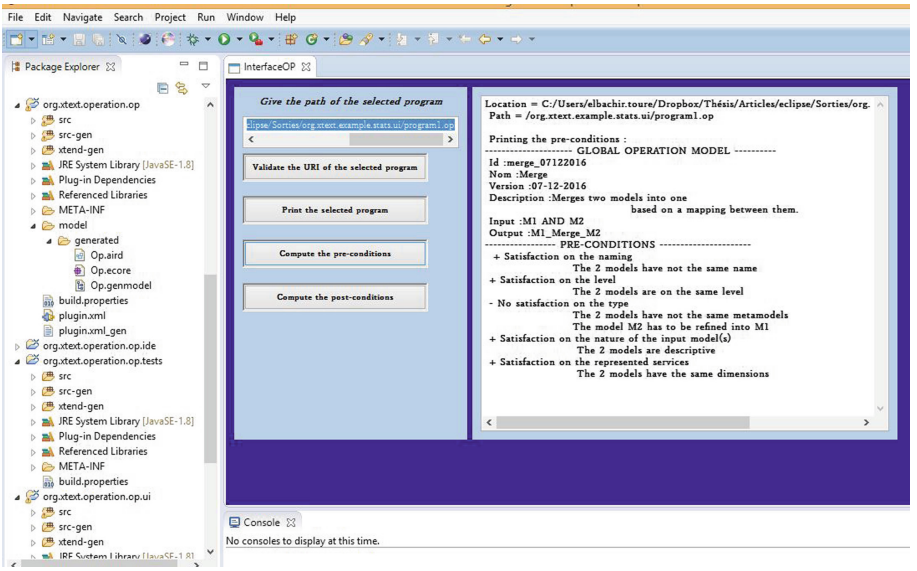

Fig. 4. A *part of the execution semantics of the MOME program*

3.4 Ensuring Correctness of the Megamodel

These changes necessitate to focus on the way the megamodel react after a change. Hoare Triples allow us to fix this problem by proposing a *formal-safe execution* and an *execution semantics* for each *global operation*.

The megamodel acts as an execution environment whose instructions are the *global operation executions*. A *global operation model* involves a set of *pre-conditions* P, a *sequence of operations* Seq, and a set of *post-conditions* Q.

$$\{P\}Seq\{Q\}$$

The *pre-conditions* represent a set of states of the system from which a given *sequence of operations* can be used.

Possibly each *operation* may have a *side effect* which refers to its impact on other components. Each operation may impact either some *component models* or other *global operation models*.

The *post-conditions* define a set of states that satisfy the required result after the execution of a global operation models.

To achieve this, as in previous works [8], we have defined a *formal-safe execution* which ensures that execution of a *global operation model* does not lead to some inconsistencies in the megamodel. We have also defined an *execution semantic* for each *global operation model*, which means its observable behavior (its side-effect).

Execution semantic for global operations

For the **Merge** *global operation* which allows to integrate $\mathbf{T}$ with $\mathbf{S}$ through their data dimension, we define its execution semantic as follows:

Operation Merge: $merge_d \longleftarrow Merge(ms_d, map_d, mt_d)$

Pre-conditions	**Post-conditions**
$P_1 ::= \mathrm{ms}_d \in M$	$Q_1 ::= \mathrm{ms}_d \in M$
$P_2 ::= level(\mathrm{ms}_d, \text{“}terminal\text{”})$	$Q_2 ::= level(\mathrm{ms}_d, \text{“}terminal\text{”})$
$P_3 ::= type(\mathrm{ms}_d, \text{“}UML\text{”})$	$Q_3 ::= type(\mathrm{ms}_d, \text{“}UML\text{”})$
$P_4 ::= dimension(\mathrm{ms}_d, \text{“}data\text{”})$	$Q_4 ::= dimension(\mathrm{ms}_d, \text{“}data\text{”})$
$P_5 ::= \mathrm{mt}_d \in M$	$Q_5 ::= \mathrm{mt}_d \in M$
$P_6 ::= level(\mathrm{mt}_d, \text{“}terminal\text{”})$	$Q_6 ::= level(\mathrm{mt}_d, \text{“}terminal\text{”})$
$P_7 ::= type(\mathrm{mt}_d, \text{“}UML\text{”})$	$Q_7 ::= type(\mathrm{mt}_d, \text{“}UML\text{”})$
$P_8 ::= dimension(\mathrm{mt}_d, \text{“}data\text{”})$	$Q_8 ::= dimension(\mathrm{mt}_d, \text{“}data\text{”})$
$P_9 ::= \mathrm{map}_d \in M$	$Q_9 ::= \mathrm{map}_d \in M$
$P_{10} ::= \mathrm{merge}_d \notin M$	$Q_{10} ::= \mathrm{merge}_d \in M$

We have the triplet:

{P} *Merge* (ms_d, map_d, mt_d) {Q}; where $\mathrm{P} = \cap \mathrm{P}_i$ and $\mathrm{Q} = \cap \mathrm{Q}_i$

Similarly, for each *global operation model*, we define its execution semantic (Fig. 4).

Setting up a *global operation model* may lead to the execution of other *global operations*, namely its *ripple effect*. In such cases, often the output of a *global operation model* may correspond to the input of another *global operation model*.

Safe executions of global operations

Before invoking the *Merge* operation, we have to call at first the global operation *Map* with the two model parameters. Indeed the third parameter (*map*) to *Merge* is a morphism describing the elements of ms_d and mt_d that are equivalent and should be "*merged*" into a single element map_d in $\mathbf{M}$. Therefore we can note a logical precedence between some *global operations* defined in the megamodel. To consider this, we have to set up a *deductive system* which enables the deduction of new theorem from some theorems already proved. A *rule of inference* takes the form "*If* $\vdash X$ *and* $\vdash Y$ *then* $\vdash$ Z", i.e. if assertions of the form X and Y have been proved as theorems, then Z also is thereby proved as a theorem.

For that we will use two inference rules presented in [10], that is the *rule of consequence* and the *rule of composition*. After that, we present an example in which these two rules are applied.

(i) Rules of consequence:

$$\textit{If} \vdash \{P\}S\{R\} \textit{ and} \vdash R \supset Q \textit{ then} \vdash \{P\}S\{Q\}$$
$$\textit{If} \vdash \{P\}S\{R\} \textit{ and} \vdash P \subset Q \textit{ then} \vdash \{Q\}S\{R\}$$

(ii) Rule of composition:

$$\textit{If} \vdash \{P\}S_1\{Q_1\} \textit{ and} \vdash \{Q_1\}S_2\{R\} \textit{ then} \vdash \{P\}(S_1; S_2)\{R\}$$
$$\frac{\vdash \{P\}S_1\{Q_1\} \quad \vdash \{Q_1\}S_2\{R\}}{\vdash \{P\}(S_1; S_2)\{R\}} \textit{ global operation}$$

We can define the inference rule for the *Merge global operation* which enables us to integrate models.

Application

Considering the previous example. We have to describe an execution of *Merge* between the two models, namely ms_d and mt_d. However before invoking the *Merge* Operation, it must be necessary to call at first the *Map* operations. For that we define an execution semantic of *Map*.

Operation. $Map : map_d \longleftarrow Map(\mathrm{ms}_d, mt_d)$

Pre-conditions	**Post-conditions**
$M_1 :: = \mathrm{mt}_d \in M$	$N_1 :: = \mathrm{mt}_d \in M$
$M_2 :: = \mathrm{ms}_d \in M$	$N_2 :: = \mathrm{ms}_d \in M$
$M_3 :: = \mathrm{map}_d \notin M$	$N_3 :: = \mathrm{map}_d \in M$
$M_4 :: = type(\mathrm{ms}_d) == type(mt_d)$	$N_4 :: = type(\mathrm{ms}_d) == type(mt_d)$
$M_5 :: = dimension(\mathrm{ms}_d) == dimension(mt_d)$	$N_5 :: = dimension(\mathrm{ms}_d) == dimension(mt_d)$

We have the triplet:

$(i)\, A_1 \longleftarrow \{M\}Map(\mathrm{ms}_d, mt_d)\{N\}$; where M $=\cap \mathrm{M}_i$ and N $= \cap \mathrm{N}_i$

Operation. $Merge : merge_d \longleftarrow Merge(\mathrm{ms}_d, map_d, mt_d)$

Pre-conditions	**Post-conditions**
$I_1 :: = \mathrm{mt}_d \in M$	$R_1 :: = \mathrm{mt}_d \in M$
$I_2 :: = \mathrm{ms}_d \in M$	$R_2 :: = \mathrm{ms}_d \in M$
$I_3 :: = \mathrm{map}_d \in M$	$R_3 :: = \mathrm{map}_d \in M$
$I_4 :: = \mathrm{merge}_d \notin M$	$R_4 :: = \mathrm{merge}_d \in M$
$I_5 :: = type(\mathrm{ms}_d) == type(mt_d)$	$R_5 :: = type(\mathrm{ms}_d) == type(mt_d)$
$I_6 :: = dimension(\mathrm{ms}_d) == dimension(mt_d)$	$R_6 :: = dimension(\mathrm{ms}_d) == dimension(mt_d)$

We have the triplet: $(ii)\, A_2 \longleftarrow \{I\}\mathrm{Merge}\ (\mathrm{ms}_d, map_d, mt_d)\{R\}$; where I $=\cap \mathrm{I}_i$ and R $= \cap \mathrm{R}_i$

Using the **rules of consequence** on (i) and (ii) we obtain:

$$\vdash \{M\}Map\{N\}\ and \vdash\ I \supset\ N\ then, \vdash \{M\}Merge\{R\}$$

We have: $A_3 \longleftarrow \{M\}Merge\{R\} \rightarrow (iii)$

More formally, using the **rules of composition** we obtain:

$$\frac{\vdash \{M\}Map\{N\}\ \ \vdash \{I\}Merge\{R\}}{\vdash \{M\}Merge\{R\}}\, Merge$$

4 Conclusion

In this paper we have proposed a runtime megamodel to represent a running software system. The approach uses a *megamodel* which represents an execution model of the running system to which it is causally connected. Indeed the megamodel is deployed with the system and used as a basis for *model management* and *changes representation*. Our approach also provides *inference rules* for reasoning about, specifying, and representing change operations. Thereby checking the *megamodel's consistency* by defining a *formal-safe execution* as well as an *execution semantic* for each operation likely to modify the *megamodel* contents.

References

1. Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G.: Abstractions for software architecture and tools to support them. IEEE Trans. Softw. Eng. **21**(4), 314–335 (1995)
2. Bislimovska, B.: Textual and content based search in software model repositories (Doctoral dissertation, Italy) (2014)
3. Bezivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: Proceedings of the 19th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications, October 2004
4. Vogel, T., Giese, H.: A language for feedback loops in self-adaptive systems: executable runtime megamodels. In: The 7th International Symposium on Software Engineering for Adaptive and Self-managing Systems, pp. 129–138, June 2012

5. Toure, E.B., Fall, I., Bah, A., Camara, M.B. Megamodel-based Management of Dynamic Tool Integration in Complex Software Systems. In Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS) (2016)
6. Song, H., Huang, G., Chauvel, F., Sun, Y.: Applying MDE tools at runtime: experiments upon runtime models. In: Models@run.time 10, vol. 641, pp. 25–36 (2010). CEUR-WS.org
7. Seidewitz, E.: What models mean. IEEE Software, September/October 2003
8. Bousso, M., Sall, O., Thiam, M., Lo, M., Toure, E.H.B.: Ontology change estimation based on axiomatic semantic and entropy measure. In: Signal Image Technology and Internet Based Systems (SITIS), pp. 458–465, November 2012
9. Vignaga, A., Jouault, F., Bastarrica, M.C., Brunelière, H.: Typing in model management. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 197–212. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02408-5_14
10. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–583 (1969)
11. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. IEEE Trans. Softw. Eng. **16**(11), 1293–1306 (1990)
12. Aßmann, U., Bencomo, N., Cheng, B.H., France, R.B.: Models@runtime (Dagstuhl seminar 11481). Dagstuhl Rep. **1**(11), 91–123 (2012)
13. Malakuti, S., Bockisch, C., Aksit, M.: Applying the composition filter model for runtime verification of multiple-language software. In: 20th International Symposium on Software Reliability Engineering, pp. 31–40. IEEE, November 2009