Software & Systems Modeling (2019) 18:3005–3023
https://doi.org/10.1007/s10270-018-00707-8

REGULAR PAPER

A unifying framework for homogeneous model composition

Jörg Kienzle[1] · Gunter Mussbacher[1] · Benoit Combemale[2] · Julien Deantoni[3]

Received: 13 January 2018 / Revised: 27 November 2018 / Accepted: 8 December 2018 / Published online: 3 January 2019

Abstract
The growing use of models for separating concerns in complex systems has lead to a proliferation of model composition operators. These composition operators have traditionally been defined from scratch following various approaches differing in formality, level of detail, chosen paradigm, and styles. Due to the lack of proper foundations for defining model composition (concepts, abstractions, or frameworks), it is difficult to compare or reuse composition operators. In this paper, we stipulate the existence of a unifying framework that reduces all structural composition operators to structural merging, and all composition operators acting on discrete behaviors to event scheduling. We provide convincing evidence of this hypothesis by discussing how structural and behavioral homogeneous model composition operators (i.e., weavers) can be mapped onto this framework. Based on this discussion, we propose a conceptual model of the framework and identify a set of research challenges, which, if addressed, lead to the realization of this framework to support rigorous and efficient engineering of model composition operators for homogeneous and eventually heterogeneous modeling languages.

Keywords Model composition · Symmetric merge · Event scheduling · Event structures · Separation of concerns

1 Introduction

Extending the time-honored practice of separation of concerns [13,39], Model-Driven Engineering (MDE) promotes the use of separate models to address the various concerns in the development of complex software-intensive systems [40]. The main objective is to chose the right level of abstraction to specify and reason about the system under development depending on stakeholder needs and system concerns. While some of these models can be defined with a single modeling language (e.g., UML), Domain-Specific Modeling Languages (DSMLs) are increasingly used to handle various concerns in system and software development [6]. To support this trend, the MDE community has developed advanced techniques for designing new DSMLs.

A consequence of separating concerns is that different, possibly heterogeneous, models need to be *composed* in order to execute the application or reason over global properties. In general, model composition unfolds along two dimensions, i.e., structure and behavior. So far, frameworks that offer composition operators had to define their own composition rules and provide custom-made implementations of their operators (e.g., through transformations).

Depending on the context of use, different or customized composition operators are needed to provide support for different development paradigms (e.g., incremental development or Software Product Line (SPL) development) and address the various objectives (e.g., analysis, compilation, runtime management, etc.) of the developer. Furthermore, due to the increasing number of application domains of interest, and the growing number of stakeholders, new DSMLs are constantly developed and new composition operators need to be developed accordingly. While dedicated foundations have been proposed in the last decade to systematically engineer modeling languages and more specifically DSMLs, this is not yet the case for defining the corresponding composition

Communicated by Dr Jeff Gray.

✉ Benoit Combemale
benoit.combemale@irit.fr

Jörg Kienzle
joerg.kienzle@mcgill.ca

Gunter Mussbacher
gunter.mussbacher@mcgill.ca

Julien Deantoni
julien.deantoni@polytech.unice.fr

1 McGill University, Montreal, Canada

2 University of Toulouse, Toulouse, France

3 Universite Cote d'Azur, I3S/INRIA, Nice, France

operators. Those foundations need to be elaborated in order to move from tedious, ad hoc crafting of composition operators to structured, streamlined engineering.

This paper proposes a framework for engineering composition operators that is based on the hypothesis that all structural composition can be expressed with symmetric *merging*, and all (discrete-event) behavioral composition can be reduced to asymmetric *event scheduling*. The framework introduces a canonical categorization of homogeneous model composition operators (i.e., weavers) and establishes a foundational set of capabilities required for most, if not all, DSMLs to support composition and modularization. Concretely, the contributions of this paper are (1) a clear definition of model composition via a clear definition of merging and event scheduling, (2) a survey of existing approaches dedicated to homogeneous model composition w.r.t. this categorization, and (3) a set of research challenges indicating how to realize the proposed model composition framework and outlining future research directions.

The structure of this paper is as follows. Section 2 illustrates the need for different homogeneous and heterogeneous composition operators by outlining examples in which structural and behavioral model composition is used in different contexts and for different purposes. Section 3 positions the proposed model composition framework in the landscape of composition techniques and describes four generic steps for composition operators. Section 4 then goes into further details of structural composition, while Sect. 5 discusses behavioral composition. Section 6 provides a description of the key concepts of the proposed model composition framework as part of the presentation of the set of research challenges. General purpose weavers are contrasted as related work with the proposed framework in Sect. 7. Finally, Sect. 8 concludes the paper.

2 Illustrating examples

This section presents concrete real-world examples in which models and model composition have been used to separate concerns during development. Each situation required the definition and implementation of composition operators, either structural or behavioral, tailored to the application context of the composition and the notation(s) that had to be composed.

2.1 Need for different homogeneous composition operators for standard modeling notations

Standard modeling notations, for example, the Unified Modeling Language, have been used extensively in the context of MDE over the last two decades. While the UML specification document published by OMG [37] standardizes the notation, it does not specify in what context or for what purpose the notation should be used during software development. For example, class diagrams have been used in very different contexts that imply different composition strategies, for example, for specifying object-oriented design structure, for specifying domain models, and even for defining the abstract syntax for modeling languages in form of metamodels.

While modeling and model-driven engineering continue to gain popularity, it soon also became clear that despite the power of abstraction of modeling, models of real-world problems and systems quickly grow to such an extent that managing the complexity by using proper modularization techniques becomes necessary [1]. As a result, many standard modeling notations have been extended with aspect-oriented mechanisms to support advanced separation of concerns. Not surprisingly, for a given notation, depending on the purpose for which models are being used, different composition operators have been proposed in the literature.

A concrete example of this, adapted from [19], is illustrated in Fig. 1. The top of the figure depicts two state diagrams, *SD1* and *SD2*. The bottom left shows the result of composing *SD1* with *SD2* using the state diagram composition operator defined by the *HiLA* approach [43]. The bottom right shows the result of composing the *same* state diagrams using the state diagram composition operator defined as part of the *Protocol Modeling* approach (PM) [30], which is in turn inspired by the CSP parallel composition operator ||.

The results of the composition are clearly different. HiLA is an approach where state diagrams are used in the context of low-level software design to describe the behavior of system components and generate code. In HiLA, states with matching names are merged, and transitions and states that only appear in one of the input models are copied. PM on the other hand is used during high-level requirements specification and analysis for simulation and test generation purpose. In this context, to ensure the tractability of protocol analysis and enable local reasoning, the || composition operator was designed to preserve the trace behavior of the input models. In other words, composing another protocol with a given protocol SD cannot override a constraint that SD says must be true. The || composition operator ignores state names and composes transitions that are *equivalents* in both input models by ensuring they are taken synchronously. In our specific example, since there are no equivalent transitions between *SD1* and *SD2*, the composition result contains the Cartesian product of the states in the two input models ($4 * 2 = 8$ states).

2.2 Need for heterogeneous composition operators between DSMLs

The development of modern complex software-intensive systems often involves the use of multiple DSMLs that capture

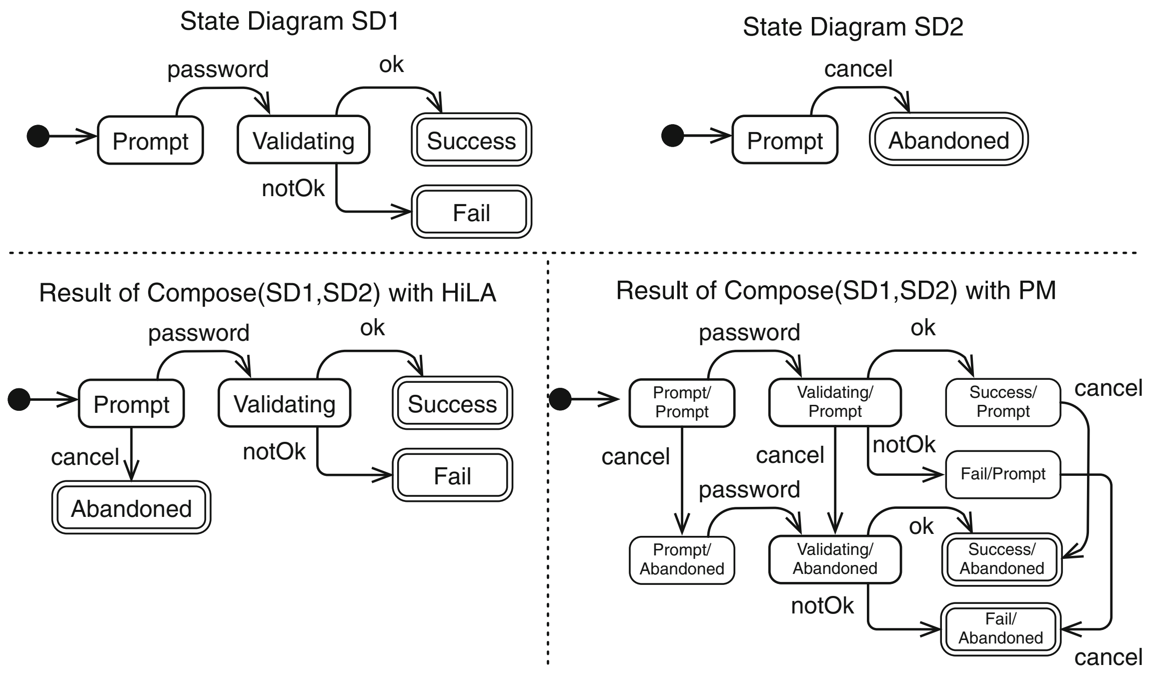


Fig. 1 Different composition operators for state diagrams

different system aspects. In addition, models of the system aspects are seldomly manipulated independently of each other. System engineers are thus faced with the difficult task of relating information presented in different models. For example, a system engineer may need to analyze a system property that requires information scattered in models expressed in different DSMLs. Current DSML development workbenches provide good support for developing independent DSMLs, but provide little or no support for integrated use of multiple composed DSMLs. The lack of support to explicitly relate concepts expressed in different DSMLs makes it very difficult for developers to reason about information spread across different models in one composed model.

Modern complex software-intensive systems increasingly use software as an integration layer. As a consequence, architectures require system-level models to integrate various engineering-specific architectures. For example, block diagrams describing device characteristics may have to be composed with class diagrams describing software systems.

Modern systems like cyber-physical systems and the Internet of Things are highly connected to the environment. Various DSMLs have been defined for describing the system behavior on the one hand, and the environment or the physical world on the other hand. For example, state machines may be used to describe the behavior of a system (e.g., ThingML[1] for embedded and distributed systems) and may have to be composed with sequence diagrams for interactions with the environment.

[1] Cf. http://thingml.org/.

3 The landscape of composition operators

The term composition is used in many situations [33]. At its most abstract, composition refers to the act of creating new entities from existing ones (e.g., by assembling together two or several smaller entities). This may occur at low levels of granularity (e.g., by adding an association or generalization between two classes) or at high levels of granularity (e.g., by connecting required and provided interfaces of components). In this paper, we are interested in composition in the context of MDE and DSMLs, i.e., in the composition of models at high levels of granularity. In other words, we are not interested in the composition of individual modeling elements such as a single class or state with another class or state, respectively, but at the composition of structural and behavioral models that represent broader concerns of interest to stakeholders. For space reasons, the paper further narrows the detailed discussions to the composition of homogeneous models (i.e., models defined by the same metamodel). While the initial focus is on homogeneous models, the intent is to investigate the applicability of our envisioned model composition framework to heterogeneous models in greater detail in the future.

Recently, aspect-oriented techniques have enabled advanced separation of concerns; i.e., they provide a developer with systematic means for the identification, separation, representation, and most importantly composition of crosscutting concerns. Aspect-oriented language extensions and dedicated composition operators (also known as *weavers*) have been defined for many programming languages and model-

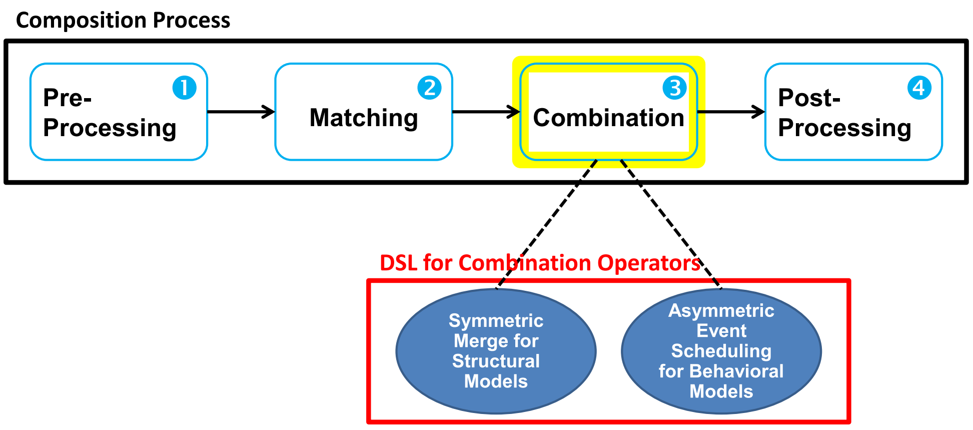

Composition Process
Pre-Processing 1
Matching 2
Combination 3
Post-Processing 4
DSL for Combination Operators
Symmetric Merge for Structural Models
Asymmetric Event Scheduling for Behavioral Models

Fig. 2 Overview of unifying framework for homogeneous model composition

ing languages. We evaluate our proposed model composition framework by mapping well-known model weavers (and hence a rich set of varied but clearly scoped composition operators) to the framework.

Many model composition operators have been proposed in the literature over the last two decades. In [29], Marchand et al. argue that any composition process can be reduced to four steps:

1. Optional preprocessing of the inputs,
2. Determining the composition location (either through implicit or explicit matching, and by intention or by extension),
3. Combining the inputs at the location(s) determined in step 2 to produce the output, and
4. Optional postprocessing of the output.

It is therefore not surprising that existing composition operators define algorithms for executing these steps, where Step 1 and Step 4 are optional. This applies to structural compositions such as UML Package Merge [38], Kompose [14], RAM Class Diagrams [21], Theme/UML Class Diagrams [7], and AoGRL [34] as well as behavioral compositions such as ADORE [31], TreMer+ [35], RAM Sequence Diagrams [21], Theme/UML Sequence Diagrams [7], HiLa [43], AoUCM [34], and RAM Protocol Models [2].

The preprocessing step (Step 1) may, for example, rename model elements as in Kompose [14]. The matching step (Step 2) may involve the evaluation of a pattern as in AoGRL and AoUCM [34] (i.e., an implicit approach) or the establishing of an explicit binding as in RAM [21]. Furthermore, matches may be defined by intention (i.e., expressed at the language level) as in TreMer+ [35] or by extension (i.e., expressed at the model level) as in Theme/UML [7]. Many other techniques have been discussed in the literature that perform different kinds of matching with varying degrees of sophistication and could be applied in Step 2. The final post-processing step (Step 4) may involve applying transformation rules to address conflicting model elements as in UML Package Merge [38].

The following two sections on structural and behavioral combination focus on Steps 2 and 3, since they are mandatory steps that all operators need to provide and because this allows us to consistently identify commonalities and differences between the operators. All aspect-oriented modeling techniques mentioned explicitly in this section are presented in more detail in the following two sections (some in greater detail than others for readability reasons). We outline how in each case, the third combination step can always be mapped to a symmetric merge operation for structural models and asymmetric event scheduling for behavioral models. The aim is to provide convincing evidence that the hypothesis on which the vision of our framework is built is indeed true.

Figure 2 illustrates the proposed framework, including the composition process and highlighting the step of the process to which symmetric merge (discussed in Sect. 4) and asymmetric event scheduling (discussed in Sect. 5) apply as well as the early DSL for the definition of reusable composition operators (as proposed in Sect. 6).

4 Structural combination: merge

This section precisely defines the structural *merge* operator of our framework and then shows in detail how it can successfully be applied to compose class diagrams. Then, further examples of how *merge* can provide composition for other modeling notations are outlined and the mathematical properties of *merge* are discussed.

4.1 Definition

Essentially, a structural model consists of elements (some of which are containers) with a set of properties, including

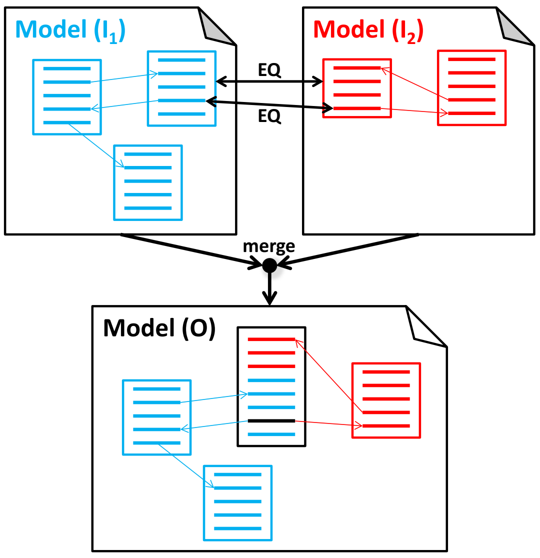


Fig. 3 $merge(I_1, I_2, EQ) \Rightarrow O$

relationships to other elements. The structural composition examples listed in Sect. 3 suggest that the combination operation for structural elements is always a symmetric merge. Formally, the merge combination operator takes two models I_1 and I_2 as inputs, as well as a set of equivalence relationships $EQ = e_1 \Leftrightarrow e_2$ with $e_1 \in I_1$ and $e_2 \in I_2$. Each model element e of input model I has a finite set of properties $\{p_e\}$, which can refer to other model elements inside the same input model I. The merge combination operator produces a new output model O that contains for each relationship between e_1 and e_2 in EQ a single model element that has as a set of properties the union of the properties of the related elements, i.e., $\{p_{e1}\}$ and $\{p_{e2}\}$. All model elements in I_1 and I_2 that are not mentioned in EQ are simply copied over into O. See Fig. 3 for an illustration of symmetric merge.

4.2 Examples of structural weavers

4.2.1 Class diagram weaving

The Reusable Aspect Models approach (RAM) [21] defines two composition operators for composing class diagrams. In RAM, the software designer that elaborates a class diagram can incorporate structural elements defined in another class diagram by either *reusing* or *extending* the other class diagram. In the case of reuse, the reused class diagram exposes a set of model elements (classes, operations, parameters) that provide structural properties that are intended to be combined with model elements from the reusing class diagram in a so-called customization interface. The designer needs to map each element from the customization interface to the desired model element in the reusing model. In the case of extension, the extending class diagram can add additional properties to any model element of the extended class diagram, as well as define additional model elements. Here, any model elements with the same signature (e.g., name for classes, name and parameter types for operations, etc.) are combined by default, but the designer can specify additional mappings between elements of the extending model and the extended model, if desired.

A simplified metamodel of RAM Class Diagrams is shown in Fig. 4 (classes highlighted in gray). *RAM Aspect* is the root of the containment hierarchy, and it contains one *ClassDiagram*, which is composed of many *Classifiers*, which in turn contain *Operations* and *Attributes*. For illustration purpose, the *visibility* and *abstract* properties for *Classifier* and *Operation* are shown. Other information, for example, type information, parameters, etc., is omitted for readability reasons. A class diagram can also contain *Associations*, which are linked to two *AssociationEnds* that are contained in the *Classifier* that they belong to.

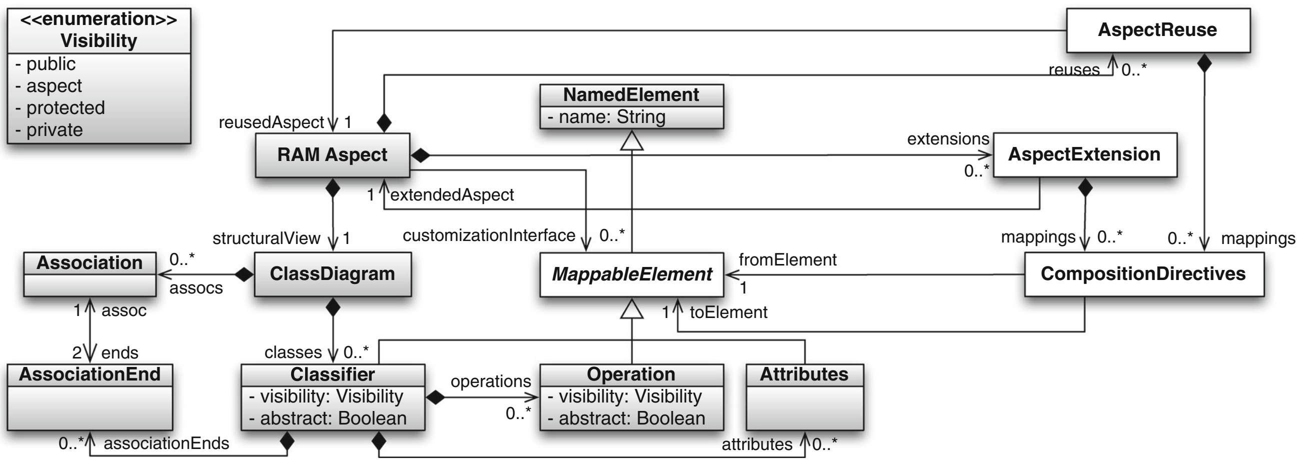


Fig. 4 Parts of the class diagram metamodel of the RAM modeling approach

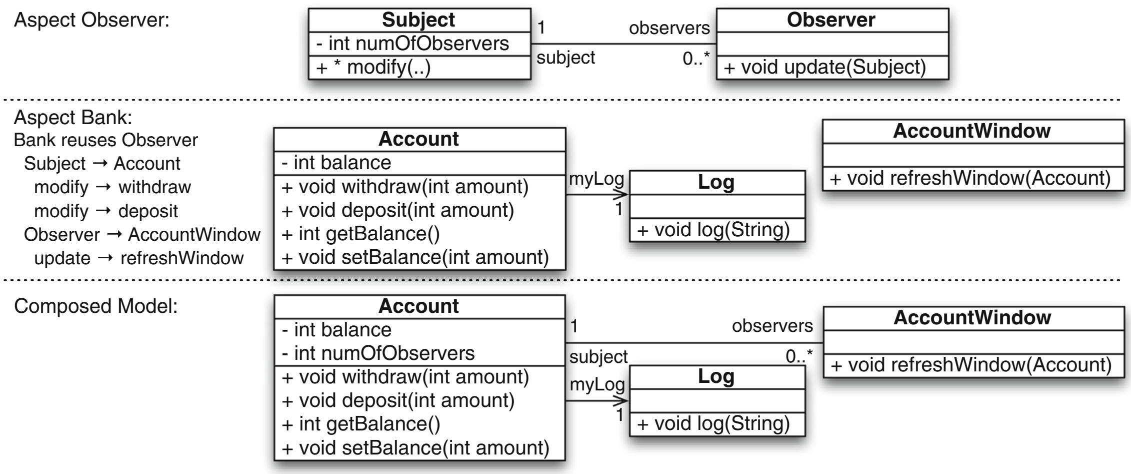


Fig. 5 Example composition of class diagrams in the RAM modeling approach

The classes in white are the ones that are used to specify composition. *AspectReuse* is used to specify a *reuse* composition, whereas *AspectExtension* is used to specify an *extends* composition. In both cases, the `mappings` (instances of *CompositionDirective*) specify which model element from the reused or extended aspect is mapped to which element in the current aspect, i.e., *fromElement* refers to a model element in the reused/extended class diagram, and *toElement* points to a model element in the current class diagram.

Figure 5 shows an example class diagram that models the structure of the *Observer* design pattern [15]. The *Bank* class diagram *reuses* the *Observer* by specifying a *reuse* composition specification that maps `Subject` → `Account`, `modify` → `withdraw/deposit`, `Observer` → `AccountWindow` and `update` → `refreshWindow`.

To produce the composed model, the RAM weaver creates a copy of the *Bank* aspect and then deep-copies or merges the model elements of the *Observer* aspect into this new model according to the *CompositionDirectives*. In other words, it performs the *merge* operation described in Sect. 4.1, where I_1 = *Observer*, I_2 = *Bank*, and EQ = `instances of CompositionDirectives` (see mappings in Fig. 4).

In the case of *reuse* composition, the preprocessing step of the composition operator (see step 1 in Sect. 3) changes the visibilities of the elements in the reused class diagram from public to package. The motivation for this is the information hiding principle [39]: The interface of the reusing class diagram should not expose structural elements of the reused class diagram. There is no need for the matching step, since the composition directives enumerate all elements that are to be combined with our proposed *merge* operator. There is also no need for postprocessing the merged output.

In the case of *extends* composition, the matching step compares the model elements in the two input models to identify element pairs with the same signature. For each pair, a composition directive that maps one to the other is created and added to the *AspectExtension*. The augmented mappings are then passed to the RAM weaver, who as a result merges the model elements with identical signatures according to the semantics of RAM model extension.

4.2.2 UML package merge

Package merge has been introduced in UML2 to improve modularity [38] and is extensively used in the UML specification itself. The package merge composition operator takes as input two class diagrams and extends the first with the second by merging their common classes and deep-copying the other ones. These common classes are identified by their names and types. The merge is done recursively, following the containment links of the models. The output of the merge replaces the first class diagram; hence, the merge is asymmetric, in the sense that only the first model is modified. However, except for the place where the merged model is stored and for conflict resolutions applied during the post-processing step, the technique is symmetric. In other words, the combination step does not depend on the order of the inputs and maps nicely to symmetric merge in the proposed framework in a similar way as explained above for RAM class diagrams.

4.2.3 Kompose

Kompose is a model composition tool [14] implemented in the Kermeta language [20]. It merges two homogeneous models by comparing the signatures of their elements. These signatures can be arbitrarily complex, using the element's name, type, field, or method names and types, and so on. Elements with the same signature are merged, while all other ones are deep-copied. Kompose proposes a system of pre-

and postdirectives to modify the models before and after the merge. The two models are called base model and aspect model, which suggests an asymmetric treatment, yet the tool is symmetric [14]. The composition operator follows the four steps described in Sect. 3, and in particular, includes the symmetric merge combination operator.

4.2.4 Theme/UML

Theme/UML [7] for class diagrams is similar to the aforementioned techniques in that structural model elements are merged symmetrically with each other, in this case based on specified binding relationships for template parameters. The properties of bound elements are merged, while non-template-parameter elements are added to the composed result. The approach of Theme/UML matches the proposed framework in its use of a symmetric merge combination operator.

4.2.5 AoGRL

The Aspect-oriented and Goal-oriented Requirement Language (AoGRL) [34] matches a parameterized goal model fragment (i.e., a pattern) against the base goal model to identify locations where aspectual goal model elements are to be inserted into the base model. While the specification of the composition is asymmetric, the actual operator is symmetric once the insertion locations have been identified, because conceptually pattern elements from the aspect are merged with matched elements from the base. AoGRL therefore adheres to the proposed framework.

4.3 Mathematical properties of merge

The merge combination operator inherits its mathematical properties from the union operator on which it is based. It is therefore commutative [$merge(a, b) = merge(b, a)$] and associative [$merge(a, merge(b, c)) = merge(merge(a, b), c)$].

Nevertheless, the composition operators of the different approaches listed above are not necessarily commutative; i.e., they are not necessarily symmetric. This stems from the fact that the preprocessing, matching, and postprocessing steps of many composition operators perform operations that are not commutative.

For example, in RAM, the *reuse* composition operator changes the visibilities of the elements in the reused class diagram from public to package in the preprocessing step. The motivation for this is the information hiding principle [39]: The interface of the reusing class diagram should not expose structural elements of the reused class diagram. This preprocessing step implies $RAM\_reuse(a, b) \neq RAM\_reuse(b, a)$, because the visibility of the model elements in the composed model would be different.

Conflict resolution also sometimes results in asymmetry, although different approaches handle conflict resolution in different ways. This is due to the fact that what constitutes a conflict depends highly on the modeling language. Conflicts can be resolved during preprocessing or postprocessing (as it is done in Kompose or UML Package Merge, where conflicting elements must be renamed before the symmetric merge happens), but also by choosing different models that are to be composed (as it is done in RAM).

Finally, some composition operators perform in-place update, i.e., they modify one of the input models during composition, which renders them asymmetric. For example, UML package merge composes one model into the other one, or in other words, the composed model replaces one of the inputs. This is orthogonal to the merge operation proposed in our framework. In our framework, after the models are composed, the user of the framework can decide whether to store the resulting model separately, or whether to overwrite one of the input models.

Some AOM approaches, for example, MATA [41], allow an aspect to also *remove* elements from the base model. Since our structural composition is based on *merge*, it supports *additive* composition only. *Removing* elements is not of compositional nature and related to program or model *slicing* [4]. Slicing is a technique used, for example, to extract parts of a system so that it can be reused in a different context, or when using negative variability in software product line development. To support removing of elements, our framework could be combined with a generic slicer such as [4], but this is out of the scope of this paper.

5 Behavioral combination: event scheduling

This section defines event structures and the behavioral *event scheduling* operator of our framework and then shows in detail how it can successfully be applied to compose sequence diagrams, state diagrams, and Aspect-oriented Use Case Maps. Then, further examples of how *event scheduling* can provide composition for other modeling notations are outlined.

5.1 Definition

A behavioral model is a model where its structure (elements and properties) represents behavior, possibly non-deterministic or concurrent. Such behavior can be obtained by applying the operational semantics of the language used to build the behavioral model on the model (i.e., by *executing the model*). Such process is for instance detailed in [26] or [10].

The behavioral composition examples listed in Sect. 3 act on behavioral input models and create a new behavioral model that specifies a particular interleaving of the input models' behaviors. Therefore, even if the composition operators take as input the behavioral models, they actually need to infer an internal representation of the execution of the input models to reason about their interleaving. Independently, this internal representation can be inferred statically (with dedicated analysis) or dynamically (simulation).

To reason on concurrent behaviors, concurrency theory introduced causal (also called true-concurrent) representations. A causal representation captures the concurrency, dependency and alternative relations among actions in a particular behavior. A well-known instance of such representation is *Event Structure* [36,42]. An event structure represents a partial order of events specifying the causality relations as well as alternative relations between actions of concurrent behavior. This fundamental model totally abstracts data and model structure to concentrate on the partial ordering of its actions. This model is not expressive enough to encode (continuous) timed or synchronous behaviors, and there exist several extensions of this model, for example, tagged signals [27] or the time model [3]. In this paper, however, we choose to present the framework based on event structures since it is expressive enough for the examples that follow and allows for simpler explanations. In any case, all the explanations and definitions given in the remainder of the paper are also appropriate for more powerful fundamental models.

Due to its nature, an event structure is a means to apply abstraction to any model by concentrating only on the observable actions in the model. An event structure is independent of the abstraction level since an event can abstract any kind of action (from the entering into a state to the call of an arbitrarily complex action). Due to this independence from the abstraction level, the same model can be represented by different, more or less detailed, event structures. For the same reason, there are usually different models that can be represented in an abstract way by the same event structure.

We believe that the combination operation for two behavioral models A and B can always be reduced to *event scheduling* that operates on the two event structures describing the behavior of A and B. Formally, the event scheduling operator takes as input two event structures es_A and es_B representing the behavior of models A and B, as well as a set of causal relationships between events of different models $CausalR := e_j \rightarrow e_k$. If e_j is an event from es_A, then e_k must be an event from es_B, or vice versa. $e_j \rightarrow e_k$ means that e_k must take place at the same time or after e_j. The result of applying the schedule combination operator is a combined event structure es_C that takes the event relationships $CausalR$ into account. See Fig. 6 for an illustration of event scheduling, where the events and dependencies of

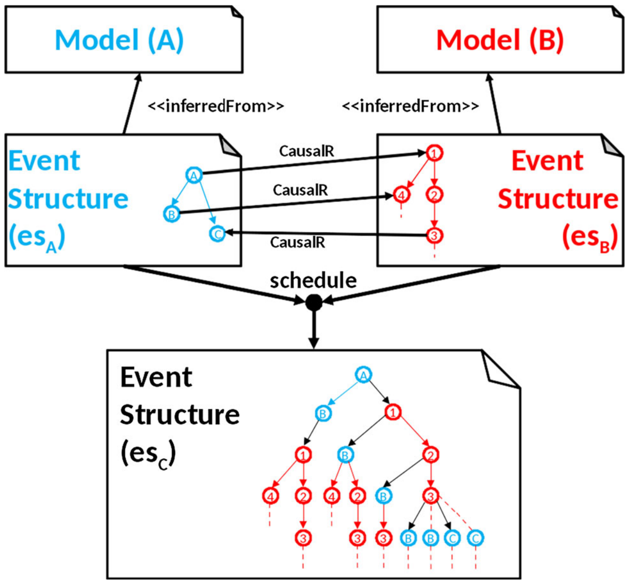


Fig. 6 $schedule(es_A, es_B, CausalR) \Rightarrow es_C$

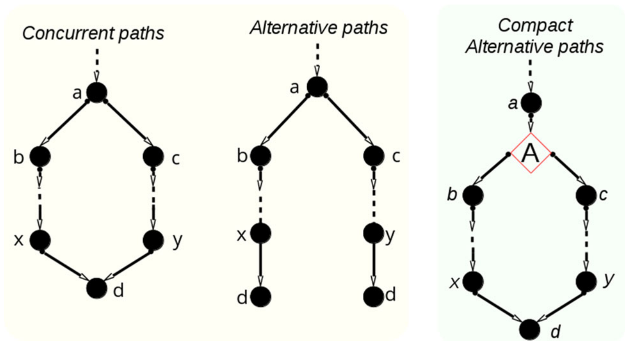


Fig. 7 Usual and compact representation of alternatives

es_A are shown in blue, the events and dependencies of es_B in red, and the causal relationships between events from es_A and es_B are shown with black arrows.

Before continuing, the reader should note that an event structure specifies all possible concurrent or alternative execution paths of the model. In an event structure, two *alternative* execution paths at a specific point in time (i.e., from a specific configuration of the event structure) are usually represented by two causalities to different event occurrences, creating two different paths that do not contain any causality to a common event occurrence (even if from some point they have the same future, see middle of Fig. 7). On the other hand, two *concurrent* execution paths at a specific point in time are represented by two causalities to different event occurrences, creating two different paths that will eventually be causality related to a common event occurrence in the event structure (see left of Fig. 7).

To represent alternative execution paths in a more compact way than duplicating all common futures, we use a dummy node marked 'A' in an event structure. The outgoing causalities of the dummy node denote exclusive execution paths, even if they are eventually causally related to a common event occurrence. The use of the dummy node is illustrated on the right side of Fig. 7. The figure on the left describes a situation where after event a two event sequences $(b..x)$ and $(c..y)$ occur potentially in parallel and then synchronize again for d. On the other hand, the event structure shown in the middle represents a situation where two possible futures exist after the occurrence of a, one starting with b and the other one starting with c. In the case where both futures have a common event sequence at some point in the future, the dummy node allows us to join the two branches at the first common event (d in our case as illustrated by the model on the right) instead of having to show two distinct sub-trees.

5.2 Examples of behavioral weavers

In this section, we illustrate the *event scheduling* operator on three different examples. These examples have been chosen to evaluate the applicability of our approach on different kinds of behavioral models (inspired by the UML behavioral models). The first one is Sequence Diagram, a message-based behavioral language, the second one is State Diagram, a state-based behavioral language, and the third one is AoUCM, a workflow-based behavioral language (as UML activity diagrams). We end the section by briefly explaining how the proposed framework can be applied on existing approaches that are proposing a composition operator.

5.2.1 Sequence diagram composition

RAM allows a software designer to express behavior using sequence diagrams [21]. In sequence diagrams, instances of objects, represented by lifelines, send messages to each other. The important behavioral events are the *message send* and *message receive* events. For each lifeline, a causal ordering is specified for the message reception and message sending events. Between lifelines, another causal ordering specifies that a receive event of a message cannot occur before the corresponding send event has occurred. While sequence diagrams can be used to depict both asynchronous and synchronous sending of messages, they are always used in RAM to specify the interactions in form of *synchronous* operation invocations that happen between objects as a result of an operation call.

To support advanced separation of concerns, RAM allows the modeler to specify object interactions pertaining to different concerns in separate sequence diagrams and use composition to generate the combined behavior when needed. For instance, if the behavior of operations `p` and `q` is described in separate sequence diagrams, and if somewhere within the model of `p` the operation `q` is invoked, then the RAM sequence diagram weaver can combine the two models to produce a new model that depicts the combined flow of execution by "inlining" the communication of `q` within `p` at the right place.

RAM also supports aspect-oriented composition of sequence diagrams, an example of which is shown in Fig. 8. On the top left of the figure, a simple RAM sequence diagram is shown that specifies the behavior of the `transfer` operation of the `Account` class previously shown in Fig. 5. It simply calls the `withdraw` operation of Account `s`, and, in case the withdraw was successful then calls `deposit` on Account `t`. The corresponding event structure $ES_{transfer}$ is depicted underneath the sequence diagram. The first event on the `:Account` lifeline is the reception event of the `transfer` call, which is followed by the send event of the `withdraw` call, followed by the reception event of the return of the `withdraw` call *as well as* the receive event of the call of `withdraw`, etc.

A separate sequence diagram, shown on the top right of Fig. 8, depicts the behavior of `Logging`. It specifies that whenever `transfer` is invoked, followed by a call to `withdraw`, followed by a call to `deposit`, then the remaining behavior of `transfer` is first executed (depicted by a rectangular box containing a $\star$). *After* that, the successful transfer is logged by calling `log` provided by `myLog`. The event structure $ES_{logging}$ corresponding to the `Logging` sequence diagram is shown underneath. Note that the rectangular box with the $\star$ was converted into two events.

To compose the two behaviors, additional causal relationships are added between the events of $ES_{transfer}$ and $ES_{logging}$. They are shown as dotted arrows in Fig. 8. For all events that match in both event structures, a causal dependency from the base model to the aspect model is created, i.e., $\forall e_i \in ES_{transfer}, e_j \in ES_{logging} | match(e_i, e_j) \implies (e_i \rightarrow e_j) \in CausalR$. In this example, this is true for the events $receiveCallTransfer$, $sendCallWithdraw$, $sendCallDeposit$, and $sendRetTransfer$. Since the location of the rectangular box with the $\star$ in the `Logging` sequence diagram determines when the execution of the remaining behavior of `transfer` should occur, two additional causal relationships have to be added to $CausalR$: from the $enterBox$ aspect model toward the first remaining event occurring on the `Bank` lifeline in the transfer model ($enterBox \rightarrow receiveRetDeposit$), as well as from the last event of the execution of `transfer` in the base model toward the $exitBox$ event in the aspect model ($receiveRetDeposit \rightarrow exitBox$).

This simple example illustrates also that our behavioral weaving framework can support semantic weaving. In case the withdraw operation is unsuccessful, the $sendCall$ $Deposit$ event never occurs, and hence the behavior of the logging aspect is never executed.

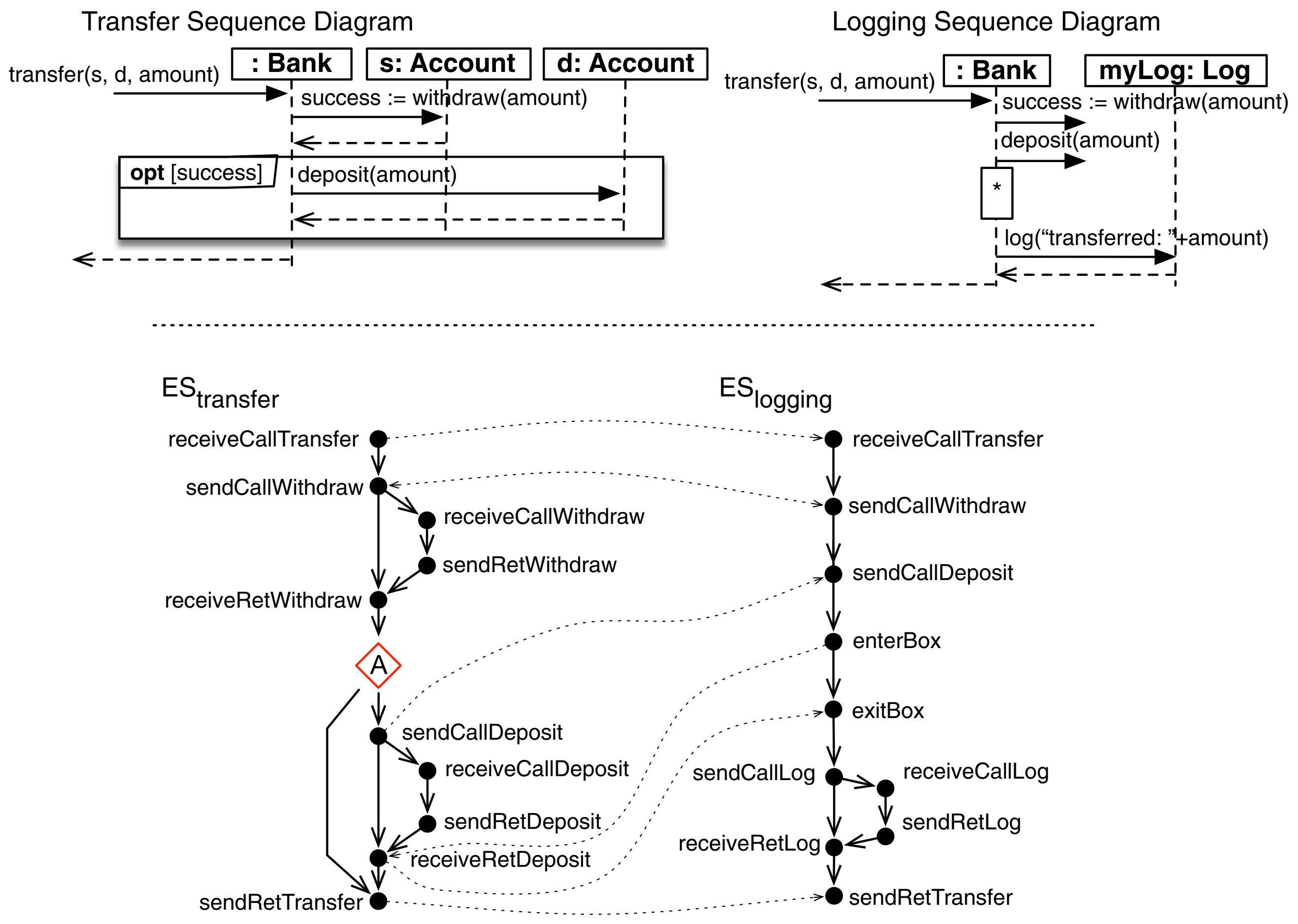


Fig. 8 RAM sequence diagram composition

5.2.2 State diagram composition

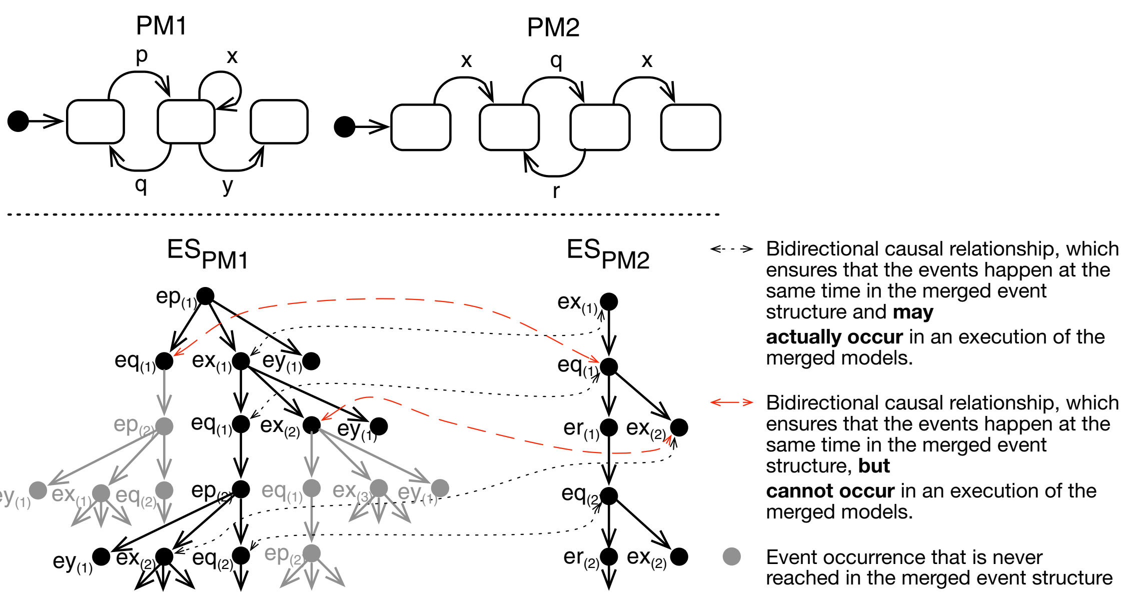
RAM allows a software designer to specify operation invocation protocols for each design class using state views [2], which are a variant of protocol state diagrams called protocol models [30]. A state view comprises a set of states and a set of named transitions that stand for the operations that the class declares. For instance, a state view comprising two states for a `File` class might specify that a file initially starts in a "Closed" state, waiting for a call to the `open` operation, which transitions to the "Opened" state. There, the operations `read` and `write` are available, until the `close` operation is called, which brings the file back to the "Closed" state. In protocol modeling terms, the "Closed" state accepts `open` and rejects `read`, `write`, and `close`, whereas the "Opened" state does the inverse. Operations that are *not* mentioned in a state view are ignored, i.e., no decision on whether to accept or reject them is taken.

A state view is mapped to an event structure by creating an event et for every transition t in the model. The event structure is then given by the tree containing all possible event occurrence scenarios acceptable by the state view starting from the initial state. For state views with cycles, the resulting event structure is usually infinite.

Figure 9 illustrates the transformation from state view to event structure on an example. In the state view $PM1$, the first transition that can occur is p. This is why the event structure ES_{PM1} has the event $ep_{(1)}$ at the root.[2] After p, transition q, x, or y can be taken in the state view. ES_{PM1} therefore contains $eq_{(1)}$, $ex_{(1)}$ and $ey_{(1)}$, and $ep_{(1)} \rightarrow eq_{(1)}$, $ep_{(1)} \rightarrow ex_{(1)}$ and $ep_{(1)} \rightarrow ey_{(1)}$. If transition q is taken, then p can be taken again. Therefore, ES_{PM1} contains another occurrence of transition p, namely the event occurrence $ep_{(2)}$ and $eq_{(1)} \rightarrow ep_{(2)}$. The resulting infinite tree represents all possible executions of the state view.

When the structural weaver of RAM merges two classes, resulting in the union of the operations of the classes as explained in Sect. 4.2, the RAM protocol weaver also needs to combine the two state views associated with the classes to yield the state view that specifies how to correctly use the new merged class.

[2] We are using the subscript $_{(i)}$ to emphasize that the dots in the event structure represent event occurrences and not event types.



<-·-> Bidirectional causal relationship, which ensures that the events happen at the same time in the merged event structure and **may actually occur** in an execution of the merged models.

<—> Bidirectional causal relationship, which ensures that the events happen at the same time in the merged event structure, **but cannot occur** in an execution of the merged models.

● Event occurrence that is never reached in the merged event structure

Fig. 9 RAM state diagram composition

The rules for combining the two state views is equivalent to the CSP parallel composition operator typically denoted ‖. It specifies that for an operation to be accepted by $I_1 \parallel I_2$, both I_1 and I_2 must either accept or ignore the operation. In other words, if either I_1 or I_2 rejects the operation, then $I_1 \parallel I_2$ also rejects the operation.

In RAM, to combine two event structures representing two state views, it suffices to add causal relationships among the event occurrences of the sets of transitions T_{PM1} and T_{PM2} that match. More precisely, $\forall t1 \in T_{PM1}, \forall t2 \in T_{PM2} | match(t1, t2) \implies (\forall et1_i \in ES_{PM1}, et2_i \in ES_{PM2}, ((et1_i \rightarrow et2_i) \in CausalR \wedge (et2_i \rightarrow et1_i) \in CausalR)$ with et_i defined as the ith event occurrence of the transition t). The composition of the state views $PM1$ and $PM2$ is shown at the bottom of Fig. 9 with the additional arrows representing some of the introduced causal relationships between the event structures ES_{PM1} and ES_{PM2}. Whenever there are causal dependencies in both directions, it is ensured that they have to happen *simultaneously*.

5.2.3 Aspect-oriented Use Case Map composition

Aspect-oriented Use Case Maps (AoUCM) [34] is a scenario/workflow notation that employs a pattern-based approach to identify locations in the base model where aspectual behavior is to be inserted. Since AoUCM models capture causal relationships, they are straightforwardly transformed into event structures by denoting each AoUCM model element m as an event em (except for those AoUCM model elements that represent purely control flow information and are hence directly mapped onto causal relations instead of events).

The AoUCM approach features an enhanced matching algorithm that takes semantic equivalences in the UCM notation into account and allows interleaving of scenarios. However, once the insertion location has been determined by the sophisticated matching algorithm, the actual composition of the aspectual and base behavior is always accomplished by the insertion of an aspect marker, which acts as a reference in the base model to the aspectual behavior that needs to be inserted.

The example AoUCM model [32] was selected, because it features a more complex composition based on interleaved scenarios. The base AoUCM model on the left of Fig. 10 shows the scenario for ordering a video online, comprised of the steps *selectMovie*, *processOrder*, *payForMovie*, and *sendMovie*. The aspectual model in the middle defines the movie points scenario, which is interleaved with the base behavior with the help of the diamond-shaped pointcut stubs. Each pointcut stub represents a pattern that is matched against the base model, for example, the *processOrder* pointcut stub is matched against the *processOrder* step in the base model (the actual definition of the pattern is not shown in the figure). The movie points scenario requires a membership form to be filled out *before* processing an order, allows movie points to be redeemed *instead of* paying for the movie, and enables earning of movie points *after* sending a movie if the transaction was not a redemption.

At the bottom of Fig. 10, the combined event structures of the AoUCM base and aspect model are shown. The event structure of the base model is represented on the left side, the one of the aspect model on the right side, and the causalities required by the combination of both models are shown in the middle. First of all, each pattern match is synchronized, for

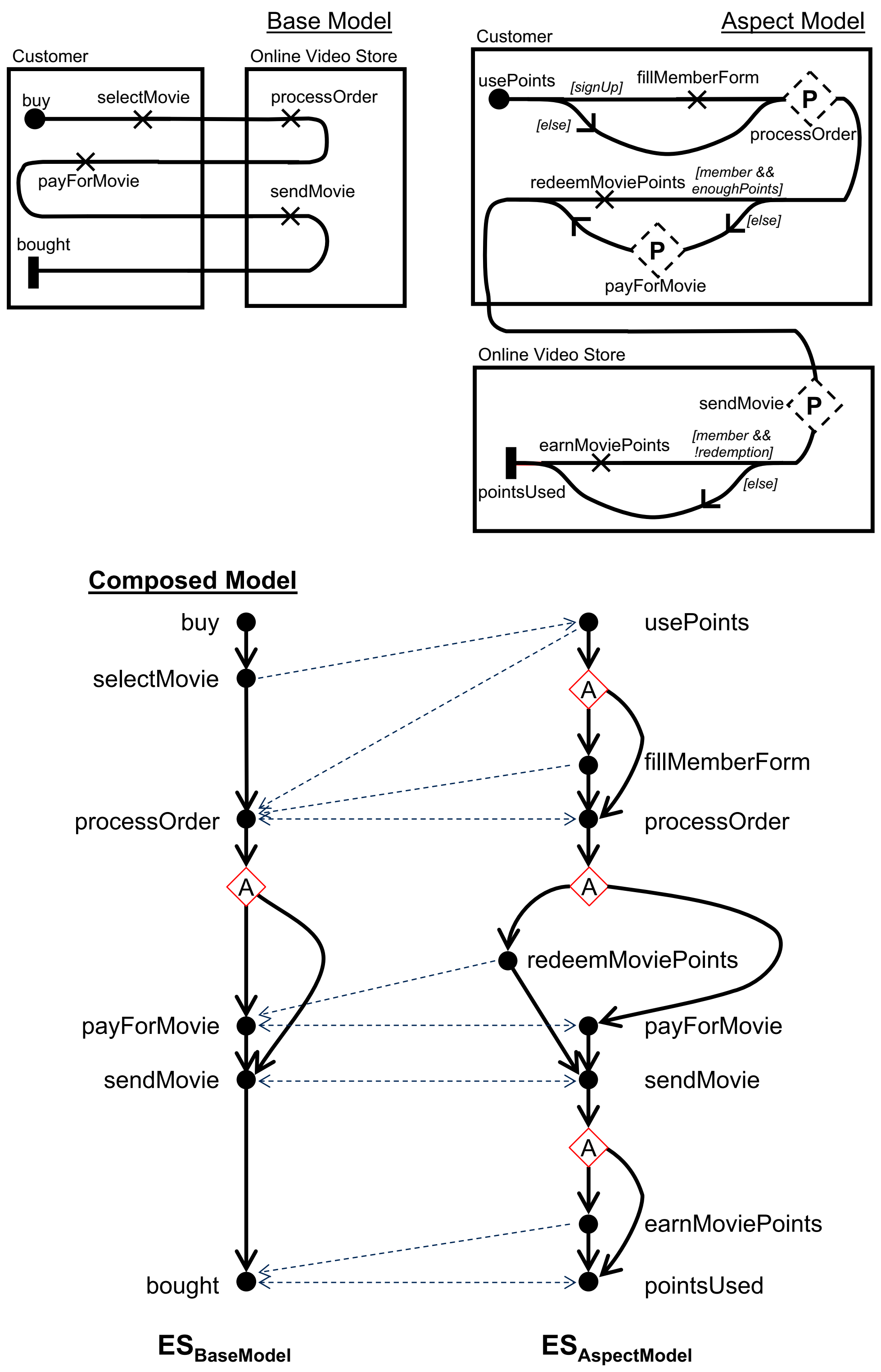


Fig. 10 AoUCM scenario/workflow composition

example, the *processOrder* event from the base model is synchronized with the *processOrder* event from the aspect model. This is equivalent to the synchronization of events performed for RAM sequence diagrams. Then, additional causal relationships are expressed in the aspectual event structure for segments created by the synchronized elements. Three types of segments exist. A segment may (a) start at a start point and end at a synchronized element (e.g., from *usePoints* to *processOrder*), (b) start and end at a synchronized element (e.g., from *processOrder* to *sendMovie*), or (c) start at a synchronized element and end at an end point (e.g., from $sendMovie$ to $pointsUsed$).

The first element in a segment has an additional causal relation from a base model element (i.e., $usePoints$) or is a synchronized element (i.e., $processOrder$, $sendMovie$). Additional causal relations never have to be specified for already synchronized elements. If the first element in a segment is not a synchronized element, then the causal relation exists from the base model element immediately preceeding the base model element synchronized with the aspectual element at the end of the segment (i.e., $selectMovie \rightarrow usePoints$).

If the segment ends with a synchronized element, then all elements in the segment have a causal relation to the base element synchronized with the end of the segment (e.g., $usePoints \rightarrow ProcessOrder$, $fillMemberForm \rightarrow ProcessOrder$, and $redeemMoviePoints \rightarrow sendMovie$). However, if the segment ends in an end point, then all elements have a causal relation to the base element immediately following the base model element synchronized with the aspectual element at the start of the segment (e.g., $earnMoviePoints \rightarrow bought$, $pointsUsed \rightarrow bought$).

AoUCM allows base behavior to be replaced by aspectual behavior (e.g., *redeemMoviePoints* may replace *payForMovie*). Therefore, the event structure of the base model needs to be augmented with alternatives to allow base elements to be skipped (see the 'A' node in the event structure of the base model).

More precisely, during the translation from a model to its event structure, the expressiveness of the weaver must be taken into account and made explicit in the resulting event structure. Because the AoUCM weaver allows for removing (i.e., skipping) some actions from the original model, the translation of any AoUCM model must contain, between each event occurrence, an alternative allowing to skip one or more event occurrences. This does not only apply to AoUCM but to any weaver of a behavioral model that allows replacement (e.g., it is also necessary for RAM sequence diagrams to support around advice). While these additional *skip causalities* may appear as a limitation of our proposed framework, we believe, on the contrary, that the skip causalities make explicit something often hidden deep inside the core of a weaver and which is now made visible to any tool in a generic way (e.g., to a generic analysis tool). Skip causalities could for instance be used by a model checker to conclude that the base model respects a property, but that a specific aspect could lead to a property violation. Even more interesting, considering property violations, the concerned alternative could be removed, effectively reducing the expressiveness of the weaver to ensure that no aspect can violate a specific property. In the composed model in Fig. 10, all possible skip causalities are not shown to simplify the representation. The shown skip causality (i.e., the 'A' node) in the event structure of the base model corresponds to the one from the aspect model that resides in the same segment and is the only one that is relevant for the composed model in this example.

5.2.4 ADORE

ADORE is a tool for service orchestration using PROLOG. It allows the scheduling of partial orchestrations (fragments) in a main orchestration using a set of user-defined relationships [31]. A fragment essentially represents partially ordered service calls and ends, which can be represented through an event structure. The relationships between fragments which describe the orchestration correspond to the asymmetric combination operation *event scheduling* of our framework, which acts thanks to event relationships between the event structures inferred from the input fragments.

5.2.5 TreMer+

In [35], Nejati et. al. propose an approach, implemented in TreMer+, to merge statechart diagrams while preserving their semantics by ensuring bisimulation. Thus, the operator is not only structural, but also ensures a particular relationship between the behaviors that represent the statechart diagrams. A statechart diagram represents a partial order of events, naturally represented by an event structure. Concurrent actions, for example, regions in statechart diagrams, can be represented by concurrent branches in the event structure, and synchronizations between concurrent machines can be ensured by causalities in the event structure Then, the combination operation infers the required event relationships ($CausalR$) between the two input event structures to ensure a bissimilar combined event structure (based on a powerful matching step).

5.2.6 Theme/UML

Theme/UML [7] for sequence diagrams is very similar to RAM in that the combined flow of execution for two sequence diagrams also involves proper "inlining". Therefore, the behavioral composition can be expressed the same way as for RAM.

5.2.7 HiLa

The High-Level Aspects for UML State Machines (HiLA) approach [43] targets the modeling of use case scenarios with UML state machines, enhanced with aspect-oriented modeling features. To derive the corresponding event structure, inputs coming from the environment are mapped to events. Synchronization between events in the base model and the aspect model can be achieved with two inverse causal relationships. HiLa also provides what is called dynamic aspects

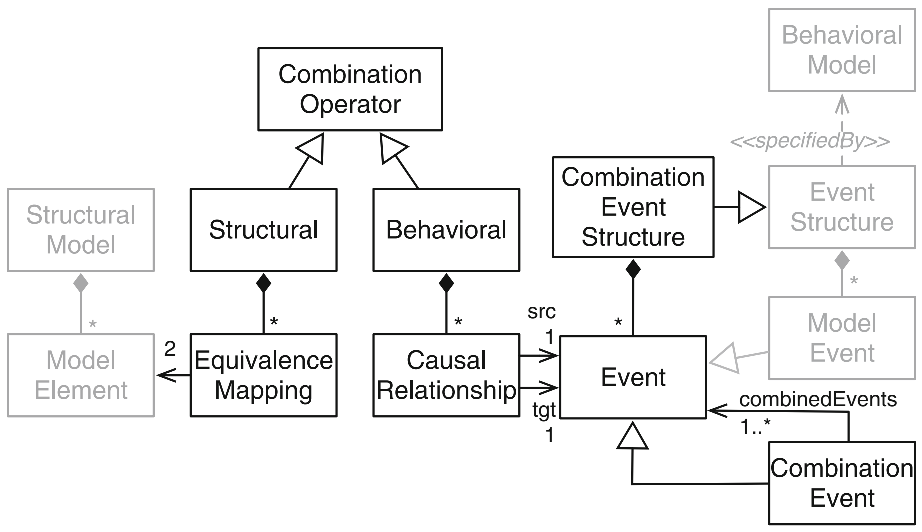


Fig. 11 Foundational primitives for model composition

that can compose state machines based on the execution trace. For instance, a so-called «history» property allows a modeler to specify a constraint that checks, for instance, that a certain state was entered more than *n* times in order to trigger behavior expressed in the aspect state machine. To achieve this composition using event structures, a causal relationship enabling the first event of the aspect state machine must be scheduled in all event orders of the base machine that visit the state more than *n* times.

6 Discussion

6.1 Conceptual model

In Sects. 4 and 5, we gave evidence of foundational primitives for structural and behavioral model combination that can handle many composition operators developed for a variety of modeling languages. In Fig. 11, we summarize these foundations as an early DSL that could be used for defining reusable model composition operators in the future.

A `Combination Operator` can be either a `Structural` combination operator between `Model Elements` of `Structural Models`, or a `Behavioral` combination operator between `Events` of the `Combination Event Structure`. The combination event structure possibly extends the `Event Structure` inferred from the execution of (i.e., `specifiedBy`) `BehavioralModels`, with additional `Combination Events` that represent a combination of events for a particular purpose of the combination operator.

A structural combination operator is composed of `Equivalence Mappings` between two unordered model elements, while a behavioral combination operator is composed of `CausalRelationship` between a source event and a target event. Of course this event relationship can be completed by more specific ones (e.g., mutual exclusion, temporal relationships).

The vision we have proposed in this paper takes the form of a complete framework to support rigorous and efficient engineering of homogeneous and potentially heterogeneous model composition operators. If successful, the proposed framework would provide a reusable building block for (modeling) language design efforts and streamline the creation and application of domain-specific languages with support for separation of concerns in the context of MDE.

6.2 Research challenges

As outlined in Sect. 3, the execution of composition operators can be decomposed into four steps: preprocessing, matching, combining, and postprocessing. The discussion of this paper has focused mostly on the most important step of this process, step 3, which addresses the actual combination of models. We argued that it can be provided in a generic way by a symmetric *merge* operator for structural models and an asymmetric *event scheduling* operator for behavioral models after they have been transformed into event structures. In order to realize the entire proposed vision, several research challenges remain:

– C1: Ensure that all behavior models (that we want to be able to compose in the context of MDE) can be mapped to corresponding event structures.
– C2: Establish proof that our hypothesis (structural combination = merge, behavioral combination = event scheduling) is correct.
– C3: Render all steps of the composition operators as generic / reusable as possible.
– C4: Provide efficient automation where possible.

– C5: Operationalize the proposed composition operator framework by integrating it with a language engineering workbench.

6.3 Proposed work items

The challenges to which the following six work items contribute are shown in parentheses in the header of each individual work item.

6.3.1 W1: mapping to event structures (C1)

We already conducted experiments that represent other behavioral models with specific kind of events structures [11,12]. To contribute further to C1, additional behavioral modeling languages should be mapped onto event structures.

6.3.2 W2: automate mapping to event structures (C4)

To contribute to C4, the application of the mapping from individual models to event structures needs to be automated by defining and implementing concrete model transformations. In this context, we have already published algorithms that create a symbolic event structure that encodes all the possible event structures of a behavioral model in [17,18,28]. Finally, to avoid constructing the symbolic event structure manually, we used a specification of the language execution semantics suitable to generate the symbolic event structure of any model conforming to the language [9,10].

6.3.3 W3: implementing composition operators (C2)

With initial tool support in place, additional evidence to support C2 needs to be provided by implementing the homogeneous composition operators surveyed in this paper according to the thoughts outlined in Sects. 4 and 5 with our framework. For heterogeneous composition, we have previously experimented with the coordinated execution of event structures in [16]. Vara Larson et al. [24] use a symbolic representation of event structures to conjointly execute different models according to a specific coordination. Most recently, we also proposed the coordination of heterogeneous models based on rules expressed at the language semantics level in [25].

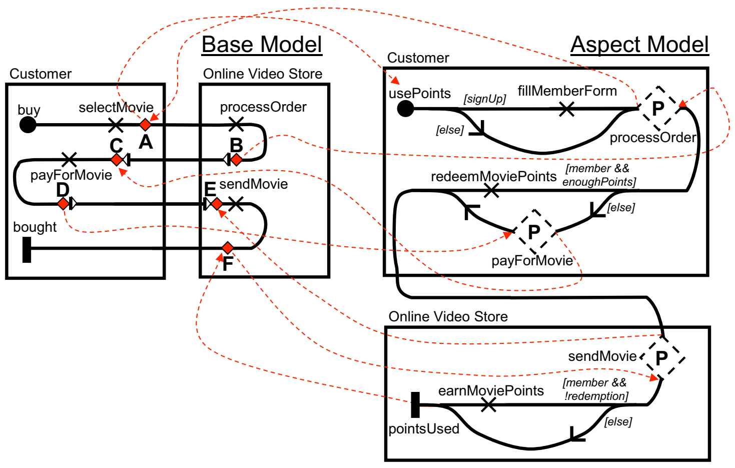
6.3.4 W4: toward generic behavioral matching (C3)

While for all the behavioral model composition operators surveyed in this paper the matching step is notation specific, we hypothesize that it does not need to be. In fact, instead of transforming a behavioral model into an event structure after the matching (step 2) and before combining (step 3), the transformation could be performed after the notation-specific preprocessing (step 1). This would allow exploring the possibility of defining a generic behavioral matcher that operates solely on event structures. Intuitively, both common input options for matching mechanisms—explicit bindings or patterns—can typically be translated into event structures following mapping rules similar to the ones used to map the input models to event structures themselves. In this case, a generic event pattern matcher would be able to identify any locations of interest in the behavior, provided that all potential matching points of the notation (i.e., the join points in aspect-oriented terms) are represented in the form of events in the event structure. Since event structures represent the execution of behavior, an additional advantage of matching at this level is that patterns that occur in the execution can be detected (i.e., the semantic interpretation of a model), as opposed to detecting patterns in the model itself (i.e., the abstract syntax of the model). For instance, such a matcher would be very convenient to match patterns that are known to require loop unrolling in the context of sequence diagram weaving as described in [22] or an understanding of the semantic equivalences of hierarchical decomposition as described in [34].

Providing generic matching capabilities in the context of our proposed framework would significantly contribute to C3.

6.3.5 W5: efficient creation of the combined event structure (C4, C5)

To further contribute to C4 and to the operationalization of the framework as stated for C5, combined event structures need to be created efficiently. In the case where the event structure obtained from a behavioral model is finite, for example, for Figs. 8 and 10, the combined event structure is also finite and can easily be created offline (i.e., not at runtime). A first work item concerns the creation of the combined event structure for behavioral models with infinite event structure (e.g., the one of Fig. 9). In this case, it is not possible to create the infinite event structure offline and consequently not possible to create the combined event structure "statically." However, it is possible to provide a framework to coordinate the execution of the base and the aspect model executions. It is then necessary to monitor the execution of each model and to set up a *runtime matcher* that triggers the creation of $EventRelationship$ elements at runtime. The simulation framework must then be able to implement such new relations on the fly. Using a runtime matcher can be costly in time so that a second work item concerns the finite encoding of infinite event structure. While this has been done earlier in [17,18,28], their encoding was very low level so that it is not well adapted for composition reasoning. Our current idea is to investigate a simple 'folding' node (a kind of parametrized jump) allowing finite representation of infinite event struc-



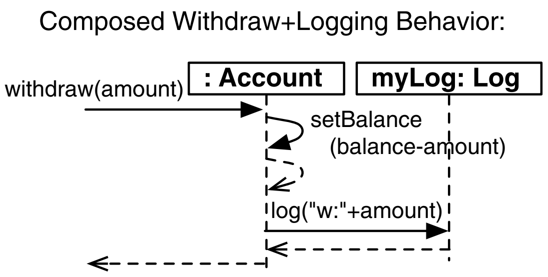

Composed Withdraw+Logging Behavior:
withdraw(amount)
: Account
myLog: Log
setBalance (balance-amount)
log("w:"+amount)

Fig. 12 Visualization of composed model with tags/markers (AoUCM) or inline (RAM sequence diagrams)

ture, while supporting generic composition. The final goal is to propose a generic, static model weaver supporting finite and infinite event structures with or without loop unrolling and other semantic-based compositions.

6.3.6 W6: visualizing the composed behavior (C5)

In addition, to be fully equivalent to existing behavioral composition operators, it does not always suffice to create a composed event structure that exhibits the correct composed behavior. To contribute to the operationalization of the framework as stated for C5, it is often important to be able to visualize the composed behavior in the original modeling formalism. The visualization takes place after the four steps discussed in this paper. A bidirectional mapping from a modeling language to event structures and back would allow the causal relations between the base and the aspect model to be indicated with tags or markers. An examination of the causal relations between the base and aspect model identifies the locations in the event structure, where aspectual behavior is added. This can then be translated into a location in the original modeling formalism with the help of the bidirectional mappings. The composed model could then be indicated by showing a visual element either before, at, or after the location in the original modeling formalism.

Consider, for example, the case of AoUCM's weaver [32,34], which results in the placement of aspect markers in the base model as shown in Fig. 12. The locations where aspect markers need to be inserted in the base model are identified by the synchronized events in the event structures of the base model and their bidirectional mappings back to the source base model. The type of aspect marker that needs to be inserted depends on the type of aforementioned segments in the event structure of the aspect model. A segment with a start point or end point results in an aspect marker with an outgoing link to the aspect model and an incoming link from the aspect model (see, e.g., aspect marker A and F in Fig. 12). For segments where both the start and the end are synchronized elements, aspect markers with only an outgoing link (e.g., B and D) or only an incoming link (e.g., C and E) are inserted into the base model depending on whether the aspect marker corresponds to the start or end of the segment, respectively.

Similarly, a weaver for RAM sequence diagrams [21] that is based on their event structures would identify the location of inserted behavior by the location of the causal relations to *enterBox* and *exitBox* and the existence of aspectual behavior specified before or after these two relations, for example, after in the case of the sequence diagram in Fig. 8. Consequently, behavior needs to be inserted into the identified base model location based on the bidirectional mappings from the event structure of the aspectual model to the source model. For notations such as sequence diagrams that can be lay outed automatically, it is then even possible to show the inserted behavior inline in the base model as shown in Fig. 12.

7 Related work: general purpose weavers

This section discusses the shortcomings of existing tools and frameworks which may be used to combine two models with each other.

7.1 GeKo

GeKo is a generic, extensible model weaver that can compose any models that conform to a common metamodel [23]. It takes as parameters a base model, a pointcut model (which can specify a pattern), and an advice model and replaces all instances of the pointcut model that are found in the base model with the advice model. The mappings between base and advice are inferred by the weaver by comparing model element properties. The combination of models that is done for each pointcut match is equivalent to our merge, i.e., it operates on the structure of the model only. As a result, GeKo, although applicable to behavioral models, cannot perform semantic-based combination.

7.2 MATA

MATA is similar to GeKo in that it can compose any models conforming to a common metamodel, but uses graph transformations to do so [41]. Similarly to GeKo, MATA operates on model structure and cannot perform semantic-based combination, which is possible with event structures.

7.3 ModMap

ModMap is a mapping language to express bidirectional translation between models conforming to object-oriented metamodels [8]. The mapping language allows for creating relationships between any metamodel element such as classes, attributes, or relations between classes. The interpretation mechanism is achieved through the assignment of a strategy to a given mapping. Strategies in the ModMap language are operations which allow the alignment of the two models involved. These operations are either predefined (rename an element, concatenate one or several elements with strings, add or remove an element from a collection) or defined by the user with a provided action language. While ModMap offers one generic merge algorithm, there is no consideration of behavioral semantics.

7.4 Event-based modularization

In [5], the authors present how an aspect-oriented approach can be augmented with an explicit specification of events as a way to express pointcuts. They show how such pointcut events can be combined to express conditions on the weaving of an advice. The authors propose few generic operators between events, but mainly focus on high-level operators for decomposition.

In contrast to existing general purpose weavers, our composition framework addresses structural models as well as the semantics of behavioral models, which is crucial as complex systems always have to be defined and evolve along both structural and behavioral dimensions.

8 Conclusion

The growing use of domain-specific languages and the need to combine instances of such languages leads to a continued demand for customized composition operators. This paper discusses a unifying framework that reduces all structural composition operators to structural merging, and all composition operators acting on discrete behaviors to event scheduling. The framework aims to support the definition and reuse of composition operators and avoid having to define them from scratch for each use. Based on a discussion of the properties of structural and behavioral composition operators and an analysis of how existing model composition operators can be mapped onto our proposed framework, we introduce a conceptual model of the framework and enumerate a set of research challenges that need to be addressed to realize the proposed framework and support rigorous and efficient engineering of model composition operators for homogeneous and eventually heterogeneous modeling languages. In future work, we will address the work items related to the five research challenges discussed in Sect. 6. Furthermore, we will investigate how heterogeneous composition operators can be supported by our proposed framework.

References

1. Aspect-Oriented Modeling Workshop Series. http://www.aspect-modeling.org/
2. Al Abed, W., Schöttle, M., Ayed, A., Kienzle, J.: Concern-oriented behaviour modelling with sequence diagrams and protocol models. In: Behavior Modeling—Foundations and Applications, vol. 6368 of *LNCS*. Springer, Berlin (2015)
3. André, C., Mallet, F., De Simone, R.: Modeling time(s). In: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems, MODELS'07, pp. 559–573, Springer, Berlin (2007)
4. Blouin, A., Combemale, B., Baudry, B., Beaudoux, O.: Kompren: modeling and generating model slicers. Softw. Syst. Model. **14**(1), 321–337 (2015)
5. Bockisch, C., Malakuti, S., Akşit, M., Katz, S.: Making aspects natural: events and composition. In: 10th International Conference on Aspect-Oriented Software Development (AOSD '11). ACM (2011)
6. Bull, C., Whittle, J.: Supporting reflective practice in software engineering education through a studio-based approach. IEEE Softw. **31**(4), 44–50 (2014)
7. Clarke, S., Walker, R.J.: Generic aspect-oriented design with Theme/UML. In: Filman, R.E., Elrad, T., Clarke, S., Aksit, A. (eds.) Aspect-Oriented Software Development, pp. 425–458. Addison-Wesley, Boston (2005)
8. Clavreul, M., Barais, O., Jézéquel, J.-M.: Integrating legacy systems with MDE. In: 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), pp. 69–78. ACM (2010)
9. Combemale, B., Deantoni, J., Larsen, M.V., Mallet, F., Barais, O., Baudry, B., France, R.: Reifying concurrency for executable metamodeling. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds) SLE–6th International Conference on Software Language Engineering, vol. 8225, pp. 365–384, Indianapolis. Springer (2013)
10. Deantoni, J., Diallo, P.I., Teodorov, C., Champeau, J., Combemale, B.: Towards a meta-language for the concurrency concern in DSLs. In: Design, Automation and Test in Europe Conference and Exhibition (DATE), Grenoble, France (2015)
11. Deantoni, J., Mallet, F.: TimeSquare: treat your models with logical time. In Furia, S.N.C.A. (ed.) TOOLS—50th International Conference on Objects, Models, Components, Patterns—2012, vol. 7304, pp. 34–41, Prague, Czech Republic. Czech Technical University in Prague, in co-operation with ETH Zurich, Springer (2012)

12. Deantoni, J., Mallet, F., Thomas, F., Reydet, G., Babau, J.-P., Mraidha, C., Gauthier, L., Rioux, L., Sordon, N.: RT-simex: retro-analysis of execution traces. In: Sullivan, K.J., Roman, G.-C. (eds.) SIGSOFT FSE, vol. ISBN 978-1-60558-791-2 of Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 377–378, Santa Fe, United States (2010)
13. Dijkstra, E.W.: A Discipline of Programming, vol. 1. Prentice-Hall, Englewood Cliffs (1976)
14. France, R., Fleurey, F., Reddy, R., Baudry, B., Ghosh, S.: Providing support for model composition in metamodels. In EDOC, pp. 253–264 (2007)
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison Wesley, Reading (1995)
16. Garcés, K., Deantoni, J., Mallet, F.: A model-based approach for reconciliation of polychronous execution traces. In: SEAA 2011—37th EUROMICRO Conference on Software Engineering and Advanced Applications, Oulu, Finland, IEEE (2011)
17. Glitia, C., Deantoni, J., Mallet, F.: Logical Time @ Work: capturing data dependencies and platform constraints. In: Kaźmierski, T.J.J., Morawiec, A. (eds) System Specification and Design Languages, vol. 106 of Lecture Notes in Electrical Engineering, pp. 223–238. Springer, New York (2012)
18. Goknil, A., Deantoni, J., Peraldi-Frati, M.-A., Mallet, F.: Tool support for the analysis of TADL2 timing constraints using TimeSquare. In: ICECCS'2013—18th International Conference on Engineering of Complex Computer Systems, Singapore, Singapore, (2013)
19. Hölzl, M., Knapp, A., Zhang, G.: Modeling the Car Crash Crisis Management System Using HiLA, pp. 234–271. Springer, Berlin (2010)
20. Jézéquel, J.-M., Combemale, B., Barais, O., Monperrus, M., Fouquet, F.: Mashup of metalanguages and its implementation in the Kermeta language workbench. Softw. Syst. Model. **14**, 905–920 (2013)
21. Kienzle, J., Al Abed, W., Klein, J.: Aspect-oriented multi-view modeling. In: 8th International Conference on Aspect-Oriented Software Development (AOSD'09), pp. 87–98. ACM Press (2009)
22. Klein, J., Hélouet, L., Jézéquel, J.-M.: Semantic-based weaving of scenarios. In: AOSD, pp. 27–38. ACM Press (2006)
23. Kramer, M.E., Klein, J., Steel, J.R.H., Morin. B., Kienzle, J., Barais, O., Jézéquel,J.-M.: Achieving practical genericity in model weaving through extensibility. In: 6th International Conference on Model Transformation (ICMT'13), vol. 7909 of *LNCS*, pp. 108–124. Springer, Berlin (2013)
24. Larsen, M.V., Goknil, A.: Railroad crossing heterogeneous model. In: GEMOC workshop 2013—International Workshop on The Globalization of Modeling Languages, Miami, Florida, USA (2013)
25. Larsen, M.E.V., Deantoni, J., Combemale, B., Mallet, F.: A behavioral coordination operator language (BCOoL). In: Lethbridge, T., Cabot, J., Egyed, A. (eds.) International Conference on Model Driven Engineering Languages and Systems (MODELS), vol. 18, pp. 462, Ottawa, Canada, September 2015. ACM (to be published in the Proceedings of the Models 2015 Conference)
26. Latombe, F., Crégut. X., Combemale, B., Deantoni, J., Pantel, M.: Weaving concurrency in executable domain-specific modeling languages. In: 8th ACM SIGPLAN International Conference on Software Language Engineering (SLE), Pittsburg. ACM (2015)
27. Lee, E.A., Sangiovanni-Vincentelli, A.: The tagged signal model-a preliminary version of a denotational framework for comparing models of computation. Memo. UCB/ERL M **96**, 71 (1996)
28. Mallet, F., Deantoni, J., André, C., De Simone, R.: The clock constraint specification language for building timed causality models. Innov. Syst. Softw. Eng. **6**(1–2), 99–106 (2010)
29. Marchand. J., Combemale, B., Baudry, B.: A categorical model of model merging and weaving. In: 4th International Workshop on Modelling in Software Engineering (MiSE 2012). IEEE (2012)
30. McNeile, A., Simons, N.: Protocol modelling: a modelling approach that supports reusable behavioural abstractions. SoSyM **5**(1), 91–107 (2006)
31. Mosser. S., Blay-Fornarino. M., France. R.: Workflow design using fragment composition. In: TAOSD VII, vol. 6210, pp. 200–233 (2010)
32. Mussbacher. G.: Aspect-Oriented User Requirements Notation. Ph.D. thesis, University of Ottawa, Canada (2010)
33. Mussbacher, G., Alam, O., Alhaj, M., Ali, S., Amálio, N., Barn. B., Bræk, R., Clark, T., Combemale. B., Cysneiros, L.M., Fatima, U., France, R., Georg, G., Horkoff, J., Kienzle, J., Leite, J.C., Lethbridge, T.C., Luckey, M., Moreira, A., Mutz, F., Padua, A., Oliveira, A., Petriu, D.C., Schöttle, M., Troup, L., Werneck, V.M.B.: Assessing composition in modeling approaches. In: Workshop CMA'12. ACM (2012)
34. Mussbacher. G., Amyot. D., Whittle, J.: Composing goal and scenario models with the aspect-oriented user requirements notation based on syntax and semantics. In: Aspect-Oriented Requirements Engineering. Springer, Berlin (2013)
35. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and merging of statecharts specifications. In: ICSE (2007)
36. Nielsen, M., Plotkin, G., Winskel, G.: Petri nets, event structures and domains. In: Semantics of concurrent computation, vol. 70 of LNCS. Springer, Berlin (1979)
37. Object Management Group.: Unified Modeling Language (v2.5.0) (2015)
38. OMG.: Uml infrastructure specification v2.4 (2010)
39. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. Assoc. Comput. Mach. **15**(12), 1053–1058 (1972)
40. Schmidt, D.C.: Model-driven engineering. IEEE Comput. **39**, 41–47 (2006)
41. Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A., Araújo, J.: MATA: a unified approach for composing UML aspect models based on graph transformation. In: Transactions on Aspect-Oriented Software Development VI, vol. 5560 of LNCS, pp. 191–237. Springer, Berlin (2009)
42. Winskel, G.: Event structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency. Springer, New York (1987)
43. Zhang, G., Hölzl, M.M.: HiLA: high-level aspects for UML state machines. In: MoDELS Workshops, vol. 6002 of LNCS, pp. 104–118. Springer (2009)

Jörg Kienzle is an associate professor at the School of Computer Science at McGill University in Montreal, Canada. He holds a Ph.D. and engineering diploma from the Swiss Federal Institute of Technology in Lausanne (EPFL). His current research interests include model-driven engineering, concern-oriented software development, reuse of models, software development methods in general, aspect-orientation, distributed systems and fault tolerance. He is actively involved in the MODELS and Modularity:AOSD communities.

Gunter Mussbacher is an Assistant Professor in the Department of Electrical and Computer Engineering at McGill University, with 100+ publications related to model-driven requirements engineering, modularity in modeling, concern-driven development and reuse, and sustainability engineering. He has co-edited with Daniel Amyot all versions of the User Requirements Notation (URN), an international requirements engineering standard published by the International Telecommunication Union as ITU Recommendation Z.151, and is currently Associate Rapporteur responsible for URN at ITU. He was Program Co-Chair for SAM'14 and Finance Chair for RE'15, and is currently the Conference Chair for ICSE 2019. He co-founded the Model-Driven Requirements Engineering (MoDRE) workshop series at RE and is a regular PC member of key conferences in his field (e.g., RE, MODELS, SLE). Gunter worked in industry as a research engineer for Mitel Networks, where he applied and taught URN concepts. He continues to teach software engineering courses and URN tutorials at university, for industry, for governmental departments, and at international conferences.

Benoit Combemale is Full Professor at the University of Toulouse since 2017, and researcher at Inria since 2018. He received his Ph.D. in Computer Science from the University of Toulouse in 2008, and his Habilitation in Computer Science from the University of Rennes in 2015. Before joining the University of Toulouse he was an Associate Professor at the University of Rennes 1 since 2009, and a postdoctoral fellow at INRIA in 2008. His research interests include model-driven engineering (MDE), software language engineering (SLE) and Validation & Verification (V&V). For more information, please visit http://people.irisa.fr/Benoit.Combemale/

Julien Deantoni is an associate professor in computer sciences at the University Cote d'Azur. After studies in electronics and micro informatics, he obtained a Ph.D. focused on the modeling and analysis of control systems, and had a post doc position at INRIA in France. He is currently a member of the I3S/Inria Kairos team. His research focuses on the join use of Model Driven Engineering and Formal Methods for System Engineering. He is particularly interested in understanding how the explicit modeling of the operational semantics of languages can be used for heterogeneous simulation and reasoning.