# Towards Business Process Models at Runtime

Thomas Johanndeiter, Anat Goldstein, Ulrich Frank

Institute for Computer Science and Business Information Systems,
University of Duisburg-Essen, Germany
{thomas.johanndeiter, anat.goldstein, ulrich.frank}@uni-due.de

**Abstract.** Business Process Management (BPM) suffers from inadequate concepts and tools for monitoring and evaluation of process executions at runtime. Conversely, models at runtime promise to give insights into the state of a software system using the abstract and concrete appearance of design time process models. Therefore, we at first advocate to use models at runtime in business process (BP) modeling. Then, we outline the implementation of a prototypical modeling framework for BP runtime models based on metaprogramming. This framework supports the integration of BP type models – models that are enhanced with statistics of runtime data – and instance models – visual representations of executed BPs – resulting in versatile process monitoring dashboards. The approach is superior to object-oriented programming, as it provides a common representation for models and code at various levels of classification, and represents an attractive alternative to object-oriented languages for the implementation of runtime models in general.

## 1 Introduction

Conceptual runtime models of a software system are able to give insights into the current state of a system using concepts of higher levels of abstraction than those of the computational model. Thus, in addition to fostering system (self)-adaptation, these models improve monitoring and understanding of a system's runtime behavior [1].

In existing research work on models at runtime, we see a focus on software architecture [2]. In this paper, we suggest extending the scope of runtime models to the area of Business Process Management (BPM). Business process (BP) modeling plays a major role in BPM. In section 2, we argue why runtime models of BP executions – which are enacted and recorded by BPM systems – present a valuable contribution to process monitoring and decision support in BPM, while only few researchers have expanded their work on runtime models to this particular domain (e.g. [3]).

The implementation of a modeling tool for respective runtime models creates challenges which can hardly be overcome with traditional object-oriented (OO) programming languages, the prevalent programming paradigm used for realizing BP modeling tools. In section 3 we discuss these challenges, which relate to the support of multiple, adaptable levels of classification in software. In section 4, we present an approach which enables overcoming them, exploiting a metaprogramming language. We demonstrate how a modeling tool allows for integrating a BP modeling language,

process type models and process instance models as adaptable runtime objects. Limitations and future research opportunities are discussed in the concluding section 5.

## 2 Background and Motivation

BPM primarily aims at facilitating a) comprehension and communication of business processes, b) continuous process improvement, c) organizational flexibility, and d) process enactment [4]. To achieve these objectives, BPM initiatives typically follow a lifecycle covering the four phases of process design, process implementation, enactment, and monitoring and evaluation [4]. In this lifecycle, conceptual models of BPs play a pivotal role [5], as they are used for representing (literally) how an organization works at the operational level. The pivotal concepts for modeling BPs are 'activity' and 'event'. An activity describes a unit of work. An event indicates completion of activities and might trigger new activities. It may also represent state changes in the external environment of a business process. Activities and events are logically ordered in a BP model with control flow constructs like sequence or fork [4]. BP models ought to be visualized using a notation catering to the perception of business stakeholders [6]. An example of a BP model is given in Fig. 1. As regards the BPM lifecycle, BP models play an important role in process design, implementation, and enactment. In these phases, BP models serve as e.g. descriptive documentations or design specifications. However, BP models created during design and implementation are surprisingly ignored for process monitoring and evaluation [7].

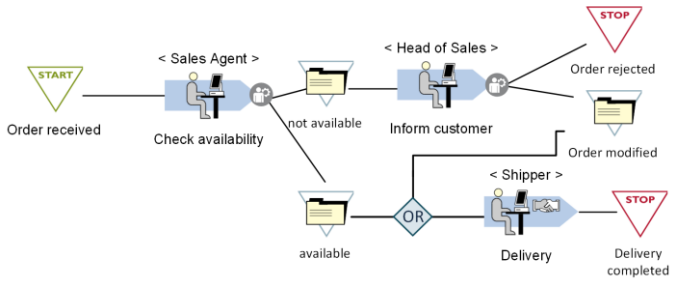


**Fig. 1.** Exemplary business process type model (MEMO OrgML notation)

### 2.1 Business Process Modeling with OrgML

In this work, we use MEMO OrgML as a BP modeling language. MEMO [8] is a multi-perspective enterprise modeling method based on a high-level framework structuring the enterprise with three generic perspectives: strategy, organization, and information system. To allow for more elaborate analyses, each perspective is associated with a domain-specific modeling language (DSML) which defines a set of diagram types focusing on different aspects of the perspective. Among the various DSMLs of MEMO, OrgML [6] accounts for BP modeling. OrgML offers the core concepts of BP modeling: (sub-) processes (i.e. activities), and events. A set of control flow con-

cepts specify the orchestration of processes, e.g. sequence and branching. OrgML offers a visual notation catering to the business domain (see Fig. 1).

The abstract syntax and semantics of MEMO DSMLs are specified using the metamodeling language MML [9]. MML is tailored for designing DSMLs and offers some unique advantages over other metamodeling languages. Most notably, MML supports intrinsic features. With intrinsic features, a MEMO DSML can include meta concepts which are not relevant for the types of a model created with the DSML (i.e. instances of meta concepts), but only for instances of these types. In other words, intrinsic features allow deferring instantiation of meta concepts. This is supported only by few metamodeling languages (e.g. the UML infrastructure does not support such a mechanism). Thus, intrinsic features create a strong motivation for using OrgML in the proposed solution, as further discussed in section 3.1 and in section 4.3. Sharing a common metamodel facilitates the integration of MEMO DSMLs, thereby supporting the integration of different perspectives. For example, OrgML references concepts of the IT perspective (e.g. used IT resources). This renders OrgML BP models more useful for comprehensive business analysis and decision making than other BP modeling languages.

### 2.2 Issues of Process Monitoring Support

For process monitoring and evaluation, process executions ought to be observed and interpreted by responsible managers with respect to different performance indicators. These indicators support e.g. identifying bottlenecks with respect to throughput times of a process. Therefore, process managers require extracting runtime information from BPM systems. However, reporting of runtime information to process managers is not supported with BP models of earlier phases of the BPM lifecycle. This dissatisfactory situation has already been acknowledged in the area of BPM [7].

In fact, the gap between process design and implementation, and runtime process monitoring and associated decision support tasks materializes on three levels. Firstly, there is a difference in domain-specific concepts employed. Design time BP models are typically constituted of activities, events, and control flow concepts (cf. beginning of section 2), whereas for decision support and analysis we typically conceptualize data in ‘facts’ and ‘dimensions’ [10]. Secondly, visualization of runtime information of process executions often diverges from those of design time BP models. Moreover, visualizations of process instances employed in BPM systems are deemed insufficient for process managers [11]. Thirdly, as BP models are not employed for process monitoring and corresponding decision support, there is a fragmentation of tools across the BPM lifecycle [11]. For instance, process managers have to apply Business Intelligence (BI) tools for offline, ex-post analysis of process performance, instead of being able to revert to modeling tools used for process design [7].

### 2.3 Prospects of Business Process Models at Runtime

Against the background of the discussed issues in BP monitoring, we propose to establish BP models at runtime as monitoring and decision support tools for process

managers. In particular, we advocate that representing runtime data of a BPM system in BP models enables the following capabilities (illustrated in Fig. 2):

1. Enhancing BP type models with information that is aggregated from the runtime environment of a BPM system, that is, from actual BP executions. For example we enrich activities of a BP with information such as their average cost (e.g. ❶ and ❷ in Fig. 2). In this way, BP modeling tools serve as managerial dashboards.
2. Visualizing executions of BPs with graphical notations of design models, establishing a more meaningful context for giving better insight into runtime behavior of BPs (e.g. ❸ and ❹ in Fig. 2).
3. Navigating (e.g. ❺ in Fig. 2) between the enhanced BP type models (type level) and the visualizations of actual executions (instance level). When problematic performance is reflected in BP type models, it is then possible to drill down to the source of the problem.

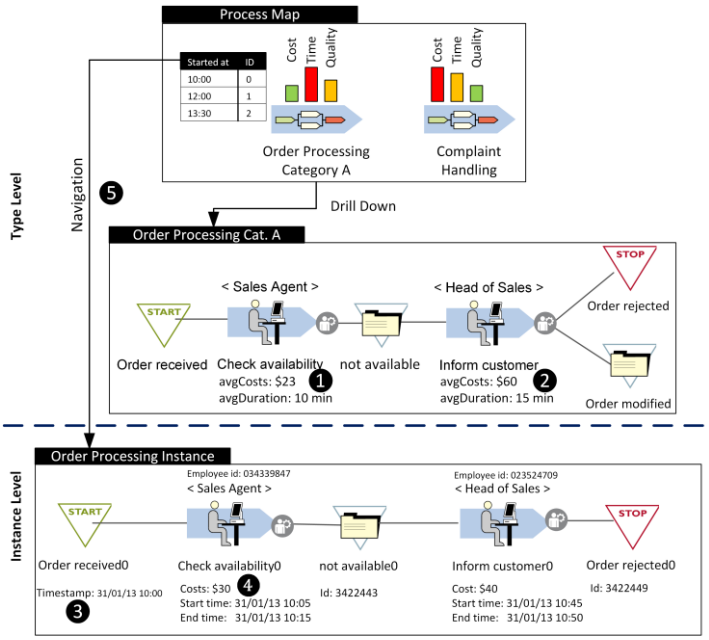


**Fig. 2.** Monitoring dashboard for BP type and instance models at runtime

The listed capabilities shall be realized within a BP modeling framework. Such a framework should offer a suitable BP modeling language and a modeling tool serving as a process monitoring dashboard. The approach presented in this paper for realizing such a modeling tool for OrgML enables various levels of classifications in software code, which are adaptable at runtime. In this approach, we highlight the realization of performance indicators in BP type models, and of instance level attributes that are defined in the modeling language. The implementation uses the metaprogramming language XMF [12].

In the presented work, we substitute interaction of runtime models and BPM systems with process execution simulations. In addition, we restrict adaptations of BP instance models after changes of BP type models, as will be discussed in section 5.

## 3 BP Modeling Framework: Shortcomings of OO Languages

Realizing a modeling tool for BP models at runtime faces unique requirements, which do not apply to "ordinary" modeling tools. In particular, we discuss the need for an implementation approach that is able to offer more than one static and one dynamic programming language level. OO languages typically do not offer this feature. To justify this requirement, we analyze the levels of abstractions required for BP models.

### 3.1 Business Process Types and Instances

The domain concepts of BP runtime modeling recommend three levels of classification. On the highest level, generic concepts of the domain, e.g. activity and event (cf. section 2), are captured in a metamodel, specifying a BP modeling language. This level is denoted $M_2$. BP models commonly refer to what is known as the type level. They describe types or classes of BPs (e.g. processing of an order) rather than particular instances (e.g. processing a specific order of a particular customer). This level is denoted $M_1$. A BP instance model is characterized by a one-to-one representation of elements of a single BP execution and located on $M_0$ [4], [14]. In addition, conceptual modeling frameworks (e.g. the MEMO framework or the UML infrastructure) comprise a fourth level of abstraction: the meta-metamodel level $M_3$ where concepts for defining modeling languages of any kind are specified (see Fig. 3).

Representing runtime information recorded in a BPM system requires accounting for the peculiarities of BPs in such a four level modeling architecture. Indeed, a runtime model must always reflect the correct state of software system. Applying this requirement to process executions, we find that the essential attributes reflecting a process instance at runtime comprise timestamps, activity cost, used resources, and source / target activities of events [7]. They may comprise further attributes like the particular employee executing the process.

The identified instance level abstractions need to be included in the BP modeling language. Ignoring instance model abstractions in the language ($M_2$) implies burdening language users with the responsibility of defining type level runtime attributes and associations, such as a timestamp and cost, in every BP model they create [15]. This hampers a shared understanding and reusability of resulting models, as an apparent abstraction is ignored, resulting in dangerous redundancies [9]. Furthermore, according to the relation of BP types and instances (cf. section 2), a type level view on runtime attributes of a BP reflects a set of runtime instances, e.g. by calculating the performance indicator 'average cost' for a set of process instances. Support of such performance indicators is facilitated by specifying selected properties of instances, like execution time or cost, in the modeling language. Only then, we are able to define that a performance indicator is always calculated from the same kind of instance level

runtime information, regardless of the actual user-created BP type model ($M_1$). To address instance level abstractions on the modeling language level, we implement intrinsic features of MEMO OrgML (cf. section 2.1) in our modeling tool.

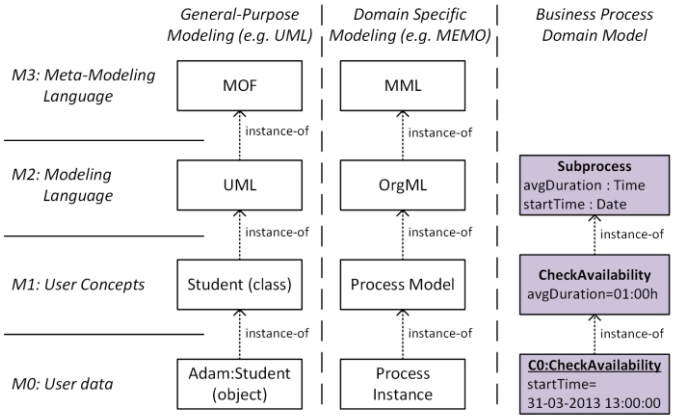


**Fig.** 3. Levels of abstraction of BP (runtime) models

In conclusion, both type and instance level shall represent runtime phenomena of BPs. BP instance models have to be represented as runtime objects in the modeling tool so that they can be adapted according to the execution of BPs over time. Additionally, the type level is subject to modifications at runtime, since it classifies runtime information in performance indicators, which shall be updated in real-time. Therefore, the type level describes properties of BP instance models, while at the same time types are runtime objects, as exemplified on the right side of Fig. 3.

### 3.2 Metamodel-based Modeling Tool Creation

The state-of-the-art approach for creating modeling editors, inter alia BP modeling tools, compiles a modeling editor for type models from a metamodel of a modeling language. Most metamodeling facilities following this approach, e.g. the Eclipse Modeling Framework (EMF) [16], rely on OO languages (e.g. Java). OO languages typically support two language levels: classes and objects. Correspondingly, during language design the metamodel is maintained in $M_0$ programming language objects. These objects are used for generating classes, which define the software structure of a modeling editor for corresponding type models. For creating instance model editors, the same "edit-generate-compile-validate"-cycle is applied to type models created by language users, as they are stored in $M_0$ runtime objects of the generated editor and thus cannot be instantiated into process instance representations [17].

We argue that this approach is not feasible a BP runtime modeling tool, as type and instance models must be maintained in adaptable, yet integrated runtime objects of the modeling environment (cf. section 3.1). Only then, performance indicators and instance model representations are able to react to changes of runtime data from a BPM

system. In the OO approach, type and instance model editors are disconnected due to the unfavorable "edit-generate-compile-validate"-cycle [17] (see Fig. 4). Hence, conceptual models and implementation code do not share a common representation, as they are connected via generation and compilation [18]. Type and instance models are depicted in disconnected editors, severely hampering runtime synchronization between type and instance level. Synchronizing the evolution of the type level with the instance level is hampered: whenever a type model changes, the described cycle needs to be iterated. Moreover, when a BP instance is created or changed, e.g. a subprocess terminates, corresponding changes should be immediately reflected in relevant type model editors, e.g. within performance indicators. This indicates that recompilation of the editor, as dictated in the OO approach, is not feasible for runtime models.

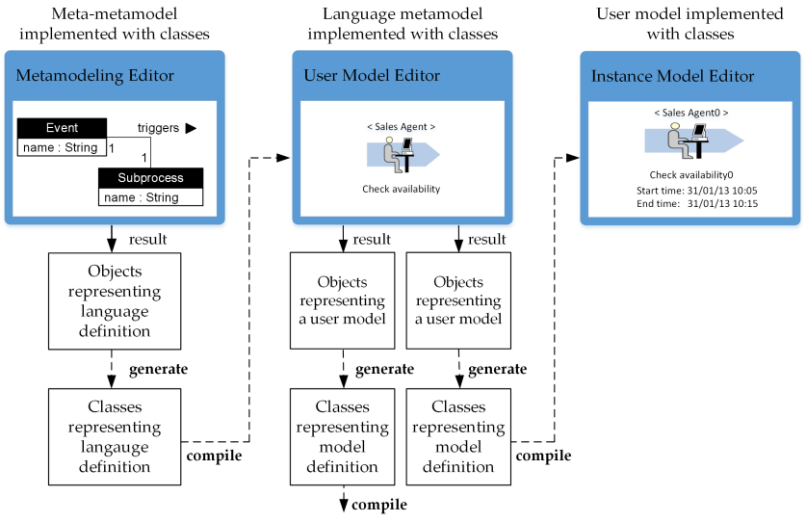


**Fig. 4.** Edit-generate-compile-validate cycle for metamodeling based model editor creation

A workaround for OO implementations is to mimic the classification relationship of types and instances with a manually implemented relationship. For example, a class representing the type features, and a class representing the instance features may be introduced and associated by a 'typeOf' relationship. Such a relationship, however, lacks the formal semantics of "true" instantiation. Thus, additional code is required, which produces a severe threat to system integrity [19].

## 4 Solution Approach

In this section we explain the core features of XMF, which serves to implement an OrgML modeling editor. Then, we describe the implementation architecture of a modeling tool for BP runtime models that supports a runtime integration and synchronization of type and instance level within a multi-level classification hierarchy.

### 4.1 XMF

A metaprogramming language is distinguished from regular programming languages through its ability to manipulate programs (e.g. the class level of OO languages) instead of program instances at runtime [13]. The metaprogramming capabilities used in the presented work derive from XMF [12]. XMF's pivotal paradigm is the application of executable metamodeling for language design, which is reflected in its core features. Firstly, XMF comprises a metamodeling language: XCore. The metamodeling language is complemented with an extended version of the Object Constraint Language (OCL), called XOCL. XOCL includes action primitives for defining executable semantics. These action primitives allow, amongst other aspects, specifying sequential execution paths, e.g. object creation and imperative control structures.

Secondly, similar to OO languages, XCore distinguishes the concepts of 'Class' and 'Object'. A particular class is instantiated into objects, which by default cannot be instantiated again and cannot inherit. However, in contrast to regular OO languages, instances of a class may again inherit the XMF concept 'Class', instead of 'Object', as 'Class' in XMF inherits 'Object'. This implies two things. Firstly, in XMF classes are always objects at the same time, i.e. they are dynamic runtime elements which can be manipulated during program execution. Secondly, as an instance of a class is able to again serve as a class, multiple consecutive instantiations are enabled. This recursive style of language definition employed in XMF, where only relative metatype to type to instance relationships are distinguished, is called golden braid architecture. Because of this architecture, XMF is more powerful than those OO languages offering a meta-object protocol (MOP), for example Smalltalk. A MOP basically consists of meta classes which allow accessing the class level ($M_1$) at runtime. However, the MOP of Smalltalk only allows for a meta-class to classify a single class ($M_1$). Therefore, for making classes adaptable at runtime in Smalltalk, every class requires a particular meta class, whereas XMF enables creating a single meta class for multiple classes on $M_1$.

The described architecture of XMF, in which instances of classes can again serve as classes, justifies why implementing a BP modeling tool with XMF avoids the "edit-generate-compile-validate"-cycle.

### 4.2 Tool Implementation Architecture

As a first step to implement a modeling editor featuring OrgML, we reconstruct MEMO MML (cf. section 2.1) with XMF metamodeling concepts. In other words, we instantiate XMF concepts like 'Class' and 'Attribute' in order to model the MML. Because of this instantiation, MML concepts possess executable features, as discussed in section 4.1, e.g. execution of operations. In addition, MML concepts are set to inherit XMF concepts in order to add executable semantics to instances of the MML meta-metamodel. Through this inheritance, instances on $M_2$ are equipped with executable features, like setters and getters, which are crucial to modeling editors.

Secondly, we model OrgML with the implemented MEMO MML and account for runtime model capabilities. For this purpose, runtime functionality relevant to all

instances of ‘MetaEntity’ (the meta meta class all concepts of OrgML are instantiated from, see Fig. 5) is captured in a concept called ‘Entity’. The functionality mainly comprises navigation from type to instance models, calculation of performance indicators, and instantiation of intrinsic features (described in section 4.3). ‘Entity’ also inherits XMF’s ‘Class’ in order to add execution semantics to instances of MetaEntity and enable their further instantiation. Thereby, every OrgML concepts transitively through ‘Entity’ inherits XMF’s ‘Class’. In combination, we can use the implementation of OrgML to create an OrgML editor at runtime with XMF. BP type models created with this editor can be directly instantiated into instance models, implying the ability to create instance model editors at runtime from BP type model specifications. The inheritance hierarchy enabling the instantiation of dynamic type level entities is given in Fig. 5. All levels can be dynamically manipulated.

‘Entity’ also encapsulates an operation for querying instances of a type. With said operation, $M_1$ level concepts are able to get a list of its instances, which serves as a foundation for integrating and navigating to the instance level at runtime.

Fig. 5 exhibits a pivotal idea of the proposed modeling tool for coping with the multiple dynamic levels implied by BP runtime models: cross-level inheritance. Indeed, ‘Entity’ inherits from ‘Class’, while being at the same time an instance of ‘Class’; all OrgML entities inherit ‘Entity’, while OrgML entities are instances of ‘MetaEntity’. ‘MetaEntity’ and ‘Entity’ are, however, on the same level of abstraction. Because of the (transitive) cross-level inheritance of XMF concepts within the presented architecture, the role XMF plays in relation to the four modeling levels of MEMO is twofold: on the one hand, XMF can be considered a fifth level of abstraction above $M_3$; on the other hand, XMF serves as a modeling infrastructure offered to all MEMO modeling levels via transitive specialization / generalization.

### 4.3 Implementation of Business Process Runtime Abstractions

The implementation of performance indicators starts with modeling them as meta-attributes in the OrgML definition. These meta-attributes are marked ‘derivable’ in order to demand a calculation (e.g. see meta-attributes of ‘AnyProcess’ in Fig. 5). For providing the calculation logic, we specify a generic operation for ‘Entity’, which offers an interface for setting data source and calculation (‘calculateDerivable()’ in Fig. 5). Additionally, we add a set of generic calculation operations to ‘Entity’, e.g. mean and maximum value. However, the specific derivation often needs to be defined by the language designer. Therefore, custom derivation operations can be added to any child class of ‘Entity’. For instance, in order to calculate the average duration of a subprocess, we add a calculation operation to ‘AnyProcess’ which is able to identify the amount of time incurred between two timestamps.

In order to include attributes for instance level runtime information in the BP modeling language level, we use MEMO’s intrinsic features (cf. section 2.1). Features such as meta-entities, meta-attributes or meta-association of the metamodel ($M_2$) can be defined as intrinsic, meaning that they are relevant only at the instance level ($M_0$), while at the type level we abstract from them. For example, defining a meta-attribute ‘cost’ for the OrgML concept of ‘Subprocess’ and marking it as intrinsic entails that

the attribute is – in contrast to regular meta-attributes – not instantiated into a particular value on the $M_1$ level, but on the level $M_0$ (see illustration in Fig. 5).

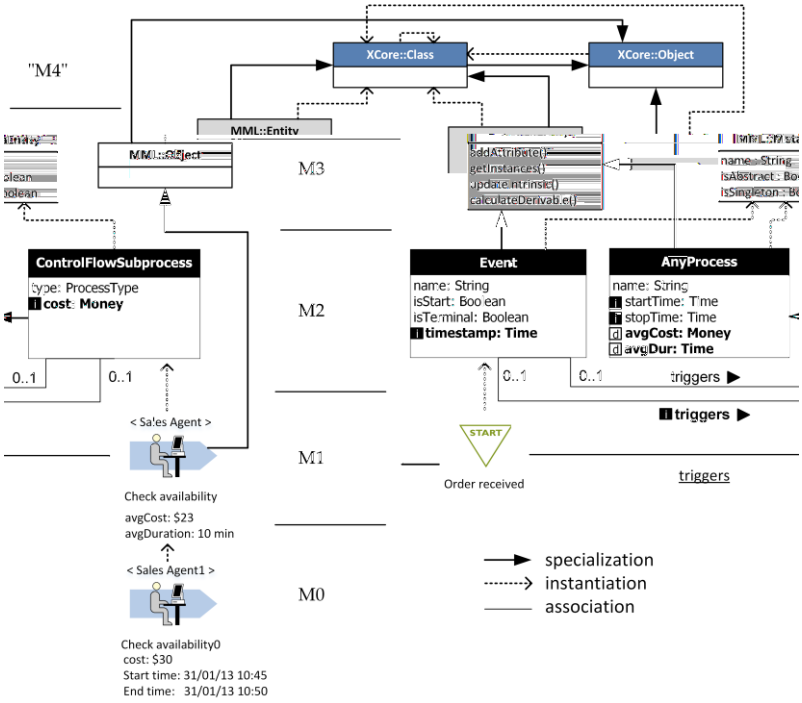


**Fig.** 5. OrgML entities are instantiated from MML and inherit from XMF concepts (boxed letter 'i' indicates intrinsic feature, boxed letter 'd' indicates derivable feature)

While deferring the instantiation of meta-attributes can be straightforwardly accounted for in the constructors of 'MetaEntity' and 'Entity', for meta-associations we require a more elaborate solution. The problem is that until BP type models are defined on $M_1$, we do not know which associations are valid on $M_0$ and what are their cardinalities. Consider the intrinsic association 'triggers' from event to subprocess in Fig. 5. Now take the instances of event ('Order received') and subprocess ('Check Availability') on $M_1$ into consideration. The concepts on $M_1$ are linked via an instance of the regular meta-association 'triggers'. Only because of this link, which is set by a BP modeler, i.e. a user of the language, the typing of the intrinsic meta-association 'triggers' can be restricted such that only instances of 'Order Received' can be connected to instances of 'Check Availability' on $M_0$. Since this information may change dynamically in the user model, the modeling tool needs to continuously observe the creation and modification of links in BP type models.

The cause of this problem is located in underspecified semantics of classifications across several modeling levels. Thus, we propose a workaround solution. Once the definition of the BP type model on $M_1$ is completed, at the behest of the modeler an operation called 'updateIntrinsics()' is executed. This operation goes over the BP type

model and sets the types of all intrinsic association of the particular $M_1$ model based on actual links. Such a workaround is practicable for BP models, as it allows changing a BP type model and instantiating new, valid BP instance model editors from the specification of said type model.

Finally, in order to automatically synchronize representations of BP type and instance models with the actual runtime information of BP instances, our prototype uses change listeners, called daemons. With daemons, we exploit that in XMF every element is aware of the higher-level concept it is an instance of. Furthermore, in our implementation every 'Entity' instance is able to query for its instances. Correspondingly, daemons can notify types on changes of their instances. Thus, when the execution of a BP instance proceeds, corresponding listeners are notified and automatically update derivable attributes in the type model. The update of the BP type model can take place at runtime, because owing to the features of XMF the 'classes' of BP instance models are stateful runtime objects at the same time.

When combining type level performance indicators and instance level attributes in integrated modeling editors, we arrive at a tool serving process design on the one side, and versatile process monitoring on the other, as exemplified in Fig. 2.

## 5 Conclusion, Limitations and Future Work

We started this paper by pointing out to the prospects potentially offered by runtime models to the field of BPM. It was shown how BP runtime models can overcome the insufficiencies of available approaches for process monitoring. Then, we elaborated on how to implement a modeling tool supporting such a BP modeling environment. It was shown that due to the two dynamically adaptable modeling levels of BPs (section 3.1), common metamodeling frameworks and OO implementation languages are not suitable for integrating BP type and instance models at runtime. This led us to propose an alternative solution based on implementing OrgML with the metaprogramming language XMF.

We advocate that the proposed approach is advantageous whenever multiple dynamic levels of abstraction need to be accounted for in a tool environment for runtime models. For example, runtime models of software architecture will presumably require a synchronization of system instance (i.e. the running system), system instance models and system configuration models.

However, we purposefully ignored some aspects of BP models at runtime in this work. Firstly, we did not address the synchronization of runtime models and BPM systems. Instead we used a simulator to imitate BP executions. Accordingly, we contemplate interfacing a BPM system for feeding operational data into BP runtime models, and invoking model-based modifications to a BPM system. For this purpose, we consider exploiting XMF's low-level messaging API to connect to external software, which requires an interface observing a BPM system and informing XMF listeners about the arrival of data. Yet, in this approach the BP model and the representation of the model in the BPM system code would still be separated, which requires complex synchronizations for enabling mutual adaptations. Therefore, we envision extending

our prototype with actual BP enactment features in order to directly use the BP runtime models as code for the enactment of processes.

Secondly, we did not discuss how BP instances and models of these instances react to changes of BP type models. In our prototype, we do not allow changes in type models to affect already executed BP instances in order to not invalidate originally legal process executions. Coping with type level changes in this respect recommends implementing policies that ensure avoiding inconsistent system and model states.

## References


1. Blair, G., Bencomo, N., France, R.B.: Models@run.time. Computer 42(10), pp. 22–27 (2009)
2. Bencomo, N.: On the use of software models during software execution. In: ICSE Workshop on MISE '09, pp. 62–67. IEEE Press, Piscataway, NJ (2009)
3. Sánchez, M., Barrero, I., Jorge, V., Deridder, D.: An Execution Platform for Extensible Runtime Models. In: 3rd Workshop on Models@run.time at MoDELS ’08 (2008)
4. Weske, M.: Business process management. Concepts, languages, architectures, 2nd ed. Springer, Heidelberg et al. (2012)
5. Kettinger, W.J., Teng, J.T.C., Guha, S.: Business Process Change: A Study of Methodologies, Techniques, and Tools. MIS Quarterly 21(1), pp. 55–80 (1997)
6. Frank, U.: MEMO Organisation Modelling Language (2): Focus on Business Processes. ICB Research Report 49. University of Duisburg-Essen (2011)
7. van der Aalst, W.M.P.: Process mining. Discovery, conformance and enhancement of business processes. Springer, Heidelberg et al. (2011)
8. Frank, U.: Multi-perspective enterprise modeling: foundational concepts, prospects and future research challenges. Soft. Syst. Model., pp. 1–22 (2012)
9. Frank, U.: The MEMO Meta Modelling Language (MML) and Language Architecture. 2nd Edition. ICB Research Report 43. University of Duisburg-Essen (2011)
10. Kimball, R.: The Data Warehouse Toolkit. Practical Techniques for Building Dimensional Data Warehouses. Wiley, New York et al. (1996)
11. Bandara, W., Indulska, M., Chong, S., Sadiq, S.: Major Issues in Business Process Management: An Expert Perspective. In: Proceedings ECIS 2007, pp. 1240–1251 (2007)
12. Clark, T., Sammut, P., Willans, J.: Applied metamodelling: a foundation for language driven development, 2nd edn. Ceteva (2008)
13. Spinellis, D.: Rational Metaprogramming. IEEE Software 25(1), pp. 78–79 (2008)
14. Kühne, T.: Matters of (meta-) modeling. Soft. Syst. Model. 5(4), pp. 369–385 (2006)
15. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. Proceedings of the 4th UML ‘01. LNCS, vol. 2185, pp. 19–33. Springer, Berlin, Heidelberg (2001)
16. Eclipse: Eclipse Modeling Project, www.eclipse.org/modeling/emf/ (2013)
17. Atkinson, C., Kühne, T.: Concepts for comparing modeling tool architectures. In: Model Driven Engineering Languages and Systems. LNCS, vol. 3713, pp. 398-413. Springer, Berlin, Heidelberg (2005)
18. Frank, U., Strecker, S.: Beyond ERP Systems: An Outline of Self-Referential Enterprise Systems. ICB Research Report 31. University of Duisburg-Essen (2011)
19. Kühne, T., Schreiber, D., New York: Can programming be liberated from the two-level style: multi-level programming with deepjava. In: Proceedings OOPSLA '07, pp. 229–244. ACM Press (2007)