Formalizing Models and Meta-models for System Development
Extended Abstract

R. Venkatesh, Purandar Bhaduri and Mathai Joseph
TRDDC, Tata Consultancy Services
54B Hadapsar Industrial Estate
Pune 411 013
{rvenky,pbhaduri,mathai}@pune.tcs.co.in

Abstract

Meta-model based development offers a promising way of managing the complexity of industrial scale software development by describing a system in terms of different 'views'. These views can then be described as instances of a single meta-model. Such views are usually not disjoint and it is essential that they are shown to be consistent. A weakness of meta-modelling tools is the lack of support for describing the behaviour of models, and this is central to demonstrating the consistency of views. We address this problem by combining meta-modelling with formal techniques for stating and verifying behavioural properties. In this paper, we describe a formalization of models and meta-models and show how this leads to automated procedures for consistency checking between views in an industrial software development framework.

1. Introduction

Industrial scale software development relies heavily on the use of diagrammatic notations to represent what are informally known as *models*. Each model describes different aspects of the system under construction. These models help in managing the complexity of problems by separating concerns. Different models often use different description techniques, each suitable for describing a particular aspect of the system. For example, the models used in UML [1, 5] are the *structural views*, e.g. classes, objects and their attributes and relationships, and the *behavioural views*, e.g. sequence and statechart diagrams, depicting inter-object collaboration and intra-object state transitions.

Each of these different models can be treated as an instance of a part of a single *meta-model*. For example, each UML diagram, which is a view of the system, is an instance of the UML meta-model [3].

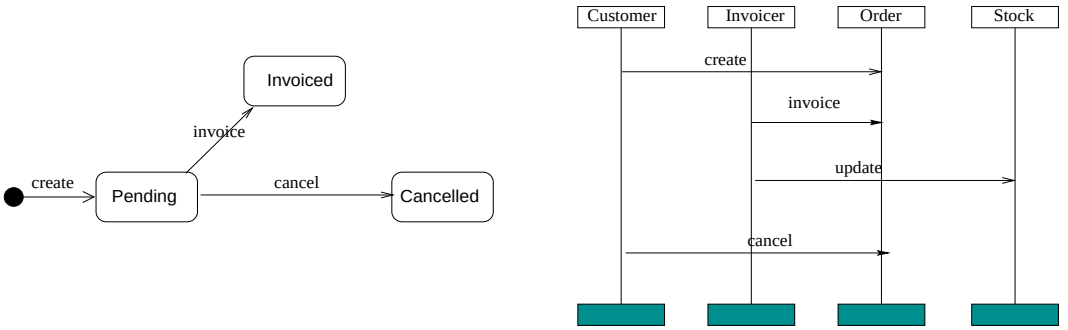
Different views will usually not be disjoint and thus may describe the same property of the system. Common properties must be *consistently defined* to ensure the integrity of the system. For example, two views can be used to describe the dynamic behaviour of a system:

1. A state machine model which describes *all* permissible sequences of events in the life of an object as well as the object's actions in these sequences. This can be considered as a projection of the system traces on a specific object.

2. A Message Sequence Chart which describes one possible time sequence of interactions (send and receive events) among a set of given objects. This describes only a set of system traces between two (usually unspecified) system states restricted to a set of objects.

EXAMPLE. Consider the simple invoicing system described in [2]. An order can be placed for a certain quantity of any one product. An order can be invoiced, i.e. its state changed from pending to invoiced, if the quantity in the order is less than the quantity of the product in stock. An order may also be cancelled. It is not clear from the description of the problem whether an invoiced order can be cancelled, and two different views might interpret it differently. Figure 1 shows two views of the invoicing system: the state machine for an *Order* and a Message Sequence Chart (MSC) depicting order cancellation. These two views are not consistent, because the state machine asserts that an invoiced order cannot be cancelled, while the MSC shows a contradictory situation.

UML has an *Object Constraint Language* (OCL) [8] for describing the static semantics of individual UML diagrams through a set of well-formedness rules. Unfortunately, OCL can be used only for stating the constraints and invariants of *individual* models. Even if the use of OCL were extended to capture the static relationships between different models,



(a) State Machine for Order

(b) MSC for cancelling an invoiced order

Figure 1. Two Views of the Invoicing System

OCL is not powerful enough to describe the dynamic behaviour of the kind shown in the example. Existing UML tools cannot detect such inconsistencies either, because they lack the *semantic* support necessary for describing the behaviour of models, which is central to checking consistency.

In this work, we provide a semantic basis for the views of a system that are derived from a meta-model by combining formal techniques with the meta-modelling approach. The goal is to formalize the *behaviour* of a system model and its views, and to be able to automate the consistency checking of different models. We demonstrate how this enables inconsistencies to be detected.

2. System Model and Views

The consistency of views of a system can be ensured by defining a unified model of the entire system. This model specifies all the possible ways in which the system might evolve over time. We refer to a particular evolution of the system as a *trace*.

Consider a labelled transition system with additional structure on the states as the system model. The *system model* is defined as a tuple $S = (States, \longrightarrow, Events)$, where $States$ is a set of states, $\longrightarrow$ is the transition relation with labels drawn from the set $Events$. A system state $s \in States$ is a collection of objects $\{ob \mid ob \in Objects\}$ and their values, where each object is a tuple $(ObjectID, set(< AttributeName, AttributeValue >), Operations)$. The event e in a system transition $s \xrightarrow{e} s'$ is either the send or receipt of a message or an operation invocation.

A system trace is a sequence of system states

$$s_0 \xrightarrow{e_0} s_1 \ldots s_i \xrightarrow{e_i} s_{i+1} \ldots$$

such that two successive states are related by a system transition. Note that the state of at most one object is modified by a system transition. We use an interleaving model of concurrency in which a system trace represents a serialization of a given set of events.

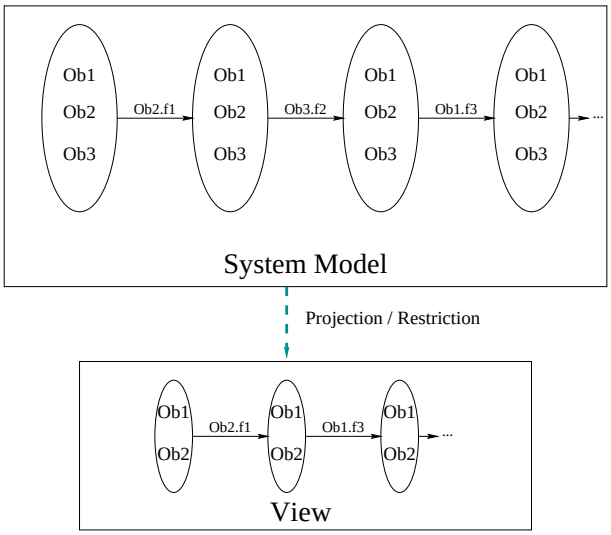


Figure 2. System Model and Views

A *view* of a system S is composed of a subset of objects, transitions and traces of S. The transitions that do not affect the objects in a view are hidden in the view. Thus, a trace in a view is a subsequence of a system trace containing exactly those transitions that affect the objects in the view. This is illustrated in Figure 2, where the view includes objects *Ob1* and *Ob2*.

The set of traces of a view defines the constraints on the behaviour of the objects contained in the view. These constraints are expressed as assertions over system states and traces that *may*, *must* and *never* happen in the evolution of the system. For example, the state machine model of an ob-

ject enumerates the *may* traces (possible sequences), along with the proviso that the list of traces not specified must *never* occur. An MSC model, on the other hand, enumerates a list of *must* traces (mandatory sequences) on a set of objects, while not claiming anything about other traces in the system. The consistency of two different views follows from the consistency of the assertions over the system contained in these views.

EXAMPLE (cont.) It follows from our formalization of consistency that the views in Figure 1 are not consistent. According to the MSC view *create,invoice,cancel* must be a valid system trace when restricted to the *Order* object. In contrast, the state machine view asserts that there is no transition labelled *cancel* from the *Invoiced* state, and hence *create,invoice,cancel* cannot be a valid trace of *Order*.

This formalization is an extension of the meta-modelling approach to software development outlined in [6] which proposes a meta-model within which the different views can be related and integrated. This is illustrated in Figure 3, where the models *M1*, *M2* and *M3* can be related by defining them in terms of a meta-model *MM*. *A1*, *A2* and *A3* are instances of models *M1*, *M2* and *M3* respectively. The program *P* implementing the system is a composition of *P1*, *P2* and *P3*, each of which implements *A1*, *A2* and *A3* respectively. In this paper, we associate meta-views *MV1*, *MV2* and *MV3* with *M1*, *M2* and *M3* respectively. Associated with *A1*, *A2* and *A3* are views *V1*, *V2* and *V3*. These are instances of *MV1*, *MV2* and *MV3* respectively. For the program *P* to be correct we require the views *V1*, *V2* and *V3* to be consistent. Further, we have provided a *generic* formal definition of a view of a system, from which all particular views can be derived as instances. The consistency of two views is also given a generic definition at the meta-level. The advantage of this generic approach lies in the ability to extend the meta-model and still be able to check for consistency. When the meta-model is extended to capture another view, a definition of the extension can be derived as an instance of this generic view.

This technique of meta-model extension is used effectively by the program development environment MasterCraft [7], where UML is used to represent some models of an application. The UML meta-model has been extended to enable additional modelling notations like the user-interaction and entity-relationship (ER) models. MasterCraft has rules that guarantee type consistency of these models. The next step is to extend the capability of the tool for checking the consistency of behavioural models.

3. Conclusion

In this work we formalize the structure and behaviour of individual models in relation to a single system model, and not in isolation. This is unlike many approaches to the formalization of UML models. We rely on a combination of meta-modelling and formalization for ensuring the integrity of view based system development. The meta-modelling framework ensures that all instances are statically correct; the formalized models on the other hand enable tool support for assuring consistency in dynamic behaviour.

This work is part of a program for formalizing system development based on meta-modelling. Once a developer has modelled an application using different views, they can be checked for consistency using tool support based on this formalization. The MasterCraft meta-modeller [4] guarantees type consistency of different meta-models and is being extended to incorporate automated procedures for consistency checking between views.

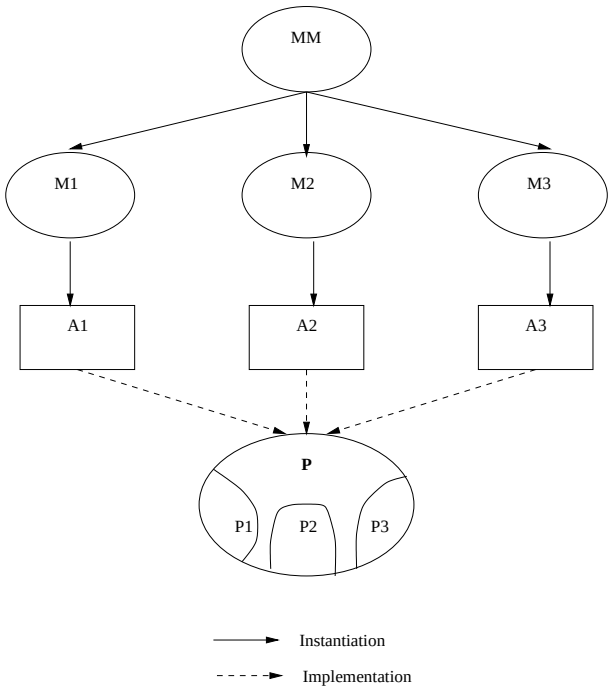


Figure 3. Different views of a system with their models and meta-model

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[2] M. Frappier and H. Habrias. *Software Specification Methods: An Overview Using a Case Study*. Springer, 2000. Case Study description available from `http://www.DMI.USherb.CA/~spec/index.html`.

[3] OMG. The Unified Modeling Language (UML) Specification - Version 1.3, June 1999. Joint submission to the Object Management Group (OMG) `http://www.omg.org/technology/uml/index.htm`.

[4] S. Reddy, A. Bahulkar, and J. Mulani. Adex - a meta modeling framework for repository-centric systems building. In *Proceedings of the Tenth International Conference on Management of Data, COMAD 2000, Pune*, Dec. 2000.

[5] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[6] A. Sreenivas, R. Venkatesh, and M. Joseph. Meta-modelling for formal software development. In C. Fidge, editor, *Computing: The Australasian Theory Symposium (CATS 2001), Electronic Notes in Theoretical Computer Science*, volume 42. Elsevier Science Publishers, 2001.

[7] Mastercraft: Integrated development framework for distributed applications. Tata Consultancy Services, 1999. `http://www.tcs.com/products/mastercraft/htdocs/mastercraft_index.htm`.

[8] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.