An Integration Concept for Complex Modelling Techniques

Benjamin Braatz
Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
bbraatz@cs.tu-berlin.de

Abstract. In this paper a concept for the integration of complex modelling techniques like e. g. UML is proposed. The integration is done by translating complex models consisting of parts following different modelling paradigms into a common low-level language, which is designed to be minimalistic enough to serve as a source for code generation and verification. On the other hand the low-level language should be expressive enough to allow the integration of the most common structural, behavioural, and constraint modelling languages. As an example for a complex modelling technique a derivation of the UML, which focuses on a small subset of the UML diagrams, but also adds some additional techniques, is considered. Moreover, a low-level language for object-oriented modelling techniques is sketched.

Keywords: UML, Semantic Integration, Code Generation

1 Introduction

Contemporary modelling techniques follow a large variety of different paradigms. For example, the Unified Modeling Language (UML, see [7]) contains state machines, activity diagrams, sequence and collaboration diagrams, and the Object Constraint Language (OCL, see [8]) to describe the behaviour of a modelled system. Other techniques like structured flowcharts as introduced by Nassi and Shneiderman (see [6]), graph transformation systems (see e. g. [4]), different kinds of process algebras and Petri nets, and temporal or modal logic formalisms are also in common use.

It would be very desirable to be able to use multiple techniques from this wealth of possibilities in one project, in order to describe the various aspects of the system with the technique, which fits best. This approach can, however, only develop its full potential, if the interconnections between the used techniques are made explicit. Ideally all applied techniques should be semantically integrated, i. e. interpreted in a common semantic domain.

This paper proposes an abstract concept introduced in Sect. 2 for integrating complex, multi-paradigm modelling techniques by translating complex models into a low-level language, which is designed to facilitate the defintion of a formal semantics, the generation of code from models, and the formal verification of models.

In Sect. 3 a complex modelling technique based on the UML is introduced, which is then translated exemplarily into an object-oriented low-level language sketched in Sect. 4. Finally, in Sect. 5 the approach is summarised, related work is discussed and some ideas for future work are given.

2 Abstract Integration Concept

The concept proposed in this paper assumes a complex modelling technique, given by a class **Mod** of models containing parts, which are described following multiple paradigms. These models are translated into a low-level language, given by a class **L3** of low-level models, leading to a function Int: **Mod** → **L3**. The low-level language should contain the essential information included in the complex models, but encode it in a paradigm-independent way. Each low-level model $L \in$ **L3** can be decomposed into its constructive part Beh(L) and its descriptive part Prop(L), which share a common structural part Str(L). (Cf. Fig. 1.)

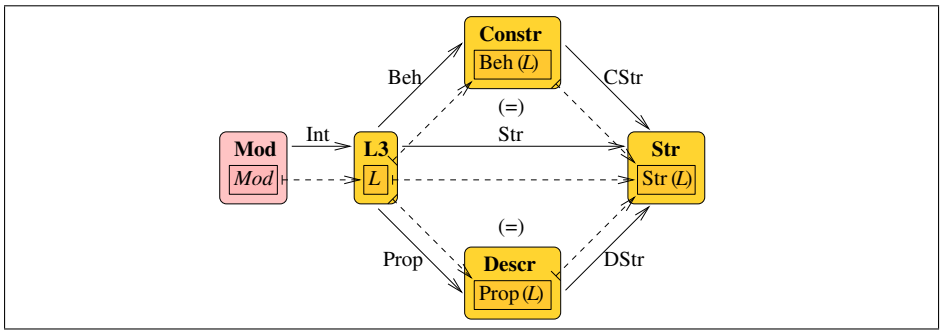


Figure 1: Relations between complex and low-level models

The rationale behind this decomposition is the strict seperation of the modelled behaviour and the properties, which have to be fulfilled by the behaviour. This seperation aids in providing tools for the low-level language. The constructive behaviour models can be used as a basis for code generation and as the axioms or models in verification, while the descriptive property models are suited to serve as a foundation for tests and as the target or specification in verification. The common structure models are needed to facilitate the connection between both parts of an **L3** model.

Note, that the seperation of concerns is established only in the low-level language. The complex modelling technique may, and most often will, combine constructive, descriptive, and structural aspects in the same diagrams. For example, a UML class diagram contains mostly structural information, but may also exhibit constraints like the multiplicities of attributes and associations or OCL pre-post-conditions and invariants, which are to be translated into the descriptive part of the corresponding low-level language model. It may even contain some OCL body constraints for query operations, which do not alter the

state of the system. Such constraints already specify the complete behaviour of the operation in question and may therefore be translated into the constructive part of the **L3** model.

A formal semantics for the low-level language will be provided by assigning a class of formal system representations **Sys**(*Str*) to each structural model *Str*, a subclass $\mathrm{Sem}_{Str}(Prop)$ of all systems satisfying the specified properties to each descriptive model *Prop*, and a single constructed system $\mathrm{Build}_{Str}(Beh)$ to each constructive model *Beh*. (Cf. Fig. 2.)

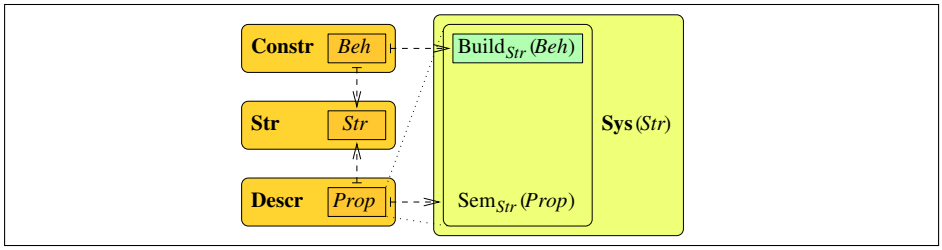


Figure 2: Semantics of low-level models

The exact definition of the semantic domain and functions is out of the scope of this paper. A draft version of such a semantics can be found in [3], which is based on transformation systems (see [5]) and adhesive high-level replacement systems (see [4]).

The formal semantics provides for the definition of consistency of a model by requiring that the behavioural part of the model satisfies the properties stated in the descriptive part, i. e. a low-level language model $L \in$ **L3** is consistent iff

$$\mathrm{Build}_{\mathrm{Str}(L)}(\mathrm{Beh}(L)) \in \mathrm{Sem}_{\mathrm{Str}(L)}(\mathrm{Prop}(L)) \ .$$

By composition with the integration function Int the formal semantics and the notion of consistency are also applicable to the complex modelling technique **Mod**.

The advantages of this approach lie in the decoupling of the complex modelling technique from the low-level language, for which the formal semantics is defined, and for which tools for generation and verification can be written. On the one hand, the semantics and tools do not have to deal with all the subtleties and "syntactic sugar" of the complex modelling technique, but can be based on the more minimalistic low-level language. On the other hand, the complex modelling technique can be enhanced by additional techniques rather easily, because only the class **Mod** and the integration function Int have to be adopted. Semantics and tools for extensions of the complex technique are then obtained automatically.

In the next section the idea of a complex integrated modelling technique will be illustrated by a small multi-paradigm modelling technique derived from the UML, while in Sect. 4 an object-oriented low-level language will be sketched.

3 CUML – The Complex Modelling Technique

As complex modelling technique we consider a derivation of the UML, which we will call CUML (Compact, Comprehensive, and Constructive UML). It uses only some of the features of UML, namely class diagrams, activity diagrams, and OCL constraints, but enhances them with transformation rules and structured flowcharts (also known as Nassi-Shneiderman diagrams, see [6]). These extensions were already proposed in [2] to yield a constructive, object-oriented modelling technique.

According to the intention of this paper, CUML is designed to allow code generation and formal verification. Therefore, we require stricter modelling than the original UML, which allows to leave a lot of features unspecified and describe requirements, actions, and other model properties in natural language.

We will describe the ideas along a small example of a shopping cart application, whose class diagram is shown in Fig. 3. The shopping cart itself is modelled by the Cart class containing Item instances, which in turn reference the corresponding Product instance. Instances of the class Catalog are used to administrate the Product instances.

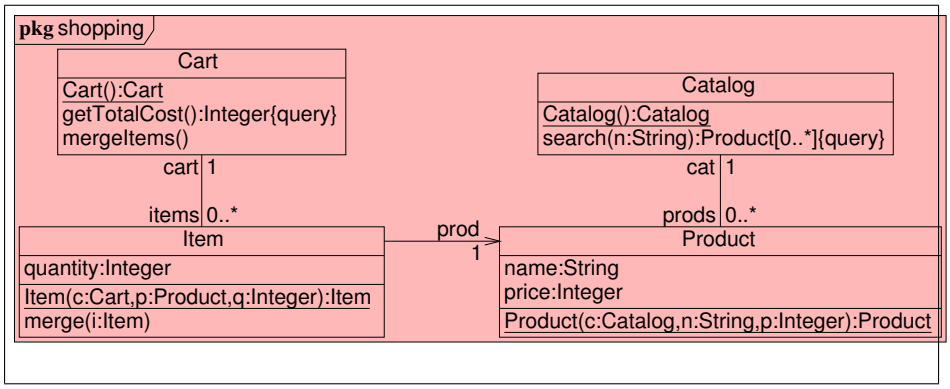


Figure 3: Class diagram of the example

Besides the usual features of class diagrams (declaring signatures of classes with properties and operations, as well as associations with multiplicities and navigability), we also use the possibility to specify if an operation is static for the constructors of the classes by underlining them and the possibility to specify if an operation is a query, i e. if it changes the state of the system.

These queries are on the one hand an operation for calculating the total cost of the items in a shopping cart, on the other hand a search operation on the products in a catalog. They are specified in Fig. 4(a) and 4(b) by OCL body constraints, which are an adequate choice because of the freeness from side effects in OCL. The total cost is calculated by iterating over the items in a shopping cart and adding the number mulitplied by the price of a single product, while searching is realised by the select operation predefined in OCL.

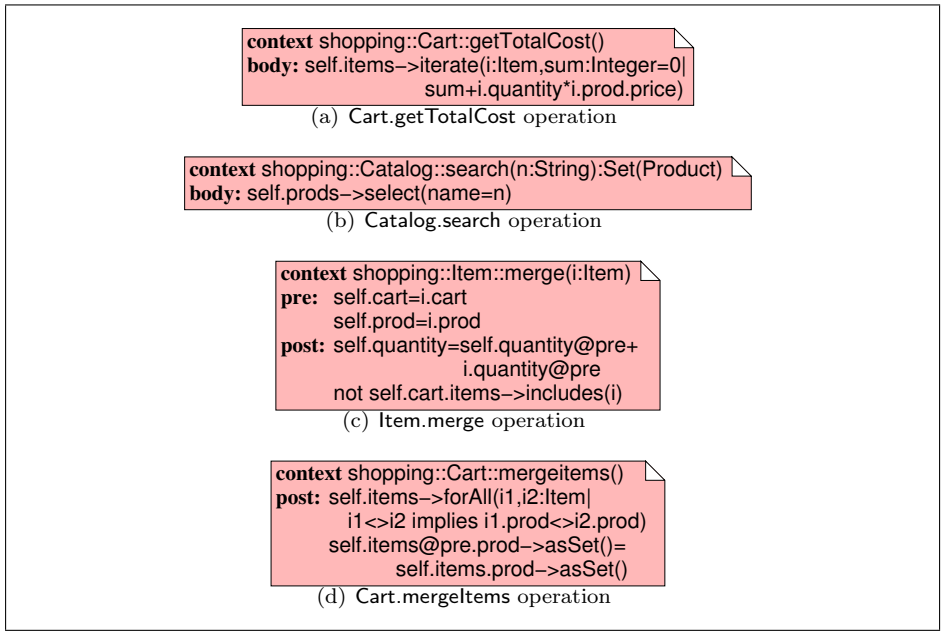


Figure 4: OCL constraints for the example

In contrast to the body constraints, the OCL pre-post conditions in Fig. 4(c) and 4(d) do not model the corresponding operations completely but merely specify some requirements. These operations are intended to merge items in the shopping cart, which reference the same product.

In order to model operations for local changes of the system, CUML uses a transformation rule notation as shown in Fig. 5. The advantage of this notation over activity diagrams and similar techniques is the declarative nature of transformation rules, which make the effects of operations on the object configuration readily visible by showing the relevant part of the system before the operation on the left-hand side and after the operation on the right-hand side.

The constructor for products in Fig. 5(a) creates a new product with the given name and price and adds the result to the given catalog, while the constructor for items in Fig. 5(b) creates a new item in the given shopping cart associated to the given product with the given quantity. The third rule in Fig. 5(c) models the behaviour of the operation for merging items in a shopping cart. The left-hand side of this rule shows that the operation is only applicable if the cart and the product of the item on which the operation is called and the parameter item are identical. The right-hand side then models the effect of the operation, where the parameter item is deleted and the quantities of self and parameter item are accumulated in self.

While OCL constraints follow a functional side-effect free paradigm suitable for modelling queries and transformation rules realise a declarative approach adequate for local changes of the system, the third behavioural technique we want to use in CUML is an imperative one, which can be used to model algo-

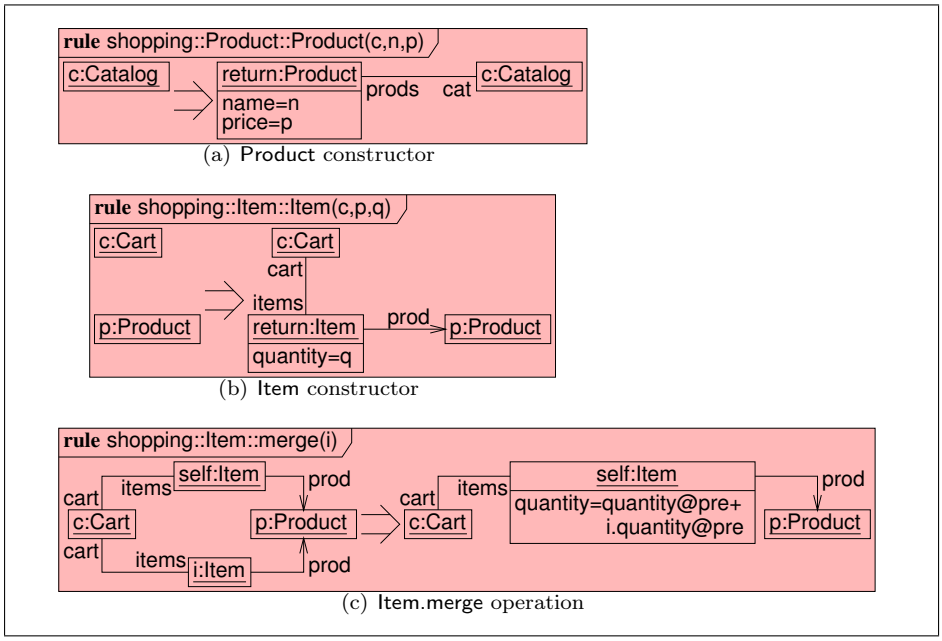


Figure 5: Transformation rules for the example

rithmic operations. In Fig. 6, we see a structured flowchart for the operation merging all items with identical products in a shopping cart.

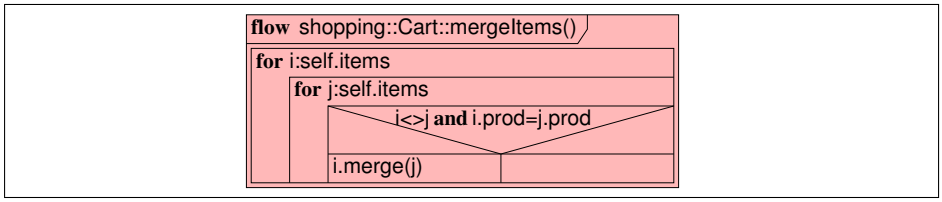


Figure 6: Flowchart for Cart.mergeItems operation

These structured flowcharts are a derivation of the flowcharts of Nassi and Shneiderman in [6]. In the example we can see that one of the advantages over graph-like techniques like activity diagrams and state machines is the visibility of the algorithmic structure with two nested iterations and a decision. Such algorithmic details are complicated to model adequately in graph-like modelling techniques.

However, we also want to integrate activity diagrams into CUML, because they are an adequate modelling technique for operations, which are less algorithmic, but rather workflow-like, though we do not have an example for such an operation in our small example.

In the next section we sketch, how the different techniques presented in this section may be integrated into a common object-oriented low-level language.

4 L3 – The Low-Level Language

As already stated in the abstract concept in Sect. 2, the low-level language will be subdivided into structural, descriptive, and constructive aspects. We will use a notation close to the UML notation for low-level models. In an implementation of this concept, low-level models would probably not be visualised at all, but rather only used in the backend, so that the developer only has to deal with (C)UML diagrams.

In Fig. 7, the structural part of the low-level model of the example is shown. It is still very similar to the class diagram, but abstracts from properties, which have to be fulfilled by the implementation rather than being ensured directly by the structure. Associations and properties are both translated into attributes, where the inverseness of the association ends will be required in the descriptive part. Likewise multiplicities are only considered in deciding if an attribute or parameter is a reference to a single object or a collection of objects, representing the multiplicities `0..1` and `0..*`, respectively. Other multiplicities will also be considered in the property model.

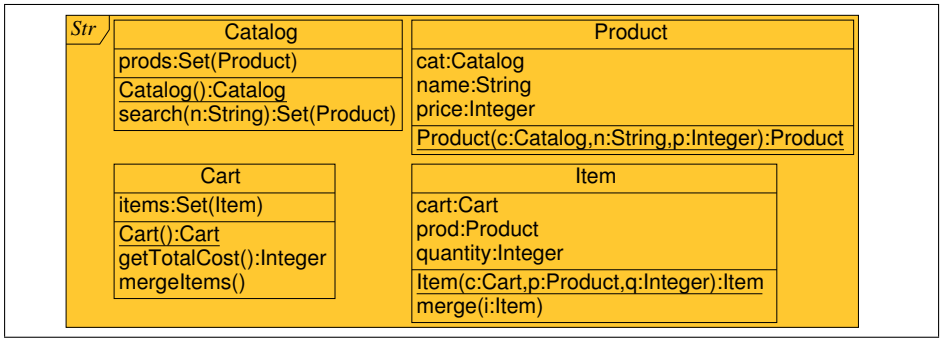


Figure 7: Structural low-level model of the example

In order to be able to represent all kinds of collections available in the UML, the low-level language should support sets, bags, sequences, and ordered sets. These collection types are retained in the low-level model rather than being flattened, because the code generation is likely to be able to translate them to structures provided by the underlying platform, e. g. the Java Collection Framework, directly.

The descriptive model in Fig. 8 contains translations of the association and multiplicity constraints from the class diagram and the OCL pre-post conditions, where all these are translated into first-order logic. A special keyword **query** is introduced to capture the property of an operation being a query.

Associations and multiplicities are invariants of object configurations. In order to be able to reuse them, abbreviations for these invariants are defined in the upper part of Fig. 8. Invariants are added to all pre and post conditions of all operations (except for the queries, since they preserve the object configuration, anyway). Moreover, the OCL pre-post conditions are translated into first-order

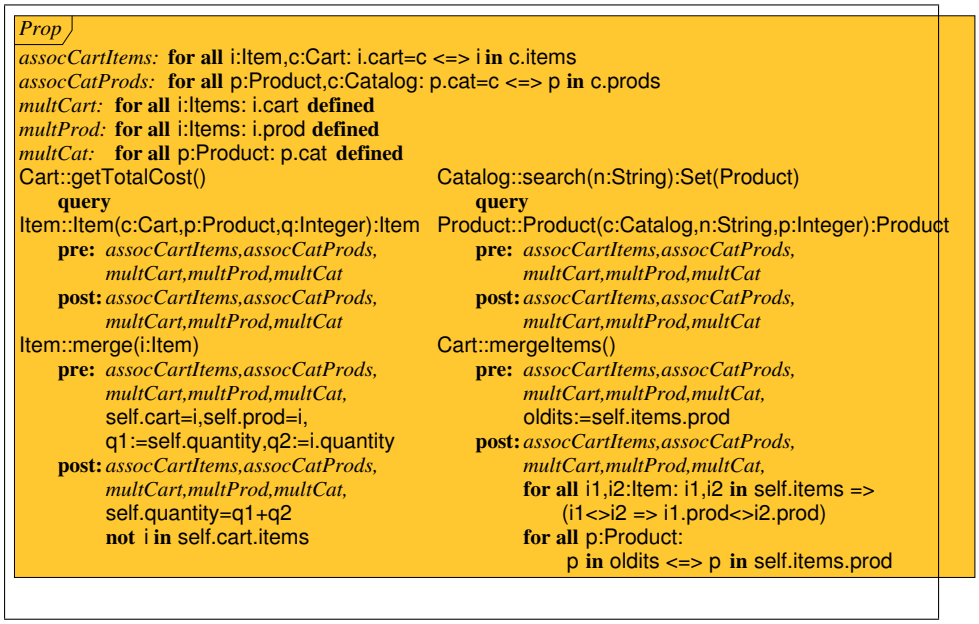


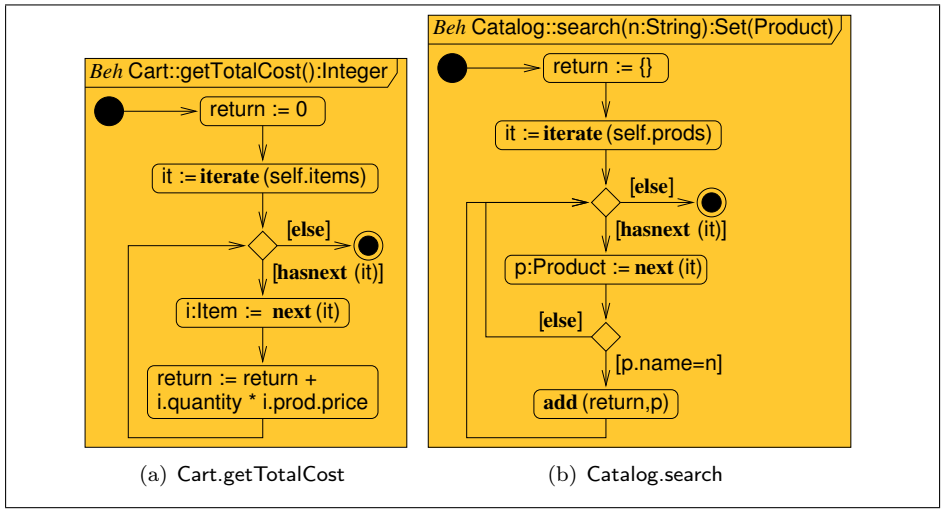
Figure 8: Descriptive low-level model of the example

logic, where the OCL @pre expressions are interpreted by using variables bound at pre time and reused at post time.

Now, the different behavioural techniques are translated into the low-level language, where constructive low-level models are visualised in the style of UML acitivity diagrams. But, while complex activity diagrams might employ a lot of features and "syntactic sugar" like e. g. complex object flows or OCL constraints as guards, the constructive low-level models may only use a very limited set of atomic actions, which we will see in the following examples.

In Fig. 9, the translations of the OCL body constraints are depicted. Since both operations iterate over a collection of objects, we need atoms to support this iteration. The actions **iterate**, **hasnext**, and **next** serve this purpose. When generating code from a low-level model, these should map relatively easy to iterator concepts on the target platform. The getTotalCost operation in Fig. 9(a) only uses assignments to the return variable and simple arithmetic calculations in addition to the iteration actions, while the search operation in Fig. 9(b) also uses atoms {} and **add** for manipulating a set of objects.

In Fig. 10, the translations of the transformation rules from Fig. 5 are given. Here, we see that the low-level language also supports parallelism. This can be employed when generating code for a target platform also supporting parallelising independent activities, like e. g. the .NET platform. The possible parallelism arises from the fact, that transformation rules do not prescribe a certain order, in which the changes to the object configuration shall be applied. The exact model transformation from the high-level transformation rules to the low-level language is out of the scope of this paper.



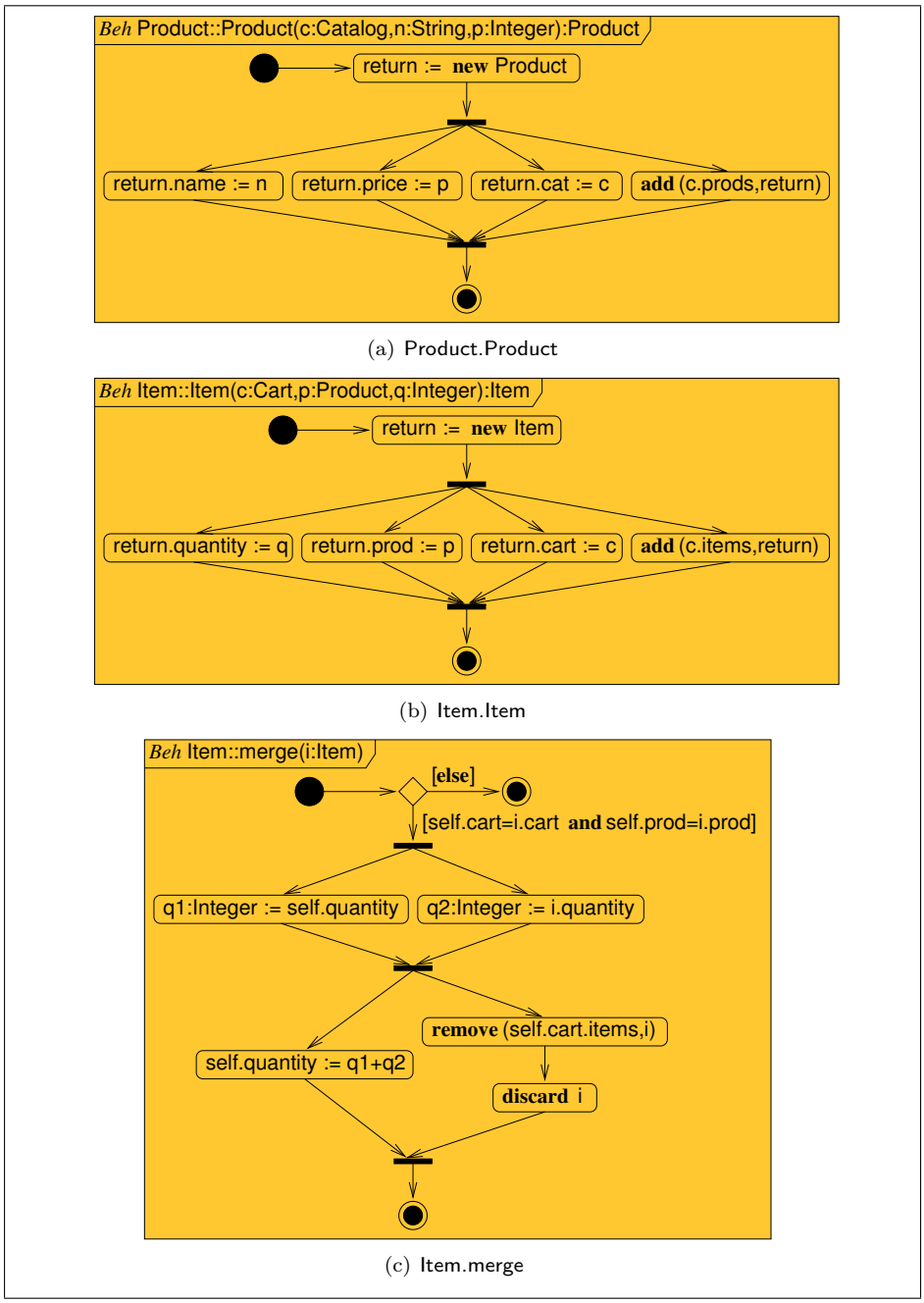
(a) Cart.getTotalCost (b) Catalog.search

Figure 9: Contructive low-level model for OCL constraints

The two constructor models in Fig. 10(a) and 10(b) make use of the atom **new**, which allocates a new uninitialised object. In Fig. 10(c), where the merge operation is modelled, we see how patterns in the left-hand side of a rule are translated into a decision constraining the applicability of the rule. Moreover, we see that attributes, which are accessed with an @pre notation in the right-hand side of a rule, are saved into variables prior to applying the changes. The merge operation also makes use of an additional atom remove for the manipulation of object sets and the **discard** atom for ending the life cycle of an object. The latter may be ignored, when generating code for a platform with garbage collection, but may be useful on other platforms like C++, where objects have to be destroyed explicitly.

Finally, in Fig. 11, the translation of the small flowchart from Fig. 6 is given. It uses two nested iterations over the same set and as a last kind of atomic action a call to another operation.

Now, we have seen the whole low-level model of our small example. As stated before, its purpose is the facilitation of code generation and verification. The suitability of the constructive model parts for code generation should be quite obvious, since the atoms used in the model are quite close to the basic instructions on common object-oriented platforms or the operations available on their collection implementations, respectively. For verification, a first approach could be the definition of a Hoare-like calculus, such that the per-operation requirements in the descriptive model could be verified along the structure of the corresponding operations.



(a) Product.Product

(b) Item.Item

(c) Item.merge

Figure 10: Contructive low-level model for transformation rules

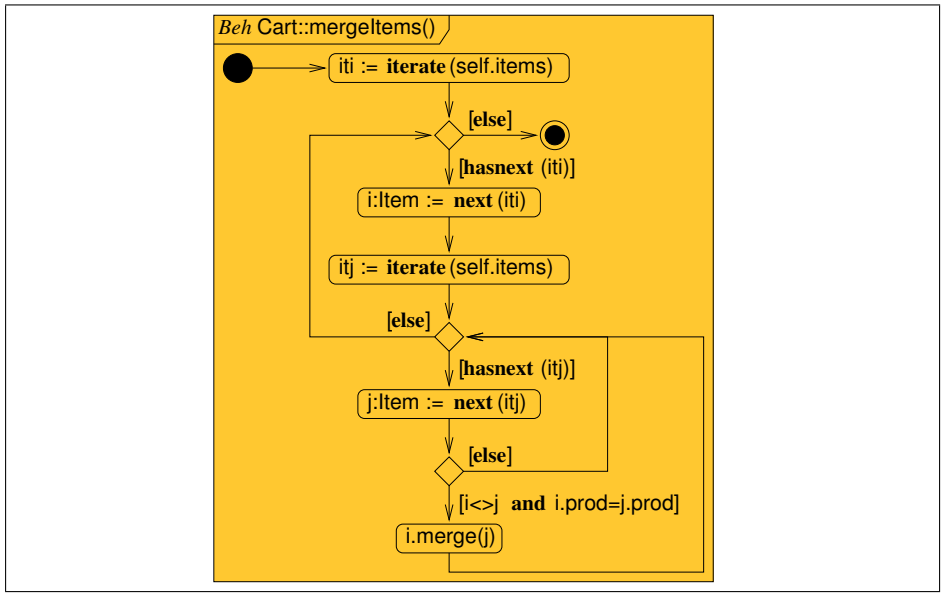


Figure 11: Constructive low-level model for flowchart

5 Summary, Related and Future Work

In this paper an abstract concept for the integration of complex multi-paradigm modelling techniques was proposed. It is build around the idea of translating complex, user-friendly models into a minimalistic, machine- and theory-friendly low-level language. This low-level language can be divided into structural, descriptive, and constructive elements, which is useful to ease code generation and verification.

As an instantiation of the concept, a complex modelling technique based on the UML and an object-oriented low-level language were sketched. The UML-based modelling technique uses behavioural techniques from different paradigms, namely functional OCL constraints, declarative transformation rules, and imperative flowcharts, while in the low-level language all these techniques are translated into the same style of low-level action flows.

A lot of formalisms have been proposed as rigorous foundations for complex modelling techniques. For example, a mapping from UML 1.3 activity diagrams to abstract state machines is proposed by Börger et al. (see [1]). Other approaches try to use process algebras as a semantic domain. While these proposals have the advantage of readily available verification and analysis tools, they need a lot of encoding to represent complex structures using the means of the algebraic formalisms, which reduces the intuition behind the translations. Moreover, the approach in this paper targets generation of code for the behavioural models, where algebraic formalisms would be a detour, given that UML activities already have a rather imperative structure.

In [9], Störrle and Hausmann evaluate the possibility to use Petri nets as a semantic domain for UML activity diagrams, which is also suggested by the UML specification itself. They come to the conlusion, that, in order to integrate all possibilities of activity diagrams, different variants of Petri nets would have to be integrated in a new formalism, which would then have neither tools nor theory available. This observation may also serve as a reason for deriving the new low-level language proposed in this paper, which is specifically designed to capture the features of complex object-oriented systems.

A first point of future work is the establishment of meta-models for both CUML and the low-level language and the development of a tool-supported model transformation between these models, which implements the translation sketched in this paper. Furthermore, the implementation of a proof-of-concept code generator for the low-level language is planned.

In order to retain compatibility with the widely used UML standard and other UML tools, we will try to formalise CUML as a UML profile, so that class diagrams, OCL expressions and activity diagrams are restricted to the subclass we consider, and transformation rules and flowcharts are realised as concrete syntaxes for special kinds of UML activities.

On the theoretical side, a formal semantics for the low-level language will be developed, which will also allow to reason about compositionality of low-level and complex models. For this purpose, concepts of visibility and imports of model elements will be introduced into the modelling techniques. A formal semantics will also allow the development of formal refinements and refactorings of models.

Moreover, formal analysis methods and tools should be developed for the low-level language, where exisiting work on verification techniques for graph transformation systems could serve as a basis, since the formal semantics will execute the low-level models by rules, which are very similar to graph rewriting rules.

Finally, the extension of the complex modelling technique with domain-specific extensions is an interesting line of future research. These extensions should be possible rather easily, because of the modular structure of the approach. The low-level language can be left unchanged and the new domain-specific language only has to be translated into this fixed low-level language, where new domain-specific languages can either be translated into corresponding UML diagrams providing for an indirect integration, formulated as an additional UML profile with its own translation into the low-level language, or equipped with a meta-model independent of the UML meta-model.

References

[1] Börger, E., A. Cavarra and E. Riccobene, *An ASM semantics for uml activity diagrams*, in: *Algebraic Methodology and Software Technology, AMAST 2000*, LNCS **1816**, Springer, 2000 pp. 293–308.

[2] Braatz, B., *A rule-based, integrated modelling approach for object-oriented systems*, in: *Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*, ENTCS (2006), to appear.

[3] Braatz, B. and A. R. Kniep, *Integration of object-oriented modelling techniques* (2006), draft version available from `http://tfs.cs.tu-berlin.de/~bbraatz/papers/BK06-TR.pdf`.

[4] Ehrig, H., K. Ehrig, U. Prange and G. Taentzer, "Fundamentals of Algebraic Graph Transformation," Monographs in Theoretical Computer Science, Springer, 2006.

[5] Große-Rhode, M., "Semantic Integration of Heterogeneous Software Specifications," Monographs in Theoretical Computer Science, Springer, 2004.

[6] Nassi, I. and B. Shneiderman, *Flowchart techniques for structured programming*, ACM SIGPLAN Notices **8** (1973), pp. 12–26, `http://www.geocities.com/SiliconValley/Way/4748/nsd.html`.

[7] Object Management Group, "UML Superstructure Specification, v2.0," (2005), `http://www.omg.org/cgi-bin/doc?formal/05-07-04`.

[8] Object Management Group, "Object Constraint Language, v2.0," (2006), `http://www.omg.org/cgi-bin/doc?formal/2006-05-01`.

[9] Störrle, H. and J. H. Hausmann, *Towards a formal semantics for UML 2.0 activities*, in: *Software Engineering, SE 2005*, 2005, available from `http://www.pst.informatik.uni-muenchen.de/~stoerrle/`.