# Towards an Institutional Framework for Heterogeneous Formal Development in UML
## — A Position Paper —

Alexander Knapp[1], Till Mossakowski[2], and Markus Roggenbach[3]

[1] Universität Augsburg, Germany
[2] Otto-von-Guericke Universität Magdeburg, Germany
[3] Swansea University, UK

**Abstract.** We present a framework for formal software development with UML. In contrast to previous approaches to equipping UML with a formal semantics, we propose an institution-based heterogeneous approach. This can express suitable formal semantics of the different UML diagram types directly, without the need to map everything to one specific formalism (let it be first-order logic or graph grammars). We provide ideas how different aspects of the formal development process can be coherently formalised, ranging from requirements over design and Hoare-style conditions on code to the implementation itself. The framework can be used to verify consistency of different UML diagrams both horizontally (e.g., consistency among various requirements) as well as vertically (e.g., correctness of design or implementation w.r.t. the requirements).

**Keywords:** UML, heterogeneous formal methods, institutions.

## 1 Introduction

In Martin Wirsing's research, the development of applicable formal methods plays an important role. Martin has examined real-world modeling and programming languages like UML and Java, and has studied suitable formal theories that can lead to increased trustworthiness due to the possibility of formal verification. In this work, we build on and extend the first author's joint work with Martin Wirsing about views in software development, heterogeneous semantics of UML, and multi-modeling languages [24,34,3,7]. In fact, the integration of different, heterogeneous views on and in a software system is a continual theme in Martin's research, be it for data bases [10], multimedia systems [20], or mobile systems [22]. We also build on joint work of Martin Wirsing with the second and third author on the design of CASL [31]; indeed, in the 1990s, Martin hosted some of the CASL/COFI meetings and provided valuable input for the CASL design. For the present paper, CASL forms a building block for the distributed ontology, modeling and specification language (DOL), which has evolved from a generalisation of CASL to heterogeneous specifications.

In the industrial design of software for critical systems, the Unified Modeling Language (UML) is an often used development mechanism. In aerospace industry, e.g., the company Aero Engine Controls (AEC)[1] uses the UML to define the software architecture

[1] AEC is the former name of Rolls-Royce Control and Data Services, `http://www.controlsdata.com`.

of aeroplane engine controllers through various levels of abstraction from a layered architecture overview to a detailed class, operation and attribute definition of the software components. This model is then used for code generation. Typically, the software components developed are either reactive or logic-based and/or stateful in nature, where notations such as UML state diagrams are used to define the required behaviour [17]. Micro-controllers in the automotive sector or, in the medical sector, ventricular assistance devices exemplify further uses of UML in the development of critical systems.

The UML is an OMG standard [32], which describes a language family of 14 types of diagrams, of structural and behavioural nature. A typical development by AEC involves about eight different UML diagrams [18]. The OMG specification provides an informal semantics of nine sub-languages in isolation. The languages are mostly linked through a common meta-model, i.e., through abstract syntax only. This situation leads to a gap between standards' recommendation to apply formal methods, and current industrial practice, which by using the UML lacks the semantic foundations to apply such methods. One common approach to deal with this gap is to define a comprehensive semantics for the UML using a system model, e.g., [4,5]. However, this is a thorny business, as every detail has to be encoded into one, necessarily quite complex semantics. Furthermore, such an approach has difficulties to cater for UML's variations of usage, leading to company or domain-specific variations. On the other hand, there are many approaches like component models (e.g. [16,2]) which are based on sound theory, but do not meet the main goal of the present work: to set up a framework in which one can eventually capture precisely the semantics and semantic variation points of the UML standard (including all its idiosyncrasies), such that a holistic model-driven development in UML can be complemented with rigorous consistency and verification conditions.

In this position paper, we outline a competing approach by providing a heterogeneous semantics, where we extend [7] by considering a subset of diagrams rich enough for industrial use, adding components and composite structures as well as behavioural and protocol state machines. We substantiate this claim by a small case study that is modelled holistically, using a variety of different diagram types. We distinguish between diagrams for properties, types and instances, where we express the meaning of a model in a sub-language/diagram directly in an appropriate semantic domain. We also distinguish diagrams for requirements, design, deployment and implementation. This degree of completeness in the coverage of the development process has not been achieved so far.

We further systematically identify meaningful connections given by the abstract syntax of the UML specification or which can be gleaned from its semantic description. The separation between the meaning of the individual diagrams and their relation allows our approach to be adopted by different methodologies, for instance an object-oriented approach or a component-based one.

## 2 Methodology

Our overall aim is to provide qualified formal methods for dependable software design for critical systems, especially embedded, reactive systems based on UML. We illustrate

this aim by first giving a case study in UML and then discussing desirable checks for consistency between the various artefacts. We then explain how languages and UML diagram types involved in a software design can be viewed as types, instances, and properties, either on the modelling level or on the implementation level. Finally, we address the topic of semantic variation points.

### 2.1 ATM Case Study

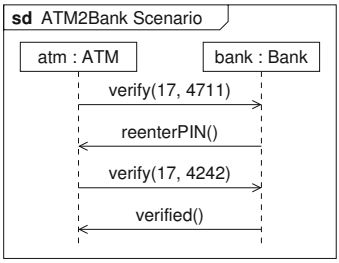
In order to illustrate our heterogeneous semantics and method, we present as a running example the design of a traditional automatic teller machine (ATM) connected to a bank. For simplicity, we only describe the handling of entering a card and a PIN with the ATM. After entering the card, one has three trials for entering the correct PIN (which is checked by the bank). After three unsuccessful trials the card is kept.

*Requirements.* Figure 1(a) shows a possible *interaction* between an atm and a bank, which consists out of four messages: the atm requests the bank to verify if a card and PIN number combination is valid, in the first case the bank requests to reenter the PIN, in the second case the verification is successful.

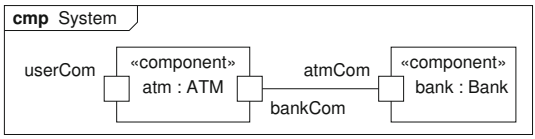
The composite structure of the ATM-bank system is specified in the *component diagram* in Fig. 1(b). In order to communicate with a bank component, the atm component has a *behaviour port* called bankCom and the bank component has a behaviour port atmCom. Furthermore, atm has a port userCom to a user. Figure 1(c) provides structural information in the form of the interfaces specifying what is provided at the userCom port of the atm instance (UserIn) and what is required (UserOut). An interface is a set of operations that other model elements have to implement. In our case, the interface is described in a *class diagram*. Here, the operation keepCard is enriched with the OCL constraint trialsNum >= 3, which refines its semantics: keepCard can only be invoked if the OCL constraints hold.

The communication protocol required between the atm's port bankCom and the bank's port atmCom is captured with a protocol state machine, see Fig. 1(d): After a verify message from atm, either verified or reenterPIN can be sent as a reply from bank; atm can request a card to be rendered invalid whenever it has not just asked to verify a card and a PIN.

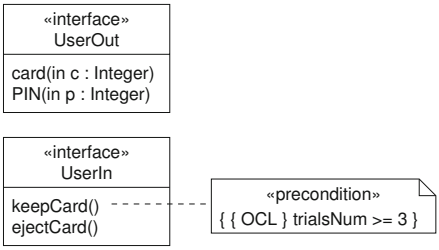
*Design.* The dynamic behaviour of the atm component is specified by the *state machine* shown in Fig. 1(e). The machine consists of five states including Idle, CardEntered, etc. Beginning in the initial Idle state, the user can *trigger* a state change by entering the card, where we indicate that the event card has to occur at port userCom. This has the *effect* that the parameter c from the card event (declared for operation card in Fig. 1(c)) is assigned to the cardId in the atm component. Entering a PIN triggers another transition to PINEntered. Then the ATM requests verification from the bank using its bankCom port. The transition to Verifying uses a *completion event*: No explicit trigger is declared and the machine autonomously creates such an event whenever a state is completed, i.e., all internal activities of the state are finished (in our example there are no such activities). In case the interaction with the bank results in reenterPIN, and the *guard* trialsNum < 3 is true, the user can again enter a PIN. If, on the other hand, trialsNum >= 3, the user



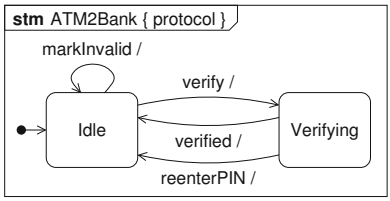

(a) Interaction



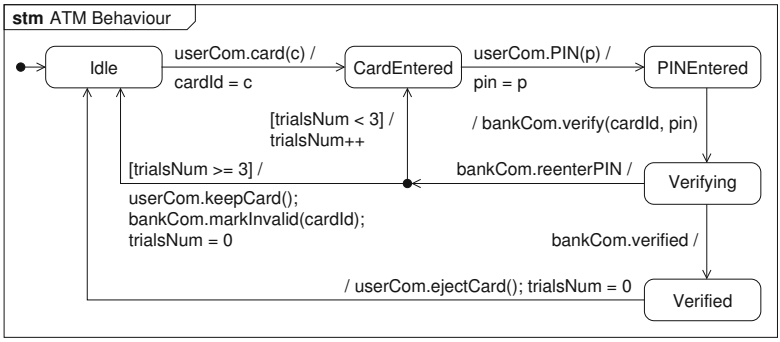

(b) Composite structure




(c) Interfaces




(d) Protocol state machine




(e) State machine

**Fig. 1.** ATM example

is informed that the card is kept by userCom.keepCard(), and the bank is informed to render the card invalid by bankCom.markInvalid(cardId).

*Deployment.* Although the UML allows to specify which component instances should run on which computational resources and which physical connectors transport their communication, we currently restrict ourselves to specifying which component and connector instances have to be present at system start. This initial configuration is described in a *composite structure diagram*, see Fig. 1(b) now interpreted at the component instance level. In particular, instances of the behaviour specifications for the components have to be deployed accordingly. The starting configuration could change

over time by changing the wiring of the connectors to ports as well as creating or deleting component instances.

*Code.* The state machine shown in Fig. 1(e) can be implemented in the programming language C, enriched with pre-/post-conditions written in the ANSI/ISO C Specification Language (ACSL). The code example below shows how the event card is encoded as a C function, where the ACSL annotations ensure that the system is in some defined state and that the number of trials to re-enter the PIN is smaller than three.

```
typedef enum states {
  EMPTY = 0, IDLE = 1, CARDENTERED = 2,
  PINENTERED = 3, VERIFYING = 4, PINVERIFIED = 5
} states_t;
int cardId = 0; int pin = 0; int trialsNum = 0;
states_t state = EMPTY;

/*@
requires state != EMPTY; requires trialsNum <= 3;
ensures state != EMPTY;  ensures trialsNum <= 3;
@*/
void card(int c) {
  switch (state) {
    case IDLE:
      cardId = c;
      state = CARDENTERED;
      break;
    default:
  }
}
```

## 2.2 Consistency and Satisfaction

A typical software devolopment for a critical system will cover requirements, design, deployment, and code using several sub-languages and diagram types summarised in Fig. 2 and classified towards a software design process. In fact, class and component diagrams also will be used for expressing requirement, as already illustrated by our small case study.

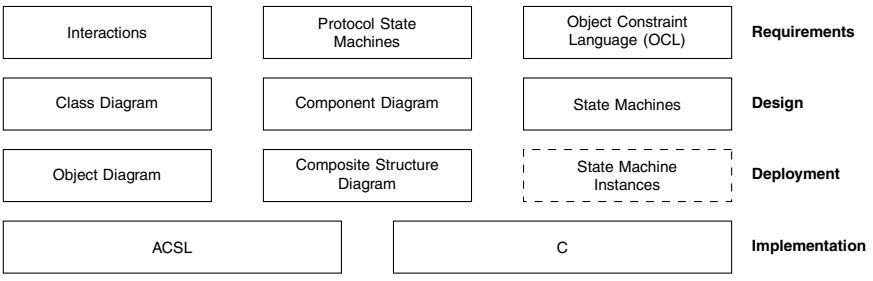



**Fig. 2.** Methodological use of languages and diagrams considered

It is desirable to detect inconsistencies at an early stage of the development in order to ease corrections and avoid costly re-engineering at a late stage (e.g. during the implementation phase). While there are some tools providing static inconsistency checks based on UML's meta-model, only few works consider dynamic checks, and generally only for specific UML diagram types, e.g. [25].

The analysis of UML models can proceed either horizontally within the requirements or within the design level checking for consistency within the level, or vertically checking for satisfaction between these two levels, see Fig. 4. A typical horizontal consistency check on the requirements level would ask if the sequential composition of actions in an interaction diagram is justified by an accompanying OCL specification. A typical vertical satisfaction check between the requirements and the design level would ask if the behaviour prescribed in an interaction diagram is realisable by several state machine (instance)s cooperating according to a composite structure diagram. The notion of a state machine instance will be explained in the next section. Code generation transforms a UML logical design to code templates with semantic annotations in the form of pre-/post-conditions and invariants. If the templates are completed satisfying the semantic annotations, it is guaranteed that the resulting code is a correct model of the logical design and therefore, by the vertical checks, also for the requirements.

Concerning Fig. 1, there are the following (succeeding) consistency and satisfiability checks: speaking horizontally, the interaction in Fig. 1(a) can be realised by the protocol state machine in Fig. 1(d), which in turn (vertically) refines to the behavioural state machine in Fig. 1(e), which in turn (vertically) refines to the C code shown at the end of Section 2.1.

A simple example for a failing horizontal check among several requirements is the interaction in Fig. 3, which cannot be realised by the protocol state machine in Fig. 1(d).

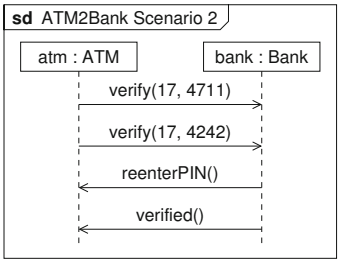



**Fig. 3.** Interaction that cannot be realised by the protocol state machine in Fig. 1(d)

### 2.3 Levels and Views

The languages and UML diagram types that we consider are restructured into different levels and views (according to the role they play in respective the languages) in Fig. 4. On the *modelling level* we use parts of the UML and the Object Constraint Language (OCL). On the *implementation level* we currently employ the programming language C and ACSL. It is left for future work to also include a proper object-oriented language such as Java together with some specification formalism.

In the *types view* of the modelling level we look at class diagrams for modelling data; component diagrams for modelling components; and state machines for specifying dynamic behaviour. These diagrams can be instantiated in the *instance view* using composite structure diagrams for showing component configurations; and object diagrams for showing concrete data. Although they are not present in UML, we also have added state machine instances (in a dashed box). Constraints on the models can be specified in the *properties view* using interactions, i.e., sequence diagrams or communication diagrams, for prescribing message exchanges between components and objects; protocol state machines for specifying port behaviour; and the OCL for detailing the behaviour of components and objects in terms of invariants and method pre-/post-conditions.

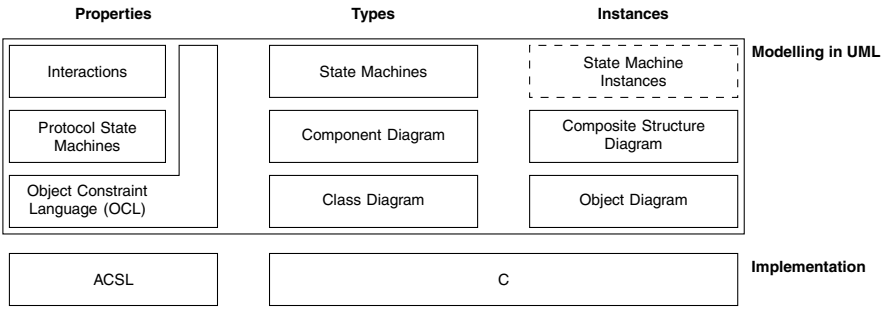



**Fig. 4.** Languages and diagrams considered

## 2.4 Semantic Variation Points

The UML specification uses the notion of "semantic variation point" whenever she does not want to fix or enforce a particular meaning for a construct, but sees room for variable but valid interpretations useful in different contexts. Examples of such semantic variation points include the behaviour of an operation invocation when a pre-condition is not satisfied; the compatibility of connectable elements, like components; the forwarding of requests at a port with several outgoing connectors; the ordering of events in event pools; the time intervals between event occurrence, event dispatching, and consumption; or the reception of an event in an unexpected situation for a protocol state machine. Different domains or implementation technologies will require different resolutions, like whether message overtaking is possible in a middle-ware. Additionally, some resolutions may enable particular validation or verification techniques, e.g., when using multi-sets or bounded queues for event pools.

However, the specification does not show clear-cut means to resolve these variation points: Not surprisingly, no parameterised semantics is explained where the resolution of a semantic variation point would simply amount to setting a particular value — which, in fact, would be quite hard more often than not. More embarrassingly, also no dedicated syntactic means are provided for at least specifying that a particular meaning is intended by the use of a feature subject to semantic variation. For the syntactical side, the most common, though rather *ad hoc*, resort is to employ stereotypes to express that,

say, the event pool for a state machine is to be realised as a queue, or that the violation of an operation's precondition will result in an error. For the semantical issues, the use of a comprehensive system model approach would require to corral all possible interpretations of a semantic variation point into a single common ground using, e.g., loose specifications.

By contrast, heterogeneous institutional semantics offer the additional possibility to provide particular stand-alone semantics for different resolutions of semantic variation points. The mutual effects of combining different resolutions can be identified and consistent resolutions can be plugged together. Also, different choices for semantic variation points can be related via abstraction maps. For example, it is easy to design an institution comorphism that abstracts the labelled transition system semantics of state machines to a trace semantics. Depending on the choice of abstraction map, the institution-independent notion of refinement [30,8] then will lead to refinement up to trace equivalence of refinement up to bisimilarity.

## 3 UML as a Basis for Heterogeneous Formal Methods, Using Institutions

In this section, we will provide some semantic foundations for model based specification and design using a heterogeneous framework based on Goguen's and Burstall's theory of institutions [14]. We handle the complexity of giving a coherent semantics to UML by sketching several institutions formalising different diagrams of UML, and several institution translations (formalised as so-called institution morphisms and comorphisms) describing their interaction and information flow. The central advantage of this approach over previous approaches to formal semantics for UML (e.g. [25]) is that each UML diagram type can stay "as-is", without the need of a coding using graph grammars (as in [12]) or some logic (as in [25]). This also keeps full flexibility in the choice of both the development method and the verification mechanisms. The formalisation of UML diagrams as institutions has the additional benefit that a notion of refinement comes for free, see [30,8]. The exact nature of the thus obtained refinement relation depends on the semantic choices that have been made.

This systematic coverage in a single semantic based meta-formalism is unique. We discuss semantic links in the form of institution (co-)morphisms, that, on the one hand, provide the basis for correct model transformations and validations, and on the other hand give rise to an integrated semantic view (via the so-called Grothendieck institution [9,26]) on the identified UML subset as well as the target implementation languages. Institution theory provides an adequate abstraction level for such a semantic integration. The framework is flexible enough to support various development paradigms as well as different resolutions of UML's semantic variation points. This is the crucial advantage of the proposed approach to the semantics of UML, compared to existing approaches in the literature which map UML to a specific global semantic domain in a fixed way.

### 3.1 Institutions and Their (Co)Morphisms

Institutions [14] are an abstract formalisation of the notion of logical system. Informally, institutions provide four different logical notions: signatures, sentences, models and

satisfaction. Signatures provide the vocabulary that may appear in sentences and that is interpreted in models. The satisfaction relation determines whether a given sentence is satisfied in a given model. The exact nature of signatures, sentences and models is left unspecified, which leads to a great flexibility. This is crucial for the possibility to model UML diagrams types (which do not at first sight look like logics) as institutions.

An important feature of institutions is the presence of signature morphisms, which can be seen as mappings between signatures. Sentences can be translated along signature morphisms, and models reduced *against* signature morphisms. The satisfaction condition states that satisfaction is invariant under change of notation and enlargement of context (along a signature morphism). For details, we refer to [14,33].

It is possible to define standard logical notions like logical consequence, logical theory, satisfiabilty etc. as well as languages for structured specification and refinement in an institution-independent way [33].

For relating institutions in a semantics preserving way, we consider institution morphisms [14]. Given institutions $\mathsf{I}$ and $\mathsf{J}$, an *institution morphism* consists of (i) a mapping from $\mathsf{I}$-signatures to $\mathsf{J}$-signatures (also for signature morphisms); (ii) a mapping from $\mathsf{J}$-sentences to $\mathsf{I}$-sentences; and (iii) a mapping from $\mathsf{I}$-models to $\mathsf{J}$-models. Again, there is a satisfaction condition governing these mappings. Dually, we consider institution comorphisms [15]. They are like institution morphisms, except that the direction of sentence and model translations are reversed.

The methodological need for these two kinds of mappings between institutions will be explained in Sect. 3.4 below. Both morphisms and comorphisms also come in a "semi" variant (i.e. semi-morphisms and semi-comorphisms) [15]. These omit both the sentence translation and the satisfaction condition. Semi-(co-)morphisms can provide a model-theoretic link between institutions that are too different to permit a sentence translation, e.g. OCL and state machines.

### 3.2 Heterogeneous Formal Semantics of Languages and Diagrams

Carrying out our program of institutionalising UML is ongoing work. In this position paper, we review this work and sketch how it can be extended to all diagrams in Fig. 2.

Building on existing UML semantics, see [25] for an overview, we want to turn UML's sub-languages and diagram types into separate institutions[2]. For substantial fragments of several UML diagram types, we have already provided a formalisation as institutions:

**Class diagrams** In [7], we have sketched an institution for class diagrams, which has been detailed in [19]. It includes a construction for stereotypes.

**Component diagrams** form an institution similar to that for class diagrams. The main difference are the connector and port types, which however are quite similar to associations.

**Object diagrams** are essentially reifications of models of class diagrams.

**Composite structure diagrams** are similar to object diagrams. The main difference are the connectors, which however are quite similar to the links of object diagrams.

[2] Alternative or complementing approaches like statecharts instead of UML state machines or SysML/MARTE components could be added to this family of institutions.

**Interactions** In [7], we have sketched an institution for interactions, as well as their interconnection (also with class diagrams) via institution comorphisms.

**OCL** In [7], we have sketched institutions for OCL. In [6], the OCL semantics is presented in more detail. An institution based on this is in preparation.

**State machines** In [23], we have provided in full detail institutions for UML state machines and protocol state machines (so far only for non-hierarchical states, a generalization to hierarchical states is in preparation). Both institutions are very similar; only their sentences differ in that UML protocol state machines have a post condition instead of an action. Post conditions can also speak about messages being sent (using OCL).

Formalising both C and ACSL as institutions is future work.

### 3.3 Institutional Interaction of Heterogeneous UML Diagrams

We now will discuss how different diagram types can be linked using the institutional approach. A characteristic example is the interplay between class diagrams, component diagrams and state machines. Here, an *environment* institution [23] provides the interface necessary to define state machines. Signatures in this environment institution fix the conditions which can be used in guards of transitions, the actions for the effects of transitions, and also the messages that can be sent from a state machine. The source of this information are the class and component diagrams: The conditions and actions involve the properties available in the classes or components, the messages are derived from the available signals and operations. The sentences of this environment institution form a simple dynamic logic (inspired by OCL). This logic can express that if a guard holds as pre-condition when executing an action, then a certain set of messages has been sent out and another guard holds as post-condition. In particular, this environment institution forms the interface to the outside; different institutions for classes and components can be linked to it via (co-)morphisms.

A family of institutions for *state machines* (and similarly for protocol state machines) is then parameterized over such environment institutions. Using a product construction on the state machine institution, *communicating state machines*, with their linkage described in a composite structure, can be captured. The essential idea behind the product construction is to control the flow of messages in such a way that each message is sent to the correct event pool.

*Example 1.* Consider the composite structure diagram in Fig. 1(b), showing instances atm and bank of the ATM and Bank components, respectively, that are connected through their bankCom and atmCom ports. In execution, atm and bank will exchange messages, as prescribed by their state machines, and this exchange is reflected by the product which internalises those events that are part of the common signature. On the other hand, messages to the outside, i.e., through the userCom port are still visible.

### 3.4 Transformations Among UML institutions

Figure 5 gives an overview of the transformations to be developed between the modeling languages, diagram types, and additional languages. We claim that the transformations

in this Figure can be formalised as institution morphisms and comorphisms. An institution morphism (represented by a solid line in the figure) roughly corresponds to a projection from a "richer" to a "poorer" logic, expressing that the "richer" logic has some more features, which are forgotten by the morphism. The main purpose of the institution morphisms is the ability to express, e.g., that an interaction diagram and a state machine are compatible because they are expressed over the same class diagram. Institution morphisms thus enable the formalisation of heterogeneous UML specifications as structured specifications over the Grothendieck institution, a flattening of the diagram of institutions and morphisms [9]. Practically, these structured Grothendieck specifications can be formulated in the distributed ontology, modeling and specification language (DOL), which currently is being standardized in the OMG (see `ontoiop.org` and [28]).

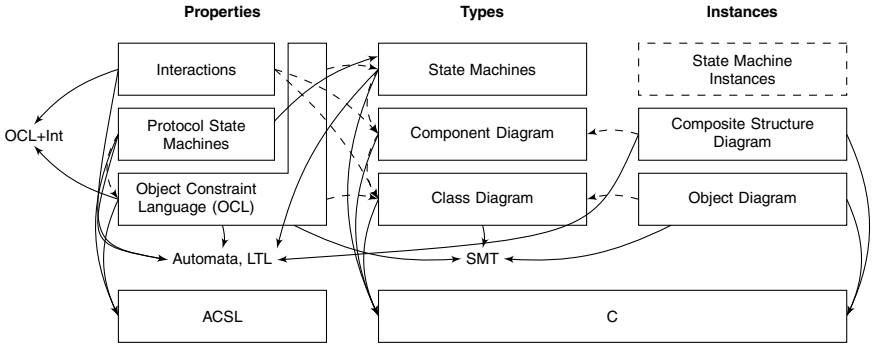



**Fig. 5.** Institution morphisms (dashed arrows) and institution co-morphisms (solid arrows) between the languages and diagrams

By contrast, institution comorphisms (represented by dashed lines in Fig. 5) are often more complex. Roughly, a comorphism corresponds to an encoding of one logic into another one. The purpose of institution comorphisms is threefold: **(1)** to provide a means for expressing the dynamic checks (see below) in the institutional framework, **(2)** to obtain tool support for the various UML diagrams by using comorphisms into tool-supported institutions, and **(3)** to transform UML diagrams into ACSL specifications and C programs.

Dynamic checks and tool support involve additional institutions (also depicted in Fig. 5, but not formalised in detail here) for certain automata, like those used in the model checker SPIN, and satisfiability modulo theories (SMT) provers, as well as linear temporal logic. The modeling language institutions can be embedded into these, paving the way for tool and prover support.

### 3.5 Consistency and Satisfaction, Revisited

The horizontal dimension of the relationship between the different models has to ensure *consistency* of the models, i.e., that the models fit together and describe a coherent

system. The same has to be checked on the implementation level for the consistency between the C program and the ACSL specification; however, here we can reuse existing theory and tools.

There are different kinds of consistency checks on the modelling level: Static checks ensuring type consistency and type correctness between types and instances. Dynamic checks include the properties and one or several cooperating instances or types. Most of the dynamic checks are theoretically undecidable, thus fully automatic tools will not be able to answer all instances. However, in many cases, useful automatic approximations are possible, while in other cases, manual effort may be involved.

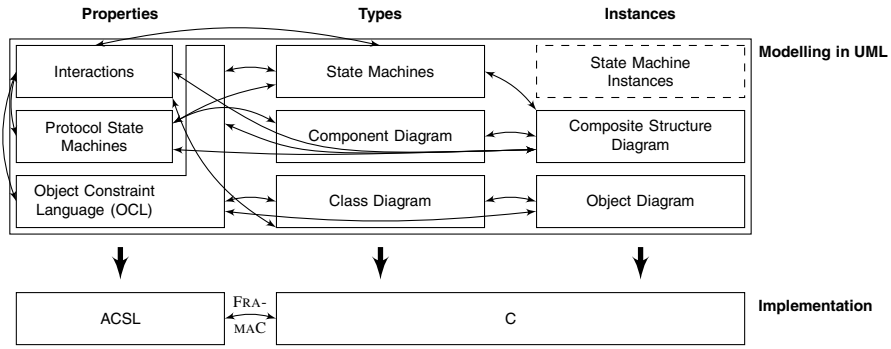



**Fig. 6.** Consistency relations (double-headed arrows) on the modelling and the implementation level; the bold arrows represent the model transformations

Figure 6 gives an overview of useful relations between different kinds of diagrams, along which consistency checks are possible. Here, we list only a few of these. Some useful static checks are:

S1. Does an OCL specification or a composite structure diagram only use the methods of a class diagram?
S2. Does a state machine or an interaction comply with the interfaces referred to in a composite structure diagram?
S3. Does an instance diagram (an object or a composite structure diagram) comply to its corresponding type diagram (a class or a component diagram)?
S4. Do the objects used in an interaction diagram form an object diagram complying to a class diagram?

Here are some useful dynamic checks:

D1. Does an object or composite structure diagram satisfy an OCL invariant? Here we use institution semi-comorphisms from the OCL institution to the object and composite structure diagram institution that turn a model of the object or composite structure diagram into a model of the OCL invariant.
D2. Does a state machine satisfy an OCL invariant or an OCL pre-/post-condition? Here we use a semi-comorphism from the OCL institution to the state machines institution that takes the runs of the state machine and selects those states and transitions that are relevant for the invariant or the method with pre-/post-conditions.

However, the UML does not specify the time point when the OCL post-condition should be evaluated; one possibility is to choose the finishing of the fired transition.

D3. Do the protocol state machines at the ends of a connector of a composite structure diagram fit together? Here we use a comorphism from the protocol state machine institution into a temporal logic institution [13], where we can form the product of the protocol state machines along the connector. However, the precise nature of compatibility may be seen as a "semantic variation point". An important question is the absence of deadlocks and buffer overruns.

D4. Is the sequential composition of methods in an interaction diagram justified by the state machines and/or the OCL specification? For the relation to an OCL specification we use a co-span of institution comorphisms between the interactions institution and the OCL institution [7]. At least two links are possible: In a strict interpretation, for each pair of successive methods in the interaction there must be a state meeting the post-condition of the first method and the pre-condition of the second method. In a more loose interpretation, a sequence of additional method calls, not prescribed but also not excluded by the interaction, must be possible to reach the pre-condition of the second method from the post-condition of the first method. For also considering state machines, the co-span approach is extended by also involving the state machines institution.

D5. Does an interaction comply with the protocol state machines? Here we proceed similarly to the case where an interaction is checked against a state machine and an OCL specification using a comorphism turning the protocol state machine into an OCL specification.

D6. Does a state machine refine the protocol state machines in a component diagram? This is expressible as a heterogeneous refinement from the protocol state machines to the state machine using a semi-comorphism which keeps signatures and models as they are (protocol state machines and state machines only differ in their sentences).

*Example 2.* Though our running example of an ATM machine is quite simple, it is rich enough to illustrate some dynamic checks. The interface UserIn in Fig. 1(c) requires the operation keepCard only to be invoked when the precondition trialsNum >= 3 holds. This property holds for the state machine in Fig. 1(e) thanks to the guard trialsNum < 3 – an illustration of check D1. This property trivially holds for the interaction shown in Figure 1(a) as keepCard is not invoked – an illustration of check D3.

## 4 Tools

The Heterogeneous Tool Set (Hets) [27,29] provides analysis and proof support for multi-logic specifications. The central idea of Hets is to provide a general framework for formal methods integration and proof management that is equipped with a strong semantic (institution-based) backbone. One can think of Hets acting like a motherboard where different expansion cards can be plugged in, the expansion cards here being individual institutions (with their analysis and proof tools) as well as institution (co)morphisms. The Hets motherboard already has plugged in a number of expansion cards (e.g., SAT solvers, automated and interactive theorem provers, model finders,

model checkers, and more). Hence, a variety of tools is available, without the need to hard-wire each tool to the logic at hand. Via suitable translations, new formalisms can be connected to existing tools.

We have just started to integrate first institutions for UML, such as class diagrams, into Hets. In order to obtain proof support for the methodology presented in this paper, beyond the individual institutions, also the morphisms and comorphisms need to be implemented in Hets. Moreover, we plan to connect Hets to the tool HugoRT [21]. HugoRT can, on the one hand, perform certain static checks on UML diagrams. Moreover, it provides transformations of UML diagrams to automata and linear temporal logic formulas, which can then be fed into model checkers like SPIN in order to check certain properties. The crucial benefit of our approach is a clear separation of concerns: verification conditions for consistency and satisfaction checks can be formulated abstractly in terms of the UML institutions and (co)morphisms described above. In a second step, these checks can then be reformulated in terms of specific logics and tools that have been connected to Hets.

## 5 Conclusion

We have outlined an institution-based semantics for the main UML diagrams. Moreover, we have sketched a methodology how consistency among UML diagrams and with implementation languages can be modeled at the institutional level and supported with tools.

Much remains to be done to fill in the details. Semantically, the greatest missing bit is certainly the institutional formalisation of programming languages and their Hoare logics, like C and ACSL, or Java and JML. Here, we want to follow the ideas sketched by A. Tarlecki and D. Sannella [33, Ex. 4.1.32, Ex. 10.1.17] for rendering an imperative programming language as an institution. The semantic basis could be a simplified version of the operational semantics of C. Ellison and G. Rosu [11]. The concepts for institutionalising a Hoare logic like ACSL on the basis of its specification [1] can be similar as for OCL. On the tools side, it is future work to make UML institutions, checks and code generation part of the tool Hets. This will allow also non-experts in institution theory to apply the suggested framework.

**Acknowledgements.** The authors would like to thank the reviewers of their valuable feedback, the editors for their considered handling of this paper, and Erwin R. Catesbeiana for pointing out the many sources of inconsistency.

## References

1. Baudin, P., Cuoq, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. Report. In: CEA 2012 (2012)
2. Bauer, S.S., Hennicker, R.: Views on Behaviour Protocols and Their Semantic Foundation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 367–382. Springer, Heidelbrg (2009)

3. Boronat, A., Knapp, A., Meseguer, J., Wirsing, M.: What Is a Multi-modeling Language? In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 71–87. Springer, Heidelberg (2009)
4. Broy, M., Cengarle, M.V., Grönniger, H., Rumpe, B.: Considerations and Rationale for a UML System Model. In: Lano (ed.) [25], ch. 3, pp. 43–60
5. Broy, M., Cengarle, M.V., Grönniger, H., Rumpe, B.: Definition of the System Model. In: Lano (ed.) [25], ch. 4, pp. 61–93
6. Cengarle, M.V., Knapp, A.: OCL 1.4/5 vs. 2.0 Expressions — Formal Semantics and Expressiveness. Softw. Syst. Model. 3(1), 9–30 (2004)
7. Cengarle, M.V., Knapp, A., Tarlecki, A., Wirsing, M.: A Heterogeneous Approach to UML Semantics. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 383–402. Springer, Heidelberg (2008)
8. Codescu, M., Mossakowski, T., Sannella, D., Tarlecki, A.: Specification Refinements: Calculi, Tools, and Applications (Submitted, 2014)
9. Diaconescu, R.: Grothendieck Institutions. Applied Cat. Struct. 10, 383–402 (2002)
10. Dosch, W., Mascari, G., Wirsing, M.: On the Algebraic Specification of Databases. In: Proc. 8th Int. Conf. Very Large Data Bases (VLDB 1982), pp. 370–385. Morgan Kaufmann (1982)
11. Ellison, C., Rosu, G.: An Executable Formal Semantics of C With Applications. In: Field, J., Hicks, M. (eds.) Proc. 39th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2012), pp. 533–544. ACM (2012)
12. Engels, G., Heckel, R., Küster, J.M.: The Consistency Workbench: A Tool for Consistency Management in UML-Based Development. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 356–359. Springer, Heidelberg (2003)
13. Fiadeiro, J.L.: Categories for Software Engineering. Springer (2005)
14. Goguen, J.A., Burstall, R.M.: Institutions: Abstract Model Theory for Specification and Programming. J. ACM 39, 95–146 (1992)
15. Goguen, J.A., Rosu, G.: Institution Morphisms. Formal Asp. Comp. 13, 274–307 (2002)
16. Hennicker, R., Janisch, S., Knapp, A.: On the Observable Behaviour of Composite Components. In: Proc. 5th Int. Wsh. Formal Aspects of Component Software (FACS 2008). ENTCS 260, pp. 125–153 (2010)
17. Hutchesson, S.: Chief software architect at AEC. Industrial case study outline (2012)
18. Hutchesson, S.: Chief software architect at AEC. Personal communication (2012)
19. James, P., Knapp, A., Mossakowski, T., Roggenbach, M.: Designing Domain Specific Languages – A Craftsman's Approach for the Railway Domain Using CASL. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 178–194. Springer, Heidelberg (2013)
20. Knapp, A., et al.: Epk-fix: Methods and tools for engineering electronic product catalogues. In: Steinmetz, R. (ed.) IDMS 1997. LNCS, vol. 1309, pp. 199–209. Springer, Heidelberg (1997)
21. Knapp, A., Merz, S., Rauh, C.: Model checking - timed UML state machines and collaborations. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 395–416. Springer, Heidelberg (2002)
22. Knapp, A., Merz, S., Wirsing, M., Zappe, J.: Specification and Refinement of Mobile Systems in MTLA and Mobile UML. Theo. Comp. Sci. 351(2), 184–202 (2006)
23. Knapp, A., Mossakowski, T., Roggenbach, M., Glauer, M.: An Institution for Simple UML State Machines. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS. Springer (to appear, 2015)
24. Knapp, A., Wirsing, M.: A Formal Approach to Object-Oriented Software Engineering. Theo. Comp. Sci. 285, 519–560 (2002)
25. Lano, K.: UML 2 — Semantics and Applications. Wiley, Chichester (2009)

26. Mossakowski, T.: Comorphism-Based Grothendieck Logics. In: Diks, K., Rytter, W. (eds.) MFCS 2002. LNCS, vol. 2420, pp. 593–604. Springer, Heidelberg (2002)
27. Mossakowski, T., Autexier, S., Hutter, D.: Development Graphs — Proof Management for Structured Specifications. J. Log. Alg. Program. 67(1–2), 114–145 (2006)
28. Mossakowski, T., Kutz, O., Codescu, M., Lange, C.: The Distributed Ontology, Modeling and Specification Language. In: Proc. 7th Int. Wsh. Modular Ontologies (WoMO 2013). CEUR-WS 1081, CEUR (2013)
29. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set, HETS. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)
30. Mossakowski, T., Sannella, D., Tarlecki, A.: A Simple Refinement Language for CASL. In: Fiadeiro, J.L., Mosses, P.D., Orejas, F. (eds.) WADT 2004. LNCS, vol. 3423, pp. 162–185. Springer, Heidelberg (2005)
31. Mosses, P.D. (ed.): CASL Reference Manual. LNCS, vol. 2960. Springer, Heidelberg (2004), Free online version available at `http://www.cofi.info`
32. Object Management Group. Unified Modeling Language. Standard, OMG (2011)
33. Sannella, D., Tarlecki, A.: Foundations of Algebraic Specification and Formal Software Development. EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg (2012)
34. Wirsing, M., Knapp, A.: View Consistency in Software Development. In: Wirsing, M., Knapp, A., Balsamo, S. (eds.) RISSEF 2002. LNCS, vol. 2941, pp. 341–357. Springer, Heidelberg (2004)