

Introduction

The common feature of model-oriented, model-based, and model-driven software development methods is to use models of the system under development as abstract representations of its desired properties. Models are used to determine the structure and behaviour of the system and to analyse its properties before it is realised in a deployed implementation. Also, for the maintenance and evolution of a system models are better suited as information sources than code or executables, which can only be inspected by running them. At least for larger projects, the usage of models for the construction, analysis, and evolution of systems is nowadays widely accepted as best practice.

Due to the complexity of large software systems, however, different kinds of models have to be constructed for these purposes. They represent different views of the system, corresponding to the different perspectives of the people involved in the process, and different formal aspects of the system, such as structure vs. behaviour, logical vs. technical design, or other classifications. In order to support this decomposition of the development process into various viewpoints, different languages and methods for the construction of the models are used, which offer those features that are needed for the definition of these views. Thus, whenever the complexity of the development process is reduced by using viewpoint models, this results in increased complexity of the models. Different languages are used, different aspects are specified, and different paradigms underlie these specifications, which necessarily lead to heterogeneous models.

The heterogeneous viewpoint models are conceptually kept together by being models of one and the same system. However, at least during the development phase, the system only exists as an idea, as an object of discourse. Therefore the development process must offer other means to tie the models together. It must be possible for instance to state whether or how elements of different viewpoint models correspond to each other and to check whether models are consistent with each other during the development. Since the system is not yet available, these issues must be directly addressed at the level of the models. Such a conceptual integration, which provides the means to con-

sider a collection of models as one model of a system, is a necessary component of any development process that incorporates viewpoint models. In order to retain the benefits of the decomposition into viewpoint models, however, integration must not rely on the concrete construction of one integrated model, but provide the means to run a model-based development with a collection of viewpoint models as if these were one model.

Although the integration must be achieved at the level of the models without explicit reference to the system, integration is inherently a semantic issue. Correspondence of model elements means that these elements refer to the same element or part of the system, i.e., they have the same meaning. Consistency means that models are free of contradictions, which holds if and only if there is at least one system that conforms to all models. This means that there must be one common semantic interpretation of the viewpoint models. Thus, to introduce a model integration approach first the semantic concepts have to be made precise. Based on this, syntactic representations and concrete methods can be defined that would become part of an integrated model-based development method.

In the following sections the viewpoint model of systems development, the conceptual integration, and the reference model approach that is introduced here are discussed in more detail.

1.1 The Viewpoint Model of Software Systems Development

The viewpoint model of software systems development comprises two major aspects: model-based development on the one hand and viewpoints on the other. The first one means that the software development should be based on models; that is, before and beyond the implementation abstract models of the system that is to be developed are constructed in order to have an abstract representation of the required structure, functionality, and properties of the system. These models allow one to state features and properties accurately without delving into the implementation details of programming languages, or even without giving a solution of how these properties can be achieved.

As mentioned above, this usage of models as a necessary abstraction layer in the development process for larger software systems is nowadays generally accepted. Appropriate languages and notation for the construction of models are offered by a variety of methods. The Unified Modeling Language (UML, see [UML03, BRJ98]) for example provides a family of visual languages for the modelling of object-oriented systems. It is a result of the unification and standardisation of notation that had been introduced earlier within different streams of object-oriented development methods. The standard that is used in its current version UML 1.5 here (see [UML03]) describes the syntax of the language formally in a boot-strapping manner, using class diagrams with constraints for the description of the language that itself contains class diagrams

and constraints. The semantics of the language is defined in natural language; the search for formal semantics of the whole language or parts of it is an ongoing research effort. Nevertheless, UML has already led to great unification within the field of object-oriented modelling concepts, and to widespread use of viewpoint models for the development of software systems.

But also more formal specification techniques are used to construct abstract models of software systems, such as the model-based techniques B [Abr96] and Z [Spi88, Spi92], process specification calculi like CCS [Mil89] and CSP [Hoa85], or the many variants of Petri nets [Pet62, Rei85, PR91]. Within the formal specification techniques a further distinction can be made between *specification formalisms* that formally define and clarify elementary notions of specification, including their formal syntax and semantics, and *formal methods* that are based on specification formalisms but have an elaborate concrete syntax and offer a method for the development of models. The model-based set-theoretic method VDM [Jon86, HW89] for example comes with an explicit development method, and object-oriented extensions of Z like Object-Z [DRS95, Smi00] introduce elaborate presentation and structuring concepts from object-oriented design. Also, the process specification formalisms CCS and CSP have been extended to formal methods, for instance in the process specification languages LOTOS [LOT87, Bri89] and ELOTOS [Que98]. Obviously, there is no clear frontier between specification formalisms and formal methods, but both can be distinguished from the more pragmatic *software modelling or specification techniques and languages* like UML, where syntactic extension is more important than precise formal semantics.

It is important to realise that for any model-based development process the models are not only auxiliary means in the early development phases before the real implementation of the system. On the contrary, they serve as adequate, readable information sources during the whole maintenance and evolution process. They have to be maintained as documentation of the system since important properties of the system and design decisions in its development process are most clearly documented by the models. Thus any further development of the system should be based on these abstract descriptions rather than on their realisation in the programming language code or the executable artefacts, where the desired structure will be hard to detect. In this way models help to reduce the complexity of developing and maintaining software systems by the abstraction of details.

The second main feature of the viewpoint model is that models are not developed for the system in its entirety, but, with each model, only a certain aspect or view of the system is represented. A class diagram for example only models the static structure of a system without giving information about its dynamic behaviour, use cases model the main functionality without revealing the structure of the system or the realisation of this functionality, process calculi focus on the temporal ordering of actions, neglecting the data types, etc. The consequence of focusing on designated viewpoints is of course that each model yields only a *partial specification* of the system, because all aspects

that are not concerned with this viewpoint are also not specified or constrained by this model.

The different viewpoints that are addressed in a modelling process may arise from different classifications. For instance, the different people taking part in the system's development play different roles and thus have different interests and expertise. Domain experts, software developers, system engineers, process managers, and clients for example need different information about the system and require different abstractions. Another classification that is based on the formal structural properties of systems and models instead of their envisaged usage is induced for example by the different UML diagram languages. These viewpoints essentially concern the static structure, the dynamic behaviour, both within and in between objects, and the implementation and deployment of the system. Yet another classification has been defined in the Reference Model of Open Distributed Processing RM-ODP [ODP, Lin91], which can be seen as one of the main sources of the viewpoint model altogether. It introduces five designated viewpoints that should be addressed when developing open distributed systems. Clearly, this particular feature of ODP is relevant for all system developments and not restricted to open distributed systems. The five viewpoints of the RM-ODP are the

enterprise viewpoint, the overall view of the system's purpose and aims, its agents and its policies;
information viewpoint, the information model of the system;
computation viewpoint, the view of the system as a set of interacting objects, adhering to the system's policies and realising its flow of information;
engineering viewpoint, the abstract machine view of the system, comprising its (technical) interaction mechanisms; and the
technology viewpoint, the configuration of software and hardware objects.

These viewpoints are of course mutually related, but no temporal order of their development is implied, as opposed to the software development phases of analysis, design, implementation, and test. In the ODP standard it is emphasised, moreover, that each of the viewpoint modellings should be supported by an adequate language, i.e., a language that allows the representation of the desired features directly without artificial encodings. Furthermore, these languages should be as formal as possible to allow for precise modelling and support the formal checking of properties with corresponding tool support.

Viewpoints contribute to the separation of concerns in the software development and thus yield a reduction in its complexity orthogonal to the contribution of the model-based approach.

The different classifications of viewpoints mentioned above are independent of each other to a large extent. This means, for example, that both software developers and domain experts are interested in information models, or that class, sequence, and state diagrams are used in the computational viewpoint. On the other hand, some combinations may be useless or excluded, so that the matrix of viewpoint classifications might have some empty entries.

Finally it is important to separate viewpoints as understood here from other decomposition or abstraction means. The decomposition of systems into manageable parts like components, modules, or classes for example is orthogonal to the concept of viewpoints, and raises completely different integration questions than the one of viewpoints (cf. the discussion in [BSBD99]). Furthermore, the view concept of databases that yields restricted views of complex data or object compounds is conceptually different from the viewpoint concept in software systems development, although technically similar integration questions may be addressed.

1.2 Integration of Specifications

Being accepted as an essential contribution to the rational development of software systems, the viewpoint model on the other hand immediately prompts the question on the relationships of the different models, especially since these may be given in very different languages, and even based on completely different modelling paradigms. There are, obviously, semantic relationships that have to be taken into consideration and clarified before syntactic support for the integration of viewpoint specifications can be developed (*Semantics first!*). As a slogan, the situation arising from the viewpoint approach could be formulated as: ‘*Many* people use *many* languages to develop *many* models of *many* views of **one** system.’

How can it be assured that indeed *one* system is modelled? The problem can be divided into two parts. The first one concerns the conceptual integration of models, which clarifies how different models can be considered as one model of a system. A model may for instance supplement the information given by another model, i.e., it can be considered as a refinement, making the first model more concrete by adding further details or further aspects that have not yet been addressed. The sequence and collaboration diagrams of UML for example refine the information given via the use cases by making explicit the dynamic interactions of the objects that realise the functionality specified by the use cases. State diagrams add information on the dynamic intra-object behaviour to class diagrams that only specify their static structure etc.

In contrast with this supplementation models may also deliver mutually overlapping information, where the same aspect of a system is concerned, but seen from different points of view. To use UML models again as examples, sequence diagrams model the interaction of objects realising certain scenarios, i.e., requirements on the objects. Statechart diagrams on the other hand model the intra-object behaviour, i.e., the capabilities of the objects. The semantic demand is then to check whether the objects—according to their statechart diagrams—are able to satisfy the requirements stated in the sequence diagrams. As opposed to the supplementation case discussed above, where models can hardly be inconsistent with each other, the semantic cor-

respondences in the case of overlapping models are much more involved and require a deeper and more precise semantic analysis.

The conceptual integration of models thus means to establish *correspondences* between model elements that express semantic relationships.

The second part of the problem concerns the *consistency* of the given models, i.e., intuitively, the question whether the set of models is free of contradictions. In the logical sense consistency means to have a common model or a common semantic interpretation. Before going into this discussion, however, a brief deviation concerning the usage of the terms *model* and *specification* is necessary at this point.

In the discussion above the term model has been used as usual in software engineering or computer science in general, where a model is an abstract representation of something made by somebody for some specific purpose (as defined in [Ste93b]). A model thus represents or describes something, a structure, a behaviour, etc., which is different from the model itself. A class diagram for example does not represent a class diagram, but possible states of objects. This distinction is made explicit in mathematical logic, where the (semantic) entities that are represented are called models, in contrast with the representing (syntactic) entities that are called specifications. Thus, beyond other distinctions that could be made, a software engineering model is a specification from the logical point of view, whereas the logical (model-theoretic) model does not have a common designation in software engineering, except perhaps the ‘meaning’ (or semantics) of the model. For that reason, and because to a large extent the terms model and specification are used as synonyms in computer science, I prefer to use the term specification instead of model in the sense of system modelling, and the term interpretation instead of model in the logical sense of a semantic entity.

Coming back to the question of the consistency of specifications, i.e., the absence of contradictions, its logical characterisation is given as follows. A set of specifications is consistent if there is a common model of all specifications. The problem with this definition in the context of heterogeneous specification languages is that the categories of models for different specification languages are often disjoint by definition, whence two specifications written in these languages may never be consistent, independently of their semantic content and intention. For example, the models of algebraic data type specifications are algebras (see for instance [EM85, BHK89, LEW96]), and the models of Petri nets are reachability graphs [Pet62, Rei85], processes, or occurrence nets [RG83, DMM89], or event structures [Win88a], depending on the chosen level of abstraction. Thus even if the data type of, say, lists is specified algebraically and some of its operations are expressed by Petri nets, the strict definition of consistency would in this case always lead to the conclusion that they are inconsistent, because an algebra is neither a reachability graph nor a process, nor even an event structure.

For that reason other definitions of consistency have been suggested. For example, a set of specifications may be considered as consistent if there is a

common (physical) implementation of all of them. This rephrases the original definition by replacing ‘common model’ with ‘common implementation’, but has the disadvantage that, in spite of its pragmatic flavour, this condition will be hard if not impossible to check. Viewpoint specifications have been introduced to deal with complex situations in the development of large systems, thus checking for consistency in this sense would amount to finishing the entire development first. A more concise approach to consistency in the context of the viewpoint model, especially the ODP viewpoints, has been developed in [BSBD99, BD99, BBDS99, BBD⁺00].¹ Different types of consistency checks are distinguished, like inter- and intra-language consistency and global vs. binary consistency, and further types of consistency checks are introduced that may depend on the specific relationships of the specifications, according to their roles in the development process and the different relations to the target system. The major contribution concerning the definition of consistency in this approach is to replace the common model and the common implementation suggested by the other definitions by ‘having a common refinement or a common development’. In order to support this approach development relations are investigated and presented in a universal formal framework that allows the study of consistency at a very general level, corresponding to the overall aims of the ODP standard. Whereas the formalisation of intra-language consistency checks, i.e., checks for specifications written in one language, is treated adequately, the inter-language check is more constrained and based on a translation between specific languages, Z and LOTOS in this case ([DBBS96]), which is not entirely satisfactory. The translation is too schematic to deal with language-specific modelling decisions and different modelling styles within one language. Recall also that for consistency checks of heterogeneous specifications correspondences need to be established first, which may depend on a specific comparison with other specifications, and usually cannot be obtained automatically. A fixed consistency check as in the mentioned approach does not offer the necessary flexibility for this purpose.

The approach to the integration and consistency checking of heterogeneous specifications introduced here claims to be as general as the consistency approach discussed, but is much more flexible, because it is based entirely on the semantics of the specifications. The basic contribution is indeed given by the definition of a common semantic domain that serves to interpret all languages that are used to construct specifications, whence each of the languages obtains its global semantics by referring to this domain. For this reason the semantic domain constructed here is called a *reference model* for the integration of specifications. With its usage the original logical definition of consistency can be taken up again; that means a set of specifications is consistent if there is a common interpretation of all specifications of the set in the reference model.

¹ See also <http://www.cs.ukc.ac.uk/research/groups/tcs/openviews/>.

In Figure 1.1 an example of an integration via a reference model is depicted. A small sample of a system is described by different specifications: a programming language construct, a class diagram with only one class, a state-chart, and an algebraic Petri net. The class defines the structure, given by the

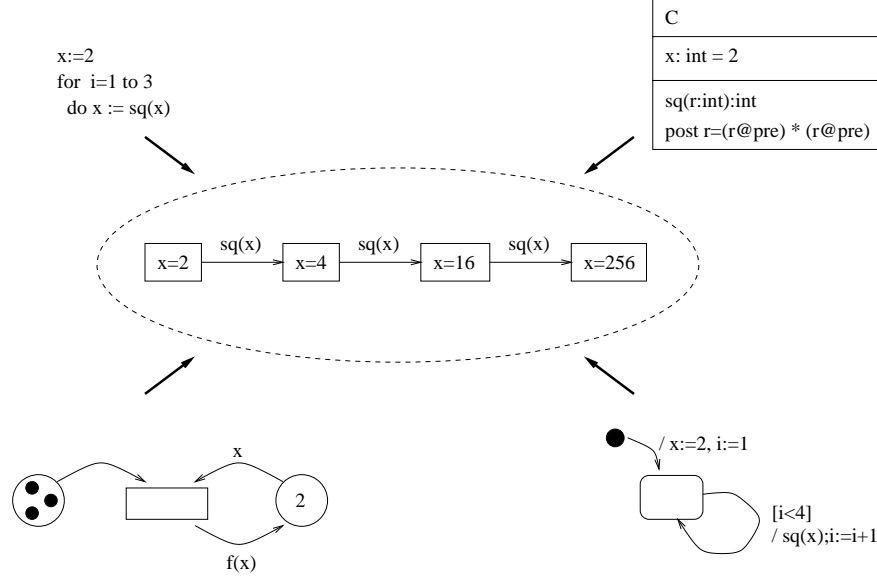


Fig. 1.1. A common interpretation of different descriptions by one formal model

integer attribute x and the operation sq of type int . Additionally it specifies an initial value $x = 2$ for x and a postcondition $r = (r@pre) \times (r@pre)$ for the operation, formulated in the object constraint language OCL of UML (see [WK98, UML03]). The expression $r@pre$ denotes the value of the parameter r before application of the operation. Beyond the pure structure, the behaviour is thus also partly specified by the class. It does not specify, however, when and how the operation should be applied. The program loop and the statechart specify the behaviour in very similar ways. Both the data states, given by the values of the attribute, and the control flow are modelled via variables whose values are manipulated. In addition to the implicit structure information given by the operations and the predicates that are applied to the variables, the dynamic behaviour is specified: the sq -operation is applied three times in sequence. The main difference between the two descriptions is that the program is textual whereas the statechart is mixed graphical and textual. Note that not only does the implicit typing of x in the program loop and the statechart overlap with the information given in the class, but so does its initialisation value. On the other hand, there is no information given about the effect of the operation sq on the variable x . Only its name indicates its

intended meaning. Finally, the algebraic Petri net models the same dynamic behaviour on the same structure in a slightly different way. Instead of introducing a new variable such as i in the other specifications, the control flow is specified here via an additional place. It holds three tokens that are consumed when applying the function f , i.e., f is applied three times. The effect of f is also not specified in the Petri net. Relating it to the sq -operation specified in the class diagram adds the information given in its postcondition to the Petri net. That means the integration of the partial viewpoint specifications enriches them mutually.

Using the reference model the consistency of these four specifications can be shown by stating one common model in the reference model as the semantic interpretation of all of them. This interpretation is depicted in the centre of Figure 1.1. There are four states containing the actual values of the static entity x , connected by three transitions labelled by the operation application $sq(x)$. This model is an admissible interpretation of the operational semantics of the program loop and the statechart, abstracting from the control flow variable i , and, analogously, it represents the operational semantics of the algebraic Petri net when interpreting the function f as $f(x) = x^2$. Moreover, it represents the possible behaviour of an object of the class. Thus the four specifications have a common interpretation, whence they are consistent.

Since the class does not specify the dynamic behaviour it obviously admits a set of models in the reference model, whereas the other specifications are usually understood as describing a single behaviour. Nevertheless, the latter can also be interpreted in different ways. The behaviour of the function f in the Petri net for instance is not specified in the net, whence it can be interpreted arbitrarily as long as no further information is given. The decision for the interpretation $f(x) = x^2$ is determined by a comparison with the class diagram that contains a postcondition for the operation the function f might correspond to. Similarly, the variables x and i in the statechart and the program loop are treated differently in the model: x is interpreted as a static entity, whereas i is resolved into the dynamics. Obviously, this distinction is influenced by the comparison with the information given in the class diagram again, which only contains the attribute x . This shows that the semantic interpretation of a specification for consistency checking and integration may depend on the interpretation of the other ones considered within this test. These local decisions about the interpretation have to be taken into account of course when different groups of specifications are compared. This discussion again supports the statement made above: that consistency checks cannot be entirely automated, but require context-dependent correspondence information.

Further, the consistency check of the four specifications in the example already contained an implicit integration in the sense of a model correspondence. The syntactic entity x in all four specifications refers to the same item in the semantic interpretation, which yields the correspondence of the specifications at these points. This is a very simple correspondence induced by

identical names, but in general correspondences may be much more complex and will hardly rely on the identity of accidentally chosen local names. The operation sq in the class, the statechart, and the program loop here correspond to the function f of the Petri net, and the three black tokens of the Petri net correspond to the loop condition *for* $i = 1$ *to* 3 in the program fragment, and the combination of the initialisation $i := 1$ and the condition $[i < 4]$ of the loop transition in the statechart.

1.2.1 Admissible Interpretations, Correspondences, and Consistency

Taking into account the above-mentioned incompleteness of viewpoint specifications that is due to their partial view of the system, a more general conception of integration is achieved. Instead of immediately searching for a common interpretation in the reference model, the following three steps in an integration process based on a common reference model are methodologically distinguished and treated explicitly.

Admissible interpretations The elements of the reference model are models of systems that cover all relevant aspects. Since a viewpoint specification constrains only one aspect of a system, it admits a set of interpretations in the reference model. The first step in an integration consists of the determination of the sets of admissible interpretations of the individual specifications. This may be induced by the language used for the construction of the specifications, for instance if these are formal specifications with formal semantics, but it might also depend on the usage, the application domain, etc. In Figure 1.2 sets of admissible interpretations are shown as subsets of the reference model, indicated by the arrows from the specifications.

Correspondences The viewpoint specifications may be developed within different name spaces, address different parts or scopes of the system, refer to different granularities w.r.t. behaviour and structure, and use different means to specify corresponding information. These semantic correspondences have to be made explicit for the integration. Within the reference model correspondences appear as transformations of the sets of admissible interpretations. System models, as elements of the reference model, may be projected onto common parts or scopes of the system, or adjusted to a common level of granularity, etc.

Consistency In order to check the consistency of the specifications, after their admissible interpretations have been determined and the correspondences have been declared, the intersection of the correspondingly transformed sets of admissible interpretations is considered. If it is empty the specifications are inconsistent w.r.t. the considered interpretations and correspondences. If there is more than one interpretation in the intersection the collection of specifications is still incomplete, i.e., the information

given w.r.t. the considered level of granularity is still not sufficient. If there is exactly one model in the intersection the specifications are consistent and complete in this sense; that means, w.r.t. the interpretations and the explicit correspondences considered in this integration.

This complete view of the integration of viewpoint specifications by the transformation and intersection of sets of admissible interpretations is illustrated on another small example in Figure 1.2. A class diagram, a statechart dia-

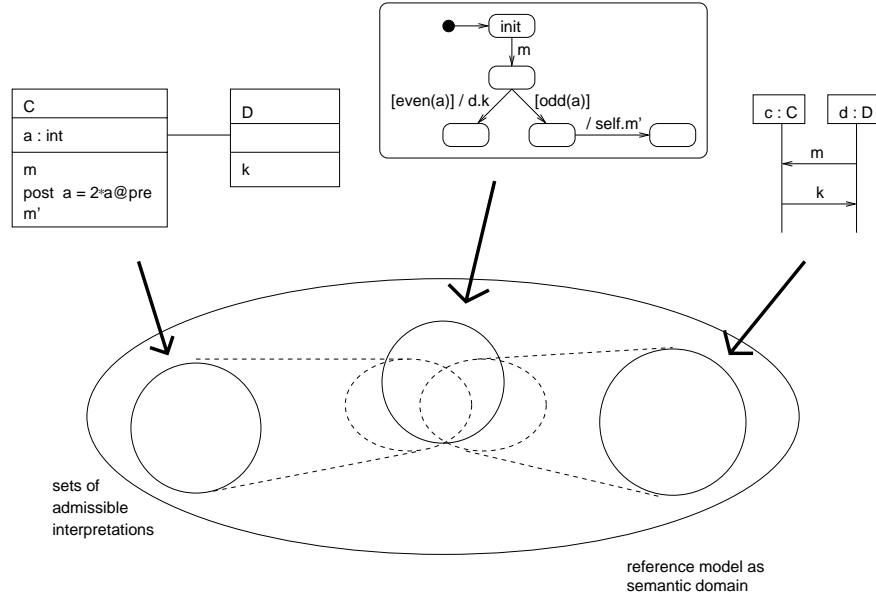


Fig. 1.2. Integration of viewpoint specifications by sets of admissible interpretations, transformations, and intersection

gram, and a sequence diagram are shown that together specify a (very small) system. The class diagram defines the structure of the objects and, by the constraint that defines the postcondition for the operation *m* in *C*, one aspect of their behaviour. The state machine specifies the simple reactive behaviour of the objects of the class. It is assumed that the state machine is associated with the the class *C*, one correspondence that is not explicit in the figure. This kind of correspondence is defined in the UML meta model for example as the context of the state machine, and realised for instance by hyperlinks in modelling tools. Finally, the sequence diagram specifies one interaction of objects of the classes *C* and *D*. For the integration all three specifications are interpreted by the full sets of system models that are admissible interpretations of the specifications. The class diagram for instance does not constrain the sequences of steps the objects can perform; that means all sequences of

executions of the operations may occur (or not occur) in a semantic interpretation of the class diagram. On the other hand, the statechart diagram and the sequence diagram do not constrain or specify the side effects of the operations on the data states of the objects or whether operations are deterministic or not. All interpretations w.r.t. these aspects are admissible.

In the second step of the integration the correspondences of the specifications have to be considered. For this example we assume that equal names have equal meanings, i.e., C , D , a , m , and m' denote the same class, attribute, and operations respectively in all specifications. Furthermore, as mentioned above, the state machine is considered to specify the behaviour of objects of class C . Such a single name space might not always be assumed. In a larger development names may be chosen locally and only in the integration of the specifications do the correspondences between the name spaces have to be declared. As shown in the examples in the following chapters these need not be one-to-one correspondences of names, but more complex constructions might be required, mapping, for instance, names to descriptions.

A further correspondence relates the scopes of the three specifications. The class diagram specifies arbitrary collections of objects, the statechart diagram specifies the behaviour of a single object of class C , and the sequence diagram specifies one desired behaviour of linked objects of classes C and D respectively. Thus to compare and integrate their semantics appropriate projections have to be applied first that map the system models in the respective sets of interpretations to the same portion of the system. For example, all system models associated with the class and sequence diagram can be projected onto systems with exactly one object of class C .

An analysis of the intersection of the transformed sets of admissible interpretations of the three specifications then shows in which way they supplement each other. Obviously, in this example the intersection contains more than one system. On the one hand, the specifications do not contradict each other, which means that there is at least one common interpretation. On the other hand, neither the initial value of the attribute a nor the side effects of the operations m' and k are specified in any diagram, which means that they can still be consistently defined in different ways. Thus there is more than one common interpretation of the three specifications.

Considering now the class diagram and the state machine together yields that, due to the postcondition for m stated in the class diagram, only the left transition from the second state (the one with the guard $even(a)$) in the state machine can be executed. This implies that, due to the state machine this time, each object of class C can perform at most one macro-step: If m is called the object calls the operation k at object d and then terminates.

In Figure 1.3 some possible interpretations of the class and the state machine diagrams are sketched. The class diagram interpretations in these samples contain two objects, one of class C and one of class D . The state machine interpretations contain one object of class C , but these also refer to operations from class D . For the comparison the class diagram interpretations must now

be projected onto systems representing a single object c of class C . On the other hand, the transition labels of the state machine interpretations must be projected onto the operations of class C . Only one system remains in the intersection of the transformed sets. This not only shows by its existence the consistency of the two diagrams, but incorporates and integrates the information spread over the two viewpoint specifications.

Taking into account the scenario specified by the sequence diagram then shows that the operation m will be called at least one time. Combining this with the information given by the state machine yields that the first message of the scenario can only be sent if the object c is in its initial state and has never done anything else before. Note that the sequence diagram describes arbitrary incomplete scenarios, i.e., ones that might not occur at the beginning of the life cycle of an object. That means, considered individually, it admits interpretations that contain the specified sequence of steps, but it also admits other steps before or after this sequence. In this example the sequence diagram does not add further information to the other two specifications. The integration, however, shows that the state machine together with the class correctly implements the scenario.

The results of such an analysis could be used to refine and develop the individual specifications, for instance by cancelling the right branch of the state machine's transition and adding the information on the initiality of c to the sequence diagram as an OCL constraint ($c.oclInState(init)$). This could be enhanced later on, when the initial value of the attribute a is known. However, the reflection of the results of the conceptual integration onto the specifications obviously depends very much on the languages they are given in. Since the overall integration approach introduced here is language independent, the reflection cannot be developed systematically in the general framework. For this purpose concrete languages or families of languages would have to be investigated in detail.

1.2.2 Language- and Method-Independent Integration

In the examples discussed just now viewpoint specifications have been compared and integrated out of context, i.e., without asking for their position or role in a development process. In fact, the viewpoint concept can be employed in many different ways in different contexts. Beyond being independent of specific modelling or specification languages, the viewpoint approach itself is also independent of any particular method that would state how to develop and use the viewpoint models. Instead, it can be deployed with a variety of methods, just as it can be instantiated with a variety of languages.

A strictly hierarchical method for instance would not only prescribe which viewpoints have to be specified, but also give an order for their development and rules that state how new viewpoint models are to be constructed based on the given ones. The advantage of such a strict hierarchy is that integration and consistency aspects can be incorporated and fixed in the method and

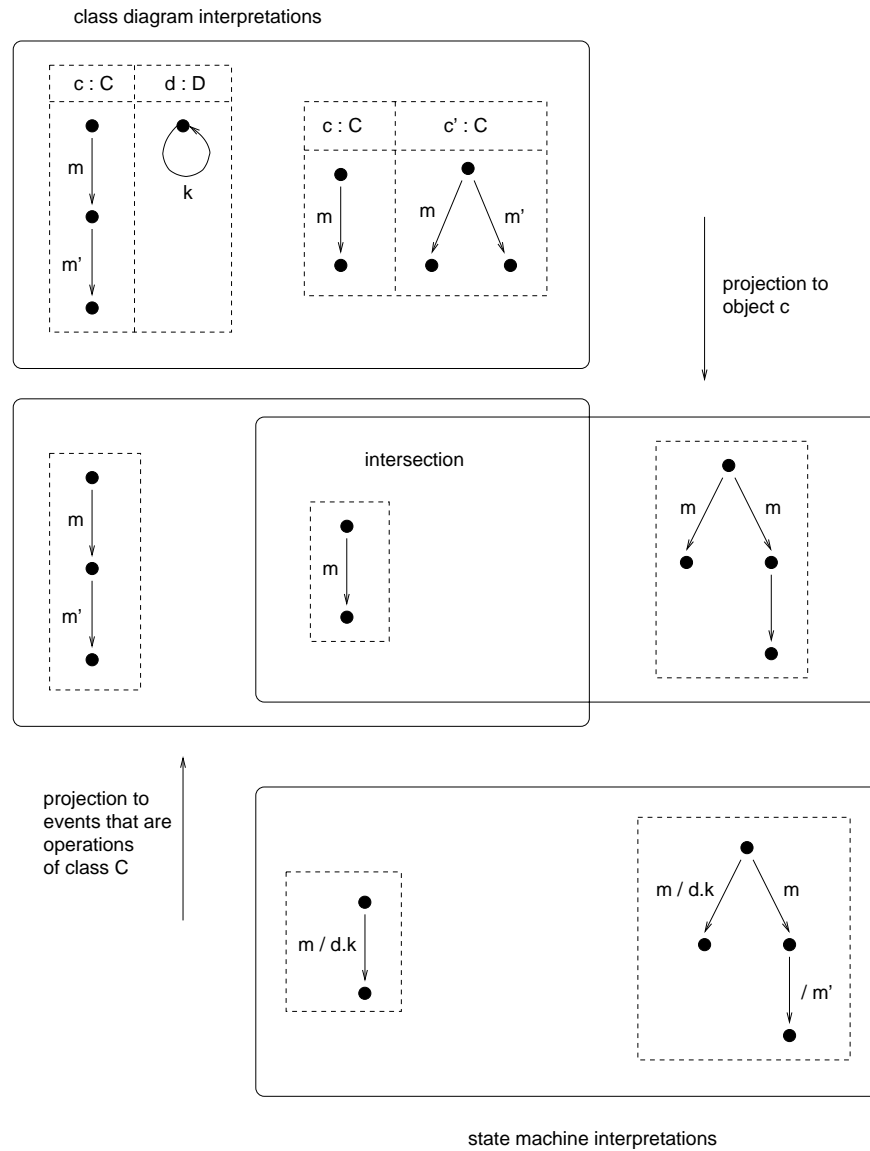


Fig. 1.3. Semantic integration of the class and the statechart diagram

need not be considered anew in each individual development. On the other hand, this does not allow deviations from the fixed order of development steps and thus limits the flexibility of both the development and the maintenance or evolution process. And if models are used continuously to support the system's evolution, new notation probably will have to be taken into account at some point and require an integration with that used so far. This concerns

especially the integration of new components whose specification might not conform to the same strict development method and might not be given in the same notation.

Less strict methods allow at least different starting points for the development process. For example, instead of requiring either the structure or the behaviour of the system to be specified first, both views may be considered initially independently of each other. UML-based software development methods, for example Catalysis [DW98] and the development method for component-based systems presented in [CD00], support this flexibility. The models developed for the different views then have to be integrated at some stage. Thus the integration problem occurs again, although in a less drastic form since the basic conceptual relationships of the models are already determined by the method.

An entirely liberal approach, where all viewpoints are equivalent, is supported by the RM-ODP. It stresses that no order should be prescribed on the specification of the viewpoints. This allows for example the integration of assets, i.e., models that have been developed before, and the flexible adaptation of the development to the given requirements. Obviously, the integration task occurs here in its most general form, since even the viewpoint specification languages are not fixed.

Heterogeneous specifications and the corresponding need for integration also arise in processes where stakeholders are used to employ specific notation that is established in their domains. Developing software for embedded systems for example involves specifications of software systems, given for instance in a UML notation like statechart diagrams, and specifications of programmable machines, given for instance in the IEC 61131-3 language Sequential Function Charts [SFC]. A flexible method should support the usage of both languages and add integration mechanisms instead of forcing the users to agree on one of the languages.

In order not to constrain the application of the integration approach unnecessarily, methodological questions as well as specific languages are separated here completely from the semantics of integration. (Separation of concerns is as important in theory design as it is in systems design.) That means that viewpoint specifications are considered independently of their envisaged usage in a development process and independently of the language that is used for their construction. The contribution of the proposal thus amounts to a framework at a meta level that delivers the fundamental notions and concepts. For an application-specific usage, given by a definite set of viewpoints, the corresponding concrete viewpoint specification languages, and a method for the concerned development process, significant instantiation effort is still required. The generality of the approach, however, allows one to use, adapt, or specialise the proposed integration framework in any method that is based on viewpoints, or takes into account heterogeneous models and specifications for any other reason. Of course, the full generality of the concepts introduced here might not be completely exploited in a specific instantiation, due to its

constraints on the viewpoints, the languages, and the usage of viewpoint specifications. The proposed integration framework serves as a foundation for the development of such methods.

1.3 Requirements of Reference Models and Their Usage

General requirements of the internal structure of a reference model for the integration of heterogeneous specifications can be stated immediately. First of all, the reference model must supply formal system models as elements that serve as semantic interpretations of the specifications. On the one hand this means that it must be possible to represent the structure and the behaviour of the specified systems with the system models. On the other hand it must be possible to adjust the granularity of the interpretation according to the desired degree of abstraction. This may be induced by the specification level (more abstract or more concrete specification) or by its scope (larger or smaller system, subsystem, component, object, etc.).

Second, there must be relations expressing the development of system models in order to trace modelling decisions made within iterative development steps. To operationalise the developments, there should be operations that allow the constructive development of more refined or more abstract views or versions of given system models.

Finally, orthogonal to the dimension of the development, there must be composition operations that allow the construction of larger, more complex system models from given ones. Defining the composition by operations in the mathematical sense means two things. First, it has to be defined how system models can be connected in order to compose them, i.e., the definition of relations on their structures and behaviours must be supported. These relations of the system models represent the *architecture* of the composed system. Second, the result of executing the composition operation has to be defined. That means the system model representing the whole composed system of interconnected models has to be given. This second requirement implies compositionality in the sense of *structural transparency*: a composition of local system models can be considered as a single system model again, and thus the internal structure of the system can be hidden.

Note that this does not mean that the result of the composition always has to be computed. Structured systems—architectures—may be retained in a development stage as connected system models (as defined in the first part of the definition of an operation). The second part of the operation’s definition, the computation of its result, only offers the possibility to consider the interconnected models as a single model again, which may be computed whenever desired.

The two dimensions of development and composition must be brought together by checking the *compositionality of developments*. This means conditions must be stated that guarantee that local developments of system models

can be composed in the same way as the system models themselves and that this yields a global development of the whole system that comprises the local developments conservatively.

As mentioned above, the intended usage of a reference model is to map viewpoint specifications to it in order to compare and integrate them semantically. This can be achieved in a uniform way by considering the specification languages and mapping them to the reference model. That just means to define their semantics in terms of the elements, relations, and operations of the reference model. If the formal semantics of a language is already defined then the redefinition in the reference model must be shown to be compatible with the given one. In general the two definitions will not coincide, because the structure of the elements of a reference model will be much richer than the one used before, due to the generality of the reference model. But it must be possible to recover the original semantics from its redefinition. It is not the idea of the reference model approach to define completely new, ‘better’ semantics with better properties. (Although a criticism of the language design or the semantics definitions may arise from their reconstruction in the reference model.) If the semantics of the languages has been defined only informally, a reference model may help to formalise them further by offering its elements and its structure as formal semantic objects. (This depends of course on the formality of the reference model itself.) Furthermore, languages may lack some of the structure or the properties of a reference model, for instance appropriate composition operations, development relations, or compositionality properties. Then the corresponding feature of the reference model might be reflected to the language by seeking syntactic representations in terms of the language or by appropriate extensions of the desired features. Analogously, the reference model may be used to investigate which of the operations support which compositionality properties.

1.4 The Transformation Systems Reference Model

In this section a first informal survey of transformation systems and their usage as a reference model for the integration of heterogeneous software specifications is given. The presentation proceeds along the requirements stated above in order to demonstrate how the different dimensions and features are covered in this approach.

A specific property of the transformation systems reference model is that it is fully formal, i.e., the elements, relations, and operations are defined mathematically. This allows us to give precise definitions, to state its properties explicitly, and to prove them. On the other hand, the correspondence of the mathematical constructions and the software specification concepts has to be established, which can be achieved only at an informal level. For that reason a large part of the book is devoted to examples, explanations, and discussions. Thus beyond the formal definitions of the reference model its basic notions

and constructs are analysed and explained at an informal level. This allows us to carry over the integration method also to applications that cannot (or should not) be handled entirely at the formal level.

1.4.1 Transformation Systems

The elements of the transformation systems reference model represent the static structure and the dynamic behaviour of systems via a two-level structure. The behaviour of a system is modelled by a transition system, given by a set of abstract control states and transitions between these states (first level). The control states demarcate reference points to the systems, i.e., these are the states at which the systems can be inspected and accessed. The internal structure of a control state, i.e., the corresponding state of the data, is attached to the control state as a label, given in the second level. All data states of a transformation system adhere to the same static structure, but may be different instances of course. That means that the static structure is the schema of which the data states are instances. The other part of the second level consists of the labels for the transitions of the transition system. Whereas a transition from one control state to another one only states that it is possible to pass from the first state to the second one, the transition label indicates which actions are performed in this step. Actions might be assignments, method calls, passive actions like events, input and output actions, etc. Using an action or a set of actions as the label of one transition means that no inspection or access to the data state is possible in between the initial and the final state of the transition. In particular, invariants need not hold in between and no other communication may take place during the step.

A transformation system—an element of the reference model—is thus an extended version of a labelled transition system (LTS), where not only the transitions but also the states are labelled. LTSs are traditionally used for the definition of operational semantics for all kinds of languages and systems, like imperative and functional programming languages, process and data type specification languages, and others. Therefore transition systems are also chosen here as formal models of the behaviour part, representing the smallest common denominator of semantic models for this aspect. Defining the labels of transitions and states appropriately and choosing the right transition system allows a flexible adjustment of this structure to the forms required by the different integration tasks.

The data states attached to the control states have been introduced as instances of one schema representing their common static structure. In the simplest case the schema is given by a list of typed static entities like the attributes of a class, program variables declared in some program, or the variables of a Z schema, for example. The corresponding instances (data states) are given by type-compatible bindings of these entities to values, i.e., elements of the corresponding types. That means a data state is given by a list of values. Making the types and their data type structure explicit leads to the definition

of data states as algebras. (This relates the transformation system approach to the states-as-algebras approaches, see [ABR99]). In addition to the data type signature there are then constants in the (static structure) signature, corresponding to the syntactic entities, such as attributes, variables, etc. Each algebra of such a signature is thus given by a data type or a collection of data types and a set of designated elements of these types. The latter represent the actual values of the static entities.

The usage of algebras then allows further generalisations, as for example parameterised constants like attributes of array types, which are programming language encodings of finite functions. (See for instance [CD00], where parameterised attributes for the specification of components and interfaces with UML are advocated.) Since data types are usually partial, and attributes/variables need not always be defined, partial algebras are used immediately as data states within transformation systems. They have essentially the same theory as total algebras (see [Bur86, Rei87, CGW95]). Finally, since the definition of transformation systems as elements of the reference model does not really depend on the choice of the labels, a generic (institution-independent) definition is given as a further generalisation that allows the usage of arbitrary data type models as data states, like first- or higher order logic structures, or order-sorted algebras, for example.

Analogous to the labelling of the control states, the transitions on the first level are labelled by sets of *actions* on the second level. The interpretation of these actions—as observations, method applications, operation calls, actions, events (passive actions), or whatever—is left open and must be taken into account when different models are compared. A set of actions attached to a transition is interpreted as the occurrence of all actions of the set in between the initial and the final state of this step. According to the interpretation of control states given above, no state inspection is possible between the initial and the final state of a transition. Thus there is no order on the actions attached to one transition, nor is it assumed that they occur at the same time. As for the control states there is also a generic version of actions, corresponding to the generic specification framework (institution) of the data state models. This allows for instance the incorporation of guards or composed actions, or distinctions of events and actions as in statecharts. For more technical reasons, implied by the generality of the transformation systems reference model, transitions are labelled furthermore by a second component, a *tracking relation*, that keeps track of the identity of data elements through state changes. (Tracking relations correspond to the partial tracking functions of *D-oids* introduced in [AZ93, AZ95].)

A transition t leading from a state c to a state d , written $t : c \rightarrow d$, is thus associated with three labels: the data states C and D attached to the control states c and d respectively, and the pair $T = (act_t, \sim_t)$ given by the action set act_t and the tracking relation \sim_t attached to t . Thus a transition $t : c \rightarrow d$ can also be seen as a data state transformation $T : C \Rightarrow D$. This point of view, which distinguishes transformation systems from ordinary labelled tran-

sition systems most clearly and is also predominant in the states-as-algebras approaches, was decisive for the designation *transformation systems*.

In Figure 1.4 a transformation system is shown that models the possible behaviour of a point object that moves erratically on a 3×3 grid. The static structure of the system is given by two attributes x and y of type $\{1, 2, 3\}$, representing the coordinates of the point and their values. The data states are given accordingly by the nine possible different values of the attributes. The transformations are interpretations of the applications of the parameterless *move*-action. The underlying transition system is given by the nine states,

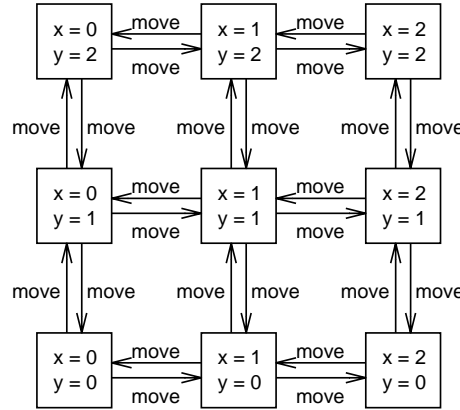


Fig. 1.4. A transformation system representing a point object moving on a grid

depicted by the rectangles, and the twenty-four transitions, depicted by the arrows. Data states and actions are given by the labels in and at the rectangles and arrows respectively.

1.4.2 Development Operations and Relations

The development of transformation systems is supported in the reference model in two ways. First, there are development operations that allow the construction of more abstract or more concrete systems from given ones. According to the two-level structure of transformation systems—the transition system level representing the dynamic behaviour and the label level corresponding to the static structure—the operations may be used to reduce or to refine the behaviour and the structure. Reduction and refinement or extension on the two levels can be combined arbitrarily. The behaviour can be restricted to certain actions, i.e., the other ones are excluded, or further steps can be added extending the behaviour. Analogously, only (public) parts of the data structure may be shown (i.e., private parts may be hidden), or further (internal, private) structure may be added. Furthermore, single actions or steps

in one system may be refined by composed steps (transactions) in another system, and static functions may be refined or implemented by compositions of static functions of the refining type. For example, an attribute *Name:String* may be refined by the composition *concat(Title,LastName,',',FirstName)* in another specification. In the opposite direction this yields the means to define interfaces of systems in the sense of more abstract views. Note that these interfaces are semantic elements, too, and that they represent not only static structure information like interfaces in class diagrams, but also behaviour information.

From a less operational point of view development relations are more adequate than development operations. For this reason they are also provided in the reference model. Given two transformation systems it can be checked with these development relations if and how one of these systems can be seen as a development of the other one. Developments are so general as to include refinements, implementations of one transformation system by another one, and interfaces as views as mentioned above. The development relations are a generalisation of the development operations in the sense that the application of a development operation always yields a system which is in development relation with the old one.

In Figure 1.5 for example the static structure of the grid-point system is refined by adding a new attribute *next* that determines the direction in which the point can move. This refinement corresponds to an inheritance in a class diagram: the static structure may be enlarged and the behaviour may be redefined. In this case, the behaviour is restricted to only one possibility per state. Also in the context of process specifications, the reduction of non-determinism is an important refinement technique, leading from abstract specifications that allow non-determinism to concrete, deterministic implementations. Implementations, as a further development relation, may also require additional internal structure, represented as a refinement by extension at the structure level.

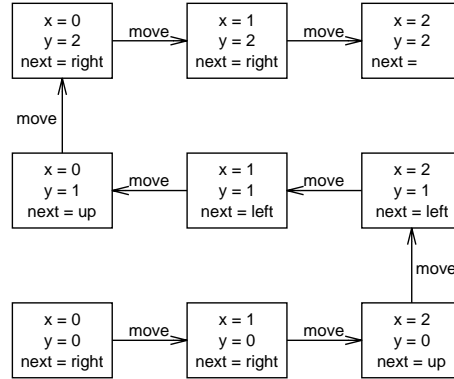


Fig. 1.5. A transformation system representing deterministic moves on a grid

1.4.3 Composition

Analogous to the development of transformation systems, their composition is based on composition operations for the two levels, transition systems (dynamic behaviour) and data state and action labels (static structure). As stated in the requirements above, the composition of systems is based on relations that describe how the local components are connected. For transformation systems these are given by two relations. A synchronisation relation on the transition systems expresses which states of the local systems can be entered simultaneously and which steps of the local systems can be performed simultaneously. An identification relation on the static structure expresses which parts of the local static structures are shared by the local systems, and thus identified in the global view. The synchronisation relation does not presume that the local systems are synchronised by a global system clock. Instead, transformation systems allow the formal synchronisation of transitions also with states, modelling situations where one component performs its actions whereas the other one remains idle in its state. Since one transition may be related to several transitions or states of the other component, synchronisation means that the transition *may be* synchronised with one of the related transitions or states. If there is exactly one related transition or state these *must be* synchronised.

Consider for example two systems \mathbf{M}_1 and \mathbf{M}_2 , each of which performs two consecutive steps t_1, t'_1 (in \mathbf{M}_1) and t_2, t'_2 (in \mathbf{M}_2) respectively (see Figure 1.6, where only the transition system level of the two transformation systems is shown). Consider first the case where \mathbf{M}_1 and \mathbf{M}_2 are completely independent of each other, i.e., all states of \mathbf{M}_1 and \mathbf{M}_2 can coexist and all transitions of \mathbf{M}_1 and \mathbf{M}_2 can take place together in one global step of the composed system. This is expressed by the full synchronisation relation, i.e., each transition and state of \mathbf{M}_1 is related to each transition and state of \mathbf{M}_2 . Then the composed system is able to perform the local steps in any order, including single and parallel executions, as shown in the centre of Figure 1.6.

To express that t_1 must be synchronised with t_2 , while t'_1 and t'_2 may still be synchronised with each other or with the corresponding states, the synchronisation relation shown in Figure 1.7 has to be given. Then the global system behaves as shown by the dashed arrows in Figure 1.6. The steps depicted by the solid arrows are excluded because they consist of pairs that are not in the synchronisation relation of Figure 1.7.

With the identification relation the sharing of common (static) data types can be expressed, such as interchange data or pervasive types, shared variables as in the access to a common store, or shared actions like the input/output actions of process calculi. In Figure 1.8 for example the two components \mathbf{M}_1 and \mathbf{M}_2 share a variable, which is called x in \mathbf{M}_1 and a in \mathbf{M}_2 , and each one has a private variable, y and b , respectively. This is expressed by the identification relation $x \text{ id } a$ on the signatures $\{x, y : \text{int}\}$ for \mathbf{M}_1 and $\{a, b : \text{int}\}$ for \mathbf{M}_2 . As indicated in the transformation systems, \mathbf{M}_1 has writing access to x , whereas

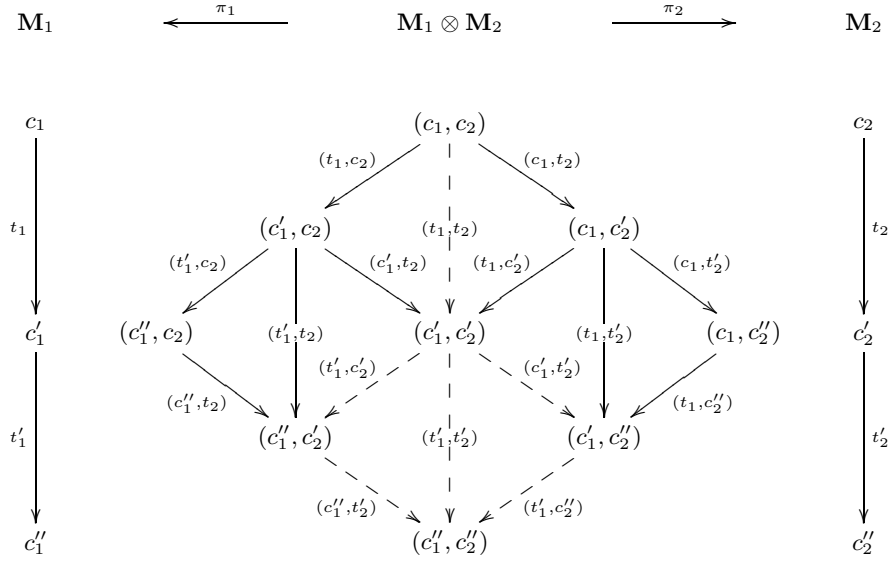


Fig. 1.6. Synchronisation of two systems

M_2 can only read a . In its first step it observes that the value of a has been changed.

The global data states of the composed system $M_1 \otimes M_2$ are then given by a superposition of the local ones. Shared parts are identified, i.e., they are represented only once in the global state, whereas the local parts are kept apart, i.e., they are added disjointly. This yields the global data states in the upper row of Figure 1.8. The actions of a step in the global system are given by the unions of the local action sets, where again names that are related via the identification relation are identified. As mentioned above, this

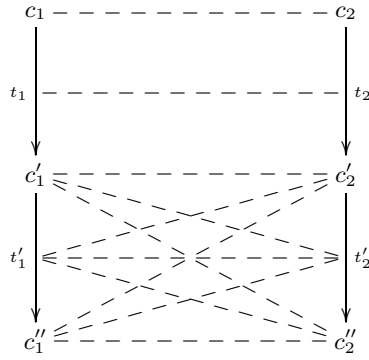


Fig. 1.7. A synchronisation relation on two transition systems

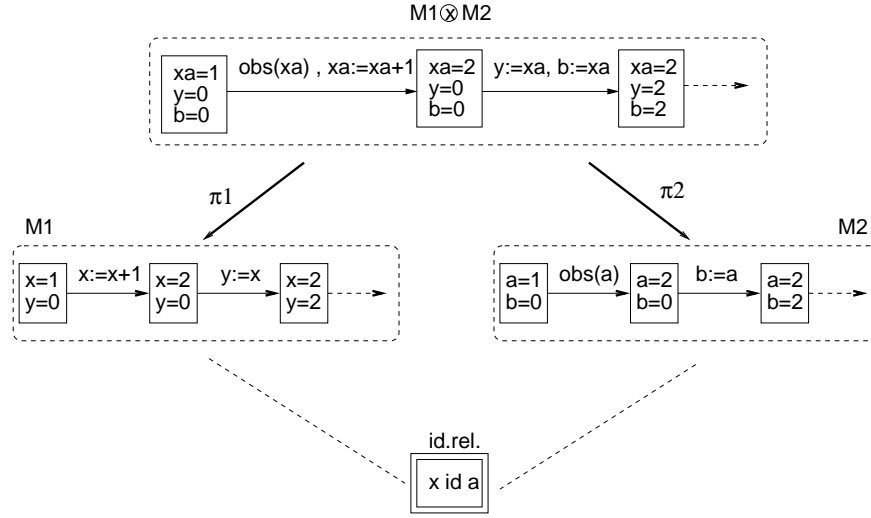


Fig. 1.8. Composition of transformation systems

sharing (identification) as well as keeping apart local parts of the structure does not depend on the names chosen in the local specifications. Instead, the identification relations are used to express arbitrary correspondences between the elements or derived elements of the local structures.

Synchronisation and identification relation need not be binary but may connect arbitrary numbers of local systems. Together they describe the connection of local systems, i.e., the architecture. This corresponds to the first part of the definition of composition operations as discussed in Section 1.3. The result of a composition, i.e., the global system seen as one transformation system again, is then given by the tuples of synchronous steps of the local systems and the corresponding superpositions of their data states and action sets as discussed above. In Figures 1.6 and 1.8 such global views have already been shown.

The abstraction mechanisms mentioned in the discussion of the development relations can also be used to obtain more asynchronous forms of behaviour composition. In Figure 1.9 for example several sequences of steps in the local components are encapsulated by introducing interfaces. Synchronising these abstract steps means designating the corresponding control states in the local systems as synchronisation points, but not constraining the local behaviour in between these points. The global behaviour induced by the synchronisation of the first and the second steps of the two interfaces is then given by two global steps. In the first one the first three steps of the left concrete system are performed and—asynchronously—the first two steps of the right concrete system are performed. At this point a synchronisation takes place. Then the last two steps of the left system and the last step of the right system are performed, again asynchronously. As opposed to the direct synchronisa-

tion (including synchronisations with states) there is no global state of the asynchronous system during the execution of the local actions that constitute one global step.

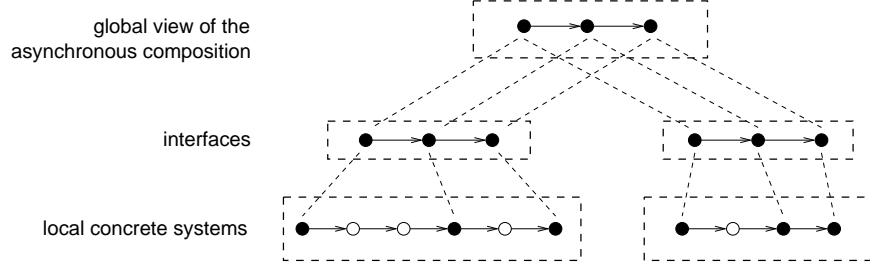


Fig. 1.9. Asynchronous composition of two systems

The structural transparency achieved by computing the result of a composition operation, i.e., the representation of a composition of systems by a single transformation system as its global view, also allows the representation of refinements and developments at the architectural level. The direct composition of two systems by a synchronisation and an identification relation for example can be refined by first introducing interfaces for the encapsulation of local steps and then composing these more abstract views as discussed above. Since both sides, the direct composition and the asynchronous composition via the interfaces, can be considered as single transformation systems in turn, the definition of the development relations for structured systems can be reduced to the basic unstructured case discussed above. That means the asynchronously composed system is a development of the directly composed system if their global views are in a development relation.

Moreover, also due to the structural transparency, the architecture of the data states and the architecture of the behavioural components need not coincide. If, for example, two processes with different behaviours exchange complex data, then the architecture of the behaviour (the two linked processes) is completely different from the architecture of the data, which may be composed of many substructures (see Figure 1.10).

1.4.4 Granularity

The examples discussed above might have led to the impression that the representation of software systems or components as transformation systems in the reference model might be rather low level. In fact, however, the granularity of the formal model can be chosen arbitrarily according to the level of abstraction required for a specific view and offered by the corresponding specification technique. The reference model itself is entirely open w.r.t. modelling decisions concerning the granularity. That means the distinction between static data

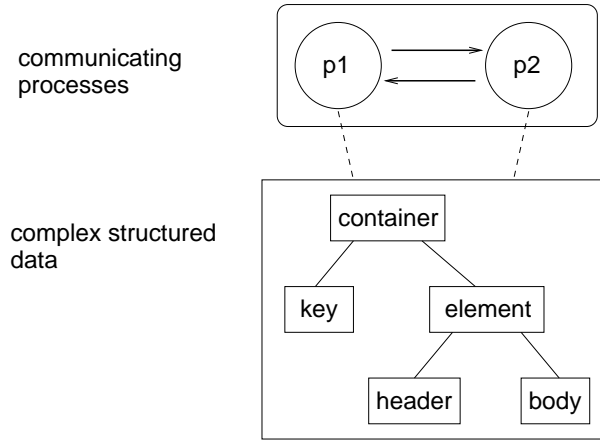


Fig. 1.10. Different architectures of behaviour and data

types and dynamic operations with side effects, and the distribution of the structure and the behaviour into local transformation systems in a concrete model are induced by the given specifications and specification techniques. They are not prescribed by the reference model.

Consider as an example the assignment of a result of a database query to an attribute of some object. If the purpose of the specification is just to state that this value is bound to the attribute, the whole query and the assignment can be considered as one data state of the system. The database query is then modelled as a static function within a static data type and the state is characterised by the equation $attr = query(par_1, \dots, par_n)$. Thus, in this case, the query is identified with its result, as in the mathematical set-theoretic understanding of a function, which is a very high abstraction. If, however, the dynamic behaviour of the query has to be modelled, for instance its execution inside the database management system, the transition system level of transformation systems has to be used, too. Then the single steps are modelled by transitions and the data states now also may contain further internal variables to store intermediate results and control information. Finally, it could be necessary to consider also the architecture level, for example if the internal structure of the database management system needs to be considered or if the query is sent by a mediator to several databases. Then the local dynamics and data structures are coordinated by a specific component, sitting on top of the other ones.

The representation of one scenario may thus use the static data type level alone, a single transformation system, or a set of interconnected transformation systems. This depends only on the desired granularity of the overall modelling.

1.5 Organisation of the Book

In the following chapter first the formal definitions of transformation systems are introduced, comprising the basic notions of control states, transitions, data states, and transformations. Their intended meaning and usage is shown in a series of examples. In Chapter 3 the properties of transformation systems are classified and investigated, introducing abstract syntactic means to formulate such properties. These refer to the notions introduced above: data invariants refer to data states, transformation rules refer to data state transformations, and control flow properties refer to the whole control flow level, possibly even including their labels. The development operations and relations are introduced in Chapter 4. They are also based on the structure of transformation systems introduced in the first chapter. Data states and transformations can be developed by adding or refining structure, or by hiding some parts, or, dually, making only designated parts visible. The behaviour can be developed by decreasing or increasing the non-determinism, adding or constraining behavioural capabilities, or refining steps by sequential or parallel composition of steps. Also the relation of developments and properties of transformation systems as discussed in Chapter 3 is presented in this chapter. That is, it is investigated what kinds of properties are preserved by which kinds of development steps, and, conditions are given that guarantee the preservation of certain properties by the corresponding development steps.

In Chapter 5 composition operations are introduced. Their definition also refers to the two-level structure of transformation systems. As discussed in Section 1.3, the first part of a composition of systems consists of their interconnection. This is given by an identification relation that addresses the data states and action sets and a synchronisation relation on the control states and transitions. The result of the application of a composition operation is given by an integration of composition operations for the data states and transformations on the one hand and transition systems on the other. Beyond the compositionality in the sense of structural transparency as discussed above, this also yields *compositional semantics* (see Section 5.4). That means the composition operations on the abstract syntactical level of transformation systems induce compositions at the associated semantic entities. The relations to the different kinds of properties of transformation systems are expressed as conditions for the *compositionality of properties* (Section 5.5), the ones to the development relations as *compositionality of developments* (Section 5.6). The former guarantees that the properties of components that can be expressed in their local languages are preserved by a composition, i.e., in the resulting global system the property also holds. Compositionality of developments means that local development steps of the components of a system induce a global development step of the system that contains the local ones, provided the local developments are compatible with the connection of the components.

An application of the integration approach to semi-formal software specifications is discussed in Chapter 6. Different UML models—class, statechart,

and sequence diagrams—are considered and analysed as viewpoint specifications. Their integration first requires them to be interpreted formally in the domain of transformation systems. Due to the complexity of the languages and the informal definition of their semantics, this obviously can be done only by examples. Moreover, the treatment will include certain assumptions on the intended meaning of the language constructs used, according to the idea of admissible interpretations. Since the idea of the chapter is just to demonstrate the integration approach and not to define a complete integrated UML semantics, this suffices, however. On the basis of the transformation system semantics the conceptual integration is then discussed. Different kinds of transformations of the semantics are presented that relate the specifications according to their relation to the global system specification. The transformations make possible the mutual adjustment of the specifications w.r.t. their name spaces, their structures, their levels of granularity, and their scopes, i.e., the parts of the system they address. These transformations thus define the syntactic and semantic correspondences of the specifications.

In Chapter 7 a conclusion and a short summary of the main concepts are given and possible further applications to concrete languages and methodological support are discussed. Then related approaches are discussed and compared with the transformation system approach. Finally some methodological issues are discussed that were crucial both for the development and the presentation of the transformation system approach.



<http://www.springer.com/978-3-540-40257-2>

Semantic Integration of Heterogeneous Software
Specifications

Große-Rhode, M.

2004, IX, 330 p., Hardcover

ISBN: 978-3-540-40257-2