Refinement and Consistency in Multiview Models

Heike Wehrheim

Department for Computing Science
University of Oldenburg
26111 Oldenburg, Germany

wehrheim@informatik.uni-oldenburg.de

Abstract. *Model transformations* are an integral part of OMG's standard for Model Driven Architecture (MDA). Model transformations should at the best allow for a seamless transition from high-level models to actual implementations. They are therefore required to be *behaviour preserving*: models (or the final implementation) at lower levels should adhere to the descriptions given in higher level models. Moreover, for complex systems models usually consists of descriptions of different views on the system. Consequently, different kinds of model transformations take place on different views, and together they should guarantee behaviour-preservation.
In this paper we discuss the applicability of formal methods to model transformations. Formal methods come with build-in notions of transformations between models, or more precisely, with *refinement* and *subtyping* concepts which provide means for comparing models on different levels with respect to their behaviour. Such notions can be applied as correctness criteria for evaluating model transformations. Moreover, refinement and subtyping concepts for *different* views can be shown to neatly fit together. This is achieved by giving a common semantics to all views which furthermore opens the possibility of checking consistency between them.

1 Introduction

The OMG's standard for model-driven architecture defines *models* to be the core concepts in software development. Model transformations are intended to provide the means for getting from high-level platform independent to lower level platform specific models and eventually implementations. Model transformations are expected to be *behaviour preserving*: lower-level models should reflect the behaviour of higher-level models.

When modelling complex systems there are usually multiple different views to be taken into account. A complex system has to fulfill several orthogonal requirements: on the static behaviour (data and operations), on its dynamic behaviour (adherence to protocols, scenarios), its timing behaviour, etc. . Thus a model of a complex system will usually consist of descriptions of several views.

Dagstuhl Seminar Proceedings 04101
http://drops.dagstuhl.de/opus/volltexte/2005/19

Consequently, a modelling language has to supply the designer with facilities for modelling multiple views and with (at the best *formal*) concepts supporting a stepwise design with multiple views. This does in particular apply to model transformations which should operate on all views, but concerns questions of consistency between views as well.

The UML partly fulfills these requirements on modelling languages. It offers possibilities for describing multiple views: the static behaviour can be modelled using class diagrams, protocols are denotable as state machines, scenarios in sequence diagrams. Concepts supporting a model-driven stepwise design with multiple views, in particular formal concepts, are however less developed. In this paper we will therefore discuss which concepts developed in the context of formal methods can be applied to a model-driven development with multiple views. It turns out that in particular *refinement* concepts, which play a central role in a formal approach to software development, can be seen to tightly match (certain forms of) model transformations. Refinement guarantees that the desired criterion of behaviour preservation is met. Thus a model transformation involving a change of a data type or a protocol, a splitting of an acitivity or an extension with new operations can be evaluated. Furthermore, questions of consistency between views (and its preservation under a transformation of the model) can be precisely studied in a formal framework.

The main focus of the paper lies on illustrating the applicability of formal methods to a model-driven design. We will therefore most often refrain from giving a precise definition of the formal concepts and instead explain where the concepts can be used. To this end we sketch some examples of model transformations on UML diagrams. For every transformation we provide a corresponding concept from formal methods covering this case.

We only cite the work which is directly applied in the examples but are aware of the fact that there are numerous other interesting approaches in this area.

2 Model transformations

In order to apply formal concepts to UML diagrams we need a formal semantics for them. For this, we first of all have to choose a semantic domain and afterwards define a translation of the diagram to this domain. The semantic domain (or formal method) should most closely reflect the modelling domain of the particular sort of diagram, i.e. a diagram for describing static behaviour should be given a semantics in terms of a state-based formal method whereas a diagram for protocols or interactions should be translated to a formal method good at modelling dynamic behaviour.

The examples of different views and model transformations elaborated on in the sequel are described by class diagrams, protocol state machines and sequence diagrams. The semantic domains for them are *Object-Z* (for class diagrams) and *CSP* (for state machines and sequence diagrams). Object-Z [12] is an object-oriented extension of Z [14], a a state-based specification language for describing states and operations on them. CSP [11, 7] is a process algebra developed for

modelling parallel communicating systems by means of process descriptions. The actual translation from the diagrams to the semantic domains is not of interest for the study undertaken in this paper. Of interest are the formal concepts coming with these languages, and whether and how they are applicable in a model-driven development.

The following model transformations will be studied. On the static model, i.e. class diagrams, we look at

- changes of data types (and corresponding operations),
- splitting of operations, and
- extension with new operations.

On the dynamic model, i.e. protocol state machines, we look at corresponding transformations which are

- changes of protocols,
- splitting of transitions, and
- extension with new transitions.

Change of a data type. Figure 1 shows a class A being part of a class diagram of one model and a corresponding class C of a different model. The change made in the model transformation concerns the type of the attribute *buffer* and consequently the definition of the method *choose* operating on this attribute. In class A attribute *buffer* is of type *set* (of some elements) and *choose* chooses just any element of this set, whereas in C *buffer* is of type (injective) *sequence* and *choose* always chooses the first element in the sequence.

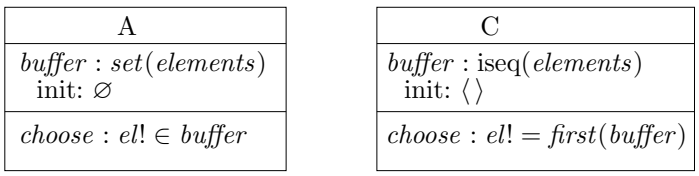


Fig. 1. Model transformation changing a data type

The question of interest for the correctness of the model transformation is the following

Is every behaviour of C a behaviour of A?

The formal concept which can be used for answering this question is that of *data refinement* coming from Object-Z [13, 1]. Data refinement is concerned with describing the allowed changes for attributes and operations when the externally observable behaviour is required to be preserved, or more precisely, when every behaviour of C has to have a corresponding behaviour in A.

Technically, this is achieved by imposing the following conditions on the two classes (which are the downward simulation conditions of Object-Z data refinement):

1. A *representation relation* R has to be given, which relates the attributes in A with corresponding ones in C. For the example, R is

   $$buffer_A = \bigcup_{1 \leq i \leq \# buffer_C} buffer_C[i]$$

   (the set *buffer* in A consists of the elements in the sequence *buffer* in C).
2. *Initialisation:* Every initial state in C must have a corresponding (via R) initial state in A (which holds since an empty sequence is related to an empty set).
3. Corresponding operations must have corresponding behaviour:
   - *Applicability: choose* in A is applicable whenever *choose* in C is applicable (which is true since both are applicable when the set/sequence, respectively, is non-empty),
   - *Correctness:* Whenever *choose* is applied in C, the result (concerning outputs and next state) corresponds with an application of *choose* in A (which holds since the first element of the sequence in C is an element of the corresponding set in A as well and thus can be chosen as output).

Change of protocol. Figure 2 shows two protocol state machines belonging to classes A and C, respectively, which are part of different models. The state machines describes the ordering of operations which are possible for a file. The state machine of A belongs to a higher-level model, it is nondeterministic and also models the case where the file to be opened is non-existent (and thus no read/write might be possible after *open*). In C the nondeterminism has been resolved, possibly by ensuring applicability of open on existing files only.

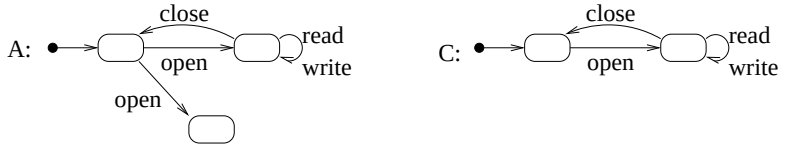


Fig. 2. Model transformation changing a protocol

The correctness criterion for such kind of changes is again:

Is every behaviour of C a behaviour of A?

and the formal concept applicable here is that of *process refinement* [11] coming from the process algebra CSP. Process refinement allows to reduce nondeterminism in a process. Depending on how discriminating the notion should be one can either use trace or failures refinement:

1. *Trace refinement:* The traces (possible sequences of operation execution) of C have to be a subset of those of A: $traces(C) \subseteq traces(A)$. This holds for the example since the state machine of C is contained in that of A.
2. *Failures refinement:* The failures of C (traces plus sets of operations which cannot be executed after a trace, i.e. are rejected) have to be a subset of those of A: $failures(C) \subseteq failures(A)$. This holds for the example but would for instance not hold for the reverse direction: the failures of A are not a subset of the failures of C since A might refuse *read* after *open* whereas C does not. Failures give additional information about the availability of operations and thus provide a more discriminating view on processes.

Splitting of operation in static model. Figure 3 shows two classes with operations for sending messages over a network. While class A contains a single operation *send*, class C uses two operations for one send, the first one being responsible for preparing the message for sending (e.g. adding certain headers) and the second one for actual transmission.

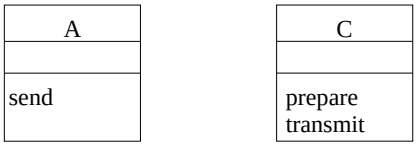


Fig. 3. Model transformation splitting an operation into two

The question to be ask on this type of model transformation is slightly different since the classes have different operations:

Has every behaviour of C a corresponding behaviour in A?

Here, corresponding means that the execution of *prepare* and *transmit* should have the same effect as that of *send*. The formal concept to be used in this case is that of *non-atomic data refinement* [2] from Object-Z. The conditions to be checked can be seen as an extension of those of ordinary data refinement:

1. Again a representation relation R has to be given.
2. With this R the usual data refinement conditions have to hold, which are 1) initialisation and 2) applicability and correctness of *prepare*; *transmit* with respect to *send*: *send* is applicable if and only if the successive execution of *prepare* and *transmit* is applicable, and the execution of *prepare*; *transmit* corresponds to one of *send*.
3. Furthermore, there are additional conditions for ruling out new behaviour in C which did not occur in A: 1) *Continuation*: once *prepare* has been executed *transmit* is applicable, and 2) *Proper starting: transmit* cannot be executed without prior execution of *prepare*.

Splitting of operation in dynamic model. Figure 4 shows a similar example of the splitting of an operation in a state machine. This time sending of messages is part of a simple protocol involving the receipt of messages as well.

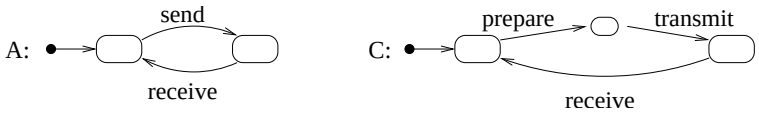


Fig. 4. Model transformation splitting a transition of a protocol

Again sending is split into preparation and transmission. The question is therefore

Has every behaviour of C a corresponding behaviour in A?

The formal concept applicable in this case is that of *non-atomic process refinement* [3] from CSP. Correspondence of behaviours in this setting is declared via an operation $\uparrow$ on traces and failures of processes which (roughly) maps the sequence *prepare*; *transmit* in traces to *send* (for an example: (*prepare*; *transmit*; *receive*; *prepare*; *transmit*) $\uparrow =$ *send*; *receive*; *send*). Technically, the conditions to be checked are an extension of ordinary process refinement:

1. Depending on whether traces or failures are to be used we either check $traces(C) \uparrow \subseteq traces(A)$ or $failures(C) \uparrow \subseteq failures(A)$.
2. In addition, two conditions corresponding to those of non-atomic data refinement have to be checked: 1) *Continuation:* after a trace of C in which *prepare* has occured but *transmit* not, *transmit* may not be refused; 2) *Proper starting:* There are no traces in C in which *transmit* occurs without a prior *prepare*.

Extension of static model. The model transformation in Figure 5 replacing class A by class C is concerned with an extension of the class with new methods. The classes model buffers with methods *put* and *get*, and class C in addition with a method *full* querying the contents of the buffer.
It is obvious that class C now cannot have exactly the same behaviour as A anymore (since *full* cannot be called on A). The question to be asked for correctness of the transformation is thus slightly rephrased. Instead of requiring behaviour preservation, we require *substitutivity*:

Can a user of A use C as if it were A?

If a client uses C as if it were A then no difference to A should be detectable. The formal concept achieving substitutivity is *subtyping* [9], in case of classes it

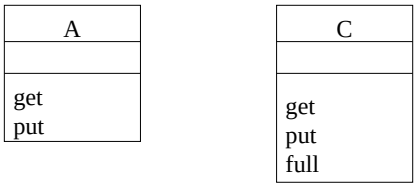


Fig. 5. Model transformation extending a class with new operations

is state-based subtyping from Object-Z [15]. Subtyping can be seen as a combination of refinement and inheritance: as far as existing methods and attributes are concerned they may be changed according to the data refinement rules. For the new methods it is required that they

- either do not modify attributes at all (which is the case in our example since *full* is a query method),
  or
- they only modify new attributes. This allows them to access values of attributes already defined in A, but not to change them.

Extension of behaviour model. The last model transformation to be considered is the case of extension for the dynamic model. Figure 6 gives the behaviour-oriented version of the extension described in the previous example. A state machine is extended with a parallel component independently executing the method *full*.

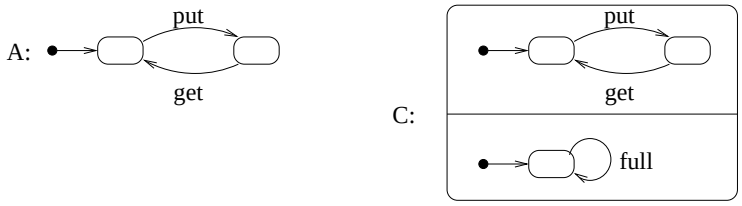


Fig. 6. Model transformation extending a protocol with new transitions

The question is again that of substitutivity:

Can a user of A use C as if it were A?

The formal concept applicable here is the behaviour-oriented version of subtyping: when the state machines are given a CSP semantics they can be compared via *behaviour-oriented subtyping* [16]. Technically, this notion is defined on the failure sets of the processes for C and A:

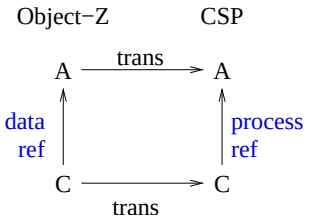
$$failures(C) \subseteq failures(A \mid\mid CHAOS(full))$$

The notation || stands for parallel composition. A weaker notion can be defined by using the traces of processes instead of the failures (see [10]). The definition says that the behaviour of C has to be a process refinement of the behaviour of A interleaved with executions of *full* at any time. Hence execution of *full* in C may not interfere with the "old" behaviour modelled in A.

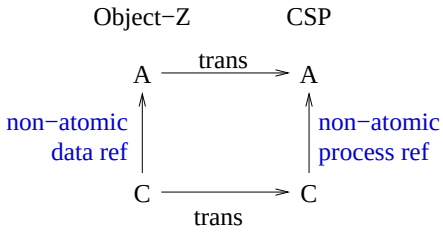
Integrating the views and transformations. These rather small examples have sketched how concepts from formal methods can be applied for evaluating model transformations once the diagrams have been supplied with a formal semantics. The question still to be answered is, however, what is the relationship between these different concepts being applied to different views? What is the impact of transformations on separate views on the overall system? In order to formally define this, *one* semantic domain has to be chosen to which all views of one model can be translated. An appropriate combination of the semantics of separate views then gives the semantics of the system model. For class diagrams and state machines a possible common semantic domain is CSP. The semantics can be obtained by translating Object-Z to CSP and afterwards combining the CSP semantics of the state machine with that of the class diagram:

$$CSP(ClassDiagram) \mid\mid CSP(StateMachine)$$

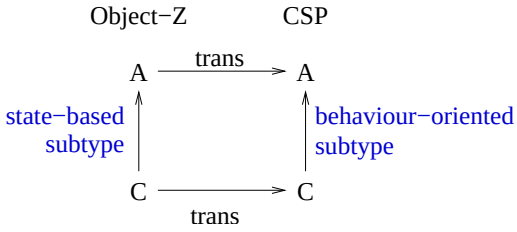
This idea follows an approach taken in CSP-OZ, a combination of CSP and Object-Z [4]. As a consequence, all Object-Z concepts which have been used for model transformations need to be mapped to CSP. Fortunately, this works quite well. For all three kinds of model transformations (change, splitting, extension) correspondence results between the state-based and the behaviour-oriented definitions have been proven. More precisely, the following result has been shown for data and process refinement [6, 8]



Whenever a class C is a data refinement of a class A then the corresponding CSP processes of C and A (obtained iva a translation from Object-Z to CSP) are in a process refinement relationship. This result carries over to the cases of non-atomic refinement [3]:



as well as subtyping [15]:

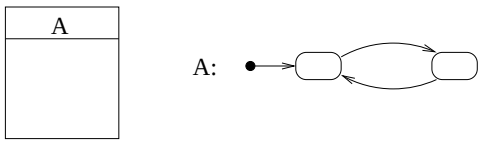


Moreover, all three notions of refinement/subtyping can be shown to be preserved under parallel composition, which is the operator used to combine the semantics of views. As a consequence, it is possible to separately apply the state-based concepts on the class diagrams and the behaviour-oriented concepts on the state machines while still achieving a correct transformation on the complete model.

3 Consistency

Correctness of model transformations is sometimes also referred as *vertical consistency*. In this section we are concerned with *horizontal consistency* between views. Views partially define the behaviour of a system. The parts they define might not necessarily be disjoint, thus the question arises whether the views within one model specify contradictory requirements. A formal semantics for views is useful for answering this type of question as well.

Again, we only sketch this on very small examples. The first example concerns classes and associated state machines.

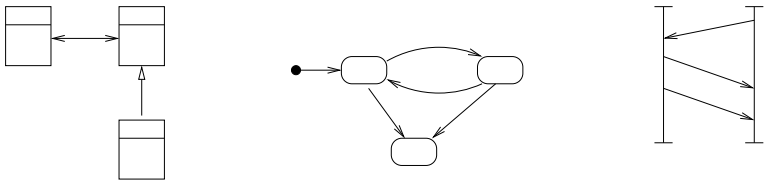


A model with a class diagram and state machines has two partially overlapping views: the state machine restricts the order of method executions and preconditions of methods might restrict it as well. For checking whether these requirements can be fulfilled at the same time, the semantics for the whole system can be checked:

- Set the system semantics to $S = CSP(ClassDiagram) \parallel CSP(StateMachine)$,
- check whether there is a *deadlock* in S but none in $CSP(ClassDiagram)$ and $CSP(StateMachine)$. If the answer is yes, then the class diagram and the state machines impose conflicting requirements on method executions.

This check can be performed automatically using the CSP modelchecker FDR [5].

The second example concerns consistency between a system model consisting of a state machine and a class diagram and a sequence diagram describing a possible scenario.



They are consistent if the scenario is possible in the model. This can be checked as follows:

- Again set $S = CSP(ClassDiagram) \parallel CSP(StateMachine)$,
- take $P = CSP(SequenceDiagram)$,
- and check whether $traces(P) \subseteq traces(S)$, i.e. whether the behaviour described in the sequence diagram is part of the behaviour of the system. Again this can be checked with FDR.

Furthermore, a formal semantics can be used to study what types of consistency are preserved under what model transformations. For the two examples given above the theory from the process algebra CSP immediately gives us the following two results:

1. Consistency between classes and state machines is preserved under refinement: If A and C are both models comprising a class and a state machine, the model A is consistent and there is a refinement relationship between the corresponding views of C and A, then C is consistent as well.
2. Consistency between a system model and a scenario might not be preserved under refinement: Refinement allows for a reduction of behaviour, thus a scenario possible in A might not be possible in C anymore, even if C is a refinement of A.

4 Conclusion

The purpose of this paper was to illustrate on several small examples from the UML what concepts coming from the area of formal methods can be applied to what questions arising in a model-based design of complex systems. In particular, several kinds of model transformations on different views as well as consistency

between views has been considered. As future work we plan a more systematic study of these issues, in particular on preservation of consistency under model transformations.

Acknowledgement. I am grateful to John Derrick and Holger Rasch for joint work which became part of this paper.

References

1. J. Derrick and E. Boiten. *Refinement in Z and Object-Z, Foundations and Advanced Application*. Springer, 2001.
2. J. Derrick and H. Wehrheim. Using coupled simulations in non-atomic refinement. In *ZB 2003: Formal Specification and Development in Z and B*, number 2651 in LNCS, pages 127–147. Springer, 2003.
3. J. Derrick and H. Wehrheim. Non-atomic refinement in Z and CSP, 2004. draft.
4. C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.
5. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, Oct 1997.
6. J. He. Process simulation and refinement. *Formal Aspects of Computing*, 1(3):229–241, 1989.
7. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
8. M.B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.
9. B. Liskov and J. Wing. A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811 – 1841, 1994.
10. E.-R. Olderog and H. Wehrheim. Specification and inheritance in CSP-OZ. In F.S. de Boer, M. Bonsague, and W.P. de Roever, editors, *Formal Methods for Components and Objects*, volume 2852 of *LNCS*, pages 361–379. Springer, 2003.
11. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
12. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.
13. G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M. Hinchey and Shaoying Liu, editors, *Int. Conf. of Formal Engineering Methods (ICFEM)*, pages 293–302. IEEE, 1997.
14. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International Series in Computer Science, 2nd edition, 1992.
15. H. Wehrheim. Relating State-based and Behaviour-oriented Subtyping. *Nordic Journal of Computing*, 9(4):405–435, 2002.
16. H. Wehrheim. Behavioral subtyping relations for active objects. *Formal Methods in System Design*, 23:143–170, 2003.