Consistency between UML Classes and Associated State Machines

Holger Rasch and Heike Wehrheim

Universität Oldenburg, Fachbereich Informatik, 26111 Oldenburg, Germany
{rasch,wehrheim}@informatik.uni-oldenburg.de

Abstract. One of the main advantages of the UML is its possibility to model different views on a system using a number of diagram types. The various diagrams can be used to specify different aspects and their combination together makes up the complete system description. This does, however, always pose the question of consistency: it may well be the case that the designer has specified contradictory requirements which can never be fulfilled together.

In this paper, we study consistency problems arising between static and dynamic diagrams, in particular between a class and its associated state machine. We propose a definition of consistency which is based on a common formal semantics for both classes and state machines, and discuss the merits and disadvantages of this definition on a simple case study. On this case study we furthermore show how consistency checks can be supported by a model checker. Finally we present consistency preserving transformations on the static as well as the dynamic part.

1 Introduction

The UML (Unified Modeling Language) [1] is an industrially accepted standard for object-oriented modelling of large, complex systems. The UML being a unification of a number of modelling languages offers various diagram types for system design. The diagrams can roughly be divided into ones describing static aspects of a system (classes and their relationships) and those describing dynamic aspects (orderings of method invocations within a class or between different classes). For instance, class diagrams fall into the first and state machines or sequence diagrams into the second category. While in general it is advantageous to have these different modeling facilities at hand, this also poses some non-trivial questions on designs. The different views on a system as described by different diagrams cannot be seen as completely orthogonal, and may thus in principle be inconsistent. The different diagrams all impose certain restrictions on the allowed system behaviour, and these restriction may at least in part be contradictory.

While it is an accepted fact that consistency is an issue in UML-based system development, appropriate definitions of consistency are still an open research

⋆ This research is partially supported by the DFG under grant OL 98/3-1.
topic. In this paper we propose and discuss definitions of consistency between static and dynamic diagrams, more precisely, between a class and its associated state machine. We aim at a formal definition of consistency, and thus will first give a formal semantics to both class definitions and state machines. This in particular requires the use of a formal specification language for classes to precisely fix the types of attributes and the semantics of methods. Since the UML does not prescribe a fixed syntax for attributes and methods of classes we feel free to choose an arbitrary one. Here, we have chosen Object-Z [12], an object-oriented specification language appropriate for describing static aspects of systems.

Since we aim at a formal definition of consistency we need a common semantic domain for classes and state machines in which we can formulate consistency. This common semantic domain is the failure divergence model of the process algebra CSP [9, 11]. This choice has a number of advantages: on the one hand a CSP semantics for Object-Z is already available [6], on the other hand CSP has a well developed theoretical background as well as tool-support in the form of the FDR model checker [8]. For (restricted classes of) state machines a CSP semantics has already been given in [3] and a CSP semantics for an even larger class of state machines is currently under development by the first author. The first step during a consistency check thus is always the translation of the class description and the state machine into CSP.

The translation gives us the CSP descriptions of two different views on a class: one view describing attributes of classes and the possible effects of method execution (data dependent restrictions) and another view describing allowed orderings of method executions. These two descriptions are the basis for defining consistency. Our main proposal for consistency is deadlock freedom of the parallel composition of these two CSP processes. Deadlock-freedom guarantees that the restrictions on the class given by the static and the dynamic diagram are not completely contradictory: the class can never reach a state in which no further action is possible anymore. This is certainly a basic requirement which consistent diagrams should always fulfill. However, as we illustrate by a small case study this might not be sufficient. It for instance allows that certain parts of the state machine may never be reached or certain methods of the class never be executed. On the case study we thus discuss how this definition of consistency may be extended to cover more than pure absence of deadlocks. We furthermore show how the model checker for CSP can be used to automatically perform such consistency checks.

During the development process a system model may gradually be altered until a model close to the actual implementation domain is reached. When consistency has been shown for a model developed in earlier phases successive transformations of the model should at the best preserve consistency. For a definition of consistency as deadlock freedom this can be achieved by using the well developed refinement concepts of the two specification languages. In Object-Z data refinement is used to replace abstract models by ones closer to an implementation, in CSP failure divergence refinement guarantees that during evolution steps no new behaviour is introduced into a model. These two types of refinement can
be separately applied on the static and the dynamic model, respectively. Since data refinement in Object-Z is known to induce failure divergence refinement on the CSP processes obtained after the translation [13, 10], these evolution steps preserve consistency (defined as deadlock freedom). This is also illustrated by means of our case study.

The paper is structured as follows. In the next section we introduce the small case study and on it give a brief introduction into Object-Z. Section 3 explains the translation of both the class and the state machine into CSP. The result of this translation is the basis for defining and discussing consistency in Sect. 4. Finally, Sect. 5 presents consistency-preserving transformations.

2 Case Study

The small case study which we use to illustrate consistency definitions concerns the modelling of an elevator class. It is a typical example of a class with a model that contains a static part (attributes and methods) as well a dynamic part which describes allowed orderings of method executions.

The static class diagram of the elevator specification shown in Figure 1 models the class with its attributes and methods.

![Class Elevator](image)

Fig. 1. Class Elevator

It attributes are requests (to store the current requests for floors), pos (the current position of the elevator) and tar (the next target). It has a method request (to make requests for particular floors), a private method start (to start the elevator once there is a pending request) and a method passed which is invoked when the elevator is moving and has passed a certain floor.

The Object-Z specification below\(^1\) gives a more precise description of this class. It formally specifies the types of the attributes and the semantics of methods. For each method we give a guard (an enabling schema) defining the states (i.e. valuations of attributes) of the class in which the method is executable and

\(^1\) To be more specific, it is the Object-Z part of a CSP-OZ specification [6].
an *effect* defining the effects of method execution on values of attributes. In general, a model can define a method to be *nondeterministic*, i.e. underspecify its effect on the attributes. This nondeterminism is usually resolved in later phases of the development (see section on model transformations).

The specification starts with the definition of type *Floor*.

\[
\begin{align*}
\text{minFloor}, \text{maxFloor} : \mathbb{N} \_1 \\
\text{minFloor} < \text{maxFloor}
\end{align*}
\]

*Floor == minFloor..maxFloor*

The class specification itself consists of an interface, a state schema, an initialisation schema and enable and effect schemas for methods. The interface consists of the method of the class itself (with keyword *method*) plus those called by the class (keyword *chan*). Input parameters of methods are marked with ?. The channels specified for our class (*closeDoor, openDoor, stop, up* and *down*) can all be seen as methods of a motor class called by the elevator to control its own movement. The schema (box) following the interface is the state schema and defines the attributes of the class with their types. The enable and effect schemas are prefixed by the corresponding keyword.

```
<table>
<thead>
<tr>
<th>Elevator</th>
</tr>
</thead>
<tbody>
<tr>
<td>method request : [ f? : Floor ]</td>
</tr>
<tr>
<td>method passed, start</td>
</tr>
<tr>
<td>chan closeDoor, openDoor</td>
</tr>
<tr>
<td>chan stop, up, down</td>
</tr>
<tr>
<td>requests : \mathbb{F} Floor</td>
</tr>
<tr>
<td>pos, tar : Floor</td>
</tr>
<tr>
<td>Init</td>
</tr>
<tr>
<td>pos = tar = minFloor</td>
</tr>
<tr>
<td>requests = \emptyset</td>
</tr>
<tr>
<td>effect_request</td>
</tr>
<tr>
<td>( \Delta(\text{requests}) )</td>
</tr>
<tr>
<td>( f? : Floor )</td>
</tr>
<tr>
<td>( f? \neq \text{pos} \Rightarrow \text{requests}' = \text{requests} \cup {f?} )</td>
</tr>
<tr>
<td>( f? = \text{pos} \Rightarrow \text{requests}' = \text{requests} )</td>
</tr>
<tr>
<td>enable_start</td>
</tr>
<tr>
<td>( \text{requests} \neq \emptyset )</td>
</tr>
<tr>
<td>( \Delta(\text{tar, requests}) )</td>
</tr>
<tr>
<td>( \text{tar}' \in \text{requests} )</td>
</tr>
<tr>
<td>( \text{requests}' = \text{requests} \setminus {\text{tar}'} )</td>
</tr>
<tr>
<td>enable_openDoor</td>
</tr>
<tr>
<td>( \text{pos} = \text{tar} )</td>
</tr>
<tr>
<td>enable_closeDoor</td>
</tr>
<tr>
<td>( \text{pos} \neq \text{tar} )</td>
</tr>
</tbody>
</table>
```
If an enabling schema for a method is left out it corresponds to a guard which is always true. Effect schemas refer to the values of attributes after execution of a method by using primed versions of the attributes. The $\Delta$-list of a method specifies the attributes which are changed by method execution. Looking at the specification of method start, we for instance see that start is only enabled when there are pending requests and upon execution of start one request is chosen (the target is set to one of the requests) and this request is eliminated.

This is the static part of the model, specified by a class diagram. It fixes all data-dependent aspects of the class. Next, we model the dynamic view on an elevator. Figure 2 shows the state machine for class Elevator. This is an extended protocol state machine, which in addition to specifying the order for calls to the methods of the corresponding class also includes the methods called by (instances of) the class.

\[ \begin{align*}
\text{enable}_\text{passed} & \quad \text{effect}_\text{passed} \\
\text{pos} \neq \text{tar} & \quad \Delta(\text{pos}) \\
\end{align*} \]

\[ \begin{align*}
\text{pos} > \text{tar} & \Rightarrow \text{pos}' = \text{pos} - 1 \\
\text{pos} < \text{tar} & \Rightarrow \text{pos}' = \text{pos} + 1 \\
\end{align*} \]

\[ \begin{align*}
\text{enable}_\text{up} & \quad \text{enable}_\text{down} \\
\text{pos} < \text{tar} & \quad \text{pos} > \text{tar} \\
\end{align*} \]

\[ \text{enable}_\text{stop} \quad \text{pos} = \text{tar} \]

\[ \text{Fig. 2. Protocol of class Elevator} \]

It consists of two submachines in parallel. The first submachine specifies the allowed sequences in the movements of the elevator. First, the elevator starts
(this means picking a target from the available requests), the door is closed, and
the elevator is sent either up or down. During movement some floors are passed
and eventually the elevator is stopped and the door opened again. Requests can
be made at any time, thus the state machine specifying requests is concurrent
to the movements state machine.\(^2\) This completes the model for class Elevator.

**Remark.** The specification is split into 2 parts, both of which are incomplete
and influence the other part. This split is natural in the sense that the order of
the methods could have been specified in the Object-Z part alone, but for this
additional flags/counters would have to be introduced, making the specification
more difficult to understand.

Furthermore, while the Object-Z diagram above is the obvious way to specify
the state space of class Elevator and the transformations on it, this can also be
done in a more UML like way using again a state machine (sketched in Fig. 3).

\[(
\begin{array}{l}
\text{entry} \quad \text{initialize according to schema Init} \\
\text{request}(f : \text{Floor}) \quad \text{store requested floor in requests} \\
\text{passed} \quad \text{pos} \neq \text{tar} \quad \text{update current position} \\
\text{start} \quad \text{requests} \neq \emptyset \quad \text{pick a target and remove it from requests} \\
\text{closeDoor} \quad \text{pos} \neq \text{tar} \\
\text{openDoor} \quad \text{pos} = \text{tar} \\
\text{stop} \quad \text{pos} = \text{tar} \\
\text{up} \quad \text{pos} < \text{tar} \\
\text{down} \quad \text{pos} > \text{tar}
\end{array}\)
\]

**Fig. 3.** Object-Z part as a state machine

In contrast to the protocol state machine of Fig. 2 this state machine does
not explicitly specify an order of events. It has only one state, since on this
abstract level the state space of size \(2^n n^2\) with \(n = \text{maxFloor} - \text{minFloor} + 1\)
is still hidden in the variables \(\text{pos}, \text{tar}\) and \(\text{requests}\). The guards and actions
 correspond to the enable and effect schemata of (Object-Z) class Elevator.

### 3 Translation into the Semantic Model

The first step in checking consistency of class definitions and state machines is
their translation into a common semantic domain. The semantic domain we have
chosen here is the failure divergence model of the process algebra CSP. Instead
of directly giving a failure divergence semantics to classes and state machines we

\(^2\) The formal parameter for method request is not shown here, since it is irrelevant for
the protocol.
translate them into CSP. This way the result remains readable and is furthermore amenable to checks with the FDR model checker for CSP. We start with a brief re-cap of CSP as far as we need it in the translation.

CSP is a formal method for specifying and reasoning about processes. Each process is built over some set of communication events, using operators like sequential composition or choice to construct more complex processes. In the object-oriented setting here, in which we use CSP, communication events are made up of method names together with parameters and return values.

**Events** take the form \( m.v_1, \ldots \) where \( m \) is the name of a method and \( v_1, \ldots \) are values of input parameters or return values.

**Prefixing** is a simple form of sequential composition. The process \( ev \rightarrow P \) can first execute the event \( ev \) and afterwards behaves like \( P \).

**Choice** is used to describe alternatives in the behaviour of processes. CSP has an operator for nondeterministic choice (\( \ominus \), under control of the process alone) as well as for external choice (\( \ominus \), influenced by the environment).

**Parallel composition**, denoted by \( ||A || \), is used to set processes in parallel, requiring synchronisation on all communication events in the set \( A \). The operator \( || \) stands for *interleaving*, i.e. parallel composition with empty synchronisation set.

### 3.1 Translation of the Object-Z specification

The translation of class Elevator follows a general translation scheme for Object-Z developed in [7]. The basic idea is that a class is translated into a parameterised process. The parameters of the process are the attributes (the state space) of the class. For each method of the class (like passed) and each method called by the class (like closeDoor) a separate channel is used. Parameters and return values are encoded as data sent on the channels.

Each method is translated to a recursion, prefixed with the appropriate event and possibly modifying the process parameters according to the effect schema of the method. All enabled methods are offered to the environment using external choice. Not offering the disabled methods, i.e. the translation of the enable schemata, is achieved by using a conditional (\( P < b \triangleright Q \) evaluates to \( P \) if \( b \) is true, \( Q \) otherwise) to choose the special process STOP – which does not communicate at all – if the enabling condition is not satisfied. Since STOP is the neutral element of \( \ominus \) this means removing the disabled method from the external choice.

At first it might seem odd that even the methods called by the class are subject to external choice, but in this case it just means that the process cannot deadlock until no method is enabled at all. Internal nondeterminism possibly needed for updating the state space or choosing parameters for method calls is always 'below' the external choice and must not influence the set of offered events.

---

3 Method calls are modeled as CSP communication.
Finally the Init schema is translated by specifying the initial process parameters. Again, this requires a nondeterministic choice over all possible valuations, if there are more than one.

The result of this part of the translation is shown below. The attributes \textit{pos}, \textit{tar} and \textit{requests} of the Object-Z are represented by the process parameters \(p\), \(t\) and \(R\), respectively.\footnote{The renaming keeps the CSP expression small and otherwise has no meaning.}

\[
\begin{align*}
\text{PROC}_Z &= Z(\text{minFloor}, \text{minFloor}, \varnothing) \\
Z(p, t, R) &= \begin{cases} 
\text{openDoor} \rightarrow Z(p, t, R) \ll p = t \gg \text{STOP} \\
\Box \text{closeDoor} \rightarrow Z(p, t, R) \ll p \neq t \gg \text{STOP} \\
\Box \text{stop} \rightarrow Z(p, t, R) \ll p = t \gg \text{STOP} \\
\Box \text{request}\? f \rightarrow Z(p, t, R \cup \{f\}) \ll f \neq p \gg R \\
\Box \left( \bigcap_{t' \in R} \text{start} \rightarrow Z(p, t', R \setminus \{t'\}) \ll R \neq \varnothing \gg \text{STOP} \right) \\
\Box \text{passed} \rightarrow Z(p + 1, p < t \gg p - 1, t, R) \ll p \neq t \gg \text{STOP} \\
\Box \text{up} \rightarrow Z(p, t, R) \ll p < t \gg \text{STOP} \\
\Box \text{down} \rightarrow Z(p, t, R) \ll p > t \gg \text{STOP}
\end{cases}
\end{align*}
\]

3.2 Translation of the protocol state machine

Now the protocol state machine for class Elevator has to be translated to CSP as well. While in general it is a non trivial task to translate a UML state machine to CSP even if it does not use parameterized events, guards and actions (as it is the case for the kind of protocol state machine used in our context), for this example a simple \textit{ad hoc} translation is possible:

\[
\begin{align*}
\text{PROC}_{SM} &= SM^0_a ||| SM^0_b \\
SM^0_a &= \text{request}\? f \rightarrow SM^0_a \\
SM^0_b &= \text{start} \rightarrow SM^1_b \\
SM^1_b &= \text{closeDoor} \rightarrow SM^2_b \\
SM^2_b &= \text{up} \rightarrow SM^3_b \Box \text{down} \rightarrow SM^3_b \\
SM^3_b &= \text{passed} \rightarrow SM^3_b \Box \text{stop} \rightarrow SM^4_b \\
SM^4_b &= \text{openDoor} \rightarrow SM^0_b
\end{align*}
\]

For each state of diagram (aside from initial or compound states) a process is defined which consists of an external choice over the events of all outgoing transitions, prefixed to the process which corresponds to the target state of
the respective transition. If a state has only one successor, the external choice is dropped. The processes corresponding to the initially active states in the concurrent state are put in parallel using interleaving.\footnote{This would not be possible, if the events sets of the concurrent submachines were not disjoint, since UML state machines always fire the maximal possible set of transitions. On the other hand synchronising on the events in the intersection would neither be possible, since this would prevent advancing in only one submachine due to one of these events.}

The parameter $f$ for event request in $SM^0$ which does not appear in the diagram is introduced here for convenience. In general renaming would be used to map all events for a method with parameters to just one event per method for the state machine, because the parameters are irrelevant for the protocol and smaller event sets make model checking faster.

### 3.3 Resulting specification

As a last step in the translation of the Object-Z class and its protocol state machine the processes obtained for each one are put in parallel

$$PROC = 
\bigparallel_{E} \bigparallel_{E} PROC_Z \parallel PROC_{SM}$$

synchronising on the set $E$ of all events (methods, including parameters) specified in the Object-Z class using method or chan declarations. In this case parallel composition can be viewed as a conjunction, that is $PROC$ accepts a method call (sent or received) iff both $PROC_Z$ and $PROC_{SM}$ accept it. This is the intended semantics of the combined specifications.

There is one problem though: with the standard UML state machine semantics it is not possible to obtain this exact behaviour, because it defines an event to fire the maximal non conflicting set of transitions, which may be empty. So in the UML events are simply discarded if there is no enabled transition. Therefore the translation given for the state machine is not correct with respect to the UML semantics, since the resulting process does not discard events.

Since our aim is formal analysis and verification of specifications like the one given as an example, including the detection of protocol violations, quietly discarding events is not an option. There are (in the model we chose) two possibilities to model protocol violations without having to introduce explicit failure states and transitions in the state machine; either they lead to undefined behaviour (divergence), or they cannot be performed at all, that is, the process deadlocks if offered the ‘wrong’ event. We decided for the latter one: an event must fire a transition of the state machine or deadlock occurs.

### 4 Notions of Consistency

Using the example given above we now discuss notions of consistency for specifications consisting of Object-Z classes and protocol state machines. We only
refer to the result of the translation, the semantics of PROC. Because of the semantic model chosen, this means the traces, deadlocks and divergences of PROC. Furthermore, consistency of either part of the specification \( \text{PROC}_Z \) or \( \text{PROC}_{SM} \) alone is not considered here. Only the effect of imposing a protocol on the Object-Z class is addressed.

The subprocesses of PROC synchronize on the set of all methods specified using chan or method declarations in the Object-Z class. This means a complete protocol must be specified; every method has to show up on a transition for a specification to be well formed. On the other hand, the protocol state machine must not contain transitions labeled with events \textit{not} corresponding to a method, since this could shadow unwanted behaviour. For the rest of this section we assume the specification to be well formed with respect to the two requirements above.

### 4.1 Basic consistency

What does consistency mean in our context? Informally, consistency here is about how the explicit specification of sequences of method invocations in the state machine and the implicit specification through the enabling conditions in the Object-Z part fit together. Formally it is some property of \( \text{PROC} \).

For the notion of basic consistency it is beneficial to view parallel composition of CSP processes as conjunction. So the consistency of \( \text{PROC} \) is the consistency of \( '\text{PROC}_Z \land \text{PROC}_{SM}' \). It is immediately clear that if this ‘formula’ is not satisfiable, the corresponding specification is \textit{inconsistent}. Translated to the terms of the semantic model this means: \( \text{PROC} \) will always deadlock, i.e. any sequence of events offered to \( \text{PROC} \) will lead to deadlock.

Strictly sticking to the analogy, consistency now would correspond to satisfiability. For the specification to be consistent it would suffice to have at least one trace of \( \text{PROC} \) not leading to deadlock. Since in general deadlock occurs several steps after performing the ‘wrong’ event, successful usage of the system specified would amount to guessing one trace from the infinite set of traces. This is clearly not an acceptable definition of consistency.

For a specification to be consistent we thus require \( \text{PROC} \) to be deadlock free; after any sequence of events performed by \( \text{PROC} \) there has to be at least one event to continue the sequence, that is, no trace may have \( E \) (the set of all methods) as the refusal set. This can be regarded the standard notion of consistency in the context of behavioural specifications and is used by other authors as well, for instance [4, 5], where it is applied to the behaviour of different entities of a model acting together.

**Definition 1.** A specification consisting of an Object-Z class and an associated state machine has the property of basic consistency iff the corresponding process in the semantic model is deadlock free.

Using the FDR model checker for CSP it can now be checked, if the example is consistent according to Def. 1. The result is, that \( \text{PROC} \) for the example given is indeed deadlock free, so the specification has the property of basic consistency.
Remark. Since in general a protocol state machine may contain final states on its top level, modeling successful termination of the class’ life cycle, the translation given is not sufficient: termination of \( \text{PROC}_{\text{SM}} \) would lead to deadlock of \( \text{PROC} \). This means that such a final state must not be translated to the special process \( \text{SKIP} \) (signaling termination), but instead to a process \( \mu t. t \rightarrow T \) with \( t \not\in E \). So termination is mapped to an infinite trace of a new event \( t \). For final states below the top level this does not apply, since they have a different meaning.

This way of handling successful termination in the state machine ensures the applicability of Def. 1 to all kinds of specifications consisting of Object-Z classes and associated protocols, including those containing finite length behaviour.

4.2 Extended Consistency

Using the example again, we now analyse the effect of a specific sequence of events on \( \text{PROC}_Z \) starting from the initial state.

\[
\begin{align*}
\text{request.1} & \rightarrow Z(0,0,\emptyset) \\
\text{start} & \rightarrow Z(0,0,\{1\}) \quad [\text{requests} \neq \emptyset] \\
\text{closeDoor} & \rightarrow Z(0,1,\emptyset) \quad [\text{pos} \neq \text{tar}] \\
\text{up} & \rightarrow Z(0,1,\emptyset) \quad [\text{pos} < \text{tar}] \\
\text{request.1} & \rightarrow Z(0,1,\{1\}] \\
\text{passed} & \rightarrow Z(1,1,\{1\}) \quad [\text{pos} \neq \text{tar}] \\
\text{stop} & \rightarrow Z(1,1,\{1\}) \quad [\text{pos} = \text{tar}] \\
\text{openDoor} & \rightarrow Z(1,1,\{1\}) \quad [\text{pos} = \text{tar}] \\
\text{start} & \rightarrow Z(1,1,\emptyset) \quad [\text{requests} \neq \emptyset]
\end{align*}
\]

At this point the elevator should be able to perform \( \text{closeDoor} \), but this method is only enabled if \( \text{pos} \neq \text{tar} \), so the only method enabled at this point is \( \text{request} \). Since \( \text{request} \) does not modify \( \text{pos} \) or \( \text{tar} \) this condition will endure infinitely. This means, the only possible trace after this prefix is \( \langle \text{request} \rangle^\ast \).

Regarding the state machine this means the above trace leads to a deadlock in one of the submachines. Again, looking at the diagram it seems to be intuitively clear, that this behaviour is not consistent, since that submachine clearly specified infinite behaviour. (It has nothing to do with leaving out fairness assumptions as well; it is perfectly acceptable behaviour for the elevator to only perform \( \langle \text{request} \rangle^\ast \) if the environment chooses to only offer \( \text{request} \). The problem is, that the environment cannot choose any other events.)

Thus one is tempted to require for a specification to be consistent, that no submachine may deadlock, even though it is not clear how to specify this property in the semantic model, quite apart from checking it with the tool FDR.

Unfortunately this requirement is unsound. Firstly, a protocol containing infinite behaviour (cycles) does in general not require this behaviour (for instance the \( \text{passed} \) loop in the example). This would mean the requirement is too strong. Secondly, concurrent submachines can always be combined into one non concurrent state machine; in the example for instance, the submachine containing the
request loop could be eliminated by adding a self-transition labeled request to all states in the other submachine. This would mean the requirement is too weak. Even worse, state machines with equivalent behaviour could give different results for the consistency check, depending on whether they use concurrency or not.

The reason for this problem is that consistency is a universal property, which we want any specification to have. The property ‘no deadlock in any submachine’ in contrast is not universal. This would pose no problem, if it could be inferred from the specification whether the property had to be established, but for this the expressive power of Object-Z and state machines is insufficient. Of course there are many other behavioural properties for which this also holds.

One solution would be to extend the protocol state machines, introducing additional elements in the specification, which declare the intentions of the modeller with respect to the behaviour.

Despite the above problems, there is still one property not yet considered, which is a universal property and thus relevant for consistency: For any method introduced in the Object-Z part using method or chan declarations there has to exist a behaviour which includes this method, that is, it must be possible for a method to be used at least once. Otherwise it could be removed from the specification without changing the semantics, and this is clearly not desired.

Even though it is not possible to obtain a significantly extended notion of consistency for the reasons described above, the last observation leads to the following definition.

**Definition 2.** A specification consisting of an Object-Z class and an associated state machine is consistent iff it has the property of basic consistency and for each method declared in the class there is a sequence of method calls allowed by the specification which includes this method.

This property can be checked using the FDR model checker, with the aid of the auxiliary process

\[ \text{ONCE}(c) = ?x:|c| \to \text{div} \]

which can perform exactly one event from the channel \( c \) it was instantiated with and then diverges.\(^6\) It models the call of method \( c \) with any valuation for the parameters. Now this process is put in parallel with \( \text{PROC} \), synchronising only on the relevant events:

\[ \text{OCC}(c) = \text{ONCE}(c) \parallel \text{PROC} \]

The specification is consistent according to Def. 2, if \( \text{PROC} \) is deadlock free and \( \text{OCC}(c) \) can diverge for each method \( c \).

\(^6\) The special process \( \text{div} \), introduced in [11], represents divergence.
5 Consistency-Preserving Transformation

Having established consistency in early phases of system development it is desirable to preserve it during successive model evolutions. For a definition of consistency as deadlock-freedom this can be achieved if all transformations are refinements (in a formal sense). Refinement [2] is one of the main concepts supporting a formal system development. Refinement concepts exist both for state-based formalisms like Object-Z as well as for process algebras like CSP. In CSP process refinement is defined as inclusion in the failure divergence model: a process \( I \) is a refinement of a process \( S \), denoted \( S \subseteq_{FD} I \), if the failures (and divergences) of \( I \) are included in those of \( S \). Intuitively this means that \( I \) is more deterministic than \( S \), it may not execute traces that \( S \) did not have nor may it refuse more methods than \( S \) did.

Process refinement enjoys a nice property that we can further exploit for our model transformations:

**Proposition 1.** Let \( P_1, P_2, Q_1, Q_2 \) be CSP processes such that \( P_1 \parallel_A P_2 \) is deadlock-free and \( P_1 \subseteq_{FD} Q_1, P_2 \subseteq_{FD} Q_2 \). Then \( Q_1 \parallel_A Q_2 \) is deadlock-free.

This essentially means that we can replace any process in a parallel composition by a refinement without losing deadlock-freedom.

For the state machine part of a model we can thus classify the consistency preserving transformations as refinements: whenever we replace a state machine SM1 by a state machine SM2, consistency is preserved if the CSP process belonging to SM2 is a process refinement of that of SM1. An interesting point for further research would be to find classes of transformations on state machines which induce refinements, as for instance [3] shows the connection between some notions of statechart inheritance and refinement.

Concerning the class definition we could give a similar definition of consistency preserving transformations. Here, we can however refrain from moving to the CSP process and instead work directly on the Object-Z specification of a class by using the refinement concept of Object-Z. In Object-Z data refinement is concerned with defining valid changes made to data representations and operations of a class. A number of conditions for data refinement ensure that although attributes of the class may have changed (and consequently methods modified) the general behaviour of the class as observed from the outside remains indistinguishable from the former behaviour. As an example we again look at our Elevator class. In later steps of the development we might want to change the type of requests to a sequence (eventually implemented by an array) and modify method \( \text{start} \) such that it always takes the first element in the sequence as next target. This converts \( \text{start} \) into a deterministic method.

```object-z
ElevImpl

method request : [f? : Floor]
method passed, start
chan closeDoor, openDoor
chan stop, up, down
```
requests : seq Floor
pos, tar : Floor

Init
pos = tar = minFloor
requests = ⟨⟩

\[\text{effect\_request}\]

\[\Delta(requests)\]
\[f? : Floor\]
\[f? \neq pos \Rightarrow requests' = requests \cup \{f?\}\]
\[f? = pos \Rightarrow requests' = requests\]

\[\text{effect\_start}\]

\[\Delta(tar, requests)\]
\[tar' = \text{head} requests\]
\[requests' = \text{tail} requests\]

When observing the class \textit{ElevImpl} from the outside we cannot distinguish it from class \textit{Elevator}: it does not display a behaviour that \textit{Elevator} did not have. \textit{ElevImpl} is a data refinement of \textit{Elevator} (and this can be checked using a number of conditions on attributes and methods, see for instance [2]).

For our aim of preserving consistency we can now use the following result about data refinement (see for instance [13, 10]).

**Proposition 2.** Let \(C_1, C_2\) be Object-Z classes such that \(C_2\) is a data refinement of \(C_1\). Then the CSP process belonging to \(C_2\) is a process refinement of the CSP process belonging to \(C_1\).

Together with the above proposition this gives us a strategy for transformations on classes at hand: during evolution a class definition may be replaced by a data refinement of it. This preserves consistency with the state machine.

6 Conclusion

In the context of modelling with the UML the notion of consistency appears almost everywhere: between different models, inside a model between different entities and even for single specifications. It appears on the syntactical and the semantical level, on one level of abstraction as well as between different levels of abstraction. It is especially prominent if several (sub-) formalisms are used to model one thing from different points of view and there is semantic overlap between the different views.

In this paper we discussed consistency for split specifications consisting of an Object-Z class describing the data aspects of a class and an associated state machine describing the allowed sequences of method calls. By means of a translation to a common semantic domain a semantics was given for the whole specification, which enabled a basic definition of consistency, namely deadlock freedom. This
definition was shown to be lacking: an example, consistent according to the definition, still contained behaviour intuitively regarded as inconsistent. After failing to strengthen the definition, it was explained why a significantly extended notion of consistency is beyond reach unless the formalisms used were extended. Finally, at least a small enhancement was given, requiring that a class must have a minimal interface with regard to the possible behaviour.

In the second part of this paper, transformations on either part of the specification during model evolution, which are compatible with consistency, were examined. For the Object-Z part the concept of data refinement was shown to be compatible; for the state machine part only a transformation in the semantic model was considered.

Although the formalism used in this paper for discussing consistency possibly seems to be quite remote from 'real world' modelling, the results still apply to specifications using languages like Java or C++. Of course formal verification of properties like consistency is normally not possible in that case, which explains our preference for a notation with formal semantics.

References