Checking Consistency in UML Diagrams: Classes and State Machines*

Holger Rasch and Heike Wehrheim

Universität Oldenburg, Department 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 range of diagram types. The various diagrams can be used to specify different aspects, and their combination makes up the complete system description. This does, however, always pose the question of *consistency*: it may very 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. By means of a simple case study we discuss several definitions of consistency which are based on a common formal semantics for both classes and state machines. We furthermore show how consistency checks can be automatically carried out by a model checker. Finally, we examine which of the consistency definitions are preserved under refinement.

1 Introduction

The UML (Unified Modeling Language) [21] 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 (sequences of interactions). For instance, class diagrams fall into the first, state machines and sequence diagrams into the second category. While in general it is advantageous to have these different modelling facilities at hand, this also poses some non-trivial questions on designs. The different views on a system as described by different diagrams are not orthogonal, and may thus in principle be inconsistent when combined.

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 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

* This research is partially supported by the DFG under grant OL 98/3-1.

E. Najm, U. Nestmann, and P. Stevens (Eds.): FMOODS 2003, LNCS 2884, pp. 229–243, 2003.

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 one. Here, we have chosen Object-Z [19], 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 failures-divergences model of the process algebra CSP [14, 18]. This choice has a number of advantages: on the one hand a CSP semantics for Object-Z is already available [9, 19, 20], on the other hand CSP has a well developed theoretical background as well as tool-support in the form of the FDR model checker [12]. For (restricted classes of) state machines a CSP semantics has already been given in [5]; we give another one for a simple form of *protocol state machines* in this paper. The first step during a consistency check is always the translation of class description and 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 several consistency definitions. The property of consistency should guarantee that the two restrictions imposed on the behaviour of (an instance of) a class are not completely contradictory. There might, however, be different opinions as to what this means, ranging from "there is at least one possible run of the model" to "every method should always eventually be offered to the environment". The various forms of consistency are formally specified, and on the case study it is discussed what the effects of requiring such forms of consistency are. Our second focus is on *tool-supported consistency checks*: for each definition we show how the FDR model checker can be employed to automatically carry out the check.

During the development process a model may gradually be altered towards one close to the actual implementation domain. When consistency has been shown for a model developed in earlier phases, successive transformations of the model should preserve consistency if possible. In a formal approach to system development *refinement* is most often used as a correctness criterion for model evolutions. The question is thus whether the proposed consistency definitions are preserved under refinement. In our case, there are two notions of refinement to be considered: *data refinement* in the state-based part and *process refinement* (viz. failures-divergences refinement in CSP) in the behaviour-oriented part. Fortunately, we can restrict our considerations to process refinement since data refinement in Object-Z is known to induce failures-divergences refinement on the CSP processes obtained after the translation [20, 15]. For each of the definitions we hence either prove preservation under refinement or illustrate by means of a counterexample that preservation cannot be achieved in general.

The paper is structured as follows. In the next section we introduce the small case study and give a brief introduction to 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 the results on preservation under refinement.

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.

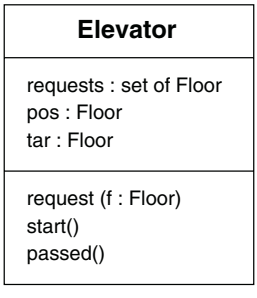

Fig. 1. Class Elevator

Its 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 an *effect* defining the effects of method execution on values of attributes.

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

$$\begin{array}{|l} minFloor, maxFloor : \mathbb{N}_1 \qquad\qquad Floor == minFloor..maxFloor \\ \hline minFloor < maxFloor \end{array}$$

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 ?.

[1] To be more specific, it is the Object-Z part of a CSP-OZ specification [9].

When 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 Δ-list of a method specifies the attributes which are changed by method execution.

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.

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 send 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. This completes the model for class Elevator.

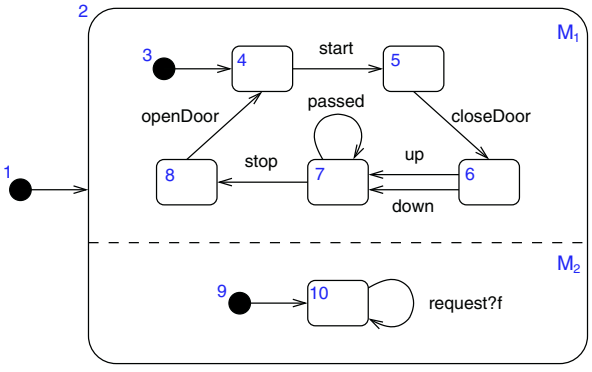

Fig. 2. Protocol of class Elevator

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 a semantic model of the process algebra CSP[2]. Instead of directly giving a (trace based) semantics to classes and state machines we translate them into CSP. This way the result remains readable and is furthermore amenable to checks with the FDR model checker for CSP.

3.1 Translation of the Object-Z Specification

The translation of class Elevator follows a general translation scheme for Object-Z developed in [11][3]. The basic idea is that a class is translated into a parameterised process. The parameters of the process correspond to the attributes 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[4]. Parameters and return values are encoded as data sent on the channels.

Each method is translated to a recursion of the main process, guarded by the event prefix $(e \rightarrow)$ corresponding to the method 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 ($\Box$). Not offering the disabled methods, i.e. the translation of the enable schemas, is achieved by using guards ($b\ \&$). Internal nondeterminism ($\sqcap$) 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. Finally, the *Init* schema is translated by specifying the initial process parameters. Again, this

[2] CSP has several semantic models. For the general semantics the *failures-divergences* model is used, but some of the checks studied here use less powerful models.
[3] This is a partial map; some abstractions cannot be handled. It can be automated.
[4] Method calls are modelled as CSP communication.

requires a nondeterministic choice over all possible valuations, if there is more than one.

The result of this part of the translation is shown below. The attributes *pos*, *tar* and *requests* of the Object-Z are represented by the process parameters p, t and R, respectively[5]. The notation $T \lhd b \rhd E$ is the operator notation of conditional choice meaning 'if b then T else E'.

$$
\begin{aligned}
PROC_Z &= Z(minFloor, minFloor, \varnothing) \\
Z(p,t,R) &= \quad p = t \;\&\; \text{openDoor} \rightarrow Z(p,t,R) \\
&\quad \Box\; p \neq t \;\&\; \text{closeDoor} \rightarrow Z(p,t,R) \\
&\quad \Box\; p = t \;\&\; \text{stop} \rightarrow Z(p,t,R) \\
&\quad \Box\; \text{request?f} \rightarrow Z(p,t,R \cup \{f\} \lhd f \neq p \rhd R) \\
&\quad \Box\; R \neq \varnothing \;\&\; \big(\sqcap_{t' \in R} \text{start} \rightarrow Z(p,t',R \setminus \{t'\}) \big) \\
&\quad \Box\; p \neq t \;\&\; \text{passed} \rightarrow Z(p+1 \lhd p < t \rhd p-1, t, R) \\
&\quad \Box\; p < t \;\&\; \text{up} \rightarrow Z(p,t,R) \\
&\quad \Box\; p > t \;\&\; \text{down} \rightarrow Z(p,t,R)
\end{aligned}
$$

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, a simple translation scheme exists for state machines which obey the following constraints:

- simple events, no guards, no actions,
- no interlevel transitions,
- only completion transitions from compound states,
- disjoint event sets in concurrent submachines,
- no pseudo states besides initial states[6].

Many, if not most, *protocol* state machines already obey these constraints[7], so we do not regard them as severe restrictions in this context.

Translation. Let SM be the state machine. For any pseudo, simple or compound state s of SM let $\mathcal{C}_s$ denote the set of all direct successors of s with respect to completion transitions and $\mathcal{T}_s$ the set of all pairs (e,t) of direct successors t of s reached via a transitions triggered by e. For any submachine M of SM (including the top level state machine M_{top}) let $\mathcal{I}_M$ denote the set of initial states for M. Now a translation function φ from states and (sub-)machines to CSP process definitions can be defined[8]:

[5] The renaming keeps the CSP expression small and otherwise has no meaning.

[6] This implies that the history mechanism is not used.

[7] They often use guards, but in CSP-OZ enable schemas are used instead.

[8] The special processes $SKIP$ and $STOP$ represent termination and deadlock; $|||$ denotes parallel composition without synchronisation (interleaving) and ; denotes sequential composition.

$$\varphi(s) \equiv \begin{cases} P_s = SKIP & \text{if } s \text{ is a final state,} \\ P_s = \Box_{(e,t)\in\mathcal{T}_s}\, e \to P_t & \text{if } s \text{ is a simple state and } \mathcal{C}_s = \varnothing, \\ P_s = \Box_{t\in\mathcal{C}_s} P_t & \text{if } s \text{ is a simple state and } \mathcal{C}_s \neq \varnothing \text{ or } s \text{ is an initial state,} \\ P_s = (|||_{i=1}^{n} P_{M_i});\ \big((\ \Box_{t\in\mathcal{C}_s} P_t) \\ \qquad \lessdot \mathcal{C}_s \neq \varnothing \gtrdot STOP\big) & \text{if } s \text{ is a compound state with submachines } M_i,\ 1 \le i \le n. \end{cases}$$

$$\varphi(M) \equiv \quad P_M = \Box_{t\in\mathcal{I}_M} P_t \qquad \text{for any (sub-) machine } M \text{ of } SM.$$

After calculating and combining the process expressions $\varphi(s)$ and $\varphi(M)$ for all states s and all (sub-) machines M of SM, the CSP semantics of SM can now be computed by evaluating $P_{M_{\text{top}}}$ using one of the CSP models.

Since the protocol state machine in Fig. 2) satisfies the constraints given above, it can be translated using the scheme above. This yields the following process definitions (to the right an equivalent simplified version of each process body is shown):

$$\begin{aligned}
P_{M_{\text{top}}} &= \Box_{i\in\{1\}} P_i && = P_1 \\
P_{M_1} &= \Box_{i\in\{3\}} P_i && = P_3 \\
P_{M_2} &= \Box_{i\in\{9\}} P_i && = P_9 \\
P_1 &= \Box_{i\in\{2\}} P_i && = P_2 \\
P_2 &= (|||_{i\in\{1,2\}} P_{M_i});\ \big((\ \Box_{t\in\varnothing} P_t) && = (P_{M_1} \mid\mid\mid P_{M_2});\ STOP \\
&\quad \lessdot \varnothing \neq \varnothing \gtrdot STOP\big) && \\
P_3 &= \Box_{i\in\{4\}} P_i && = P_4 \\
P_4 &= \Box_{(e,t)\in\{(\text{start},5)\}} e \to P_t && = \text{start} \to P_5 \\
P_5 &= \Box_{(e,t)\in\{(\text{closeDoor},6)\}} e \to P_t && = \text{closeDoor} \to P_6 \\
P_6 &= \Box_{(e,t)\in\{(\text{up},7),(\text{down},7)\}} e \to P_t && = \text{up} \to P_7 \,\Box\, \text{down} \to P_7 \\
P_7 &= \Box_{(e,t)\in\{(\text{passed},7),(\text{stop},8)\}} e \to P_t && = \text{passed} \to P_7 \,\Box\, \text{stop} \to P_8 \\
P_8 &= \Box_{(e,t)\in\{(\text{openDoor},4)\}} e \to P_t && = \text{openDoor} \to P_4 \\
P_9 &= \Box_{i\in\{10\}} P_i && = P_{10} \\
P_{10} &= \Box_{(e,t)\in\{(\text{request?f},10)\}} e \to P_t && = \text{request?f} \to P_{10}
\end{aligned}$$

With $PROC_{SM} = P_{M_{\text{top}}}$ we now have a CSP translation of the state machine part of the specification. Simplifying it again, the readable (but equivalent) version of $PROC_{SM}$ looks like this:

$$
\begin{aligned}
PROC_{SM} &= P_4 \;|||\; P_{10} \\
P_4 &= \text{start} \rightarrow \text{closeDoor} \rightarrow (\text{up} \rightarrow P_7 \;\Box\; \text{down} \rightarrow P_7) \\
P_7 &= \text{passed} \rightarrow P_7 \;\Box\; \text{stop} \rightarrow \text{openDoor} \rightarrow P_4 \\
P_{10} &= \text{request?f} \rightarrow P_{10}
\end{aligned}
$$

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

$$PROC \quad = \quad PROC_Z \underset{\Sigma}{\parallel} PROC_{SM}$$

synchronising on the set Σ 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 with respect to UML protocol state machines, i.e., the protocol state machine restricts possible behaviour.

4 Notions of Consistency

Using the example given above we now discuss different 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 $PROC$.

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 $PROC$. In the sequel we present several possible definitions of consistency and for each of them develop a technique for proving it using the FDR model checker (in case of finite state specification)[9].

4.1 Basic Consistency

When talking about consistency it is beneficial to view parallel composition of CSP processes as conjunction. So the consistency of $PROC$ is the consistency of '$PROC_Z \wedge PROC_{SM}$'. It is immediately clear that if this 'formula' is not satisfiable, the corresponding specification is *inconsistent*. Translated to the terms of the semantic model this means: $PROC$ will always deadlock, i.e. any sequence of events offered to $PROC$ will lead to deadlock. Seen the other way round: for the specification to be satisfiable it suffices to have at least one trace of $PROC$ not leading to deadlock.

[9] We only consider reactive systems without termination; to use these for systems which include finite behaviour, termination has to be mapped to an infinite iteration of some extra event.

Definition 1. *A specification consisting of an Object-Z class and an associated state machine has the property of* satisfiability *iff the corresponding process in the semantic model has at least one non-terminating run.*

This property can be automatically checked as follows. We assume Σ to be the alphabet of the class (i.e. all events occurring in $PROC$), the event acc not to be in Σ, and $PROC$ to be given in machine readable CSP so that the FDR model checker can be used. In CSP, properties are checked by comparing the system process with another CSP process specifying the property. The comparison is a *refinement test* in one of the models of CSP: traces, failures or failures and divergences. For our first property we use the following property process INF, parameterised in the event (or method) m under consideration:

$$INF(m) = m \rightarrow INF$$

For checking satisfiability we test whether

$$PROC[acc/m] \sqsubseteq_{\mathcal{T}} INF(acc)$$

holds. The process $INF(acc)$, only executing acc events, is a trace refinement of $PROC$ in which all events are renamed to acc if $PROC$ contains at least one nonterminating run. Performing this check for our elevator example tells us that the specification is satisfiable.

Satisfiability does, however, not exclude the case that the specification deadlocks. 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 a sufficient form of consistency.

For a specification to have basic consistency we thus require $PROC$ to be deadlock free; after any sequence of events performed by $PROC$ there has to be at least one event to continue the sequence, that is, no trace may have Σ (the set of all methods) as the refusal set. This can be regarded as the standard notion of consistency in the context of behavioural specifications and is used by other authors as well, for instance [7, 6], where it is applied to the behaviour of *different* entities of a model acting together.

Definition 2. *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 whether the example is consistent according to Def. 2 (by using FDR's predefined test for deadlock-freedom). The result is, that $PROC$ for the example given is indeed deadlock free, so the specification has the property of basic consistency.

4.2 Execution of Methods

Now that we have defined a first form of consistency we again look at our example to see whether this is sufficient. Consider the following sequence of events on $PROC_Z$ starting from the initial state.

$$
\begin{array}{rcll}
 & & Z(0,0,\varnothing) & \\
request.1 & \rightarrow & Z(0,0,\{1\}) & \\
start & \rightarrow & Z(0,1,\varnothing) & [\, requests \neq \varnothing \,] \\
closeDoor & \rightarrow & Z(0,1,\varnothing) & [\, pos \neq tar \,] \\
up & \rightarrow & Z(0,1,\varnothing) & [\, pos < tar \,] \\
request.1 & \rightarrow & Z(0,1,\{1\}) & \\
passed & \rightarrow & Z(1,1,\{1\}) & [\, pos \neq tar \,] \\
stop & \rightarrow & Z(1,1,\{1\}) & [\, pos = tar \,] \\
openDoor & \rightarrow & Z(1,1,\{1\}) & [\, pos = tar \,] \\
start & \rightarrow & Z(1,1,\varnothing) & [\, requests \neq \varnothing \,]
\end{array}
$$

At this point the elevator should be able to perform *closeDoor*, but this method is only enabled if $pos \neq tar$, so the only method enabled at this point is *request*. Since *request* does not modify *pos* or *tar* this condition will endure infinitely. This means, the only possible trace after this prefix is $\langle request \rangle^*$. The specification has the property of basic consistency but still the combination of state-based part and protocol state machine prevents certain executions which we expect from the elevator. What we additionally need are certain forms of *liveness* of methods.

As a first approach we might require that every method specified in the interface of the class can be executed at least once.

Definition 3. *A specification consisting of an Object-Z class and an associated state machine has the property of* method executability *iff in the corresponding process in the semantic model every method is executed at least once.*

Executed at least once means that there is some trace in which an event corresponding to the method occurs. In FDR this can be checked as follows. We define a tester process

$$ONCE(m) = m \rightarrow STOP$$

and check the trace inclusion

$$PROC \setminus (\Sigma \setminus \{m\}) \sqsubseteq_{\mathcal{T}} ONCE(m)$$

This test has to be carried out for every method in the interface of the class. Since in the above example trace of the elevator method *closeDoors* has already been executed, this definition does however not have the desired effect: Our specification has the property of method executability for all methods in the interface of *Elevator*.

Although method executability is a weak requirement on specifications, it is a fundamental one: it amounts to the detection of 'dead code' in the specification, i.e., if a method m fails this test, anything can be substituted for it without changing the semantics of the specification. This is almost always an error.

A stronger definition might require that all methods (or at least certain methods marked to be live) should be executed infinitely often:

Definition 4. *A specification consisting of an Object-Z class and an associated state machine has the property of* method liveness *iff in the corresponding process in the semantic model every method will always eventually be executed again.*

This requirement is close to the definition of *impartiality* of [17] (or unconditional fairness of [13]) which states that every process in a concurrent system takes infinitely many moves. Clearly, method liveness is not fulfilled by our example. It can be tested using again the property process *INF* defined above:

$$INF(m) \sqsubseteq_{\mathcal{FD}} PROC \setminus (\Sigma \setminus \{m\})$$

There is still some drawback in this strong liveness requirement. For object-oriented systems it is not adequate to require that methods are always executed since an execution requires a request from the environment, and there might well be traces on which a method is never executed simply because it is never requested. What we really would like to have is liveness with respect to an *offering* of methods to a client of the class.

Definition 5. *A specification consisting of an Object-Z class and an associated state machine has the property of* method availability *iff in the corresponding process in the semantic model every method will always eventually be enabled again.*

Unfortunately, this kind of unconditional liveness is not expressible within the failures-divergences model. We have to approximate it by a form of *bounded* availability, fixing an upper bound N on the number of steps in between two offerings of a method.

$$\begin{aligned} OFFER(i,m) = \big(& \sqcap_{Ev \subseteq \Sigma} \ \Box_{ev \in Ev \cup \{m\}} \ ev \rightarrow OFFER(N,m) \big) \\ & \Box \ i > O \ \& \ \sqcap_{Ev \subseteq \Sigma \setminus \{m\}} \ \Box_{ev \in Ev} \ ev \rightarrow OFFER(i-1,m) \end{aligned}$$

After at most N steps process $OFFER(N,m)$ reaches a state in which method m is not refused. Regarding the other events the process can freely choose to refuse as well as execute them. The check for bounded method availability of m is then

$$OFFER(N,m) \sqsubseteq_{\mathcal{FD}} PROC$$

A last remark on this test concerns efficiency. Since process *OFFER* contains a choice over all possible subsets of Σ the state space of *OFFER* will be exponential in the size of the class' alphabet. For larger specifications this might make it unrealistic to actually carry out the check. Fortunately, for this test all events besides m in $OFFER(N,m)$ and $PROC$ can be regarded as equivalent, so renaming can be used to reduce the size of the alphabet to two, without changing the outcome of the test. Giving the reduced form as actual input to FDR results in clearly reduced runtime comparable to the other checks.

Summarising, we have proposed and discussed five definitions of consistency which can be used for classes and associated state machines. Which ones are

actually used depend on the designer of the specification. In our opinion, basic consistency and method executability should always should always be fulfilled. The extended liveness conditions are in general not to be used for all methods, since some kinds of methods, e.g., those performing initialisation, are not intended to have these properties in the first place. Since this cannot be inferred from the specification as it is now, it might be useful to include additional elements in the diagrams to indicate the intentions of the modeller with respect to the intended behaviour.

5 Consistency-Preserving Transformation

Having established consistency in early phases of system development it is desirable to preserve it during successive model evolutions. In a formal approach to system development model evolutions from high-level specifications to lower-level ones are supported by the concept of refinement [4]. Refinement defines correctness criteria for allowed changes between different levels of abstraction. With regard to consistency refinement should preserve consistency, or rather the other way round: consistency should be defined such that it is preserved under refinement. In the sequel we will examine which of our five definitions of consistency are preserved under refinement.

Since we have a state-based and a behaviour-oriented part there are in principle two kinds of refinement to be considered: *process refinement* in CSP, defined as inclusion in the failures-divergences model, and *data refinement*, proven via simulation rules between classes. Fortunately, data refinement in Object-Z induces failures-divergences refinement on its CSP semantics [20, 15, 10]. Hence it is sufficient to study preservation of consistency under process refinement. In the sequel we let refinement stand for failures-divergences refinement.

We pass through our definitions in the order in which they are defined in the last section. For the first one we get:

Proposition 1. *Satisfiability is not preserved under refinement.*

This fact can be illustrated by the following CSP process P over the alphabet $\Sigma = \{a, b, c\}$ representing some process $PROC$:

$$\begin{aligned} P &= a \rightarrow STOP \\ &\sqcap\ b \rightarrow Run \\ Run &= \Box_{ev \in \Sigma \setminus \{a\}}\ ev \rightarrow Run \end{aligned}$$

P has the property of satisfiability: a nonterminating run is $b, c, c, c, \ldots$. Consider now the process P' defined as $a \rightarrow STOP$. P' is a failures-divergences refinement of P (due to the internal choice at the start of P), but is has no nonterminating run.

Concerning basic consistency we get a better result. The following proposition follows from standard CSP theory.

Proposition 2. *Basic consistency is preserved under refinement.*

Concerning the execution of methods there is one negative and two positive results.

Proposition 3. *Method executability is not preserved under refinement.*

Starting with the same process P which we used for the counterexample about satisfiability we now refine it to a process $P'' = b \rightarrow Run$. Process P has the property of method executability for all methods in Σ, including a. P'' is a refinement of P but allows for no execution of a.

The stronger requirements of method liveness and availability are however preserved:

Proposition 4. *Method liveness and method availability are preserved under refinement.*

Proof. Referring to the processes used for checking these two requirements the results easily follow from transitivity of refinement and monotony of all CSP operators with respect to refinement.

Together with the discussion in the last section this gives strong hints as to what a reasonable definition of consistency might be. Basic consistency should always be fulfilled in order to achieve a meaningful model. This should be complemented with liveness and/or availability of methods, where, however, it might make sense to restrict these requirements to some of the methods.

For the state machine part of a model we can then classify one kind of consistency preserving transformations via 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 [5] shows the connection between some notions of statechart inheritance and refinement. For the state-based part of the model (the class) data refinement is a consistency preserving transformation.

6 Conclusion

In this paper we discussed consistency for 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 number of consistency definitions. For every such definition we proposed a technique for automatically checking it with a model checker, and we furthermore showed which of the definitions are preserved under refinement.

Related Work. For the UML, consistency is a heavily discussed topic, see for instance the workshop on "Consistency Problems in UML-based Software Development" [16]. The approaches in this workshop discuss a wide variety of consistency issues arising in UML, ranging from mainly syntactic ones to others

involving a semantic analysis of the model. The work closest to ours is that of [7, 6] who also use CSP as their semantic basis (and FDR for model checking). However, whereas they are comparing different behavioural views of a UML model (state machines and protocols in UML-RT) we define consistency between a static and a dynamic diagram.

The basis for our work is CSP-OZ, an integration of Object-Z and CSP, which – together with the translation of state machines to CSP – allows for a common analysis of state-based and behaviour-oriented views. Besides CSP-OZ there are a number of other integrated specification languages around, for an overview and comparison of integrations of Z and process algebras see [9].

Consistency of different views is (or has been) an issue in other areas as well, especially in the ODP ISO reference model which allows for a specification of distributed systems by different viewpoints. The approaches taken there are, however, different from ours. [8, 1] achieve consistency by transformations between viewpoints, [2] define consistency between viewpoints by the existence of a common implementation of both viewpoints (using a variety of possible implementation/refinement relations). The latter approach has also been taken by Davies and Crichton for defining consistency between sequence diagrams and system models in UML [3] (also using CSP as a semantic domain). While for the comparison of sequence diagrams specifying certain scenarios of a system a refinement based consistency definition seems reasonable (in order to find out whether the system sometimes/always exhibits the scenario), for class and state diagrams it might turn out to be inadequate: Since both the traces model and the stable failures model of CSP have top elements with regard to the respective refinement order, a common refinement always exists, whereas a specification, which is consistent according to some of our definitions, typically does not have a common refinement in the failures-divergences model. Moreover, in our opinion our consistency definitions more naturally capture a practitioners point of view on consistency.

References

1. C. Bernardeschi, J. Dustzadeh, A. Fantechi, E. Najm, A. Nimour, and F. Olsen. Transformations and Consistent Semantics for ODP Viewpoints. In *Proceedings of Second IFIP conference on Formal Methods for Open Object-based Distributed Systems - FMOODS'97*. Chapman & Hall, 1997.
2. H. Bowman, M.W.A. Steen, E.A. Boiten, and J. Derrick. A formal framework for viewpoint consistency. *Formal Methods in System Design*, 21:111–166, 2002.
3. J. Davies and Ch. Crichton. Concurrency and Refinement in the Unified Modeling Language. *Electronic Notes in Theoretical Computer Science*, 70(3), 2002.
4. J. Derrick and E. Boiten. *Refinement in Z and Object-Z, Foundations and Advanced Application*. Springer, 2001.
5. G. Engels, R. Heckel, and J. Küster. Rule-based Specification of Behavioral Consistency based on the UML Meta-Model. In Martin Gogolla, editor, *UML 2001*. Springer, 2001.

6. G. Engels, R. Heckel, J. M. Küster, and L. Groenewegen. Consistency-preserving model evolution through transformations. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002 – Model Engineering, Concepts, and Tools*, volume 2460 of *LNCS*, pages 212–226. Springer-Verlag, 2002.
7. G. Engels, J. Küster, R. Heckel, and L. Groenewegen. A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In *9th ACM Sigsoft Symposium on Foundations of Software Engineering*, volume 26 of *ACM Software Engineering Notes*, 2001.
8. K. Farooqui and L. Logrippo. Viewpoint Transformation. In *Proc. of the International Conference on Open Distributed Processing*, pages 352–562, 1993.
9. 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.
10. C. Fischer and S. Hallerstede. Data-Refinement in CSP-OZ. Technical Report TRCF-97-3, University of Oldenburg, June 1997.
11. C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods (IFM)*, pages 315–334. Springer, 1999.
12. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, Oct 1997.
13. N. Francez. *Fairness*. Texts and Monographs in Computer Science. Springer, 1986.
14. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
15. M.B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.
16. L. Kuzniarz, G. Reggio, J. L. Sourrouille, and Z. Huzar, editors. *UML 2002 – Workshop on Consistency Problems in UML-based Software Development*, volume 06 of *Blekinge IOT Research Report*, 2002.
17. D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, Justice and Fairness: The Ethics of Concurrent Termination. In G. Goos and J. Hartmanis, editors, *Automata, Languages and Programming*, number 115 in LNCS, pages 264 –277. Springer, 1981.
18. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
19. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.
20. 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.
21. OMG Unified Modeling Language specification, version 1.5, March 2003. http://www.omg.org.