Checking the validity of scenarios in UML models*

Holger Rasch and Heike Wehrheim

Department of Computer Science
University of Paderborn
33098 Paderborn, Germany
{hrasch,wehrheim}@uni-paderborn.de

Abstract. In the UML, sequence diagrams are used to state scenarios, i.e., examples of interactions between objects. As such, sequence diagrams are being developed in the early design phases where requirements on the system are being captured. Their intuitively appealing character and conceptual simplicity makes them an ideal tool for formulating simple properties on a system, even for non-experts. Besides guiding the development of a UML model, sequence diagrams can thus furthermore be used as a starting point for the *verification* of the UML model.
In this paper, we show how the requirements on the system as stated in sequence diagrams can be (semi-automatically) validated for UML models consisting of class diagrams, state machines and structure diagrams. The sequence diagrams that we consider can be universally or existentially quantified or negated, i.e., state scenarios that should always, sometimes or never occur. For validating them in a UML model, we translate both model and sequence diagrams into a formal specification language (the process algebra CSP), and develop procedures for employing the standard CSP model checker (FDR) for checking their validity.

1 Introduction

The complexity of software is steadily increasing. Models of software systems have to reflect this complexity in that they precisely describe all different aspects making up the functionality of a complex system. The UML is a modelling language which supports modelling with different views. Its various diagram types allow for the description of different though not necessarily disjoint aspects of a system: Class diagrams model the static behaviour (data and operations), state machines the dynamic behaviour (protocols), structure diagrams the architectural composition and sequence diagrams typical application scenarios (plus possibly further diagrams for other aspects). Together they model the system to be built. Such a complex model composition immediately poses the question of consistency: is the architectural composition consistent with the interface description of the components, are static and dynamic behaviour non-contradictory, is the scenario as stated in the sequence diagrams actually allowed in the model, etc. In this paper, we develop techniques which can be used for answering the latter question.

* This research was partially supported by the DFG project ForMooS (grants OL98/3-2 and WE2290/5-1)

The starting point for our study are UML 1.5 sequence diagrams [24] which we extend with features for stating *negation* and *universal* and *existential* quantification (partially coming from UML 2.0 sequence diagrams[1]). These facilities allow to dinstinguish between different types of scenarios: those never occuring, sometimes occuring (i.e., in at least one run) or always occuring (in all runs). The remaining part of the UML models will consist of class diagrams, state machines and structure diagrams. To achieve the necessary precision in the model (which is mandatory for a verification) we additionally use the Z notation [21, 13] for writing attributes and methods in class diagrams. The question is then whether the sequence diagrams are consistent with the UML model in that the restrictions on the overall behaviour (as laid down in the diagrams) do not prevent desired or allow forbidden scenarios. We develop a technique which allows to automatically check for this kind of consistency. To this end, we translate both sequence diagrams and the rest of the UML model into a formal specification language. The translation of class diagrams, state machines and structure diagrams follows a technique proposed in [16], the translation of the sequence diagrams is inspired by [10] and given in this paper. Since the properties stated in the sequence diagrams all refer to orderings in the communication between objects we have chosen the process algebra CSP for this purpose. CSP [12, 18] has been developed to model and analyse systems exhibiting a large degree of parallelism and communication. Moreover, there is a model checker for CSP (FDR [9]) which can be used for automatically analysing CSP processes. The translation provides us with a semantics of UML model and sequence diagrams in terms of the semantic model of CSP. On this semantic model we formally define validity of a sequence diagram (in the UML model) with respect to existential and universal quantification; negation is obtained by negating the definition of existential quantification. For these validity definitions we develop procedures for automatic checks using the FDR model checker: the validity checks have to be formulated as *refinement checks* between CSP processes which is the type of analysis supported by FDR. To this end we develop testers out of the CSP semantics of sequence diagrams which are then checked against the CSP semantics of the UML model.

The paper is structured as follows: The next section will present an example of a UML model together with a number of allowed or forbidden scenarios stated by sequence diagrams. Section 3 describes the translation of model and sequence diagram to CSP. Section 4 formally defines validity and develops procedures for checking validity via the FDR model checker. The last section concludes.

2 Example

In this section we start with introducing the example which will be used for illustrating our technique for checking the validity of scenarios. The example concerns the modelling of cash machines and banks. For the modelling we use a UML profile for reactive systems proposed in [16] and inspired by the ROOM method [19]. This

[1] We do not treat other new features of UML 2.0 sequence diagrams here (like combined fragments) since our main interest is in checking validity not in developing a semantics for UML 2.0.

profile allows to describe reactive systems as being built out of processes (active objects) working concurrently and communicating with each other. Each process has an associated interface which describes its communication capabilities. An interface description contains both the methods callable on the active object/process as well as those called by the object.

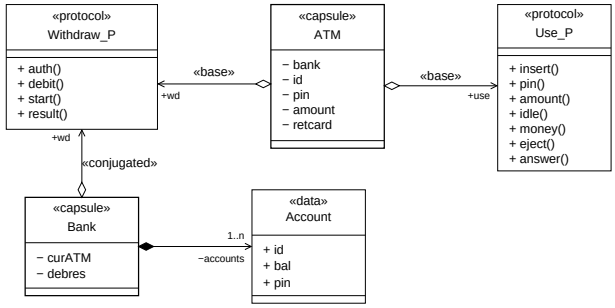

Fig. 1. Class Diagram for capsules Bank and ATM

In the UML profile these active object or processes are modelled by special *capsule* classes and their interfaces by *protocol* classes. Capsules which share a common protocol can be connected to each other. The dynamic behaviour of the processes, viz. capsules, are given by UML state machines. Figure 1 gives the class diagram for capsules ATM (the cash machine) and Bank. Capsule Bank has one protocol which describes its interface, namely the methods *auth*, *debit* and *result* that it offers to other processes. Class Account is a passive component which is associated to Bank (every bank has a number of accounts). Capsule ATM has two protocols, one for communication with the Bank and the second one for communication with users. The stereotypes «base» and «conjugated» describe the direction of communication for the protocols.

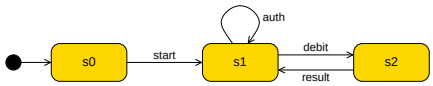

Fig. 2. State Machine for Capsule Bank

Figures 2 and 3 depict the protocol state machines for ATM and Bank. They model the allowed ordering of method invocation for objects of class Bank and ATM, respectively. The reactive system itself is modelled by a structure diagram. Structure diagrams describe the architecture of systems, i.e., their components and their interconnection. A capsule in a structure diagram is drawn as a rectangle with ports (white or black boxes) indicating their protocols. Two ports (and thus two capsules) can be connected

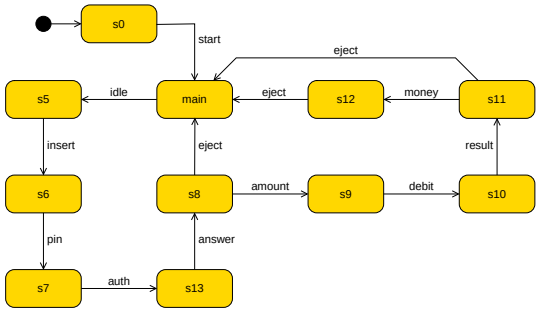

Fig. 3. State Machine for Capsule ATM

if they refer to the same protocol. A port residing on the border of the outermost capsule (capsule System) describes the interface of the system towards its environment.

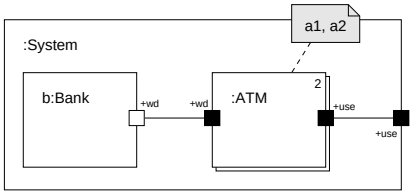

Fig. 4. Structure Diagram

Here, our banking system consists of one bank b connected with two cash machines a_1 and a_2 communicating over the joint protocol Withdraw_P. The interface to users of the banking system is given by the protocol Use_P.

In order to get the necessary precision in the model the UML profile furthermore allows to formally specify the signatures of methods in interfaces, and the attributes and methods in classes (both for capsules and passive classes). For this purpose the specification formalism Z [21, 13] is employed. If such a specification is supplied for all methods and attributes a precise and unambigious meaning can be given to the UML model. This is the prerequisite for formally checking the validity of scenarios. For our example, we only show the *signatures* of methods in interface since we will refer to them when translating sequence diagrams. For the Bank the interface specification in Z is (assuming given types ID and PIN):

$$\texttt{method}\ auth : [from : ATM;\ to : \{\texttt{self}\};\ id? : ID;\ pin? : PIN;\ ok! : \mathbb{B}]$$
$$\texttt{method}\ debit : [from : ATM;\ to : \{\texttt{self}\};\ id? : ID;\ amt? : \mathbb{N}]$$
$$\texttt{chan}\ start : [from : \{\texttt{self}\};\ to : ATM]$$
$$\texttt{chan}\ result : [from : \{\texttt{self}\};\ to : ATM;\ ok! : \mathbb{B}]$$

There are two types of operations in the interface: those being declared as `method` are methods of the Bank itself and can be called by other objects; those declared by `chan` are methods that Bank calls on other objects. The parameters of these operations can be divided into *input* (marked with ?), *output* (!) or *simple* parameters. The latter one are used for addressing particular objects. Every method must have two simple parameters specifying the caller (parameter *from*) and callee (parameter *to*) of the method. The value `self` refers to the object itself. If a parameter has type {`self`} then the only possible value that the parameter can take is `self`.

The interface of ATM towards Bank is specified in a complementary way (channels and methods, and input and output reversed):

$$\texttt{method}\ \mathit{start} : [\mathit{from} : \mathit{Bank};\ \mathit{to} : \{\texttt{self}\}]$$
$$\texttt{method}\ \mathit{result} : [\mathit{from} : \mathit{Bank};\ \mathit{to} : \{\texttt{self}\};\ \mathit{ok}? : \mathbb{B}]$$
$$\texttt{chan}\ \mathit{auth} : [\mathit{from} : \{\texttt{self}\};\ \mathit{to} : \mathit{Bank};\ \mathit{id}! : \mathit{ID};\ \mathit{pin}! : \mathit{PIN};\ \mathit{ok}? : \mathbb{B}]$$
$$\texttt{chan}\ \mathit{debit} : [\mathit{from} : \{\texttt{self}\};\ \mathit{to} : \mathit{Bank};\ \mathit{id}! : \mathit{ID};\ \mathit{amt}! : \mathbb{N}]$$

Additionally the interface contains operations for interaction with a user. We always assume to have one class (here called *User*) which models the environment of our system. The interface of this class can be determined from the structure diagram: all protocols of ports residing on the borders of the outermost capsule System are also protocols of the environment class (complementing methods and channels, inputs and outputs). The behaviour of the environment remains unspecified, thus we assume it to behave chaotically (all behaviour allowed). Thus for our interface of ATM towards a user we assume a class User to be given.

$$\texttt{method}\ \mathit{insert} : [\mathit{from} : \mathit{User};\ \mathit{to} : \{\texttt{self}\};\ \mathit{id}? : \mathit{ID}]$$
$$\texttt{method}\ \mathit{pin} : [\mathit{from} : \mathit{User};\ \mathit{to} : \{\texttt{self}\};\ \mathit{pin}? : \mathit{PIN}]$$
$$\texttt{method}\ \mathit{amount} : [\mathit{from} : \mathit{User};\ \mathit{to} : \{\texttt{self}\};\ \mathit{amt}? : \mathbb{N}]$$
$$\texttt{chan}\ \mathit{answer} : [\mathit{from} : \{\texttt{self}\};\ \mathit{to} : \mathit{User};\ \mathit{ok}! : \mathbb{B}]$$
$$\texttt{chan}\ \mathit{idle} : [\mathit{from} : \{\texttt{self}\};\ \mathit{to} : \mathit{User}]$$
$$\texttt{chan}\ \mathit{money} : [\mathit{from} : \{\texttt{self}\};\ \mathit{to} : \mathit{User};\ \mathit{amt}! : \mathbb{N}]$$
$$\texttt{chan}\ \mathit{eject} : [\mathit{from} : \{\texttt{self}\};\ \mathit{to} : \mathit{User}]$$

This completes the UML model. The development of the model might have been preceeded by the modelling of typical (allowed or forbidden) scenarios of the system to be modelled. Such scenarios can be described by sequence diagrams. Here, we use a very simple form of sequence diagrams since our main focus is not on giving a semantics but on checking their validity (for semantics for message sequence charts, the precursors of sequence diagrams, see for instance [10, 14, 15]; for a semantics for UML 2.0 interactions diagrams see [22]). For the banking system we might for instance like to specify that a user never gets money when the ATMs question for enough credit is answered with "no". Thus the scenario in Figure 5 is forbidden for our system.

In general, sequence diagrams consist of a number of lifelines for objects. These lines are connected by arrows labelled with methods. The sequence diagram thus models orderings for interactions between objects. We use particular objects here (although the scenario should be forbidden for all banks, ATMs and users) since we need to refer to specific objects in our addressing parameters. In principle object

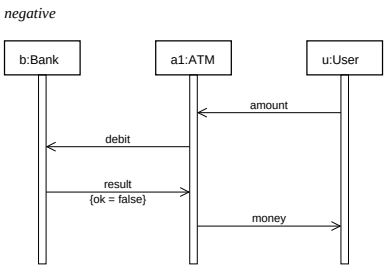

Fig. 5. Forbidden Behaviour Bank-ATM-User

names could be left out first and later be instantiated when the validity of scenarios is checked.

The next sequence diagram shows a possible scenario between bank, ATM and user: when the answer of the bank following the request for an authentification of a pin is not ok then the card should be ejected.

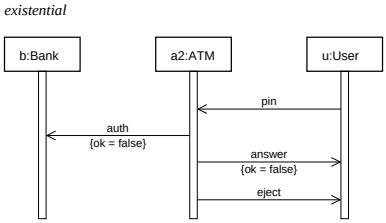

Fig. 6. Possible Behaviour Bank-ATM-User

The previous scenario showed some behaviour which is possible but must not necessarily occur in all runs. The following is a scenario which should always occur: after ejecting and idling for a while an insert of another card should be possible.

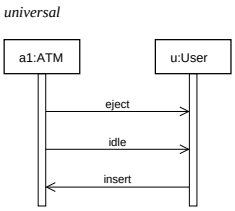

Fig. 7. Required Behaviour ATM-User

3 Translating the UML model to CSP

Given a UML model of a system and a number of sequence diagrams specifying allowed or forbidden scenarios we are interested in knowing whether these scenarios are actually possible (or not possible) in the model. We refer to this as the *validity* of the scenario in the model. Checking the validity of scenarios should at the best be fully automatic; here we propose a technique which is partially automatic. For checking the validity we first of all have to compute the semantics of model and sequence diagrams. This so far has to be done by hand, but can be automated. Given this semantics the check can be carried out with the CSP model checker FDR.

3.1 Translating class diagrams, state machines and structure diagrams

As a semantic domain for model and sequence diagrams we have chosen the process algebra CSP. A translation of models in our specific UML profile to CSP can be found in [16]. Roughly, the translation proceeds as follows: The class diagram together with the Z formalisation of interfaces, attributes and methods is translated to Object-Z [20]. The Object-Z classes of capsules are then augmented with CSP process descriptions which are derived from the state machines. A specific CSP process is computed for the outermost capsule *System*. This process describes the architecture of the system and consists of the parallel composition of the capsules in the system. Together, these classes form a CSP-OZ [8] specification. CSP-OZ is a combination of Object-Z and CSP and has a semantics in terms of the semantic model of CSP. Thus we hereby end with a CSP semantics for our UML model. To show at least a small part of the resulting CSP process:

$$\begin{aligned} \texttt{main} &= Bank[b] \;||_{\{|\ auth,debit,result,start\ |\}}\; Clients \\ Clients &= ATM[a1] \;|||\; ATM[a2] \end{aligned}$$

is the CSP process of class *System* (derived from the structure diagram). It describes the behaviour of the overall system. The operator $|||$ is the interleaving operator of CSP (parallel composition with no synchronisation) whereas $||_A$ (A being a set of methods) is a parallel composition which requires joint execution of methods in A. This synchronisation set is derived from the joint protocol (Withdraw_P) of Bank and ATM.

3.2 Translating sequence diagrams

In the next step we equip the sequence diagrams with a CSP semantics as well. To this end, we first formalise sequence diagrams, i.e., give description of their syntax. This will be the basis for the translation to CSP. As a language for formalising the diagrams we again employ Z.

We start with the definition of some basic data types (in particular for the names appearing in the diagrams).

$[Name,$ [Channel names]
$Param,$ [Parameters]
$Object,$ [Object names]
$D,$ [Data values for parameters]
$UID]$ [Unique identifier for arrows]

The sets which can actually be used here depend on the UML model and can be derived from it. In the banking example, we for instance would have $auth \in Name$, $amt \in Param$, $a1 \in Objects$ etc. The set D is the basic type of values for parameters. As we have already explained, all channels must have two special parameters which are used for adressing the partners in a communication. These are declared in the following axiomatic definition:

$$to, from : Param$$

Communication channels between components are described by *protocols* in the UML model. A protocol describe the interface of a class, i.e. the methods it provides (with their signature) and the methods it requires. Components that are to be connected with each other in the structure diagram share one (or more) protocols. From this interface information in the protocols we just need the names of channels and their parameters here.

Channel

$$name : Name$$
$$params : \mathbb{P}\, Param$$

$$\{to, from\} \subseteq params$$

CSP processes are built over *events*. An event consists of a channel name together with values for parameters, e.g. $answer.b.a1.true$ is an event consisting of the name *answer* plus values b (for parameter *from*), $a1$ (for *to*) and *true* (for *ok*). The notation ?_ denotes that any value can be accepted for a parameter. A *partial event* is one in which some of the values for parameters are missing, e.g. $answer.b.a1$ is only a partial event. Given a set of partial events Ev we use the CSP notation $\{|\ Ev\ |\}$ to denote the set of completions of Ev. Formalised in Z events are as follows.

Event

$$ch : Channel$$
$$val : Param \pfun D \cup \{?\_\}$$

$$\mathrm{dom}\, val = ch.params$$

In sequence diagrams events will appear as labels of arrows between objects. Whenever values for parameters are left out (which is most often the case) we assume the value to be ?_.

Arrows in sequence diagrams are connecting the lifelines of two objects and are labelled with events. To distinguish two arrows with the same label connecting the same lines we attach a unique identifier to each arrow. The values for parameters *to* and *from* of an event attached to an arrow have to agree with the objects connected by the arrow.

Arrow

$$
\begin{array}{l}
from, to : Object \\
event : Event \\
uid : UID \\
\hline
to \neq from \qquad \text{[no loops]} \\
event.to = to \land event.from = from
\end{array}
$$

These definitions form the basis for formalising sequence diagrams. A sequence diagram simply consists of a set of arrows and lifelines. Each line belongs to an object and has a number of arrows going out of it or coming in. On one line arrows are always ordered (hence we describe them as a sequence). A number of additional conditions ensure well-formedness of sequence diagrams. Additionally, every sequence diagram is equipped with an occurrence condition stating whether the specified scenario should never/sometimes or always happen.

$$Condition ::= negative \mid existential \mid universal$$

SQ

$$
\begin{array}{l}
c : Condition \\
arrows : \mathbb{F}\, Arrow \\
lines : Object \pfun \operatorname{iseq} Arrow \\
\hline
arrows = ran \bigcup \operatorname{ran} lines \\
\qquad\qquad \text{[the set } arrows \text{ contains exactly those appearing on lifelines]} \\
\forall\, a : arrows \bullet \\
\quad \#(\{a.from, a.to\} \cap \operatorname{dom} lines) = 2 \land \\
\qquad\qquad \text{[an arrow belongs to exacty 2 different lifelines]} \\
\quad a \in \operatorname{ran}(lines\ \ a.from) \cap \operatorname{ran}(lines\ a.to) \land \\
\qquad\qquad \text{[}to \text{ and } from \text{ are set to the correct lifelines]} \\
\quad a \notin \operatorname{ran} \bigcup \operatorname{ran}(\{a.from, a.to\} \ndres lines) \\
\qquad\qquad \text{[an arrow cannot belong to a wrong lifeline]} \\
\exists\, R : Arrow \rightarrow Arrow \bullet \\
\quad \forall\, a_1, a_2 : arrows \mid a_1 \neq a_2 \bullet \neg(a_1\ \underline{R^+}\ a_1) \land \\
\qquad (a_1\ \underline{R}\ a_2 \Leftrightarrow \exists\, s : \operatorname{ran} lines \bullet s \upharpoonright \{a_1, a_2\} = \langle a_1, a_2 \rangle) \\
\qquad\qquad \text{[arrows cannot go back in time]}
\end{array}
$$

This formalisation of sequence diagrams is the basis for our translation to CSP. Next we define a function from sequence diagrams to CSP processes which defines the

translation. The range of the function is the set of CSP processes defined by the following given type:

$$[CSP]$$

The precise syntax of processes will not be defined here, for an introduction to CSP see [18]. In the translation we use two operators of CSP: $\rightarrow$ is the *prefix* operator for modelling sequencing and $||$ is the parallel composition. Here we employ *alphabetised* parallel composition: for every process in a parallel composition its alphabet of events is given, and synchronisation has to take place on events in the intersection of alphabets. Syntactically this takes the form $|| (P_i, \alpha_i)$ where i ranges over some index set and the P_i are CSP processes with alphabets α_i. When there are just two processes we write $P_A||_B\ Q$. In contrast to ordinary parallel composition alphabetised parallel composition is associative which is convenient here.

The translation proceeds in two steps. The CSP process of the sequence diagram is the parallel composition of the CSP processes belonging to every lifeline of an object. These processes synchronise on shared events. Due to the parameters *to* and *from* in events, an event belongs to the alphabet of exactly two objects (and thus to exactly two CSP processes).

$$\begin{array}{|l} trans : SQ \rightarrow CSP \\ \hline \forall\, sq : SQ \bullet \\ \quad trans\ sq = || \{o : \mathrm{dom}\, lines \bullet (transLine\ o, alpha\ (o, sq))\} \end{array}$$

The alphabet of an object in a sequence diagram consists of the events over channels appearing on arrows of the object's lifeline with values for parameters *to* and *from* properly filled in.

$$\begin{array}{|l} alpha : Object \times SQ \rightarrow \mathbb{P}\, Event \\ \hline \forall\, o : Object, sq : SQ \bullet \\ \quad alpha(o, sq) = \{|\ a : \mathrm{ran}(sq.lines\ o) \bullet \\ \qquad\qquad (a.event.ch)_\bullet (a.event.val)(from)_\bullet (a.event.val)(to)\ |\} \end{array}$$

Note that the bold dots are those used for separating values of parameters in CSP events. We cannot just plainly use $a.event$ here since FDR is not accepting the notation $?\_$ in sets of events, only in process expressions.

The CSP processes of the lifelines are simply the sequential composition of the events on their arrows.

$$\begin{array}{|l} transLine : \mathrm{iseq}\, Arrow \rightarrow CSP \\ \hline transLine\langle\,\rangle = SKIP \\ \forall\, sa : \mathrm{iseq}\, Arrow, a : Arrow \bullet \\ \quad transLine\langle a\rangle \frown sa = a.event \rightarrow transLine\ sa \end{array}$$

The occurrence condition plays no role in the translation to CSP. It will be used for defining validity. As an example for the translation consider the sequence diagram in Figure 5 (here u is the identity of the user):

$$
\begin{array}{l}
\| \{ \\
(debit.a1.b?\_?\_ \rightarrow result.b.a1.false \rightarrow SKIP, \\
\qquad \{| \ debit.a1.b, result.b.a1 \}), \\
(amount.u.a1?\_ \rightarrow debit.a1.b?\_?\_ \rightarrow result.b.a1.false \rightarrow money.a1.u?\_ \rightarrow SKIP, \\
\qquad \{| \ amount.u.a1, debit.a1.b, results.b.a1, money.a1.u \ |\}), \\
(amount.u.a1?\_ \rightarrow money.a1.u?\_ \rightarrow SKIP, \\
\qquad \{| \ amount.u.a1, money.a1.u \ |\})\}
\end{array}
$$

4 Checking Validity

Having defined the CSP semantics of simple sequence diagrams and UML models, it is now possible to check the validity of the scenario in the UML model. For the check we employ the CSP model checker FDR [9]. FDR performs checks for deadlock- and divergence-freedom as well as *refinement* checks between processes. It is based on the semantic models of CSP. To see how we can use FDR for checking validity of sequence diagrams we first give a short summary of CSP's semantic models.

We assume Σ to be a global set of events (of type *Event*). The traces model $\mathcal{T}$ identifies a process P with the (prefix closed) set $traces(P) \subseteq \Sigma^*$ of sequences of events it can perform. Refinement in $\mathcal{T}$ is defined as $P \sqsubseteq_{\mathcal{T}} Q \Leftrightarrow traces(P) \supseteq traces(Q)$. A more powerful model is the stable failures model $\mathcal{F}$, which, in addition to $traces(P)$, records for any trace t of P the set of events R that P can stably refuse after performing t. The set of all these pairs $(t, R) \in \Sigma^* \times \mathbb{P}\,\Sigma$ is called $failures(P)$. A pair (t, Σ) identifies deadlock in $\mathcal{F}$, i.e., a trace after which P refuses all events[2]. The standard model for CSP is the failures-divergences model $\mathcal{N}$. Besides deadlock this model can deal with divergence. The (extension closed) set $divergences(P) \subseteq \Sigma^*$ contains all traces after which P can diverge. The failures in $\mathcal{N}$ differ slightly from those in $\mathcal{F}$; $failures_\perp(P)$ includes all (t, X), $X \subseteq \Sigma$, for any $t \in divergences(P)$ in addition to the stable failures. All three models are supported by the FDR model checker, and refinement checks in these models all amount to checking inclusion between the semantics of processes. Finally, we need the set $infinites(P)$, which belongs to the infinite traces models of CSP. It contains all infinite traces of P including the infinite extensions of divergent traces. It is not supported by FDR, but this does not concern our use of the set.

As a first step in the validation, we are interested in the language of a sequence diagram $sq : SQ$: $\mathcal{L}(sq) = runs(trans\ sq)$, where for a CSP process P $runs(P)$ is used for 'all runs of P'. With 'run' we denote only those finite traces which cannot be extended (apart from termination) and the infinite traces, i.e,

$$runs(P) = \{t : \mathrm{seq}\,\Sigma \mid (t, \Sigma) \in failures_\perp(P)\} \ \cup \ infinites(P)$$

Note that this differs from $traces(P)$ since the latter set also contains all prefixes. For the simple sequence diagrams defined in this paper, there are of course only finite runs; in fact, all of the examples here define exactly one sequence of events, i.e., a

[2] We do not explicitly treat successful termination here.

singleton set as their respective language. The focus in this section is therefore on validating an occurrence condition for a single word; generalization to a set of words is discussed at the end.

4.1 Occurrence of a Single Word

In the following let P denote the CSP process of the UML model (specified in the class diagram, state machine and structure diagrams) obtained by the translation sketched in Section 3.1. We assume E to be the alphabet of P and let $\mathrm{Seq}\, E$ denote the set of finite and infinite sequences of elements in E.

Definition 1. *Let $sq : SQ$ be a sequence diagram with $\mathcal{L}(sq) = \{s\}$ and P be the CSP process belonging to the UML model. The sequence diagram sq is* valid in P *iff the following hold:*

- $sq.condition = negative \Rightarrow \neg\, \exists\, t : runs(P);\ u : seq\, E;\ v : Seq\, E \bullet t = u^\frown s^\frown v$,
- $sq.condition = existential \Rightarrow \exists\, t : runs(P);\ u : seq\, E;\ v : Seq\, E \bullet t = u^\frown s^\frown v$,
- $sq.condition = universal \Rightarrow \forall\, t : runs(P) \bullet \exists\, u : seq\, E;\ v : Seq\, E \bullet t = u^\frown s^\frown v$.

To establish these occurrence conditions, it is sufficient to verify the following assertions:

$$\mathbf{A}_\exists : \qquad \exists\, t : traces(P);\ u : \mathrm{seq}\, E \bullet t = u^\frown s$$

for *negative* and *existential* conditions and

$$\mathbf{A}_\forall : \qquad \forall\, t : runs(P) \bullet \exists\, u : \mathrm{seq}\, E;\ v : \mathrm{Seq}\, E \bullet t = u^\frown s^\frown v$$

for *universal* conditions[3].

Since this is neither a refinement check in one of CSP's semantic models nor a check for deadlock- or divergence-freedom, we cannot directly use FDR but first have to reformulate the problem in a way tractable by FDR. The general idea is to use an auxiliary process ('tester'), which performs pattern matching for the sequence s on the stream of events it is offered. This process is then put in parallel to the process P for the system to be analysed, synchronizing on the whole alphabet of the system.

The pattern matching consists of keeping track of the longest prefix of s already matched, and calculating the resulting longest prefix after performing the next event. For this, a function δ is defined, which, for some alphabet $E : \mathbb{P}\, Event$ and a word $s : \mathrm{seq}\, E$, maps an already matched prefix s_1 of s together with an event e to the maximal prefix of s resulting from appending e to s_1, e.g., $\delta(\{a, b, c\}, \langle a, b, a, c\rangle, \langle a, b, a\rangle, b) = \langle a, b\rangle$.

[3] Although $runs(P)$ contains all possible (infinite) extensions of divergent traces, this is not relevant for $\mathbf{A}_\forall$, since for any infinite trace included in $runs(P)$ due to divergence, the finite prefix leading to divergence is already included. If s is contained in such a prefix, then all extensions contain it, too, and if it is not contained, then this prefix alone suffices as a counterexample, regardless of the extensions.

$$\delta : (\mathbb{P}\, Event \times \operatorname{seq} Event \times \operatorname{seq} Event \times Event) \nrightarrow \operatorname{seq} Event$$

$$\forall E : \mathbb{P}\, Event;\ s, s_1, s_2 : \operatorname{seq} E;\ e : E \mid s_1 \operatorname{prefix} s \bullet \delta(E, s, s_1, e) = s_2 \Leftrightarrow$$
$$s_2 \operatorname{prefix} s \wedge s_2 \operatorname{suffix} s_1 \frown \langle e \rangle \wedge \neg\, \exists\, x : E \bullet \langle x \rangle \frown s_2 \operatorname{suffix} s_1 \frown \langle e \rangle$$

This is basically the transition function of a deterministic finite automaton, where states are represented as sequences of events; s_1 is the current state, s the final state, e the current input and E the alphabet of the automaton.

Checking Assertion $\mathbf{A}_\exists$

In order to perform this check for some specific sequence of events s, a process S_{match} is constructed from s, which always accepts all events in E until it has performed s; in that case it performs an event $match \notin E$ and stops (deadlocks). Define $S_{match} = S_{\langle\,\rangle}$ with

$$S_s = match \rightarrow STOP$$
$$S_{s_1} = \Box_{e:E}\, e \rightarrow S_{\delta(E,s,s_1,e)}$$

for all true prefixes $s_1 \subset s$. The *traces* model ($\mathcal{T}$) is sufficient for this test, where S_{match} looks like this:

$$traces(S_{match}) = \{t, t_1 : \operatorname{seq} E \mid \neg(s \operatorname{infix} t) \wedge (t_1 \subseteq t \frown s \frown \langle match \rangle) \bullet t_1\}$$

Example. As an example, S_{match} is now constructed for the sequence diagram of Fig. 5. Using the abbreviations $A \equiv amount.u.a1?\_$, $D \equiv debit.a1.b?\_?\_$, $M \equiv money.a1.u?\_$ and $R \equiv result.b.a1.false$, as well as $E = \{|\ amount, debit, result, money\ |\}$, the language defined by that diagram is the set $\{\langle A, D, R, M\rangle\}$ and we therefore only need to check for the single word in that set, i.e., we can use the construction described above, yielding:

$$S_{match} = S_{\langle\,\rangle}$$
$$S_{\langle\,\rangle} = A \rightarrow S_{\langle A\rangle} \,\Box\, \Box_{e:E\setminus\{A\}}\, e \rightarrow S_{\langle\,\rangle}$$
$$S_{\langle A\rangle} = D \rightarrow S_{\langle A,D\rangle} \,\Box\, \Box_{e:E\setminus\{D\}}\, e \rightarrow S_{\langle\,\rangle}$$
$$S_{\langle A,D\rangle} = R \rightarrow S_{\langle A,D,R\rangle} \,\Box\, \Box_{e:E\setminus\{R\}}\, e \rightarrow S_{\langle\,\rangle}$$
$$S_{\langle A,D,R\rangle} = M \rightarrow S_{\langle A,D,R,M\rangle} \,\Box\, \Box_{e:E\setminus\{M\}}\, e \rightarrow S_{\langle\,\rangle}$$
$$S_{\langle A,D,R,M\rangle} = match \rightarrow STOP$$

Next we put P and S_{match} in parallel, synchronising on the whole alphabet E of P and hide all of E, since we are only interested in the occurrence of *match*. Then we use FDR for a refinement check in the traces model in order to check for said occurrence:

$$(P \parallel_E S_{match}) \setminus E \sqsubseteq_{\mathcal{T}} match \rightarrow STOP \qquad (*)$$

This is correct since the following correspondence holds

$$(P \parallel_E S_{match}) \setminus E \sqsubseteq_{\mathcal{T}} match \rightarrow STOP \quad \Leftrightarrow \quad \mathbf{A}_{\exists}$$

because $(P \parallel_E S_{match}) \setminus E$ cannot, due to hiding E, perform any event but *match* and *match* can be performed at most once due to the construction of S_{match}; furthermore, the refinement relation holds, if and only if $(P \parallel_E S_{match}) \setminus E$ can perform *match* at least once, which again means that P must have a trace which contains s.

Thus, for checking validity of sequence diagrams with negative conditions we use FDR to check $(*)$, if this fails the sequence diagram is valid in the UML model; for sequence diagrams with existential conditions we use $(*)$ as well and validity holds if the test is successful.

Checking Assertion $\mathbf{A}_{\forall}$

This task is a little more complicated, because of the universal quantification. We cannot use the simple traces model here, but need to take deadlock and divergence into account: if s is to occur on all runs of P, then P must neither be able to deadlock nor to diverge until it has performed s. Furthermore, each (non diverging) infinite run of P has to contain s. Therefore we split this task into two checks. For the first one a process S_{div} is constructed from s, which differs from S_{match} only in the definition

$$S_s = \mathbf{div}$$

that is, it diverges immediately after performing s instead of performing an additional event and stopping. The test using this process is carried out in the *stable failures* model ($\mathcal{F}$), where S_{div} is described as

$$\begin{aligned} traces(S_{div}) &= \{t, t_1 : \operatorname{seq} E \mid \neg(s \operatorname{infix} t) \wedge (t_1 \subseteq t^\frown s \bullet t_1\} \\ failures(S_{div}) &= \{t : traces(S_{div}) \bullet (t, \varnothing)\} \end{aligned}$$

Proposition 1. *$P \parallel_E S_{div}$ deadlock free ($\mathcal{F}$) $\Leftrightarrow$ P cannot deadlock (in a stable state) until it has performed s.*

Proof. Suppose $P \parallel_E S_{div}$ is deadlock free in $\mathcal{F}$. S_{div} is constructed as not to constrain the behaviour of P unless it has performed s, in which case it diverges and thus prevents P from performing any further events without introducing deadlock. Since S_{div} cannot cause deadlock, it follows that P cannot deadlock in a stable state on all behaviours allowed by S_{div}, i.e, until it has performed s. If on the other hand, P cannot deadlock in a stable state until it has performed s, it follows that $P \parallel_E S_{div}$ is deadlock free in $\mathcal{F}$, since S_{div} does not refuse any event until it has performed s, and then stops P by diverging, which (in $\mathcal{F}$) does not introduce deadlock. □

The second test requires an auxiliary process S_{not}, which always accepts any event in E, except when it has already performed all but the last event of s, in which case S_{not} refuses exactly that event, but no other. Let $s_0^\frown\langle f\rangle = s$, then

$$\begin{aligned} S_{s_0} &= \Box_{e:E\setminus\{f\}} e \rightarrow S_{\delta(E,s,s_0,e)} \\ S_{s_1} &= \Box_{e:E} e \rightarrow S_{\delta(E,s,s_1,e)} \end{aligned}$$

for any prefix $s_1 \subset s$ with $\#s_1 \leqslant \#s - 2$. Finally, $S_{not} = S_{\langle\rangle}$.

This test is carried out using the *failures-divergences* model ($\mathcal{N}$). In this model S_{not} looks like this:

$$\begin{aligned} \mathit{failures}_{\perp}(S_{not}) = \{t, s_1, s_2 : \operatorname{seq} E;\ e : E;\ R : \mathbb{P}\{e\} \mid \\ \neg(s \operatorname{infix} t) \wedge s = s_1 \frown s_2 \frown \langle e \rangle \bullet \\ (t \frown s_1, \text{if } s_2 = \langle\rangle \text{ then } R \text{ else } \varnothing)\} \end{aligned}$$

$$\mathit{divergences}(S_{not}) = \varnothing.$$

Proposition 2. *$(P \parallel_E S_{not}) \setminus E$ divergence free $\Leftrightarrow$ P cannot diverge until it has performed s and P has no infinite trace (non-diverging), which does not contain s.*

Proof Suppose $(P \parallel_E S_{not}) \setminus E$ is divergence free. Since S_{not} does not constrain the behaviour of P unless P wants to perform the last event of s, it follows that P cannot diverge unless it has performed s. Furthermore, since the hiding of E turns any infinite behaviour of $(P \parallel_E S_{not}) \setminus E$ into divergence, it follows that $(P \parallel_E S_{not})$ does not have infinite traces, i.e., any infinite trace of P contains s. On the other hand, if P cannot diverge until it has performed s and does not have an infinite trace, which does not contain s, then $(P \parallel_E S_{not})$ is divergence free and has not infinite traces. Thus $(P \parallel_E S_{not}) \setminus E$ divergence free. □

Summarising, we have

$$P \parallel_E S_{div} \text{ deadlock free } (\mathcal{F}) \ \wedge \ (P \parallel_E S_{not}) \setminus E \text{ divergence free}$$

if and only if

- P cannot deadlock until it has performed s and
- P cannot diverge until it has performed s and
- P has no infinite trace (non-diverging) which does not contain s,

i.e., any finite or (non-diverging) infinite run of P contains s. For the assertion $\mathbf{A}_\forall$ we thus get the following

Result

$$P \parallel_E S_{div} \text{ deadlock free } (\mathcal{F}) \ \wedge \ (P \parallel_E S_{not}) \setminus E \text{ divergence free} \quad \Leftrightarrow \quad \mathbf{A}_\forall$$

Validity of sequence diagrams with universal condition can hence be checked using the above two tests.

4.2 Occurrence of a Set of Words

Besides single words the language of a sequence diagram can also contain more than one word. This is the case if the sequence diagram specifies certain interactions to be concurrent. The corresponding CSP process will then contain all possible interleavings in its set of runs. Given not just a single word but a (finite) set $L = \mathcal{L}(sq)$ of words

from a sequence diagram sq, there are several possible definitions for 'occurrence' both in the existential as well as in the universal case: does 'possible' mean *in some run* r *some* $s \in L$ *occurs* ($\exists\, r\, \exists\, s$), *each* $s \in L$ *occurs in some run* r ($\forall\, s\, \exists\, r$) or *in some run* r *all* $s \in L$ *occur* ($\exists\, r\, \forall\, s$)? Does 'required' mean *on all runs* r *some* $s \in L$ *occurs* ($\forall\, r\, \exists\, s$), *some* $s \in L$ *occurs on all runs* r ($\exists\, s\, \forall\, r$) or *on all runs* r *all* $s \in S$ *occur* ($\forall\, r\, \forall\, s$)?

The third version for each case ($\exists\, r\, \forall\, s$, $\forall\, r\, \forall\, s$) is clearly too strong as a general interpretation, but all other versions can be justified. In our case, though, since L is derived from a sequence diagram, the actual interleaving of events for which no ordering is implied by the lifelines is irrelevant. Therefore, the weakest versions ($\exists\, r\, \exists\, s$, $\forall\, r\, \exists\, s$) are sufficient here.

The simplest, yet slightly inefficient way to perform the necessary tests for a set L, is to construct the respective processes for all the $s \in L$ and to use them all at once, i.e., to put them in parallel, synchronising on E.

5 Conclusion

In this paper we proposed a method for checking the validity of (simple) sequence diagrams in a UML model. To this end we supplied both UML model and sequence diagrams with a CSP semantics and developed procedures for emplyoing CSP's model checker FDR for validity checking. The technique can easily be extended to UML 2.0 sequence diagrams by developing a CSP semantics for them or using the semantics of [22] which defines it as a set of (timed) traces over actions. Our approach of developing testers for validity checking can then stay as it is.

Related work. The question of consistency in models with multiple views (of which our issue is one special aspect) is widely studied. A general approach for defining consistency in models with multiple views is studied in [2]. In the context of UML an annual workshop with the topic of consistency is carried out [5]. The use of CSP as a common semantic domain for multiple views in the study of consistency is chosen in [7, 17, 1]. While the first two do not consider sequence diagrams the work of Bolton and Davies addresses sequences diagrams and develops tests for checking whether a scenario is possible in a UML model. They do, however, only treat the case that the scenario starts in the *initial* state; their tests do not cover scenarios happening sometime later after initialisation. As a consequence, checking validity simply amounts to checking for trace refinement and thus FDR thus be directly employed.

Syntactic checks of consistency between class and sequence diagrams are proposed in [23]. Using labelled trasition systems, [4] defines behavioural consistency for combinations of UML behavioural diagrams (including statecharts and sequence diagrams) as deadlock freedom. They use the SPIN model checker for analysis, but do not give a formal translation from UML to SPIN's input language.

The work most closest to us is that carried out in the context of life sequence charts (LSCs) [6]. LSCs are an extension of sequence diagrams with special features for modelling liveness requirements. The work [3] proposes validity checking for LSCs by translating them to the temporal logic LTL and checking them against Statemate models. Based on LSCs, the *play in play out* approach [11] uses a collection of 'played

in' examples to specify a whole system, instead of using them only as requirements for an explicit model.

References

1. C. Bolton and J. Davies. Using relational and behavioural semantics in the verification of object models. In S. Smith and C. Talcott, editors, *FMOODS'00: Formal Methods for Open Object-Based Distributed Systems*, pages 163–182. Kluwer, 2000.
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. M. Brill, R. Buschermöhle, W. Damm, J. Klose, B. Westphal, and H. Wittke. Formal verification of LSC's in the development process. In H. Ehrig, W. Damm, M. Grosse-Rhode, W. Reif, E. Schnieder, and E. Westkmper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, number 3147 in LNCS. Springer, 2004.
4. Y. Choi and C. Bunse. Behavioral consistency checking for component-based software development using the KobrA approach. In *Consistency Problems in UML-based Software Development III*, Lisbon, Portugal, October 2004. revised papers.
5. Consistency problems in UML-based software development III – understanding and usage of dependency relationships, October 2004. `http://uml04.ci.pwr.wroc.pl/`.
6. W. Damm and D. Harel. LSC's: Breathing life into Message Sequence Charts. In P. Ciancarini and R. Gorrieri, editors, *FMOODS'99: Formal Methods for Open Object-based Distributed Systems*, 1999.
7. 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.
8. 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.
9. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement – FDR2 User Manual*, 4th edition, August 1998.
10. T. Gehrke, M. Huhn, A. Rensink, and H. Wehrheim. An algebraic semantics for message sequence chart documents. In S. Budkowski, A. Cavalle, and E. Najm, editors, *FORTE/PSTV'98: Formal Description Techniques & Protocol Specification, Testing and Verification*, pages 3–18, 1998.
11. D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: The Play In/Play-Out Approach. *Software and System Modeling*, 2:82–107, 2003.
12. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1985.
13. International Organisation for Standardization. *Information technology – Z formal specification notation – Syntax, type system and semantics*, 1st edition, July 2002. ISO/IEC 13568:2002 (E) International Standard.
14. P.B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.
15. S. Mauw and M.A. Reniers. An Algebraic Semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.
16. M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Linking CSP-OZ with UML and Java: A case study. In *Integrated Formal Methods*, number 2999 in Lecture Notes in Computer Science, pages 267–286. Springer, March 2004.
17. H. Rasch and H. Wehrheim. Checking Consistency in UML Diagrams: Classes and State Machines. In *FMOODS 2003: Formal Methods for Open Object-based Distributed Systems*, number 2884 in LNCS, pages 229–243. Springer, 2003.

18. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science. Prentice Hall Europe, 1998.
19. B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.
20. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
21. J. M. Spivey. *The Z Notation: A Reference Manual*. Oriel College, Oxford, 2nd edition, 1998.
22. H. Störrle. Semantics of Interactions in UML 2.0. In *Intl. Workshop on Visual Languages and Formal Methods*, 2003. at HCC'03, Auckland, NZ.
23. A. Tsiolakis and H. Ehrig. Consistency Analysis of UML Class and Sequence Diagrams using Attributed Graph Grammars. In H. Ehrig and G. Taentzer, editors, *GRATRA 2000*. TU Berlin, FB Informatik, TR No. 2000-2, pp. 77-86, March 2000.
24. OMG Unified Modeling Language specification, version 1.5, March 2003. `http://www.omg.org`.