# Using Use Cases in Executable Z

Wolfgang Grieskamp Markus Lepper
*Technische Universität Berlin, FB13*
{wg,lepper}@cs.tu-berlin.de

## Abstract

*Use Cases are a wide-spread* informal *method for specifying the requirements of a technical system in the early development phase. Z is a* formal *notation which aims to support, beside others, the specification of early requirements. In this paper, we develop a representation of Use Cases in Z and apply it to several examples. Our focus is on instrumenting the formalization for* black-box test evaluation *in* Executable Z, *a computation model and implementation for Z based on concurrent constraint resolution*

## 1. Introduction

Use Cases [9] are a wide-spread *informal* method for specifying the requirements of a technical system in the early development phase. They provide a methodology for the loose but nevertheless systematic description of aspects of a system's behavior. Z [8, 10] is a *formal* notation which aims to support, beside others, the specification of early requirements. Combining Use Cases and Z is therefore an interesting experiment: from Use Cases we may inherit the methodology – from Z, we inherit a formal meaning, and thus the possibility to apply tools for consistency check and validation.

In this paper, we develop a shallow encoding of Use Cases in Z and apply it to several examples. As a surplus of our approach, we get consistency checks by Z tools such as type checkers. However, our focus in instrumenting the formalization is on *black-box test evaluation*: given a set of Use Cases in Z, some input data describing a test-case, and the output data from a run of the system's implementation on the given input, we check by executing the Use Cases whether the implementation confirms to the requirements – as far as they are formalized. To this end, we use the ZAP (version 2) plugin of the ZETA tool environment [2] which allows for the execution of significant parts of the Z notation using concurrent constraint resolution techniques [6].

## 2. Use Cases in Z

**What Are Use Cases?** There is an ongoing discussion about syntax, semantics and methodology of Use Cases in the software engineering community (see e.g. [1]). Opposed to the graphic formalisms for *combining* Use Cases, e.g. by the "Use Case Diagrams" offered by UML [9], the means for specifying the *contents* of a single Use Case is not agreed upon at all.

The UML semantics state that "a Use Case can be described in plain text, using operations, in activity diagrams, by a state-machine, or by other behavior description techniques...."(UML semantics, cited from [4]). For our purpose of applying formal techniques we need an unambiguous description technique which is amenable to exact reasoning such as test evaluation. We therefore develop a model related to *temporal interval logic*, explicitely using *nondeterministic choice*, *repetition* and *interruption*.

In summary, we use the following informal definition of Use Cases, near to the one found in [3]:

- The systems we observe are characterized by sequences of *interactions*. Sequences of interactions are called *dialogues*.
- Each interaction has assigned a certain *actor*. The actors are often one human and one technical system, but several humans can also talk to several machines, or machines can talk to each other. The important methodological principle is that we only look at the *observable* behavior of each actor and that all internal state is hidden.
- Use Cases are described by a so-called *fragments* of dialogues between two or more actors. A fragment schematically specifies a set of possible dialogues by a "pattern" of interactions.
- Fragments can be combined by sequential composition, nondeterministic choice, repetition, and interruption.
- We have an (observable) global system state which all Use Cases share. This extension compared to a "puristic" idea of Use Cases allows us to abstract Use Cases over some data state. In the fragments we can specify how this data state is transformed.

Note that we do not restrict our model to only two actors, as often found in the literature, eg. [3]. Moreover, we do not impose *a priori* that actors in dialogues do alternate. Finally, we do not have a builtin concept of "idle" states.

**Example: Cash Dispenser.** We look at the fragments of (simplified) dialogues between a user and a cash dispenser (Spec. 1). The following basic constructors for fragments are used (a formal definition follows later on):

- $actor \diamond action$ constructs a fragment containing the single interaction where *actor* performs *action*.
- $actor \diamond action \,/\, rel$ is the general version of constructing singleton fragments. In addition to $actor \diamond action$, a transformation on the system state is given by the relation *rel*. We have $actor \diamond action = actor \diamond action \,/\, \mathrm{id}$.
- $frag \curvearrowright frag'$ is the sequential composition of fragments. After the dialogues described by *frag* the ones of $frag'$ must follow.
- $\mathbf{select}\, frags$ describes a choice between several fragments. *frags* can be an enumeration of fragments, but also a set-comprehension: we use the pattern $\mathbf{select}\{x : A \bullet frag\}$ to introduce a locally bound variable *x* in fragments, which semantically is the choice between all possible instantiations of *x*.
- $\mathbf{repeat}\, frag$ describes the repetition of *frag* for zero or more times.
- $frag\ \mathbf{except}\ frag'$ describes that *frag* can be "interrupted" at some interaction which overlaps with the first interaction of $frag'$. It is then continued with the behavior of $frag'$.

For the cash dispenser in Spec. 1, the type *ACTOR* defines the actors, *user* and *dispenser*. The type *ACTION* lists the actions performed. The type *State* defines the system state, which is the money reserve of the dispenser (in an extended version, we might represent here the accounts of individual card holders). We define two operations *Draw* and *CantDraw* to be used in the fragments which work on the system state[1].

From a methodological point of view, we have to justify that the visible representation of *reserves* is not a violation of our principle that inner system state must *not* be modelled in Use Cases. We argue that the reserves *are* observable, since the user may be confronted with the situation that the dispenser cannot satisfy his requests because of low reserves.

The fragments are given as follows. *Normal* describes the usual process of a disposition. *InvalidCard* is an exceptional fragment which can interrupt *Normal* at the point where the user has inserted his card; the card is immediately rejected and the dialogue ends if the user has removed his card. *NoReserves* is a further exceptional behavior; it can continue at the point where the user enters the amount to draw, *and* where this amount cannot be served because of low reserves. Note the use of our choice operator, **select**, to bind the input variable *amount*? in *Normal* and *NoReserves*. The overall behavior of the system is described by a repetition of the *Normal* fragment which can be interrupted by *InvalidCard* or by *NoReserves*.

**Specification 1** Cash Dispenser

$\mathbf{section}\ CashDispenser\ \mathbf{parents}\ UseCases$

$$\begin{array}{l} ACTOR ::= user \mid dispenser \\ ACTION ::= askCard \mid putCard \mid ejectCard \mid takeCard \mid \\ \qquad askAmount \mid putAmount\langle\langle \mathbb{Z} \rangle\rangle \mid \\ \qquad ejectMoney\langle\langle \mathbb{Z} \rangle\rangle \mid takeMoney \end{array}$$

State
$$reserves : \mathbb{Z}$$

CantDraw
$$\begin{array}{l} \Xi State;\ amount? : \mathbb{Z} \\ \hline reserves < amount? \end{array}$$

Draw
$$\begin{array}{l} \Delta State;\ amount? : \mathbb{Z} \\ \hline reserves \geq amount?;\ reserves' = reserves - amount? \end{array}$$

$$\uparrow == \lambda\, Op : \mathbb{P}(\Delta State) \bullet \{Op \bullet (\theta State, \theta State')\}$$

$$\begin{array}{l} Normal, InvalidCard, NoReserves, System : \\ \quad Fragment[ACTOR, ACTION, State] \\ \hline \begin{array}{lll} Normal & = & dispenser \diamond askCard \\ & & \curvearrowright user \diamond putCard \\ & & \curvearrowright dispenser \diamond askAmount \\ & & \curvearrowright \mathbf{select}\{amount? : \mathbb{Z} \bullet \\ & & \quad user \diamond putAmount(amount?) \,/ \\ & & \qquad\quad \uparrow [\Delta State \mid Draw] \\ & & \curvearrowright dispenser \diamond ejectCard \\ & & \curvearrowright user \diamond takeCard \\ & & \curvearrowright dispenser \diamond ejectMoney(amount?) \\ & & \curvearrowright user \diamond takeMoney\} \\ InvalidCard & = & user \diamond putCard \\ & & \curvearrowright dispenser \diamond ejectCard \\ & & \curvearrowright user \diamond takeCard \\ NoReserves & = & \mathbf{select}\{amount? : \mathbb{Z} \bullet \\ & & \quad user \diamond putAmount(amount?) \,/ \\ & & \qquad\quad \uparrow [\Delta State \mid CantDraw] \\ & & \curvearrowright dispenser \diamond ejectCard \\ & & \curvearrowright user \diamond takeCard\} \\ System & = & \mathbf{repeat}(Normal\ \mathbf{except}\ InvalidCard \\ & & \qquad\qquad\qquad \mathbf{except}\ NoReserves) \end{array} \end{array}$$

**Formal Model.** We define the formal model of our version of Use Cases in Z. An *interaction* is given by the schema *Interaction*, generic over the type of actors $\alpha$ and of actions $\pi$. A *dialogue* is a sequence of interactions:

[1] These operations are conveniently written as $\Delta$-schemata. For sake of type correctness these schemata have to be lifted to binary relations between undecorated schemata, which is done by the $\uparrow$-operator. Unfortunately this operator cannot be formulated in a generic way.

$\textbf{section}\ \mathit{UseCasesModel}$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Interaction}[\alpha,\pi]} \\ \hline
\mathit{actor} : \alpha;\ \mathit{action} : \pi \\ \hline
\end{array}$$

$$\mathit{Dialogue}[\alpha,\pi] == \mathrm{seq}\,\mathit{Interaction}[\alpha,\pi]$$

A *pattern* is a sequence of interactions paired with a state transition relation over the state type $\Sigma$. A *fragment* is a set of patterns:

$$\begin{array}{l}
\mathit{Pattern}[\alpha,\pi,\Sigma] == \mathrm{seq}(\mathit{Interaction}[\alpha,\pi] \times (\Sigma \leftrightarrow \Sigma)) \\
\mathit{Fragment}[\alpha,\pi,\Sigma] == \mathbb{P}\,\mathit{Pattern}[\alpha,\pi,\Sigma]
\end{array}$$

Our basic constructor functions for fragments are defined as follows[2] :

$$\begin{array}{l}
\textbf{function}\ 65(\_ :: \_) \\
\textbf{function}\ 60(\_ \diamond \_) \\
\textbf{function}\ 60(\_ \diamond \_ \,/\, \_)
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{[\alpha,\pi]} \\ \hline
\_ :: \_ == \lambda\, \mathit{actor} : \alpha;\ \mathit{action} : \pi \bullet \theta\mathit{Interaction}[\alpha,\pi] \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{[\alpha,\pi,\Sigma]} \\ \hline
\_ \diamond \_ \,/\, \_ == \lambda\, \mathit{actor} : \alpha;\ \mathit{action} : \pi;\ r : \Sigma \leftrightarrow \Sigma \bullet \\
\qquad\qquad\qquad \{\langle(\mathit{actor} :: \mathit{action}, r)\rangle\} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{[\alpha,\pi,\Sigma]} \\ \hline
\_ \diamond \_ == \lambda\, \mathit{actor} : \alpha;\ \mathit{action} : \pi \bullet \mathit{actor} \diamond \mathit{action} \,/\, \mathsf{id}[\Sigma] \\ \hline
\end{array}$$

For the sequential composition, $f \curvearrowright f'$, all combinations of the patterns in $f$ and $f'$ are concatenated:

$$\textbf{function}\ 50\ \mathrm{rightassoc}\ (\_ \curvearrowright \_)$$

$$\begin{array}{|l}
\multicolumn{1}{l}{[\alpha,\pi,\Sigma]} \\ \hline
\_ \curvearrowright \_ == \lambda\, f, f' : \mathit{Fragment}[\alpha,\pi,\Sigma] \bullet (\_ \frown \_)(\!|\, f \times f' \,|\!) \\ \hline
\end{array}$$

For the repetition, $\textbf{repeat}\, f$, we construct a relation which concatenates some pattern $p_1$ with some pattern $p_2 \in f$. The image of the transitive closure of this relation on the empty fragment represents all possible concatenations of the patterns of the repeated fragment:

$$\begin{array}{|l}
\multicolumn{1}{l}{[\alpha,\pi,\Sigma]} \\ \hline
\textbf{repeat} == \lambda\, f : \mathit{Fragment}[\alpha,\pi,\Sigma] \bullet \\
\quad \{p_1 : \mathit{Pattern}[\alpha,\pi,\Sigma];\ p_2 : f \bullet (p_1, p_1 \frown p_2)\}^* (\!|\, \{\langle\rangle\} \,|\!) \\ \hline
\end{array}$$

For the interruption operator, $f\ \textbf{except}\ f'$, we enrich $f$ by all patterns consisting of a prefix of $p \in f$ concatenated with a continuation $p' \in f'$ such that the interaction at the end of the prefix of $p$ coincides with the interaction at the beginning of $p'$. Note that the state transition relation at this overlapping point is taken from $p'$, not from $p$:

$$\textbf{function}\ 40\ \mathrm{leftassoc}\ (\_\ \textbf{except}\ \_)$$

$$\begin{array}{|l}
\multicolumn{1}{l}{[\alpha,\pi,\Sigma]} \\ \hline
\_\ \textbf{except}\ \_ == \lambda\, f, f' : \mathit{Fragment}[\alpha,\pi,\Sigma] \bullet f \cup \\
\quad \{p : f;\ p' : f';\ i : \mathbb{N} \mid i \in \mathsf{dom}\, p;\ \mathit{first}(p\, i) = \mathit{first}(p'\, 1) \\
\qquad \bullet ((1 \mathinner{..} i-1) \lhd p) \frown p'\} \\ \hline
\end{array}$$

Our last construction operator for fragments, $\textbf{select}\, \mathit{fs}$, is just an alias for generalized union, collecting all patterns from from all fragments $f \in \mathit{fs}$:

$$\begin{array}{|l}
\multicolumn{1}{l}{[\alpha,\pi,\Sigma]} \\ \hline
\textbf{select} == \bigcup[\mathit{Fragment}[\alpha,\pi,\Sigma]] \\ \hline
\end{array}$$

So far we have seen how fragments are *constructed*. The *satisfaction relation* on fragments, $(d,\sigma) \in_F f$, relates the dialogue $d$ and intial state $\sigma$ with the fragments $f$ they confirm to:

$$\textbf{relation}\ (\_ \in_F \_)$$

$$\begin{array}{|l}
\multicolumn{1}{l}{[\alpha,\pi,\Sigma]} \\ \hline
\_ \in_F \_ : \mathit{Dialogue}[\alpha,\pi] \times \Sigma \leftrightarrow \mathit{Fragment}[\alpha,\pi,\Sigma] \\ \hline
\forall\, d : \mathit{Dialogue}[\alpha,\pi];\ \sigma : \Sigma;\ f : \mathit{Fragment}[\alpha,\pi,\Sigma] \bullet \\
\quad (d,\sigma) \in_F f \Leftrightarrow \\
\qquad (\exists\, p : f \bullet \\
\qquad\quad \sigma \in \mathsf{dom}(\mathit{fold}(\_ \fatsemi \_)(\mathit{second} \circ p)) \land \mathit{first} \circ p = d) \\ \hline
\end{array}$$

Thus, a dialogue and initial state confirms to a fragment if there exists a pattern in the fragment such that the initial state is in the domain of the composition of all state transitions in the pattern and the interactions of the pattern match the dialogue (where $\mathit{fold}\, f\, \langle x_1, \ldots, x_n\rangle$ denotes $x_1\, f\, \ldots\, f\, x_n$.)

## 3. Executing Use Cases

In [5, 6] a computation model based on concurrent constraint resolution has been developed for Z. A high-performance virtual machine has been derived, which is implemented as part of the notation and tool integration environment ZETA [2]. In this implementation, all idioms of Z which are related to *functional* and *logic* programming languages are executable. Below, we illustrate the basic features, and develop an encoding of fragments which is executable.

**Executing Z.** As sets are paradigmatic for the specification level of Z, they are for the execution level. Set objects

[2] The template declarations for user-defined functions and relations are new features of standard Z. So is the **section**-construct which allows seperate type checking and execution of parts of this document.

– relations or functions – are executable if they are defined by (recursive) equations, as in the following example:

**section** *ExecExamples*

$N ::= Z \mid S\langle\!\langle \{x : N\} \rangle\!\rangle$ $\qquad$ $three == S(S(S(Z)))$

$$\begin{array}{|l} add : \mathbb{P}((N \times N) \times N) \\ \hline add = \{y : N \bullet ((Z, y), y)\} \cup \\ \qquad \{x, y, z : N \mid ((x, y), z) \in add \bullet ((S\,x, y), S\,z)\} \end{array}$$

$$\begin{array}{|l} less == \{x, y, z : N \mid ((x, S\,z), y) \in add \bullet (x, y)\} \end{array}$$

We may now execute queries such as the following, where we ask for the pair of sets containing all those $N$ less resp. greater than *three*:

$$(\{x : N \mid (x, three) \in less\}, \{x : N \mid (three, x) \in less\})$$

```
⇝ ({Z,S(Z),S(S(Z))},{S(S(S(S(x))))})
```

Note that the second value of the resulting pair is a singleton set containing the free variable $x$. These capabilities are similar to logic programming. In fact, we can give a translation from any clause-based system to a system of recursive set-equations in the style given for *add*, where we collect all clauses for the same relational symbol into a union of set-comprehensions, and map literals $R(e_1, \ldots, e_n)$ to membership tests $(e_1, \ldots, e_n) \in R$.

The functional paradigm comes into play as follows: as known, a binary relation $R$ can be *applied* in Z, written as $R\,e$, which is syntactic sugar for the expression $\mu\, y : X \mid (e, y) \in R$. For computing goals such as application or $\mu$-values, we use *encapsulated search*. During encapsulated search free variables from the enclosing context are not allowed to be bound. A constraint requiring a value for such variables *residuates* until the context binds the variable.

As a consequence, if we had defined the recursive path of *add* as $\{x, y, z : N \mid z = add(x, y) \bullet ((S\,x, y), S\,z)\}$ (instead of using $((x, y), z) \in add$), backwards computation would not be possible:

$$\{x : N \mid (x, three) \in less\}$$

```
⇝  unresolved constraints:
    LTX:cpinz(48.24-48.31)
        waiting for variable x
```

Here, the encapsulated search for $add(x, y)$ cannot continue since it is not allowed to produce bindings for the context variables $x$ and $y$. This way, we can control evaluation order.

The elegance of the functional paradigm comes from the fact that functions are first-order citizens. In our implementation of execution for Z, sets are full first-order citizens as well. For example, we can implement operators such as relational image as follows:

$$\begin{array}{|l} [X, Y] \\ \hline \_(\!|\_|\!) == \lambda R : \mathbb{P}(X \times Y);\ S : \mathbb{P}\,X \bullet \\ \qquad \{x : X;\ y : Y \mid x \in S \wedge (x, y) \in R \bullet y\} \end{array}$$

We can now, for instance, query for the relational image, $R(\!|S|\!)$, of the *add* function over the cartesian product of the numbers less then three:

$$\mathbf{let}\ ns == \{x : N \mid (x, three) \in less\} \bullet add(\!|ns \times ns|\!)$$

```
⇝ {Z,S(Z),S(S(Z)),S(S(S(Z))),
    S(S(S(S(Z))))}
```

It is also possible to define the arrow types of Z, as shown below for the set of partial functions:

$$\begin{array}{|l} [X, Y] \\ \hline \_ \mathrel{+\!\!\!\!\to} \_ == \\ \quad \{R : \mathbb{P}(X \times Y) \mid \\ \qquad (\forall x : X \mid x \in \mathrm{dom}\, R \bullet \exists_1\, y : Y \bullet (x, y) \in R)\} \end{array}$$

This example makes use of universal and unique existential quantification, which are a source of non-executability in our setting. These quantifiers are resolved by encapsulated search, and we must be able to finitely enumerate the quantified range. Thus, if we try to check whether *add* is a function, we get in a few seconds:

$$add \in N \times N \mathrel{+\!\!\!\!\to} N$$

```
⇝ still searching after 200000 steps
   gc # 1 reclaimed 28674k of 32770k
   ...
```

In enumerating *add* our computation diverges. However, for finite relations it works:

$$(\lambda\, x, y : N \mid (x, three) \in less;\ (y, three) \in less \bullet add(x, y)) \\ \in N \times N \mathrel{+\!\!\!\!\to} N$$

```
⇝ *true*
```

The example also illustrates a rough edge of our approach. The Z semantics defines the schema text $f : N \mathrel{+\!\!\!\!\to} N$ to be equivalent to $f : \mathbb{P}(N \times N) \mid f \in N \mathrel{+\!\!\!\!\to} N$. This treatment causes serious problems for executability, as we have seen. In the implementation of executable Z, we therefore *discard* constraints introduced by declarations; they are treated as *assumptions* which may be utilized by the compiler. If a declared membership is actually a constraint required for execution, the user has to place it in the constraint part of schema text.

**Executable Encoding of Uses Cases.** The satisfaction relation for fragments, $(d, \sigma) \in_F f$, where $d$ is a dialogue, $\sigma$ a system state and $f$ a fragment, is not executable in its descriptive definition from the previous section. We could perhaps define an executable version of the constructors for fragments and of the satisfaction relation: however, the representation of fragments as sets of patterns is a dead-end regarding *efficient* executability. The problem comes apparent in the definition of $f \curvearrowright f' = (\_ \frown \_)(\!| f \times f' |\!)$: a common prefix $p \in f$ is not shared in the composition and needs to be "parsed" again for every $p' \in f'$. A better representation would use *trees*, preserving common prefixes in fragments. Based on this idea we now will develop an encoding of fragments allowing the efficient execution of the satisfaction relation.

For a tree-like representation, we encode fragments as a set of *branches*. A branch is either eod – indicating that a dialogue may end here – or $\mathsf{br}(i, r, f)$, where $i$ is the interaction at the head of this branch, $r$ the state transition, and $f$ the followup fragment:

**section** *UseCases*

$$\begin{array}{|l}
\hline
\textit{Interaction}[\alpha, \pi] \\
\textit{actor} : \alpha;\ \textit{action} : \pi \\
\hline
\end{array}$$

$$\textit{Dialogue}[\alpha, \pi] == \mathrm{seq}\,\textit{Interaction}[\alpha, \pi]$$

$$\begin{array}{l}
\textit{Branch}[\alpha, \pi, \Sigma] ::= \\
\quad \mathsf{eod} \mid \\
\quad \mathsf{br}\langle\!\langle \textit{Interaction}[\alpha, \pi] \times (\Sigma \leftrightarrow \Sigma) \times \textit{Fragment}[\alpha, \pi, \Sigma] \rangle\!\rangle \\
\textit{Fragment}[\alpha, \pi, \Sigma] == \mathcal{P}\textit{Branch}[\alpha, \pi, \Sigma]
\end{array}$$

This definition makes use of an extension of the Z of the ZETA system, allowing generic free types. Note that the power operator used for fragments cannot be the general powerset in order to let the construction be consistent – with $\mathcal{P}$ we denote the "executable" power-sets.

Based on the tree encoding, we redefine the operations on fragments. The operator templates and constructors for interactions remain the same and are not repeated. Basic fragments are constructed as follows:

$$\begin{array}{|l}
\hline\hline
[\alpha, \pi, \Sigma] \\
\_ \diamond \_ / \_ == \lambda\,\textit{actor} : \alpha;\ \textit{action} : \pi;\ r : \Sigma \leftrightarrow \Sigma \bullet \\
\qquad\qquad \{\mathsf{br}(\textit{actor} :: \textit{action}, r, \{\mathsf{eod}\})\} \\
\hline
\end{array}$$

For the definition of sequential composition, we use a technique which is paradigmatic for the tree encoding of fragments: the composition is lazily "pushed" through the construction of the tree:

$$\begin{array}{|l}
\hline\hline
[\alpha, \pi, \Sigma] \\
\_ \curvearrowright \_ : \textit{Fragment}[\alpha, \pi, \Sigma] \times \textit{Fragment}[\alpha, \pi, \Sigma] \to \\
\qquad\qquad \textit{Fragment}[\alpha, \pi, \Sigma] \\
\hline
(\_ \curvearrowright \_) = \lambda f_1, f_2 : \textit{Fragment}[\alpha, \pi, \Sigma] \bullet \\
\quad (\mathbf{if}\ \mathsf{eod} \in f_1\ \mathbf{then}\ f_2\ \mathbf{else}\ \varnothing) \cup \\
\quad \{i : \textit{Interaction}[\alpha, \pi];\ r : \Sigma \leftrightarrow \Sigma \\
\quad\ f_1' : \textit{Fragment}[\alpha, \pi, \Sigma] \mid \mathsf{br}(i, r, f_1') \in f_1 \\
\quad\ \bullet \mathsf{br}(i, r, f_1' \curvearrowright f_2)\} \\
\hline
\end{array}$$

In the definition of the **repeat** operator, we embed the recursive expansion of the operator in a set comprehension:

$$\begin{array}{|l}
\hline\hline
[\alpha, \pi, \Sigma] \\
\mathbf{repeat} : \textit{Fragment}[\alpha, \pi, \Sigma] \to \textit{Fragment}[\alpha, \pi, \Sigma] \\
\hline
\mathbf{repeat} = \lambda f : \textit{Fragment}[\alpha, \pi, \Sigma] \bullet \\
\quad \{\mathsf{eod}\} \cup (f \curvearrowright \{b : \textit{Branch}[\alpha, \pi, \Sigma] \mid b \in \mathbf{repeat}\, f\}) \\
\hline
\end{array}$$

The definition of $f_1$ **except** $f_2$ uses similar techniques:

$$\begin{array}{|l}
\hline\hline
[\alpha, \pi, \Sigma] \\
\_\ \mathbf{except}\ \_ : \textit{Fragment}[\alpha, \pi, \Sigma] \times \textit{Fragment}[\alpha, \pi, \Sigma] \to \\
\qquad\qquad \textit{Fragment}[\alpha, \pi, \Sigma] \\
\hline
(\_\ \mathbf{except}\ \_) = \lambda f_1, f_2 : \textit{Fragment}[\alpha, \pi, \Sigma] \bullet \\
\quad (\mathbf{if}\ \mathsf{eod} \in f_1\ \mathbf{then}\ \{\mathsf{eod}\}\ \mathbf{else}\ \varnothing) \cup \\
\quad \{i : \textit{Interaction}[\alpha, \pi];\ r_1 : \Sigma \leftrightarrow \Sigma \\
\quad\ f_1' : \textit{Fragment}[\alpha, \pi, \Sigma] \\
\quad\ \mid \mathsf{br}(i, r_1, f_1') \in f_1 \bullet \mathsf{br}(i, r_1, f_1'\ \mathbf{except}\ f_2)\} \cup \\
\quad \{i : \textit{Interaction}[\alpha, \pi];\ r_1, r_2 : \Sigma \leftrightarrow \Sigma \\
\quad\ f_1', f_2' : \textit{Fragment}[\alpha, \pi, \Sigma] \\
\quad\ \mid \mathsf{br}(i, r_1, f_1') \in f_1;\ \mathsf{br}(i, r_2, f_2') \in f_2 \bullet \mathsf{br}(i, r_2, f_2')\} \\
\hline
\end{array}$$

The third case in the set union describes the actual interruption, where we continue with $f_2$, provided that there is an overlapping between a current interaction of $f_1$ and the first interaction of $f_2$.

The definition of the choice, **select**, is the same as in the model semantics ($\mathbf{select} = \bigcup$). Generalized union is executable by the definition $\bigcup SS = \{S : \mathbb{P}A;\ x : A \mid S \in SS;\ x \in S \bullet x\}$ as provided by the standard Z toolkit.

We finally define satisfaction: $(d, \sigma) \in_F f$ "parses" a dialogue and initial state by trying the branches of a fragment tree:

$$\begin{array}{|l}
\hline\hline
[\alpha, \pi, \Sigma] \\
\_ \in_F \_ : \mathbb{P}((\textit{Dialogue}[\alpha, \pi] \times \Sigma) \times \textit{Fragment}[\alpha, \pi, \Sigma]) \\
\hline
(\_ \in_F \_) = \\
\quad \{\sigma : \Sigma;\ f : \textit{Fragment}[\alpha, \pi, \Sigma] \mid \mathsf{eod} \in f \\
\quad\ \bullet (\langle\rangle, \sigma) \mapsto f\} \cup \\
\quad \{i : \textit{Interaction}[\alpha, \pi];\ d : \textit{Dialogue}[\alpha, \pi];\ \sigma, \sigma' : \Sigma \\
\quad\ f, f' : \textit{Fragment}[\alpha, \pi, \Sigma];\ r : \Sigma \leftrightarrow \Sigma \\
\quad\ \mid \mathsf{br}(i, r, f') \in f;\ (\sigma, \sigma') \in r;\ (d, \sigma') \in_F f' \\
\quad\ \bullet (\langle i \rangle \frown d, \sigma) \mapsto f\} \\
\hline
\end{array}$$

**Example: Testing The Cash Dispenser.** We can now test satisfaction of given dialogues regarding the cash dispenser's Use Cases (Spec. 1). Let some test dialogues be defined as follows:

**section** *CashDispenser*

$$\begin{aligned} d_1 == \langle & \mathit{dispenser} :: \mathit{askCard}, \mathit{user} :: \mathit{putCard},\\ & \mathit{dispenser} :: \mathit{askAmount}, \mathit{user} :: \mathit{putAmount}(400),\\ & \mathit{dispenser} :: \mathit{ejectCard}, \mathit{user} :: \mathit{takeCard},\\ & \mathit{dispenser} :: \mathit{ejectMoney}(400), \mathit{user} :: \mathit{takeMoney}\rangle \end{aligned}$$

$$\begin{aligned} d_2 == \langle & \mathit{dispenser} :: \mathit{askCard}, \mathit{user} :: \mathit{putCard},\\ & \mathit{dispenser} :: \mathit{ejectCard}, \mathit{user} :: \mathit{takeCard}\rangle \end{aligned}$$

$$\sigma_1 == \langle\!| \ \mathit{reserves} == 600 \ |\!\rangle; \ \sigma_2 == \langle\!| \ \mathit{reserves} == 800 \ |\!\rangle$$

Here are some query results:

$$\begin{array}{ll} (d_1, \sigma_1) \in_F \mathit{System} & \Rrightarrow \texttt{*true*}\\ (d_1 \frown d_2, \sigma_1) \in_F \mathit{System} & \Rrightarrow \texttt{*true*}\\ (d_1 \frown d_2 \frown d_1, \sigma_1) \in_F \mathit{System} & \Rrightarrow \texttt{*false*}\\ (d_1 \frown d_2 \frown d_1, \sigma_2) \in_F \mathit{System} & \Rrightarrow \texttt{*true*} \end{array}$$

In the third case, the reserves are too low to serve two subsequent requests of the amount of $400$ units. In the fourth case, the reserves are raised, such that the requests can be satisfied.

The efficiency of the execution of such queries scales to larger test-data input. Dialogues of length 1000 are processed in approximately 10 seconds on a Pentium-II/400 for the cash dispenser example. In general, efficiency depends on the kind of specification, and the amount of backtracking required to recognize a dialogue.

Thus the execution of the dispenser's Use Cases causes no problems. In general, given an input dialogue and initial system state, we can expect to execute a large subset of Use Case definitions in the presented style. Restrictions are the followings:

- For the state transition relations $r$, we must be able to *enumerate* solutions to $(\sigma, \sigma') \in r$. $r$ may be a true relation, and the solutions can be enumerated symbolically. However, *if* the enumeration happens to be infinite, our method is not complete, and since our implementation of Z uses depth-first search, not even semicomplete.
- The use of $\mathbf{select}\{x : A \bullet f\}$ in order to introduce local variables has some restrictions. We cannot write fragments of the kind

  $$\mathbf{select}\{x? : \mathbb{Z} \bullet A \diamond \mathit{get}(x? - 1) \curvearrowright B \diamond \mathit{put}(x? + 1)\}$$

  The reason is that our implementation of executable Z currently does not provide arithmetic constraints, and a term like $\mathit{get}(x? - 1)$ cannot be constructed until the variable $x?$ is bound (technically, the according concurrent constraint "residuates"). However, we may write

  $$\begin{aligned} &\mathbf{select}\{x?, x! : \mathbb{Z} \mid x? + 1 = x! - 1\\ &\qquad \bullet A \diamond \mathit{get}(x?) \curvearrowright B \diamond \mathit{put}(x!)\} \end{aligned}$$

  The resolution of the constraint in the choice is deferred until all necessary information is available, that is, $x?$ and $x!$ are bound.

## 4. Concurrency And Its Application

The model given in the previous sections is adequate for the loose description of systems like the cash dispenser where two or more actors communicate in a fixed order. It touches its limits, however, if the dialogues we want to describe consist of an *interleaving* of interactions of different *threads*.

As an example, consider the problem of describing an elevator system. In such a system, we have *n* users which interact with one elevator. When a user calls the elevator, until this request is served, other users may be served which are "on the way" of the elevator from its current floor to the first user's floor. Though we may model such a behavior by an according system state, this would not be in the spirit of Use Cases. Instead, we want to be able to use descriptive fragments of the kind $\mathit{user}\,n \diamond \mathit{call} \curvearrowright \mathit{elevator} \diamond \mathit{open\_door}$ – which describes the service offered to *some* user *n*, independent of services which might be provided at the same time (resp. interleaved with this service). This motivates the development of a simple model of concurrency, which is applied to the problem of an elevator system in this section.

**A Simple Model Of Concurrency.** To model concurrency, we conservatively extend our current encoding by a new operator for parallel composition. The definition uses the same "trick" as before, pushing the composition lazily through tree construction:

$$\mathbf{function}\ 45(\_ \parallel \_)$$

$$\begin{array}{l} [\alpha, \pi, \Sigma] \\ \hline \_ \parallel \_ : \mathit{Fragment}[\alpha, \pi, \Sigma] \times \mathit{Fragment}[\alpha, \pi, \Sigma] \rightarrow \\ \qquad \mathit{Fragment}[\alpha, \pi, \Sigma] \\ \hline (\_ \parallel \_) = \lambda f_1, f_2 : \mathit{Fragment}[\alpha, \pi, \Sigma] \bullet \\ \quad (\mathbf{if}\ \mathsf{eod} \in f_1 \mathbf{then}\ f_2 \\ \qquad\qquad \mathbf{else\,if}\ \mathsf{eod} \in f_2\ \mathbf{then}\ f_1\ \mathbf{else}\ \varnothing) \cup \\ \quad \{i : \mathit{Interaction}[\alpha, \pi];\ r_1, r_2 : \Sigma \leftrightarrow \Sigma \\ \quad\ f_1', f_2' : \mathit{Fragment}[\alpha, \pi, \Sigma] \\ \quad\ \mid \mathsf{br}(i, r_1, f_1') \in f_1;\ \mathsf{br}(i, r_2, f_2') \in f_2 \\ \quad\ \bullet \mathsf{br}(i, r_1 \cap r_2, f_1' \parallel f_2')\} \cup \\ \quad \{i : \mathit{Interaction}[\alpha, \pi];\ r_1 : \Sigma \leftrightarrow \Sigma \\ \quad\ f_1' : \mathit{Fragment}[\alpha, \pi, \Sigma] \\ \quad\ \mid \mathsf{br}(i, r_1, f_1') \in f_1 \bullet \mathsf{br}(i, r_1, f_1' \parallel f_2)\} \cup \\ \quad \{i : \mathit{Interaction}[\alpha, \pi];\ r_2 : \Sigma \leftrightarrow \Sigma \\ \quad\ f_2' : \mathit{Fragment}[\alpha, \pi, \Sigma] \\ \quad\ \mid \mathsf{br}(i, r_2, f_2') \in f_2 \bullet \mathsf{br}(i, r_2, f_1 \parallel f_2')\} \end{array}$$

Our parallel composition allows the synchronous as well as the interleaved combination of fragments. In the definition, this is realized by the four cases:

- eod is in one of the fragments $f_i$; then it is continued with the other fragment (interleaved composition)
- both fragments synchronously step on the same interaction $i$; the state transitions are joined as $r_1 \cap r_2$, and thus must be compatible (this is in difference to approaches which use *racing* to handle conflicts in synchronous transitions, e.g [2]).
- one of the fragments proceeds (interleaved composition)

**Example: Elevator System.** As an example applying the concurrency model, we define Use Cases for a (simplified) elevator system. The basic domains used are defined in Spec. 2. We represent *location* and *time* by natural numbers. The constant *MINOPENTIME* specifies the minimal time an elevator's door should be kept open. A *floor* is defined as an abstraction over the number of the floor. The function *floorLoc* associates a location with each floor. A *direction* can be either *up* or *down*.

**Specification 2** Basic Domains

$$\textbf{section}\ \mathit{Elevator}\ \textbf{parents}\ \mathit{UseCases}$$

$$\begin{array}{ll}
\mathit{LOCATION} & == \mathbb{N} \\
\mathit{TIME} & == \mathbb{N} \\
\mathit{MINOPENTIME} & == 20 \\
\mathit{MAXFLOOR} & == 4 \\
\mathit{FLOOR} & ::= \mathit{floor}\langle\!\langle 1 \mathrel{..} \mathit{MAXFLOOR} \rangle\!\rangle \\
\mathit{floorLoc} & == \lambda\, f : \mathit{FLOOR} \bullet (\mathit{floor}^{\sim})f * 3 \\
\mathit{DIR} & ::= \mathit{up} \mid \mathit{down}
\end{array}$$

**Specification 3** System State and Transitions

*State*

$$\begin{array}{|l}
\mathit{time} : \mathit{TIME};\ \mathit{location} : \mathit{LOCATION} \\
\mathit{requests} : \mathrm{seq}\, \mathit{FLOOR}
\end{array}$$

$$\uparrow == \lambda\, \mathit{Op} : \mathbb{P}(\Delta \mathit{State}) \bullet \{\mathit{Op} \bullet (\theta \mathit{State}, \theta \mathit{State}')\}$$

$$\begin{array}{|l}
\mathit{queueRequest} : \mathit{LOCATION} \times \mathrm{seq}\, \mathit{FLOOR} \times \\
\qquad\qquad \mathit{FLOOR} \times \mathit{DIR} \to \mathrm{seq}\, \mathit{FLOOR}
\end{array}$$

*AddRequest*

$$\begin{array}{|l}
\Delta \mathit{State};\ \Xi(\mathit{State} \setminus (\mathit{requests})) \\
\mathit{floor}? : \mathit{FLOOR};\ \mathit{dir}? : \mathit{DIR} \\
\hline
\mathit{requests}' = \mathit{queueRequest}(\mathit{location}, \mathit{requests}, \mathit{floor}?, \mathit{dir}?)
\end{array}$$

*RemoveRequest*

$$\begin{array}{|l}
\Delta \mathit{State};\ \Xi(\mathit{State} \setminus (\mathit{requests})) \\
\hline
\mathit{requests} \neq \langle\rangle;\ \mathit{location} = \mathit{floorLoc}(\mathit{head}\, \mathit{requests}) \\
\mathit{requests}' = \mathit{tail}\, \mathit{requests}
\end{array}$$

*Move*

$$\begin{array}{|l}
\Delta \mathit{State};\ \Xi(\mathit{State} \setminus (\mathit{location}));\ \mathit{target}? : \mathit{LOCATION} \\
\hline
\mathit{requests} \neq \langle\rangle \\
\exists\, \mathit{goal} == \mathit{floorLoc}(\mathit{head}\, \mathit{requests}) \bullet \\
\quad \mathit{target}? \neq \mathit{goal} \Rightarrow \\
\qquad \mathsf{abs}(\mathit{target}? - \mathit{goal}) < \mathsf{abs}(\mathit{location} - \mathit{goal}) \land \\
\qquad (\mathit{target}? > \mathit{location} \Rightarrow \mathit{goal} \notin \mathit{location} \mathrel{..} \mathit{target}?) \land \\
\qquad (\mathit{target}? < \mathit{location} \Rightarrow \mathit{goal} \notin \mathit{target}? \mathrel{..} \mathit{location}) \\
\mathit{location}' = \mathit{target}?
\end{array}$$

*Tick*

$$\begin{array}{|l}
\Delta \mathit{State};\ \Xi(\mathit{State} \setminus (\mathit{time})) \\
\mathit{duration}? : \mathit{TIME} \\
\hline
\mathit{time}' = \mathit{time} + \mathit{duration}?
\end{array}$$

Spec. 3 introduces the system state, *State*, and operations on it. The state is given by a time stamp, a location of the cabin and a queue of requests, containing floors in the order the cabin shall approach. We suppose this state to be visible to users of an elevator (for example, the location of the cabin can be visualized by lamps); hence the principle of observability is not violated.

The function $\mathit{queueRequest}(l, \mathit{reqs}, f, d)$ queues a request in the right order, given the situation that the elevator is at $l$ and is requested to serve $f$ in the direction $d$. We suppose this function inserts $f$ into the queue of requests in a "fair" way, serving requests "on the way" to *head reqs* if possible. The definition is left open in this presentation.

The operation *RemoveRequest* removes the next request; its precondition demands that there is actually a request, and that the cabin is at the floor of this request. The operation *Move* checks wether a change of the location of the cabin to *target*? confirms to the current request queue: if the request queue is empty, no change is allowed; if it is non empty, the cabin shall approach the first floor in sequence, and it shall not outrun a floor which is requested. Finally, the operation *Tick* describes a change of the time stamp.

**Specification 4** Actors and Actions of the Elevator

$$\begin{array}{lll}
\mathit{ACTOR} & ::= & \mathit{user}\langle\!\langle \mathbb{N} \rangle\!\rangle \mid \mathit{cabin} \mid \mathit{clock} \\
\mathit{ACTION} & ::= & \mathit{tick}\langle\!\langle \mathit{TIME} \rangle\!\rangle \\
 & \mid & \mathit{call}\langle\!\langle \mathit{FLOOR} \times \mathit{DIR} \rangle\!\rangle \\
 & \mid & \mathit{select}\langle\!\langle \mathit{FLOOR} \rangle\!\rangle \\
 & \mid & \mathit{moved}\langle\!\langle \mathit{LOCATION} \rangle\!\rangle \\
 & \mid & \mathit{opened} \mid \mathit{closed}
\end{array}$$

**Specification 5** User's View

$$\begin{array}{l}
\mathit{UserCalls}, \mathit{UserSelects}, \mathit{ElevatorServes}, \mathit{User} : \\
\quad \mathbb{N} \rightarrow \mathit{Fragment}[\mathit{ACTOR}, \mathit{ACTION}, \mathit{State}] \\
\hline
\mathit{UserCalls} = \lambda\, n : \mathbb{N} \bullet \\
\quad \mathbf{select}\{\mathit{floor}? : \mathit{FLOOR};\ \mathit{dir}? : \mathit{DIR};\ t : \mathit{TIME} \\
\qquad \bullet\ \mathit{user}\, n \diamond \mathit{call}(\mathit{floor}?, \mathit{dir}?) \,/ \\
\qquad\qquad \uparrow [\Delta \mathit{State} \mid \mathit{AddRequest}] \\
\qquad \curvearrowright \mathit{cabin} \diamond \mathit{opened} \,/ \\
\qquad\qquad \uparrow [\Delta \mathit{State} \mid \mathit{RemoveRequest};\ t = \mathit{time}] \\
\qquad \curvearrowright \mathit{cabin} \diamond \mathit{closed} \,/ \\
\qquad\qquad \uparrow [\Xi \mathit{State} \mid \mathit{time} - t \geq \mathit{MINOPENTIME}]\} \\
\mathit{UserSelects} = \lambda\, n : \mathbb{N} \bullet \\
\quad \mathbf{select}\{\mathit{floor}? : \mathit{FLOOR};\ \mathit{dir}? : \mathit{DIR};\ t : \mathit{TIME} \\
\qquad \bullet\ \mathit{user}\, n \diamond \mathit{select}\ \mathit{floor}? \,/ \\
\qquad\qquad \uparrow [\Delta \mathit{State} \mid \\
\qquad\qquad\quad \mathit{dir}? = \mathbf{if}\ \mathit{floorLoc}\ \mathit{floor}? < \mathit{location} \\
\qquad\qquad\qquad\qquad \mathbf{then}\ \mathit{down}\ \mathbf{else}\ \mathit{up} \\
\qquad\qquad\quad \mathit{AddRequest}] \\
\qquad \curvearrowright \mathit{cabin} \diamond \mathit{opened} \,/ \\
\qquad\qquad \uparrow [\Delta \mathit{State} \mid \mathit{RemoveRequest};\ t = \mathit{time}] \\
\qquad \curvearrowright \mathit{cabin} \diamond \mathit{closed} \,/ \\
\qquad\qquad \uparrow [\Xi \mathit{State} \mid \mathit{time} - t \geq \mathit{MINOPENTIME}]\} \\
\mathit{User} = \lambda\, n : \mathbb{N} \bullet \\
\quad \mathbf{repeat}(\mathbf{select}\{\mathit{UserCalls}\, n, \mathit{UserSelects}\, n\})
\end{array}$$

**Specification 6** Cabin's View

$$\begin{array}{l}
\mathit{Cabin} : \mathit{Fragment}[\mathit{ACTOR}, \mathit{ACTION}, \mathit{State}] \\
\hline
\mathit{Cabin} = \\
\quad \mathbf{repeat}(\mathbf{select}\{\mathit{target}? : \mathit{LOCATION} \\
\qquad \bullet \mathit{cabin} \diamond \mathit{moved}\ \mathit{target}? \,/ \uparrow [\Delta \mathit{State} \mid \mathit{Move}]\})
\end{array}$$

**Specification 7** Clock's View

$$\begin{array}{l}
\mathit{Clock} : \mathit{Fragment}[\mathit{ACTOR}, \mathit{ACTION}, \mathit{State}] \\
\hline
\mathit{Clock} = \\
\quad \mathbf{repeat}(\mathbf{select}\{\mathit{duration}? : \mathit{TIME} \\
\qquad \bullet \mathit{clock} \diamond \mathit{tick}\ \mathit{duration}? \,/ \uparrow [\Delta \mathit{State} \mid \mathit{Tick}]\})
\end{array}$$

Spec. 4 defines the actors and the actions of the elevator system. We have *n* users, the *cabin*, and the *clock*. The *clock* performs the *tick* action, the cabin moves to a location and opens or closes the door, and *user n* calls the cabin at a given floor for a certain direction, or selects a floor from inside the cabin.

The *user view* on the elevator system is defined by the fragments in Spec. 5. Each fragment is parameterized over the number *n* of a user; in the sequel we will instantiate these views in a parallel composition $\mathit{User}\,1 \parallel \mathit{User}\,2 \parallel \ldots$. A user repeatedly calls from a floor or selects a floor, and is then served by the elevator, which stops at the floor and keeps its door open for at least *MINOPENTIME*.

The *cabin view* is given in Spec. 6. It just describes how the cabin moves from target to target, using *Move* at each step to test if the move is valid and to update the location. Note that the interactions $\mathit{cabin} \diamond \mathit{opened}$ and $\mathit{cabin} \diamond \mathit{closed}$ belong to the user view, and not to the cabin view. The *clock view*, Spec. 7, finally defines how the clock repeatedly ticks, updating the time stamp in the system state.

Our overall model is given by a parallel composition

$$\mathit{User}\,1 \parallel \ldots \parallel \mathit{User}\,n \parallel \mathit{Cabin} \parallel \mathit{Clock}$$

where the interactions described by the individual fragments may appear in arbitrary interleaving or synchronously, provided there exists a valid system state transformation which fulfills this combination. Each of the fragments in the composition can be thought of an individual *thread*; communication between these threads is realized via the system state or by synchronous interactions. We support only a static number of such threads, and thus must know in advance how many users appear in a given dialogue before we can test for conformance of this dialogue to the use case specification.

We make some evaluation experiments. Let the following test data be given:

$$\begin{array}{l}
\sigma == (\mu[\mathit{State} \mid \mathit{time} = 0;\ \mathit{location} = 0;\ \mathit{requests} = \langle\rangle]) \\
d_1 == \lambda\, \mathit{duration} : \mathit{TIME} \bullet \\
\quad \langle \mathit{clock} :: \mathit{tick}\, 10, \mathit{user}\, 1 :: \mathit{call}(\mathit{floor}\, 2, \mathit{up}), \\
\quad\; \mathit{cabin} :: \mathit{moved}(\mathit{floorLoc}(\mathit{floor}\, 2)), \\
\quad\; \mathit{cabin} :: \mathit{opened}, \mathit{clock} :: \mathit{tick}\ \mathit{duration}, \mathit{cabin} :: \mathit{closed} \rangle
\end{array}$$

Test evaluation yields in:

$$\begin{array}{l}
(d_1\, 60, \sigma) \in_F \mathit{User}\, 1 \parallel \mathit{Cabin} \parallel \mathit{Clock} \Rrightarrow \texttt{*true*} \\
(d_1\, 10, \sigma) \in_F \mathit{User}\, 1 \parallel \mathit{Cabin} \parallel \mathit{Clock} \Rrightarrow \texttt{*false*}
\end{array}$$

In the second case, the time the door was kept open is to small.

The next set of test data describes the situation where a user which calls the cabin at a floor which is on the cabin's way is served in correct order ($d_2$) and in invalid order ($d_3$):

$$\begin{array}{l}
d_2 == \langle \mathit{user}\, 1 :: \mathit{call}(\mathit{floor}\, 3, \mathit{up}), \mathit{user}\, 2 :: \mathit{call}(\mathit{floor}\, 2, \mathit{up}), \\
\qquad\quad \mathit{cabin} :: \mathit{moved}(\mathit{floorLoc}(\mathit{floor}\, 2)), \\
\qquad\quad \mathit{cabin} :: \mathit{opened}, \mathit{clock} :: \mathit{tick}\, 40, \mathit{cabin} :: \mathit{closed}, \\
\qquad\quad \mathit{cabin} :: \mathit{moved}(\mathit{floorLoc}(\mathit{floor}\, 3)), \\
\qquad\quad \mathit{cabin} :: \mathit{opened}, \mathit{clock} :: \mathit{tick}\, 40, \mathit{cabin} :: \mathit{closed} \rangle
\end{array}$$

$$\begin{aligned} d_3 == \langle & \mathit{user}\,1 :: \mathit{call}(\mathit{floor}\,3, \mathit{up}), \mathit{user}\,2 :: \mathit{call}(\mathit{floor}\,2, \mathit{up}), \\ & \mathit{cabin} :: \mathit{moved}(\mathit{floorLoc}(\mathit{floor}\,3)), \\ & \mathit{cabin} :: \mathit{opened}, \mathit{clock} :: \mathit{tick}\,40, \mathit{cabin} :: \mathit{closed}, \\ & \mathit{cabin} :: \mathit{moved}(\mathit{floorLoc}(\mathit{floor}\,2)), \\ & \mathit{cabin} :: \mathit{opened}, \mathit{clock} :: \mathit{tick}\,40, \mathit{cabin} :: \mathit{closed} \rangle \end{aligned}$$

As to be expected, we get

$$\begin{aligned} (d_2, \sigma) \in_F \mathit{User}\,1 \parallel \mathit{User}\,2 \parallel \mathit{Cabin} \parallel \mathit{Clock} \Rrightarrow \texttt{*true*} \\ (d_3, \sigma) \in_F \mathit{User}\,1 \parallel \mathit{User}\,2 \parallel \mathit{Cabin} \parallel \mathit{Clock} \Rrightarrow \texttt{*false*} \end{aligned}$$

In the examples above, we had no synchronous interactions (where two parallel fragments consume the same interaction of a dialogue). The following test data describes a situation where two users are served at the same floor. In this case, the $\mathit{cabin} \diamond \mathit{opened} \ldots$ dialogue needs to be shared by these users:

$$\begin{aligned} d_4 == \langle & \mathit{user}\,1 :: \mathit{select}(\mathit{floor}\,3), \mathit{user}\,2 :: \mathit{call}(\mathit{floor}\,3, \mathit{up}), \\ & \mathit{cabin} :: \mathit{moved}(\mathit{floorLoc}(\mathit{floor}\,3)), \\ & \mathit{cabin} :: \mathit{opened}, \mathit{clock} :: \mathit{tick}\,40, \mathit{cabin} :: \mathit{closed} \rangle \end{aligned}$$

$$(d_4, \sigma) \in_F \mathit{User}\,1 \parallel \mathit{User}\,2 \parallel \mathit{Cabin} \parallel \mathit{Clock} \Rrightarrow \texttt{*true*}$$

## 5. Related Work and Conclusion

We have presented a setting which allows for the integration of Use Cases and Z in requirement specifications. The benefits of both approaches seem to be preserved, compensating the flaws of each other. For Use Cases, we do not find an exact semantics in the literature, which makes their instrumentation by tool support hard, and no standard way for describing system states, which is often required in real-world applications. Both are taken from Z in our integrated setting. The Z methodology for sequential systems, on the other hand, misses a way how to combine state transitions in specifications, and how to define I/O behavior. This is taken over from Use Cases to the world of Z.

In [3] a specification of Use Cases in Z has been given. The focus is on understanding Use Cases, not on instrumenting them for specification in combination with Z, as has been done in this paper.

The possibility to execute our integrated Use Case and Z specifications for the purpose of test evaluation shows the power of the implementation of executable Z [6]. This power is mainly achieved by the combination of *higher-orderness* (which supports suitable abstractions) with *concurrent constraint resolution*, which allows to suspend goals as long as enough information is available to resolve them. The computational setting is comparable to that of logic functional languages [7], but achieves its unique flavor by its set-orientation.

Our executable encoding of fragments by infinite trees, which are incrementally unrolled, is not only suited for Use Cases, but can be used to encode other kinds of positive trace logics. For example, we have applied a similar model to an encoding of the positive subset of discrete temporal interval logics. A disadvantage is, however, that the current implementation of Executable Z does not always preserve sharing and does not perform memorization. Future work on Executable Z thus aims at supporting these features. Assuming they would be present, the encoding by infinite trees is probably as efficient as the much harder to maintain representation by automatons.

The introduction of concurrency into Use Cases by a combination of interleaving and synchronicity is a promising approach to strengthen the power of this kind of specifications. However, further validation is required whether this approach scales up to larger examples, both from a methodological point of view as from the point of feasibility for execution. Regarding the last aspect, the complexity seems to be manageable as long as no deep backtracking becomes necessary; that is, the “right” interleaving is decided early in a branch.

On the meta-level of software engineering the experiment of joining an informal and a formal specification technique grants benefits to both sides. We experienced that notions and rules from the informal world are lifted to a new level of higher exactness as soon as a mathematical pendant is being constructed. On the other side the informal context requires a certain amount of flexibility and looseness, for which the formal techniques have to modify their expressivness accordingly and which can serve as a measure for feasibilty in future practice.

## References

[1] E. V. Berard. Be careful with ”use cases”. Technical report, The Object Agency, Inc., 1998. `http://www.toa.com/pub/use\_cases.htm`.

[2] R. Büssow and W. Grieskamp. A Modular Framework for the Integration of Heterogenous Notations and Tools. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proc. of the 1st Intl. Conference on Integrated Formal Methods – IFM'99*. Springer-Verlag, London, June 1999.

[3] G. Butler, P. Grogono, and F. Khende. A Z specification of use cases. In *Proc. of the Asia-Pacific Software Engineering Conference and International Computer Science Conference*, pages 505–506. IEEE Computer Society Press, 1997.

[4] D. Coleman. A use case template: draft for discussion, 1998. Hewlett-Packard Software Initiative.

[5] W. Grieskamp. *A Set-Based Calculus and its Implementation*. PhD thesis, Technische Universität Berlin, 1999.

[6] W. Grieskamp. A Computation Model for Z based on Concurrent Constraint Resolution. To appear in ZB2000 – International Conference of Z and B Users, September 2000.

[7] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19(20), 1994.

[8] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[9] Uml semantics version 1.3. `http://www.rational.com/uml/index.jtmpl`.

[10] Drafts for the Z ISO standard. Ian Toyn (editor). URL: `http://www.cs.york.ac.uk/~ian/zstan`, 1999.