Encoding Temporal Logics in Executable Z: A Case Study for the ZETA System

Wolfgang Grieskamp and Markus Lepper

Technische Universität Berlin, FB13, Institut für Kommunikations- und Softwaretechnik, Sekr. 5–13, Franklinstr. 28/29, D–10587 Berlin, E-mail: {wg,lepper}@cs.tu-berlin.de

Abstract. The ZETA system is a Z-based tool environment for developing formal specifications. It contains a component for *executing* the Z language based on the implementation technique of *concurrent constraint resolution*. In this paper, we present a case-study for the environment, by providing an executable encoding of *temporal interval logics* in the Z language. As an application of this setting, test-case evaluation of trace-producing systems on the base of a formal requirements specifications is envisaged.

1 Introduction

The ZETA system [2] is a tool environment for developing formal specifications based on the Z notation [10]. It contains a component for *executing* the Z language, using a computation model of *concurrent constraint resolution*, described in [5]. A wide range of Z's logic can be executed within this approach, integrating the power of higher-order functional and logic computation.

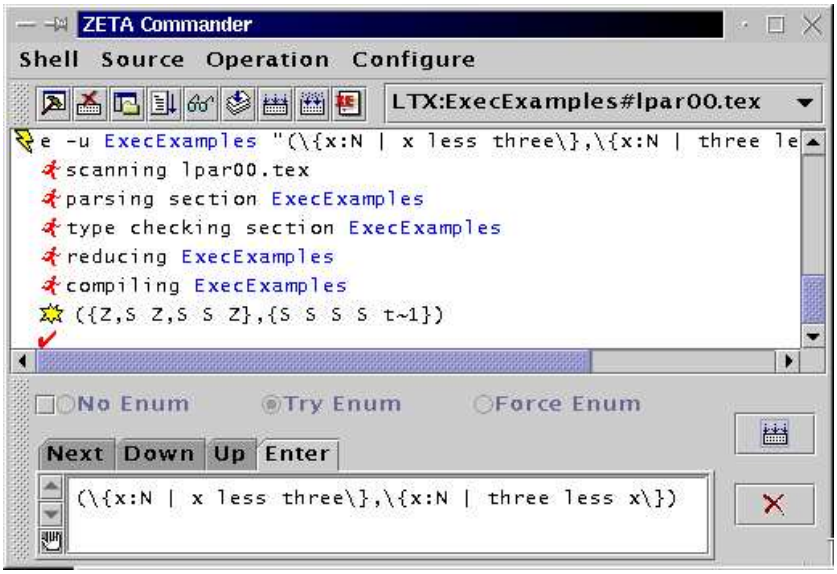
In this paper, we present a case study of the system. We develop an executable encoding of *discrete temporal interval logics* (in the style of Moszkowski's logic, [8]), and illustrate it by animation in the ZETA system. The example demonstrates the interplay of logical search and of higher-orderness, the last one allowing us to build abstractions by passing predicates (generally represented as sets in Z) to functions and storing them in data values.

As an application of our encoding of temporal logics we briefly look at *test-case* evaluation for *safety-critical* embedded systems. Given a formal requirements specification which uses temporal logics, 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 specification whether the implementation meets its requirements. This application stems from the context of a research project funded by Daimler-Chrysler.

This paper is organized as follows. In Sec. 2, we introduce the basic features of executing Z in the ZETA system. In Sec. 3 we develop the encoding of temporal logics, and describe the application to test-case evaluation, where we use the example of an *elevator controller*. In Sec. 4 we give a conclusion, discussing the results and related work.

ISBN 3-540-41286-9

Fig. 1 ZETA's Graphical User Interface After Executing an Expression



2 Executing Z under ZETA

In [4, 5] 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 a component of the ZETA system. In this implementation, all idioms of Z which are related to *functional* and *logic* programming languages are executable. Below, we look at some examples to illustrate the basic features. We assume some knowledge of Z (see e.g. [10]; the Z implemented by ZETA actually confirms to the forthcoming Z ISO Standard [14], which, however, does not make a significant difference in our application).

As sets are paradigmatic for the specification level of Z, they are for the execution level. Set objects – relations or functions – are eventually defined by (recursive) equations, as in the following example, where we define natural numbers as a free type, addition on these numbers and an order relation:

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

$$\mid add : \mathbb{P}((N \times N) \times N)$$

$$\mid add = \{y : N \bullet (Z, y) \mapsto y\} \cup \{x, y, z : N \mid (x, y) \mapsto z \in add \bullet (S\,x, y) \mapsto S\,z\}$$

$$\mid \_less\_ == \{x, y : N \mid (\exists\, t : N \bullet (x, S\,t) \mapsto y \in add)\}$$

A few remarks on the syntax. With ::= a free type is introduced in Z. The declaration form $n == E$ declares and defines a (non-recursive) name simultaneously. The form $x \mapsto y$ is just an alternative notation for (x, y). A set-comprehension in Z, $\{x : T \mid P \bullet E\}$, describes the values E such that P holds

ISBN 3-540-41285-9

for the possible assignments of x; if the $\bullet$ is omitted (as in the definition for *less*, where the $\bullet$ actually belongs to the existential quantor), the set of tuples of the assignments to the variables in the declaration part is denoted (thus $\{x, y : T \mid P\} = \{x, y : T \mid P \bullet (x, y)\}$).

We may now execute under ZETA queries such as the following, where we ask for the pair of sets less and greater than *three*:

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

$\longrightarrow$ `({Z,S(Z),S(S(Z))},{S(S(S(S(t~))))})`

The query as it is entered into the ZETA GUI is visualized in Fig. 1. In the sequel, however, we will use a conceptual notation as above.

As the result of the query, we get the pair of the numbers less than and greater than *three*, where the second value of the resulting pair is a singleton set containing the free variable `t~` (the `~` results from internal variable renaming). These capabilities are obviously 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. A binary relation R can be *applied*, written as $R\,e$, which is syntactic sugar for the expression $\mu\, y : X \mid (e, y) \in R$. This expression is defined iff their exists a unique y such that the constraint is satisfied; it then delivers this y. The set *add* is a binary relation (since it is member of the set $\mathbb{P}((N \times N) \times N)$), and therefore we can for example evaluate $\mathit{add}(\mathit{three}, \mathit{three}) \Rrightarrow$ `S(S(S(S(S(S(Z))))))`.

Note the semantic difference of $(e, y) \in R$ and $y = R\,e$: the first is not satisfied if R is not defined at e, or produces several solutions for y if R is not unique at e, whereas the second is *undefined* in these cases. This difference is resembled in the implementation: application, μ-expressions, and related forms are realized by *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 $\{x, y, z : N \mid z = \mathit{add}(x, y) \bullet (S\,x, y) \mapsto S\,z\}$ (instead of using $(x, y) \mapsto z \in \mathit{add}$), backwards computation is not be possible:

$\{x : N \mid x \mathit{\ less\ three}\}$

$\longrightarrow$ `unresolved constraints:`
`LTX:cpinz(48.24-48.31) waiting for variable x`

Here, the encapsulated search for $\mathit{add}(x, y)$, solving $\mu\, z : N \mid ((x, y), z) \in \mathit{add}$, cannot continue, since it is not allowed to produce bindings for the context variables x and y.

The elegance of the functional paradigm comes from the fact that functions are first-class citizens. In our implementation of execution for Z, sets are first-class citizens as well. For example, we define a function describing relational image as follows:

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

A query for the relational image of the *add* function over the cartesian product of the numbers less then three yields in:

$$\textbf{let}\ ns == \{x : N \mid (x, three) \in less\} \bullet add(\!| ns \times ns |\!)$$
$$\longrightarrow \texttt{\{Z,S(Z),S(S(Z)),S(S(S(Z))),S(S(S(S(Z))))\}}$$

Universal quantification is executable if it deals with finite ranges. For example, we can define the operator denoting the set of partial functions in Z, $A \nrightarrow B$, as follows:

$$[X, Y]$$
$$\_ \nrightarrow \_ == \{R : \mathbb{P}(X \times Y) \mid (\forall x : X \mid x \in \mathsf{dom}\, R \bullet \exists_1 y : Y \bullet (x, y) \in R)\}$$

Universal and unique existential quantification are resolved by enumeration. Thus, if we try to check whether *add* is a partial function, we get in a few seconds:

$$add \in N \times N \nrightarrow N$$
$$\longrightarrow \texttt{still searching after 200000 steps}$$
$$\texttt{gc \#1 reclaimed 28674k of 32770k}$$
$$\texttt{...}$$

In enumerating the domain of *add* our computation diverges. However, if we restrict *add* to a finite domain it works:

$$\exists_1 ns == \{x : N \mid (x, three) \in less\} \bullet ((ns \times ns) \lhd add \in N \times N \nrightarrow N)$$
$$\longrightarrow \texttt{*true*}$$

Above, $A \lhd R$ restricts the domain of R to the set A; the existential quantor is used to introduce a local name in the predicate.

3 Encoding of Temporal Interval Logics

Temporal interval logics [8, 3] is a powerful tool for describing requirements on traces of the behavior of real-time systems. For a discrete version of this logic, related to Moszkowski's version of ITL, an embedding into Z has been described

in [1]. Here, we develop an executable shallow encoding for the positive subset of this kind of ITL. The encoding supports resolution for timing and observation constraints (going behind Moszkowski's Tempura implementation), demonstrating some of the capabilities of Executable Z in the ZETA system.

3.1 The Encoding

We define temporal formulas generic over a state type Σ, such that the behaviors we look at have type $\mathrm{seq}\,\Sigma$ ($\mathrm{seq}\_$ is Z's type constructor for sequences). A *predicate* over a state binding is a unary relation, $p \in SP[\Sigma] = \mathbb{P}\,\Sigma$:

$$SP[\Sigma] == \mathbb{P}\,\Sigma$$

A *temporal formula* is encoded by a set of so-called arcs, $w \in TF[\Sigma] = \mathcal{P}ARC[\Sigma]$[1], which basically model a transition relation. An arc is either a proper transition, $tr(p, w)$, where p is the guard for this transition and w a followup formula, or the special arc *eot* which indicates that an interval which satisfies this formula may end at this point:

$$TF[\Sigma] == \mathcal{P}ARC[\Sigma] \qquad ARC[\Sigma] ::= eot \mid tr\langle\!\langle SP[\Sigma] \times TF[\Sigma] \rangle\!\rangle$$

$xs \in_T w$ is the satisfaction relation of this encoding of temporal formulas, and is defined as follows:

$$\begin{array}{|l}
\hline
[\Sigma] \\
_ \in_T _ : \mathrm{seq}\,\Sigma \leftrightarrow TF[\Sigma] \\
\hline
(_ \in_T _) = \{w : TF[\Sigma] \mid eot \in w \bullet (\langle\rangle, w)\} \cup \\
\qquad \{x : \Sigma;\ xs : \mathrm{seq}\,\Sigma;\ p : SP[\Sigma];\ w, w' : TF[\Sigma] \mid \\
\qquad\qquad tr(p, w') \in w;\ x \in p;\ xs \in_T w' \bullet (\langle x \rangle \frown xs, w)\} \\
\hline
\end{array}$$

Thus, if *eot* is an arc of the transition relation, then the empty interval is valid. Moreover, all intervals are valid such that their exists a transition whose predicate fulfills the head of the interval, and the tail of the interval satisfies the followup formula of this transition.

We know define the operators of our logic, which construct values of type $TF[\Sigma]$. The formula which is satisfied exactly by the empty trace is encoded by the singleton transition containing the *eot* arc. The formula $\uparrow p$ lifts a state predicate p to an interval formula which holds exactly for those intervals of length 1 containing a state which satisfies p:

[1] We use the powerset-constructor $\mathcal{P}$ which models a computable powerset *domain*. Using the general power, $\mathbb{P}$, our free type definition of ARC would be inconsistent in Z, since a free type's constructor cannot have a general powerset of the type in its domain.

ISBN 3-540-41285-9

$$[\Sigma]$$
$$\mathsf{empty} == \{eot[\Sigma]\}$$

$$[\Sigma]$$
$$\uparrow == \lambda\, p : SP[\Sigma] \bullet \{tr(p, \mathsf{empty})\}$$

Next we look at disjunction, $w_1 \sqcup w_2$, and its generalized form. Disjunction is realized by simply mapping it to the union of the arc sets of both formulas:

$$[\Sigma]$$
$$\_ \sqcup \_ == \lambda\, w_1, w_2 : TF[\Sigma] \bullet w_1 \cup w_2$$

$$[\Sigma]$$
$$\bigsqcup == \lambda\, ws : \mathbb{P}\, TF[\Sigma] \bullet \bigcup ws$$

Beware that the generalized disjunction operator can be used for introducing "local variables", as on $\bigsqcup\{x : T \bullet TF[x]\}$.

Conjunction, $w_1 \sqcap w_2$, constructs new arcs by pairwise combination of all arcs of w_1 and w_2 – the conjunction is recursively "pushed" through these combinations:

$$[\Sigma]$$
$$\_ \sqcap \_ : TF[\Sigma] \times TF[\Sigma] \to TF[\Sigma]$$
$$(\_ \sqcap \_) = \lambda\, w_1, w_2 : TF[\Sigma] \bullet$$
$$(\textbf{if } eot \in w_1 \wedge eot \in w_2 \textbf{ then } \mathsf{empty} \textbf{ else } \varnothing) \sqcup$$
$$\{p_1, p_2 : SP[\Sigma];\ w_1', w_2' : TF[\Sigma]$$
$$\mid tr(p_1, w_1') \in w_1;\ tr(p_2, w_2') \in w_2 \bullet tr(p_1 \cap p_2, w_1' \sqcap w_2')\}$$

$w_1 \fatsemi w_2$ is sequential composition ("chop"). The followup-formula w_2 is recursively pushed through the arcs of w_1 until *eot* is reached:

$$[\Sigma]$$
$$\_ \fatsemi \_ : TF[\Sigma] \times TF[\Sigma] \to TF[\Sigma]$$
$$(\_ \fatsemi \_) = \lambda\, w_1, w_2 : TF[\Sigma] \bullet$$
$$(\textbf{if } eot \in w_1 \textbf{ then } w_2 \textbf{ else } \varnothing) \sqcup$$
$$\{p : SP[\Sigma];\ w_1' : TF[\Sigma] \mid tr(p, w_1') \in w_1 \bullet tr(p, w_1' \fatsemi w_2)\}$$

w^* is the repetition of w for zero or more times, w^+ for one or more times. In the definition of $\_^*$, we need to embed the recursive reference to $\_^*$ in a set-comprehension, since our implementation of Z imposes a *strict* (eager) evaluation order. The formula skip holds for arbitrary singleton intervals. Temporal truthness, satisfied by any interval, is the repetition of skip. Temporal falsity is described by the empty set of arcs:

$$[\Sigma]$$

$$\_^* : TF[\Sigma] \to TF[\Sigma]$$

$$(\_^*) = \lambda\, w : TF[\Sigma] \bullet \mathsf{empty} \sqcup ((w \setminus \mathsf{empty}) \fatsemi \{a : ARC[\Sigma] \mid a \in w^*\})$$

$$[\Sigma]$$

$$\_^+ == \lambda\, w : TF[\Sigma] \bullet w \fatsemi w^*$$

$$[\Sigma]$$

$$\mathsf{skip} == \uparrow \Sigma$$

$$[\Sigma]$$

$$\mathsf{true} == \mathsf{skip}[\Sigma]^*$$

$$[\Sigma]$$

$$\mathsf{false} == \varnothing[ARC[\Sigma]]$$

We animate the encoding of some formulas. Suppose type Σ is instantiated with $\mathbb{Z}$. Recall that our observation predicates are sets, hence we can use e.g. $\{1\}$ as a predicate which is exactly true for the state value 1:

$$\begin{array}{ll} \uparrow\{1\} \fatsemi \mathsf{empty} \fatsemi \uparrow\{2\} \fatsemi \mathsf{empty} & \Rrightarrow \texttt{\{tr(\{1\},\{tr(\{2\},\{eot\})\})\}} \\ \uparrow\{1\}^* & \Rrightarrow \texttt{\{eot,tr(\{1\},\{eot,tr(\{1\},...)\})\}} \\ \uparrow\{2,3\}^* \sqcap (\uparrow\{1,2\} \fatsemi \uparrow\{3,4\}) & \Rrightarrow \texttt{\{tr(\{2\},\{tr(\{3\},\{eot\})\})\}} \end{array}$$

The first example shows neutrality of empty on chop. The next example illustrates how the repetition operator incrementally "unrolls" its operand (the ZETA displayer has stopped unrolling after a certain depth). In the last example, the effect of conjunction is shown.

Using the satisfaction relation $t \in_T w$, we can now test whether a trace t fulfills a formula w and – provided the state predicates are finite – also generate the set of traces which satisfy a formula. Here are some examples

$$\begin{array}{ll} \langle 1,2,3,1,2,1\rangle \in_T (\mathsf{true} \fatsemi \uparrow\{x : \mathbb{Z} \mid x \geq 2\}^+)^* & \Rrightarrow \texttt{*false*} \\ \langle 2,2,2,1,2,2\rangle \in_T (\mathsf{true} \fatsemi \uparrow\{x : \mathbb{Z} \mid x \geq 2\}^+)^* & \Rrightarrow \texttt{*true*} \\ \{t : \mathrm{seq}\,\mathbb{Z} \mid t \in_T \uparrow\{1,2\}^+\} & \Rrightarrow \texttt{\{<1>,<2>,<1,1>,<1,2>,...\}} \end{array}$$

In the first two examples above, the formula states that the interval must be partitionable into zero or more sub-interval such that in each sub-interval, from some point only numbers greater or equal two appear. This is not satisfied by the first trace, but by the second, choosing the right partitioning. The third example shows the generation of traces.

Our encoding allows the use of free variables in state predicates. For example, we can define a formula which is satisfied by all traces which contain adjacent values. The variable can be existential quantified, or as in the example below, bound by a set comprehension to enumerate its possible bindings:

$$\{x : \mathbb{Z} \mid \langle 4,1,1,3,2,2\rangle \in_T \mathsf{true} \fatsemi \uparrow\{x\} \fatsemi \uparrow\{x\} \fatsemi \mathsf{true}\} \Rrightarrow \texttt{\{1,2\}}$$

We will use this feature in the next section in order to introduce timing constraints.

ISBN 3-540-41285-9

3.2 Timing Constraints

Due to the higher-orderness of Z and our implementation, it is easily possible to add new temporal operators. Suppose that our state type Σ contains a duration stamp describing the time distance to the next observation[2], and that this stamp is selected by the function $getd : \Sigma \to \mathcal{T}$. We then can define a duration operator $DUR(getd, d)$ which holds for those intervals whose duration is d[3]:

$$\mathcal{T} == \mathbb{Z}$$

$$[\Sigma]$$

$$DUR : (\Sigma \to \mathcal{T}) \times \mathcal{T} \to TF[\Sigma]$$

$$\begin{aligned} DUR = \lambda\, getd : \Sigma \to \mathcal{T};\ d : \mathcal{T} \bullet \\ \bigsqcup\{\sigma : \Sigma \mid getd\, \sigma = d \bullet \uparrow\{\sigma\}\} \sqcup \\ \bigsqcup\{\sigma : \Sigma;\ d' : \mathcal{T} \mid d = getd\, \sigma + d' \bullet \uparrow\{\sigma\} \fatsemi DUR(getd, d')\} \end{aligned}$$

This definition makes use of the "generalized disjunction" for temporal formulas, $\bigsqcup$ (see 3.1), to introduce local variables σ and d'. In general, the set-comprehension $\{x : T \mid P \bullet w\}$, where w is a temporal formula, denotes the set of all formulas for instances of x which satisfy P. Since a temporal formula is a set of arcs, the generalized disjunction simply collects all arcs from all formulas, by its definition $\bigsqcup = \bigcup$. The name $\bigcup$ is in turn defined in the Z standard library as $\bigcup SS = \{x : X;\ S : \mathbb{P}\, X \mid S \in SS;\ x \in S \bullet x\}$. Our implementation enumerates the solutions to $S \in SS$ symbolically; henceforth $\bigcup$ also works if SS is not finite, as in the definition of DUR.

In the definition of $DUR(getd, d)$ two cases are distinguished. Either the interval contains exactly one state with duration d, or d is the result of adding $getd\, \sigma$ of the heading state and d' for the remaining states.

As an example, we calculate the partitions of an interval with equal duration, using repetition on the duration operator (where our state contains only durations, and the identity function id selects them):

$$\{d : \mathcal{T} \mid \langle 1, 1, 2, 2, 2\rangle \in_T DUR(\mathsf{id}, d)^*\}$$
$$\longrightarrow \texttt{\{2,4,8\}}$$

Note that the partitionings are not of equal length regarding the number of states in an interval. For the duration 2, we use $\langle 1, 1\rangle$ and the remaining three $\langle 2\rangle$ partitions. For the duration 4, we have $\langle 1, 1, 2\rangle$ and $\langle 2, 2\rangle$. For duration 8, one partition containing all states is recognized.

3.3 Application

Fig. 2 gives a very simplified example how to apply our temporal logics for requirements specification. The specification defines some aspects of the behavior

[2] Currently, in our implementation of Z only integral numbers are supported – hence we define time as integral numbers.

[3] Beware that we do not support an "overlapping chop"; therefore intervals which limits fall between two data samples of the given behavior are never considered.

ISBN 3-540-41285-9

Fig. 2 Elevator Requirements

$$POS == \mathbb{N} \qquad FLOOR == \{0, 20, 40, 60, 80\}$$

$$delay == 15$$

$$STATE$$

$$dur : \mathcal{T};\ pos : POS;\ open, request : \mathbb{P}\, FLOOR$$

$$\begin{array}{lcl}
getd & == & \lambda\, \sigma : STATE \bullet \sigma.dur \\
Safety & == & \uparrow[STATE \mid \forall f : FLOOR \mid f \in open \bullet pos = f]^* \\
Serve & == & \lambda f : FLOOR \bullet \\
 & & \quad \uparrow[STATE \mid f \notin request] \sqcup \\
 & & \quad (\uparrow[STATE \mid f \in request \wedge f \neq pos]^+_9 \\
 & & \quad \bigsqcup\{d : \mathcal{T} \mid d < delay \bullet \uparrow[STATE \mid f = pos]^* \sqcap DUR(getd, d)\}^\circ_9 \\
 & & \quad \uparrow[STATE \mid f \in open]^+) \\
Liveness & == & Serve(0)^* \sqcap Serve(20)^* \sqcap Serve(40)^* \sqcap Serve(60)^* \sqcap Serve(80)^* \\
Reqs & == & Safety \sqcap Liveness
\end{array}$$

of a (much simplified) elevator controller. The elevator's state is modeled by a set of sensors which are combined with a duration stamp into the system state *STATE*. The sensors are the current position of the elevator and two sets which represent the state of doors at each floor and of request buttons. Floors are modeled as a subset of positions.

Our requirements are composed from the conjunction of sub-requirements:

- *Saftey*: a door must be only open if the elevator is at the floor of the door.
- *Serve*: describing the service requirements for a given floor f: Either the floor is not requested, or – if the elevator is requested at this floor – the elevator can be anywhere else. But as soon as it reaches the floor, it must stop there and open the door at least after *delay* seconds. (The specification does not handle error situations, where the elevator does not work for some reason.)
- *Liveness*: is simply the conjunction of all service requirements for all floors.

Such a specification can now be used for test-evaluation, feeding it with the concrete traces produced by an implementation of the controller. For example, let some test traces (parameterized over a duration stamp) be defined as follows:

$$\begin{array}{lcl}
t_1 == \lambda\, d : \mathcal{T} \bullet & \langle\!| & dur == d, pos == 0, open == \varnothing, request == \{20\} \,|\!\rangle, \\
 & & |\!| \ dur == d, pos == 20, open == \varnothing, request == \{20\} \,|\!\rangle, \\
 & & |\!| \ dur == d, pos == 20, open == \{20\}, request == \varnothing \,|\!\rangle\rangle \\
t_2 == \lambda\, d : \mathcal{T} \bullet & \langle\!| & dur == d, pos == 0, open == \varnothing[\mathbb{Z}], request == \{20\} \,|\!\rangle, \\
 & & |\!| \ dur == d, pos == 20, open == \varnothing, request == \{20\} \,|\!\rangle, \\
 & & |\!| \ dur == d, pos == 40, open == \varnothing, request == \{20\} \,|\!\rangle\rangle
\end{array}$$

Here are some evaluation results:

$$t_1\ 10 \in_T Reqs \Rrightarrow \texttt{*true*};\ t_1\ 40 \in_T Reqs \Rrightarrow \texttt{*false*};\ t_2\ 10 \in_T Reqs \Rrightarrow \texttt{*false*}$$

ISBN 3-540-41285-9

In the second case, the elevator stopped at the requested floor but did not opened the door in time. In the third case, the elevator passed a requested floor without stopping.

The performance of test-evaluation highly depends on the kind of specification. For the above specification we check traces of around thousand elements in approx. 30 seconds. However, it is possible to formulate specifications which are intractable to execution since deep backtracking is required to recognize traces. These specifications involve constructs such as $(\mathsf{true}^{\circ}_{9} decision_1) \sqcup (\mathsf{true}^{\circ}_{9} decision_2)$.

ISBN 3-540-41285-9

4 Conclusion and Related Work

We have presented a case study of the ZETA system, a practical, working setting for developing specifications based on the Z language, which allows for executing a subset of Z based on concurrent constraint resolution. The example of encoding temporal interval logics showed that higher-orderness is a key feature for an environment where we can add new abstractions and notations in a convenient and consistent way: in that temporal formulas are first-class citizens, we could define the operators of the logic as functions over formulas. Below, we discuss some further aspects.

Animating Z. Animation of the "imperative" part of Z is provided by the ZANS tool [7], imperative meaning Z's specification style for sequential systems using state transition schemas. This approach is highly restricted. An elaborated *functional approach* for executing Z has been described in [11], though no implementation exists today, and logic resolution is not employed. Other approaches are based on a mapping to Prolog (e.g. [12, 13]), but do not support higher-orderness. The approach presented in this paper goes beyond all the others, since it allows the combination of the functional and logic aspects of Z in a higher-order setting.

Functional and Logic Programming Languages. There is a close relationship of our setting to functional logic languages such as Curry [6] or Oz [9]: in these languages it is possible to write functions which return constraints, enabling abstractions as have been used in this paper. However, our setting provides a tighter integration and has a richer predicate language as f.i. Curry, including negation and universal quantification which are treated by encapsulated search. The role of a function as a special kind of relation as a special kind of set, and of application $e\,e'$ just as an abbreviation for $\mu\, y \mid (e', y) \in e$, makes this tight integration possible.

Integrating Specific Resolution Techniques. Currently, our implementation is not very ambitious regarding the basic employed resolution techniques. Central to the computation model is not the basic solver technology (which is currently mere term unification) but the management of abstractions of constraints via sets. However, the integration of specialized solvers for arithmetic, interval and temporal constraints is required for our application to test-evaluation. The extension of the model to an architecture of cooperating basic solvers is therefore subject of future work.

ISBN 3-540-41285-9

References

1. R. Büssow and W. Grieskamp. Combinig Z and temporal interval logics for the formalization of properties and behaviors of embedded systems. In R. K. Shyamasundar and K. Ueda, editors, *Advances in Computing Science – Asian '97*, volume 1345 of *LNCS*, pages 46–56. Springer-Verlag, 1997.
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. Z. Chaochen, C. A. R. Hoare, and A. Ravn. A calculus of durations. *Information Processing Letters*, 40(5), 1991.
4. W. Grieskamp. *A Set-Based Calculus and its Implementation*. PhD thesis, Technische Universität Berlin, 1999.
5. 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.
6. M. Hanus. Curry – an integrated functional logic language. Technical report, Internet, 1999. Language report version 0.5.
7. X. Jia. An approach to animating Z specifications. Internet: `http://saturn.cs.depaul.edu/~fm/zans.html`, 1996.
8. B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, 1986. updated version from the authors home page.
9. G. Smolka. Concurrent constraint programming based on functional programming. In C. Hankin, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, vol. 1381, pages 1–11, Lisbon, Portugal, 1998. Springer-Verlag.
10. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
11. S. Valentine. The programming language Z^{--}. *Information and Software Technology*, 37(5–6):293–301, May–June 1995.
12. M. M. West and B. M. Eaglestone. Software development: Two approaches to animation of Z specifications using Prolog. *IEE/BCS Software Engineering Journal*, 7(4):264–276, July 1992.
13. M. Winikoff, P. Dart, and E. Kazmierczak. Rapid prototyping using formal specifications. In *Proceedings of the Australasian Computer Science Conference*, 1998.
14. Drafts for the Z ISO standard. Ian Toyn (editor). URL: `http://www.cs.york.ac.uk/~ian/zstan`, 1999.