Constraint Programming in Z

Wolfgang Grieskamp, Markus Lepper, and Jacob Wieland

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

Abstract. We present a setting for "programming" in the set-based specification language Z which is based on concurrent constraint resolution. The setting is implemented as part of the tool integration environment ZETA where it is used for executing specifications for the purpose of test-data evaluation and simulation.

1 Introduction

The automatic evaluation of *test cases* for *safety-critical systems* is an interesting application that can help to put formal methods into industrial practice. Some studies report that more than 50% of development costs in this application area go into testing. A setting for test-case evaluation that can improve this situation is as follows: given a requirements specification, 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.

In this paper, we present a setting for "programming" in the set-based specification language Z [13] which is based on concurrent constraint resolution as described in [5]. The setting is implemented as part of the tool integration environment ZETA [3] where it is used for executing specifications for the purpose of test-case evaluation and simulation. Constraints are represented as finite or infinite sets, and sets are first-order citizens – a key feature of the approach. As an example showing the capabilities, a shallow encoding of discrete temporal interval logics (in the style of Moszkowski's logic [10]) is given – an application which stems from a research project funded by Daimler-Chrysler for the purpose of test-case evaluation for embedded systems.

The paper starts with a short introduction into Z (Sec. 2). We then explore the basics of constraint programming in Z (Sec. 3). In Sec. 4, the encoding of temporal interval logics is defined and illustrated by examples. Related work is discussed in the conclusion. In the appendix, a complete formal definition of the computation model of Z is given for the interested reader. The model is based on reducing Z to a small intermediate calculus and defining a natural semantics for the calculus.

2 A Sketch of Z

Z is a relatively wide-spread formal notation whose ISO standard is currently forthcoming [18]. Though Z's usual application domain is for high-level requirements specification, its orientation on *sets* makes it well suited for expressing constraint problems. We give a short introduction into the basic design of the language, and refer for further information for instance to [13].

The semantic model of Z is based on a *typed set theory*. Given a collection of so-called *given types*, Z types are constructed by *power set*, $\mathbb{P}\,\tau$, *cartesian product*, $\tau_1 \times \ldots \times \tau_n$, and so-called *schemas*, $[x_1 : \tau_1;\ \ldots;\ x_n : \tau_n]$. Schemas denote a set of *bindings*, which are tuples with named components.

A distinguishing feature of Z is that types are "first-order citizens" of the language. A type is also an expression which denotes a set. As a set, it just has the property of being the *largest* such one containing elements of the same type. For example, the given type $\mathbb{Z}$ is the largest set containing (integral) numbers; the set of natural numbers, $\mathbb{N}$, is a subset of these. We can use both $\mathbb{Z}$ and $\mathbb{N}$ in declarations such as $x : \mathbb{Z}$, $y : \mathbb{N}$, or in expressions like $\mathbb{P}\,\mathbb{N}$ or $\mathbb{Z} \times \mathbb{N}$.

A central constituting language element of Z is so-called *schema text*, which denotes a constrained binding set. Schema text, $D \mid P$, is built from a set of declarations D and a set of properties P. Properties are given by first-order predicate formulas over relational propositions. For instance, $x, y : \mathbb{N} \mid x \leq y$ represents the set of bindings where both components x and y are constrained to be natural numbers by the declaration, and where x is constrained to be less than or equal to y by the property.

Schema text is used in manifold ways in Z. (1) We use it to directly denote sets of bindings, as in $[x, y : \mathbb{N} \mid x \leq y]$, which means the set $\{\!|\ x == 0, y == 0\ |\!\}, \{\!|\ x == 0, y == 1\ |\!\}, \ldots\}$. (2) We use it in quantifiers. For example, the universal quantifier is written as $\forall\, D \mid P \bullet Q$, where $D \mid P$ spawns the range of quantified values for which the property Q must hold. This notation is equivalent to $\forall\, D \bullet P \Rightarrow Q$. Similarly, $\exists\, D \mid P \bullet Q$, which is equivalent to $\exists\, D \bullet P \wedge Q$. (3) We use it in function abstractions, $\lambda\, D \mid P \bullet E$, where $D \mid P$ denotes the domain of the function. The binding set $D \mid P$ is converted into a set of tuples by removing the names of the components. For instance, the domain of the abstraction $\lambda\, x : \mathbb{N};\ y : \mathbb{N} \mid y \neq 0 \bullet x \,\mathsf{div}\, y$ is the set of pairs (x representing the first and y the second component) with the specified property ($y \neq 0$). Functions are just special sets, such that the above abstraction is an element of the set $\mathbb{P}((\mathbb{N} \times \mathbb{N}) \times \mathbb{N})$. (4) Schema text s used in set comprehension, $\{D \mid P\}$. The binding set $D \mid P$ is converted into a set of tuples by removing the names of the components. A variant of comprehension, $\{D \mid P \bullet E\}$, allows us to explicitly construct the result value, as e.g. in $\{x : \mathbb{N} \mid x \,\mathsf{mod}\, 2 = 0 \bullet (x, x+1)\}$, which describes the set of pairs where an even number is related with its direct successor. (5) Finally, schema text is also used for introducing global constants. Using the notation $\begin{array}{|l} D \\ \hline P \end{array}$ the names of the bindings of $D \mid P$ are introduced as global constants. If $D \mid P$ contains more then one binding, the constants are loosely specified. If $D \mid P$ is empty, the specification is inconsistent.

Z provides a powerful so-called *mathematical toolkit*, defining notions such as numbers, relations, and functions – a fairly standard account of basic set theory. The toolkit contains set constructors such as $E \leftrightarrow E'$ (the set of binary relations, an abbreviation for $\mathbb{P}(E \times E')$), $E \nrightarrow E'$ (the set of partial functions), $E \rightarrow E'$ (the set of total functions), $\mathsf{ran}\, E$ resp. $\mathsf{dom}\, E$ (the range and domain of a binary relation or function), $E^\sim$ (relational inversion), and so on.

The forthcoming Z Standard defines several lexical representations, one of them is LaTeX, which we use in this paper. Thus the LaTeX source of this document is also the source for processing the examples it contains under the ZETA system.

3 Basics of Constraint Programming in Z

In [5] a computation model based on concurrent constraint resolution has been developed for Z (the model is formally defined in the appendix). A high-performance virtual machine has been derived, which is implemented as part of the notation and tool integration environment ZETA [3]. 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.

3.1 Sets ⊃ Relations ⊃ Functions

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 the free type of natural numbers, an addition function on them, and a less-then relation. We avoid syntactic sugar to get the principles right:

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

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. Recall that after the $\bullet$ in a set-comprehension a generator expression follows.

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

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

$\Rrightarrow$ `({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 $(e_1, \ldots, e_n) \in R$. (Clause-style definition is provided as syntactic sugar in ZETA, which is eliminated by such a translation).

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 $add(three, three) \Rrightarrow \mathtt{S(S(S(S(S(Z)))))}$.

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 $\{x, y, z : N \mid z = add(x, y) \bullet ((S\,x, y), S\,z)\}$ (instead of using $((x, y), z) \in add$), backwards computation is not be possible:

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

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

Here, the encapsulated search for $add(x, y)$, solving $\mu\, z : N \mid ((x, y), z) \in add$, cannot continue, since it is not allowed to produce bindings for the context variables x and y. This can be seen as a restriction – but actually we see it as a clean way a user can influence execution order in our framework, similar as discussed for the functional logic paradigm in [6].

3.2 Sets of Sets

The elegance of the functional paradigm comes mostly from the fact that functions are first-order citizens. In our implementation of execution for Z, sets are full first-order citizens as well – in particular we can build sets of sets. For example, we can implement operators such as domain projection, inversion, and relational image as follows (where the definition below uses Z's boxing style for introducing generics):

$[X, Y]$

$\mathsf{dom} == \lambda\, R : \mathbb{P}(X \times Y) \bullet \{x : X;\ y : Y \mid (x, y) \in R \bullet x\}$
$\_^{\sim} == \lambda\, R : \mathbb{P}(X \times Y) \bullet \{x : X;\ y : Y \mid (x, y) \in R \bullet (y, x)\}$
$\_(\!|\_|\!) == \lambda\, R : \mathbb{P}(X \times Y);\ S : \mathbb{P}\, X \bullet \{x : X;\ y : Y \mid x \in S \land (x, y) \in R \bullet y\}$

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:

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

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

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

This example makes use of universal and unique existential quantification, which are of non-executability in our setting. These quantors 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:

$$add \in N \times N \pfun N$$
$$\Rrightarrow$$
```
still searching after 200000 steps
gc # 1 reclaimed 28674k of 32770k (peak was 34818k)
...
```

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

$$(\lambda\, x, y : N \mid (x, three) \in less;\ (y, three) \in less \bullet add(x, y)) \in N \times N \pfun N$$
$$\Rrightarrow \texttt{*true*}$$

The example also illustrates a rough edge of our approach. The Z semantics defines the schema text $f : N \pfun N$ to be equivalent to $f : \mathbb{P}(N \times N) \mid f \in N \pfun N$. This treatment causes serious problems for executability, as we have seen. In the implementation 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 required constraint, the user has to place it in the constraint part of schema text.

3.3 Restrictions

Our implementation of constraint programming in Z has several restrictions. The most important for our current application to test-case evaluation is that no sub-solvers for arithmetic constraints are employed. Work is in progress to integrate such solvers.

Moreover, we currently do not incorporate finite set-unification [1]. As comes apparent in the formal definition of the computation model in the Appendix, a constraint which needs to unify sets tries to enumerate them and residuates until free variables in a set's extension are bound. There is no conceptual or technical

reason why set unification could not be added to the model – it is just not there because in our running applications for test-case evaluation it is not required.

4 Encoding of Temporal Interval Logics

Temporal interval logics [10, 4] 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 Motszkowski version of ITL, an embedding into Z has been described in [**?**]. Here, we develop an executable shallow encoding for the positive subset of this kind of ITL – that is a representation which does not require language or implementation extensions. The encoding supports resolution for timing and observation constraints (going behind Moszkowski's Tempura implementation) and is illustrated by some examples. The purpose of the exercise is to demonstrate how the flexibility of the Z language and the higher-order features of our constraint programming implementation facilitate domain-specific extensions in a quite natural way.

4.1 The Encoding

We define temporal formulas over some abstract type $STATE$, such that the intervals we look at have type $\mathrm{seq}\, STATE$ (seq _ is Z's type for sequences). The type $STATE$ will be refined later on.

The encoding of interval logic is given in Fig. 1. A *predicate* over a state binding is a unary relation, $p \in SP = \mathbb{P}\, STATE$. A *temporal formula* is encoded by a set of so-called arcs, $as \in TF = \mathcal{P} ARC$ [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 a interval which satisfies this formula may end at this point. $xs \in_T w$ is the satisfaction relation of this encoding of temporal formulas, and defined as follows: 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.

Fig. 2 defines the basic operators of our logic. The formula which is satisfied exactly by the empty trace is encoded by the singleton transition containing the *eot* arc. Temporal falsity is described by the empty set of arcs. $w_1 \sqcup w_w$ and $w_1 \sqcap w_2$ model disjunction and conjunction. $w_1 \fatsemi w_2$ is sequential composition ("chop"). w^* is the repetition of w for zero or more times. Since our implementation of Z imposes a *strict* (eager) evaluation order, we need to embed the recursive reference to $\_^*$ in a set-comprehension, $\{a : ARC \mid a \in w^*\}$. The formula $!\, p$ holds for those intervals of length 1 for which p holds at the singleton state.

[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.

Fig. 1 Encoding of Interval Logic

$$SP == \mathbb{P}\, STATE \qquad TF == \mathcal{P} ARC$$

$$ARC ::= eot \mid tr\langle\langle SP \times TF \rangle\rangle \qquad | \; eot?\_ == \{w : TF \mid eot \in w\}$$

$$\_ \in_T \_ : \operatorname{seq} STATE \leftrightarrow TF$$

$$(\_ \in_T \_) = \{w : TF \mid eot?\, w \bullet (\langle\rangle, w)\} \cup \{x : STATE;\ xs : \operatorname{seq} STATE;\ p : SP;\ w, w' : TF \mid tr(p, w') \in w;\ x \in p;\ xs \in_T w' \bullet (\langle x \rangle \frown xs, w)\}$$

Fig. 2 Operators of Interval Logic

$$\mathsf{empty} == \{eot\} \qquad | \; \mathsf{false} == \varnothing[ARC]$$

$$\_ \sqcup \_ == \lambda\, w_1, w_2 : TF \bullet w_1 \cup w_2 \qquad | \; \bigsqcup == \lambda\, ws : \mathbb{P}\, TF \bullet \bigcup ws$$

$$\_ \sqcap \_ : TF \times TF \to TF$$

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

$$\_ \fatsemi \_ : TF \times TF \to TF$$

$$(\_ \fatsemi \_) = \lambda\, w_1, w_2 : TF \bullet (\textbf{if } eot?\, w_1 \textbf{ then } w_2 \textbf{ else } \mathsf{false}) \sqcup \{p : SP;\ w_1' : TF \mid tr(p, w_1') \in w_1 \bullet tr(p, w_1' \fatsemi w_2)\}$$

$$\_^* : TF \to TF$$

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

$$! == \lambda\, p : SP \bullet \{tr(p, \mathsf{empty})\} \qquad | \; \mathsf{skip} == !\, STATE$$

$$\mathsf{true} == \mathsf{skip}^* \qquad | \; \lceil \_ \rceil == \lambda\, p : SP \bullet !\, p \fatsemi (!\, p)^*$$

$$\_ \leadsto \_ == \lambda\, p : SP;\ w : TF \bullet !\{x : STATE \mid x \notin p\}^* \fatsemi ((!\ p \fatsemi w) \sqcup \mathsf{empty})$$

The formula skip holds for arbitrary singleton intervals. Temporal truthness, satisfied by any interval, is the repetition of skip. With $\lceil p \rceil$ those intervals of length greater 0 are described in which p holds at each point. $p \leadsto w$ describes temporal implication: it is satisfied for those intervals that when at some point p is observed then afterwards until the end of the interval w holds.

We take a look at the encoding of some formulas. Suppose type *STATE* 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 1:

$$!\{1\} \fatsemi \mathsf{empty} \fatsemi !\{2\} \fatsemi \mathsf{empty} \Rrightarrow \texttt{\{tr(\{1\},\{tr(\{2\},\{eot\})\})\}}$$
$$!\{1\}^* \Rrightarrow \texttt{\{eot,tr(\{1\},\{eot,tr(\{1\},...)\})\}}$$
$$!\{2,3\}^* \sqcap (!\{1,2\} \fatsemi !\{3,4\}) \Rrightarrow \texttt{\{tr(\{2\},\{tr(\{3\},\{eot\})\})\}}$$

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 used predicates are finite – also generate the set of traces which satisfy a formula:

$$\langle 1,2,3,1,2,1\rangle \in_T (\{2\} \rightsquigarrow \lceil\{x : \mathbb{Z} \mid x \geq 2\}\rceil)^* \Rrightarrow \texttt{*false*}$$
$$\langle 2,2,2,1,2,2\rangle \in_T (\{2\} \rightsquigarrow \lceil\{x : \mathbb{Z} \mid x \geq 2\}\rceil)^* \Rrightarrow \texttt{*true*}$$
$$\{t : \mathrm{seq}\, STATE \mid t \in_T \lceil\{1,2\}\rceil\} \Rrightarrow \{\texttt{<1>,<2>,<1,1>,<1,2>,...}\}$$

In the first two examples above, the formula states that the interval must be partitionable into n subintervals such that in each subinterval, whenever a state 2 is encountered, the remaining states until the end of the subinterval are greater or equal 2, and (since $\lceil p \rceil$ is only satisfied for intervals with length greater 0) that there *are* remaining states. Note that this requires to choose the right partitioning between several possibilities.

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

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

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

4.2 Timing Constraints

Due to the higher-orderness of Z and our implementation, it is easily possible to add new temporal operators. Suppose that our data space $STATE$ is partitioned such that it contains a time stamp $t?$[2], holding the current absolute system time, a duration stamp $d?$, describing the distance to the next state, and some sensors, given by a schema $SENSORS$ (to be refined for the concrete application) which is included into $STATE$:

$$\mid \mathcal{T} == \mathbb{Z} \qquad\qquad \mid STATE == [t?, d? : \mathcal{T};\ SENSORS]$$

We can now define a duration operator $DUR(d)$ which holds for those intervals whose duration is d:

$$\begin{array}{l} \mid DUR == \lambda\, d : \mathcal{T} \bullet \\ \quad !([STATE \mid d = d?]) \sqcup \\ \quad \bigsqcup\{t_0 : \mathcal{T} \bullet\ !([STATE \mid t_0 = t?]) \fatsemi \mathsf{true} \fatsemi !([STATE \mid d = t? - t_0 + d?])\} \end{array}$$

This definition uses the "generalized disjunction" for temporal formulas (Fig. 2) to introduce a local variable t_0 which binds the time stamp when entering the

[2] Currently, in our implementation of Z only integral numbers are supported – hence we define type $\mathcal{T}$ to be $\mathbb{Z}$.

interval satisfying $DUR(d)$. Semantically, the set-comprehension above denotes the set of all formulas such that t_0 has some binding according to the given constraints. 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 toolkit 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.

We take a look at an example. We calculate the partitions of an interval which have equal duration, using repetition on the duration operator:

$$\{d : \mathcal{T} \mid d < 8;\ \langle \langle\!|\ t? == 0, d? == 1\ |\!\rangle, \langle\!|\ t? == 1, d? == 1\ |\!\rangle, \langle\!|\ t? == 2, d? == 2\ |\!\rangle, \\ \langle\!|\ t? == 4, d? == 2\ |\!\rangle, \langle\!|\ t? == 6, d? == 2\ |\!\rangle\rangle \quad \in_T DUR(d)^*\}$$

$\Rrightarrow$ `{2,4}`

Note that for the duration 2, the first partition contains two states, whereas the remaining partitions contain one.

4.3 Application: Requirements Specification

Fig. 3 gives an example how to apply our temporal logics for requirements specification. The specification defines some aspects of the behavior of a (much simplified) elevator controller. Such a specification can then be used for test-evaluation, feeding it with the concrete traces produced by an implementation of the controller.

The elevator's state is modeled by a set of sensors which are combined with time stamps into the system state $STATE$ as described in the previous section. The sensors are the current location of the elevator, the velocity 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 locations.

Auxiliary state predicates $At\, f$, $Open\, f$ and $Request\, f$ are introduced, to characterize the according observations. Our requirements are composed from the conjunction of three sub-requirements:

- $Saftey_1$: in each state, if the elevator is moving ($velocity \neq 0$), then all doors must be closed
- $Saftey_2$: in each state, the door of a floor can only be open if the elevator is At this floor (note that this implies that at each time at maximal one door can be open)
- $Serve$: describing a service situation – if the elevator is requested from some floor, then it cannot pass this floor without stopping at it. If it has once reached the floor, it must open the door at this floor at least after $delay$ seconds. (The specification does not handles error situations, where the elevator does not work for some reason.)

4.4 Application: Debugging

A further application of the encoding of interval logics lies in the analysis of traces of system behavior on a lower level than by a requirements specification,

Fig. 3 Elevator Requirements

$$
\begin{array}{l}
| \; LOCATION == \mathbb{N} \qquad\qquad | \; VELOCITY == \mathbb{N} \\
| \; FLOOR == \{0, 20, 40, 60, 80, 100\} \qquad | \; delay == 30 \\
\hline
\textit{SENSORS} \\
location : LOCATION;\ velocity : VELOCITY;\ open, request : \mathbb{P}\, FLOOR \\
\hline
| \; At == \lambda f : FLOOR \bullet [STATE \mid location = f] \\
| \; Open == \lambda f : FLOOR \bullet [STATE \mid f \in open] \\
| \; Request == \lambda f : FLOOR \bullet [STATE \mid f \in request] \\
| \; Safety_1 == \lceil [STATE \mid velocity \neq 0 \Rightarrow open = \varnothing] \rceil \\
| \; Safety_2 == \lceil [STATE \mid \forall f : FLOOR \mid Open\, f \bullet At\, f] \rceil \\
| \; Serve == \bigsqcup \{ f : FLOOR;\ d : \mathcal{T} \mid d \leq delay \bullet \\
\qquad\qquad Request\, f \rightsquigarrow (!(\neg\, At\, f))^* \fatsemi (\lceil At\, f \rceil \sqcap DUR\, d \sqcap (\mathsf{true} \fatsemi \lceil Open\, f \rceil))\} \\
| \; Requirements == Safety_1 \sqcap Safety_2 \sqcap Serve^* \\
\hline
\end{array}
$$

for example for the purpose of debugging. Again, the higher-orderness of our setting allows us to define suitable abstractions which facilitate such an activity.

As an example, the following abstraction binds all time instances on which the derivation over time of two subsequent samples exceeds a certain value (below, the Z form $\theta STATE$ extracts a state binding in the current schema context):

$$
\begin{array}{l}
| \; spike == \lambda\, diff : \mathbb{Z};\ at : \mathcal{T};\ f : (STATE \to \mathbb{Z}) \bullet \\
\qquad \bigsqcup \{ v_0 : \mathbb{Z};\ t_0 : \mathcal{T} \bullet \\
\qquad\qquad !([STATE \mid v_0 = f(\theta STATE);\ t_0 = t?])^{\circ}_{9} \\
\qquad\qquad !([STATE \mid at = t?;\ abs(\,(f(\theta STATE) - v_0)\ \mathsf{div}\ (at - t_0)\,) > diff])\}
\end{array}
$$

For the elevator, for example, we might want to check whether their are spikes in the velocity: an acceleration of more then $2\frac{m}{s^2}$ indicates a sensor hardware error. To find points in time where such an error occurs we execute:

$$
\begin{array}{l}
\{t : \mathcal{T} \mid \langle \{\!|\ t? == 0, velocity == 2, \ldots\ |\!\}, \{\!|\ t? == 1, velocity == 16, \ldots\ |\!\}, \\
\qquad\qquad \{\!|\ t? == 2, velocity == 16, \ldots\ |\!\}, \{\!|\ t? == 3, velocity == 48, \ldots\ |\!\} \rangle \\
\qquad\qquad\qquad \in_T \mathsf{true} \fatsemi spike(2, t, \lambda\, STATE \bullet velocity) \fatsemi \mathsf{true}\} \\
\Rrightarrow \texttt{\{1,3\}}
\end{array}
$$

5 Conclusion and Related Work

We have presented a practical, working setting for constraint programming based on the Z specification language. The application to animation and test-evaluation of requirement specifications has been illustrated. The Z language, though primarily designed for specification, has been shown to be a suitable environment

for embedding constraint programming. The examples given proved that higher-orderness is a key feature for an environment where we can add new abstractions and notation such as temporal logics in a convenient and consistent way. A formal definition of the underlying computation model is found in the appendix.

Executing Z. Animation of the "imperative" part of Z is provided by the ZANS tool [8], 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 [14], though no implementation exists today, and logic resolution is not employed. Other approaches are based on a mapping to Prolog (e.g. [16, 17]), 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 Logic Languages and Logic Functional Languages. There is a close relationship of our setting to functional logic languages such as Curry [7] or Oz [12]: 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 (in the appendix it is shown that μ again is just a special case of the more general concept of homomorphisms over sets, realizing encapsulated search).

The relation to logic languages with functional features, such as RELFUN [2] or λ-Prolog [11], is less close. These languages are oriented at a clause-based notational style and implementation technique and focus more on relations which can take functions as parameters, whereas we are more interested in functions which take relations as parameters.

Constraint 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, which are coordinated by our higher-order framework, is therefore subject of our current work.

Acknowledgment. Thanks go to Petra Hofstedt for fruitful discussions in the CP working group at the TUB.

References

1. P. Arenas-Sanchez and A. Dovier. A minimality study for set unification. *Journal of Functional and Logic Programming*, 1997(7), 1997.

2. H. Boley. *A Tight, Practical Integration of Relations and Functions*, volume 1712 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1999.
3. 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.
4. Z. Chaochen, C. A. R. Hoare, and A. Ravn. A calculus of durations. *Information Processing Letters*, 40(5), 1991.
5. W. Grieskamp. *A Set-Based Calculus and its Implementation*. PhD thesis, Technische Universität Berlin, 1999.
6. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19(20), 1994.
7. M. Hanus. Curry – an integrated functional logic language. Technical report, Internet, 1999. Language report version 0.5.
8. X. Jia. An approach to animating Z specifications. Internet: `http://saturn.cs.depaul.edu/~fm/zans.html`, 1996.
9. G. Kahn. Natural semantics. In *Symposium on Theoretical Computer Science (STACS'97)*, volume 247 of *Lecture Notes in Computer Science*, 1987.
10. B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, 1986. updated version from the authors home page.
11. G. Nadathur and D. Miller. An overview of λProlog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*. MIT Press, 1988.
12. 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.
13. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
14. S. Valentine. The programming language Z^{--}. *Information and Software Technology*, 37(5–6):293–301, May–June 1995.
15. D. H. D. Warren. The extended andorra model with implicit control. In *ICLP'90 Parallel Logic Programming Workshop*, 1990.
16. 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.
17. M. Winikoff, P. Dart, and E. Kazmierczak. Rapid prototyping using formal specifications. In *Proceedings of the Australasian Computer Science Conference*, 1998.
18. Drafts for the Z ISO standard. Ian Toyn (editor). URL: `http://www.cs.york.ac.uk/~ian/zstan`, 1999.

A Computation Model

In this appendix, we give the formal definition of the computation model of Z. First, we construct an intermediate calculus, called μZ, to which the Z language is reduced. Next, computation in μZ is defined in the style of natural semantics [9].

Let x be variable symbols and let ρ be (term) constructor symbols. The syntax of μZ is defined as follows:

$$e \in \mathbb{E} ::= x \mid \rho(\overline{e}) \mid \mathsf{0} \mid \{e\} \mid e \cup e' \mid \{p \setminus \overline{x} \mid \phi\} \mid \mathsf{hom}_\delta\, e$$
$$p \in \mathbb{P} \subseteq \mathbb{E};\ \phi \in \mathbb{C} ::= e \subseteq e' \mid \mathsf{true} \mid \phi \wedge \phi';\ \delta \in \Delta == \{\mu, \mathsf{E}, \ldots\}$$

The form $\rho(\overline{e})$ describes a term constructor application, where $\overline{e}$ is a sequence of expressions. The set of patterns, $\mathbb{P}$, are those expressions which are solely built from variables x and constructor applications. The form $\mathsf{0}$ is the empty set, The form $\{e\}$ a singleton set, the form $e \cup e'$ set union. $\{p \setminus \overline{x} \mid \phi\}$ denotes a set comprehension, and is the set of values which match the pattern p such that there exists a solution for the local existential variables $\overline{x}$ such that the constraint ϕ is satisfied under the substitution of the match. We will write $\{p \mid \phi\}$ for the case that $\#\overline{x} = 0$. Constraints ϕ are tautologies, subset constraints and constraint conjunction.

The expression form $\mathsf{hom}_\delta\, e$ denotes a *homomorphism* on the set e, which enumerates the elements of e to compute a result from them. The μZ calculus provides a fixed (but extensible) number of builtin homomorphisms, whose treatment in the computation model is homogeneous. Here, we will only use two of them. The μ-homomorphism determines the μ-value of a set, and is defined iff the set is not empty and contains no distinct elements; the unique element is then delivered. The E-homomorphism forces the extensional enumeration of elements of a set: it is defined iff the set is finitely enumeratable, then yielding the set itself (in its extensional representation).

It is possible to map all constructs of Z to the μZ calculus in a natural way. The most interesting part here are Z properties, whose mapping is performed in a context of a μZ set comprehension, $\{p \setminus \overline{x} \mid \mathcal{C}\ q\}$, where $\mathcal{C}$ is a conjunctive context of μZ constraints, and q is a not yet mapped Z property (the restriction to a comprehension context is not a real one, since an entire Z specification is treated as a top-level set comprehension). In order to represent truth values, sets over a singleton type are used, whose element is constructed by tt, such that truth is $\{\mathsf{tt}\}$ and falsity $\mathsf{0}$:

$$\begin{array}{lcl}
\{p \setminus \overline{x} \mid \mathcal{C}(q \vee q')\} & \rightsquigarrow & \{p \setminus \overline{x} \mid \mathcal{C}(\{\mathsf{tt}\} \subseteq \{\mathsf{tt} \mid q\} \cup \{\mathsf{tt} \mid q'\})\} \\
\{p \setminus \overline{x} \mid \mathcal{C}(q \Rightarrow q')\} & \rightsquigarrow & \{p \setminus \overline{x} \mid \mathcal{C}(\{\mathsf{tt} \mid q\} \subseteq \{\mathsf{tt} \mid q'\})\} \\
\{p \setminus \overline{x} \mid \mathcal{C}(\neg\, q)\} & \rightsquigarrow & \{p \setminus \overline{x} \mid \mathcal{C}(\{\mathsf{tt} \mid q\} \subseteq \mathsf{0})\} \\
\{p \setminus \overline{x} \mid \mathcal{C}(\exists\, y : e \mid q \bullet q')\} & \rightsquigarrow & \{p \setminus \overline{x}, y \mid \mathcal{C}(q \wedge q')\} \\
\{p \setminus \overline{x} \mid \mathcal{C}(\forall\, y : e \mid q \bullet q')\} & \rightsquigarrow & \{p \setminus \overline{x} \mid \mathcal{C}(\{y \mid q\} \subseteq \{y \mid q'\})\}
\end{array}$$

Note that constraints introduced by declarations, $y : e$, are discarded, as discussed in Sec. 3.2. Further concepts of Z, for example the schema calculus, introduce no difficulties, and their mapping is therefore left open here.

We now define the natural semantics of μZ. Values are a normal form of expressions as specified by the following grammar:

$$v \in \mathbb{E}_V ::= \rho(\overline{v}) \mid \mathsf{0} \mid \{v\} \mid \{p \setminus \overline{x} \mid \phi\} \mid v \cup v' \mid x$$

Values can be "cyclic", that is regularly infinite. With $\mathsf{freeze}\ X\ v$ the variables $x \in X$ are "frozen" in v, mapping each x to a unique variable x_f. With $\mathsf{unfreeze}\ X\ v$ variables are unfrozen. The function $\mathsf{free}\ v$ delivers those free variables in a value which are not frozen, the function $\mathsf{frozen}\ v$ those which are. A *substitution* is a mapping from variables to values, defined in the usual way. With $\mathsf{bound}\,\sigma = \{x : \mathsf{dom}\,\sigma \mid \sigma x \neq x\}$ the bound variables of a substitution are denoted. With $\sigma[x := v]$ we extend a substitution by a variable assignment. A (partial) *equality* on values, written as $v \sim v'$, is available. Variables (frozen or not) are equal only by name, set comprehensions never. The equality takes commutativity, associativity and idempotence of set union into account. The relation $v \not\sim v'$ indicates inequality, and is *not* the reverse of $v \sim v'$: inequality of variables of different names cannot be decided, as well as inequality of set comprehensions. A *goal* is a substitution paired with a constraint, written as $\theta = \sigma :: \phi$. A *choice* is a sequence of goals, $\overline{\theta} = \langle \theta_1, \ldots, \theta_n \rangle$. The propagation of a substitution over a choice is defined as $\sigma \rhd \langle \sigma_1 :: \phi_1, \ldots, \sigma_n :: \phi_n \rangle = \langle \sigma \circ \sigma_1 :: \phi_1, \ldots, \sigma \circ \sigma_n :: \phi_n \rangle$

Computation is defined by two relations which are mutually dependent. The first relation, $e \xrightarrow{\sigma} e'$, describes a reduction step on expressions under the substitution σ. The second, $\overline{\theta} \rightsquigarrow \overline{\theta}'$, describes a resolution step by mapping a choice into a choice. The rules for $e \xrightarrow{\sigma} e'$ are given in Fig. 4. $\mathcal{S}$ describes a strict reduction context, which is specified by a "grammar with a hole". The intermediate expression form $\mathsf{HOM}_\delta(hs, X, x, \overline{\theta})$ is used to represent the state of homomorphism reduction, where hs is the current internal state of the according homomorphism δ. *start*, *next*, *stop* and *end* describe the behavior of homomorphisms, and are provided with definitions for μ and E, where we assume that unmentioned argument cases are not in the domain of the according functions (the domain test is used for driving the rules). We have not used the *stop* possibility, since both homomorphisms need to enumerate the entire set – but other homomorphisms may stop enumeration after the first solution is found.

The rules for $\overline{\theta} \rightsquigarrow \overline{\theta}'$ are given in Fig. 5. $\mathcal{C}$ describes a conjunctive context where the next resolution step is to be applied. The choice of the context is non-deterministic, indicating that constraints are concurrently reduced. In Rule $C9$, we try to resolve equality on sets which contain set-comprehensions by enumeration. Rule $C12$ models the import of a set comprehension into a resolution context – this is the place where we *instantiate* constraints. $C14$ creates a choice point – in an implementation, we defer applying this rule as long as possible, employing the Andorra Principle [15]. The (intended) restrictions of the model become apparent in the kind of subset-constraints *not* handled: non-trivial constraints where free variables appear at the left or the right side and unions at the right side where the left set is not a singleton.

Fig. 4 Expression Reduction Rules

$$\mathcal{S}\cdot ::= (\mathcal{S}\cdot)\cup e \mid e\cup(\mathcal{S}\cdot) \mid \{\mathcal{S}\cdot\} \mid \rho(\ldots,\mathcal{S}\cdot,\ldots) \mid \mathsf{hom}_\delta(\mathcal{S}\cdot) \mid \cdot$$

$$E1\ \frac{e \xrightarrow{\sigma} e'}{\mathcal{S}\ e \xrightarrow{\sigma} \mathcal{S}\ e'} \qquad E2\ \frac{x \in \mathsf{bound}\,\sigma}{x \xrightarrow{\sigma} \sigma x}$$

$$E3\ \frac{X = \mathsf{free}\,v;\ x \notin X \cup \mathsf{frozen}\,v}{\mathsf{hom}_\delta v \xrightarrow{\sigma} \mathsf{HOM}_\delta(\mathit{start}\,\delta, X, x, \langle\sigma :: \{x\} \subseteq \mathsf{freeze}\ X\ v\rangle)}$$

$$E4\ \frac{\sigma \rhd \overline{\theta} \rightsquigarrow \overline{\theta}'}{\mathsf{HOM}_\delta(hs, X, x, \overline{\theta}) \xrightarrow{\sigma} \mathsf{HOM}_\delta(hs, X, x, \overline{\theta}')}$$

$$E5\ \frac{(hs, \sigma'\ x) \in \mathsf{dom}(\mathit{next}\,\delta)}{\mathsf{HOM}_\delta(hs, X, x, \langle\sigma' :: \mathsf{true}\rangle \frown \overline{\theta}) \xrightarrow{\sigma} \mathsf{HOM}_\delta(\mathit{next}\,\delta\,(hs, \sigma'\ x), X, x, \overline{\theta})}$$

$$E6\ \frac{(hs, \sigma'\ x) \in \mathsf{dom}(\mathit{stop}\,\delta)}{\mathsf{HOM}_\delta(hs, X, x, \langle\sigma' :: \mathsf{true}\rangle \frown \overline{\theta}) \xrightarrow{\sigma} \mathsf{unfreeze}\ X\ (\mathit{stop}\,\delta\,(hs, \sigma'\ x))}$$

$$E7\ \frac{hs \in \mathsf{dom}(\mathit{end}\,\delta)}{\mathsf{HOM}_\delta(hs, X, x, \langle\rangle) \xrightarrow{\sigma} \mathsf{unfreeze}\ X\ (\mathit{end}\,\delta\,hs)}$$

$$\begin{array}{lll} \mathit{start}\,\mathsf{E} & = 0 & \mathit{start}\ \mu & = \varnothing \\ \mathit{next}\,\mathsf{E}\,(v, v') & = v \cup \{v'\} & \mathit{next}\ \mu\,(\varnothing, v) & = \{v\}\ \textbf{if}\ \mathsf{free}(v) = \varnothing \\ \mathit{end}\,\mathsf{E}\,v & = v & \mathit{next}\ \mu\,(\{v\}, v') & = \{v\}\ \textbf{if}\ v \sim v' \\ & & \mathit{end}\ \mu\,\{v\} & = v \end{array}$$

Fig. 5 Constraint Resolution Rules

$$\mathcal{C}\cdot ::= (\mathcal{C}\cdot)\wedge\phi \mid \phi\wedge(\mathcal{C}\cdot) \mid \cdot \qquad C1\ \frac{\langle\sigma :: \phi\rangle \rightsquigarrow \langle\sigma_1 :: \phi_1, \ldots, \sigma_n :: \phi_n\rangle}{\langle\sigma :: \mathcal{C}\,\phi\rangle \frown \overline{\theta} \rightsquigarrow \langle\sigma_1 :: \mathcal{C}\,\phi_1, \ldots, \sigma_n :: \mathcal{C}\,\phi_n\rangle \frown \overline{\theta}}$$

$$C2\ \frac{i, j \in \{1,2\};\ i \neq j;\ \phi_i = \mathsf{true}}{\langle\sigma :: \mathcal{C}(\phi_1 \wedge \phi_2)\rangle \frown \overline{\theta} \rightsquigarrow \langle\sigma :: \mathcal{C}\,\phi_j\rangle \frown \overline{\theta}} \qquad C3\ \frac{e_i \xrightarrow{\sigma} e_i';\ i, j \in \{1,2\};\ i \neq j;\ e_j' = e_j}{\langle\sigma :: e_1 \subseteq e_2\rangle \rightsquigarrow \langle\sigma :: e_1' \subseteq e_2'\rangle}$$

$$C4\ \frac{}{\langle\sigma :: \{\rho(v_1, \ldots, v_n)\} \subseteq \{\rho(v_1', \ldots, v_n')\}\rangle \rightsquigarrow \langle\sigma :: \{v_1\} \subseteq \{v_1'\} \wedge \ldots \wedge \{v_n\} \subseteq \{v_n'\}\rangle}$$

$$C5\ \frac{\rho \neq \rho'}{\langle\sigma :: \{\rho(\ldots)\} \subseteq \{\rho'(\ldots)\}\rangle \rightsquigarrow \langle\rangle} \qquad C6\ \frac{\begin{array}{c} i, j \in \{1,2\};\ i \neq j \\ v_i\ \textit{not frozen variable};\ v_i \notin \mathsf{bound}\,\sigma \end{array}}{\langle\sigma :: \{v_1\} \subseteq \{v_2\}\rangle \rightsquigarrow \langle\sigma[v_i := v_j] :: \mathsf{true}\rangle}$$

$$C7\ \frac{v \sim v'}{\langle\sigma :: \{v\} \subseteq \{v'\}\rangle \rightsquigarrow \langle\sigma :: \mathsf{true}\rangle} \qquad C8\ \frac{v \not\sim v'}{\langle\sigma :: \{v\} \subseteq \{v'\}\rangle \rightsquigarrow \langle\rangle}$$

$$C9\ \frac{i, j \in \{1,2\};\ i \neq j;\ v_i\ \textit{non-extensional set};\ v_i' = \mathsf{hom}_\mathsf{E}\,v_i;\ v_j' = v_j}{\langle\sigma :: \{v_1\} \subseteq \{v_2\}\rangle \rightsquigarrow \langle\sigma :: \{v_1'\} \subseteq \{v_2'\}\rangle}$$

$$C10\ \frac{}{\langle\sigma :: 0 \subseteq v\rangle \rightsquigarrow \langle\sigma :: \mathsf{true}\rangle} \qquad C11\ \frac{}{\langle\sigma :: \{v\} \subseteq 0\rangle \rightsquigarrow \langle\rangle} \qquad C12\ \frac{\textit{rename}\ p, \overline{x}, \phi\ \textit{wrt}\ \sigma}{\langle\sigma :: \{v\} \subseteq \{p \setminus \overline{x} \mid \phi\}\rangle \rightsquigarrow \langle\sigma :: \{v\} \subseteq \{p\} \wedge \phi\rangle}$$

$$C13\ \frac{}{\langle\sigma :: v_1 \cup v_2 \subseteq v\rangle \rightsquigarrow \langle\sigma :: v_1 \subseteq v \wedge v_2 \subseteq v\rangle} \qquad C14\ \frac{}{\langle\sigma :: \{v\} \subseteq v_1 \cup v_2\rangle \rightsquigarrow \langle\sigma :: \{v\} \subseteq v_1, \sigma :: \{v\} \subseteq v_2\rangle}$$

$$C15\ \frac{}{\sigma :: \{p \setminus \overline{x} \mid \phi\} \subseteq v \rightsquigarrow \sigma :: \mathsf{hom}_\mathsf{E}\{p \setminus \overline{x} \mid \phi\} \subseteq v}$$