Transition Specifications & the Algebraic Core of the Z Notation

Martin Große-Rhode, Alfio Martini

Bericht-Nr. 96-31

Transition Specifications & the Algebraic Core of the Z Notation

Martin Große-Rhode Alfio Martini *

Technische Universität Berlin, Fachbereich 13 Informatik, Franklinstrasse 28/29
D-10587 Berlin, Germany

e-mail:{alfio,mgr}@cs.tu-berlin.de

Abstract

The *Z-notation* is a powerful tool for the formal specification of software systems, based on a model theoretic approach. *Transition specifications* are an extension of algebraic data type specifications to dynamic abstract data types, based on the idea of states as environments. Our aim in this paper is to define an (informal) translation between an *algebraic core* of the Z-notation and transition specifications. This allows to compare the approaches, relate their mutual benefits as well as to clarify how some main concepts – *locality, compositionality,* and *separation of concerns* – that are realized in transition specifications, could be carried over Z.

1 Introduction

The *Z–notation* is a powerful tool for the formal specification of software systems, based on a model theoretic approach. Z–specifications define (classes of) set theoretic models of systems using a standardized notation for set operations and logical descriptions. A specification is organized as a hierarchy of *schemas*, each of which may introduce new names (variables) with constraints on their properties. In this way a model is constructed, given by assignments of the variables of the specification to sets representing their types. Both static and dynamic aspects of the system, its data types, admissible states, and operations that change the state, are defined by the same means of schemas. The dynamic state transformation given by an operation is thereby modeled as a pair of sets, representing the state of the system before and after the application of the operation. This unique modeling is an implication of the general semantic assumption of Z, that everything in the model of a system is a set: types, functions, states, transitions et.

Transition specifications are an extension of algebraic data type specification to dynamic abstract data types, based on the idea of states as environments. These are modeled as partial

*Research supported in part by a CNPq-grant 200529/94-3.

functions from *references* to their contents, where references (identifiers) are contained in the data type and dereferencing is simply a partial function from references into their value sort. Formally an algebraic data type specification is extended by a specification of operations, whose semantics is defined in terms of the way they redefine the dereferencing functions. As opposed to a Z–specification there is thus a distinction between the dynamic and static part of the system that is already fixed on the syntactic level.

The specification framework of transition specifications has the same basic structural properties as the one of algebraic data type specification. Each specification defines a category of models, which has an initial model (a prototype or standard interpretation). There is a sound and complete deduction calculus, and there are structuring operations on the syntactic level, like parameterization and actualization, renaming, union, etc., with a compositional semantics.

Our aim in this paper is to define an (informal) translation between an *algebraic core* of the Z–notation and transition specifications that allows to compare the approaches and relate their mutual benefits. The algebraic core, in hindsight, is the part of Z that is translatable into an (algebraic) transition specification. It is obvious that the general expressive power, given by the general logical formulae of Z, cannot be reached by an algebraic approach in the classical sense of (conditional) equational theories. In general only positive conditions on the behavior of a system can be specified in an algebraic framework.

Our intention is not to write a complete compiler from Z to transition specifications. Z as a *specification language* is a fully worked out environment with a lot of details concerning the notation, syntactic macros, and implicit assumptions that are necessary to use it in practice. On the other hand, transition specifications are a *specification theory* whose purpose is to show the general possibility of algebraic specification of dynamic data types in a systematic approach. The purpose of this paper is to make clear how some main concepts — *locality*, *compositionality*, and *separation of concerns* — that are realized in transition specifications could be carried over to Z.

Locality Transition specifications, like algebraic specifications, are structured into small, single specifications with their categories of local models (resp. initial local models). Each single specification constitutes a name space for the sort–, function– and operation names introduced in this specification, and these names do not have a meaning outside this specification. If a connection between two (or several) specifications shall be introduced this has to be specified explicitly be a specification morphism that relates the items of the specifications concerned. This specification morphism is independent of the names chosen locally, in particular, accidental coincidence of local names does not imply that they denote the same object in the global specification.

Compositionality Putting large specifications together from smaller local parts is the general composition mechanism supported by transition specifications (as well as algebraic specifications). Local specifications are connected by morphisms as discussed above, thus on the syntactic level a (general) composition operation can be expressed by forming a diagram of specifications. Its result is the *colimit* of this diagram. Compositional semantics now means that the local models of the component specifications can be put together in the same way as the specifications themselves to obtain a global model of the

complete specification. Vice versa each global model can be decomposed into related local models, that is, global models are completely determined by their local components. This also implies that local correctness proofs can be put together to show the overall correctness of the complete specification.

Separation of Concerns As already mentioned above the distinction between static and dynamic properties of a system is already done on the syntactic level in a transition specification. That means, if a function is declared as static there is no possibility that it ever may have side effects on the state. If state changes (side effects) should at least possible the function has to be declared as a method, and its semantics has to defined in terms of the state changes it causes, not in terms of equations. This distinction supports the approach to specify purely functional as long as possible, and to introduce and use a state only if necessary on top of this purely functional part.

A further topic, that has been investigated for transition specifications more recently is the description of concurrent behavior of operations ([5]). However, in this paper we do not address this issue.

The paper is organized as follows. In the next section transition specifications are introduced as far as necessary for the translation. Then the translation of Z–specifications into transition specifications is developed by two examples. In the first, simpler one each schema is translated step by step and the correctness of the translation is (informally) shown by discussing the formal semantics of the Z–schema and its translation. The second, more involved example introduces some further Z issues and the possibility of their translation. In the appendix an algebraic specification of sets is presented, that is used in the translation as algebraic counterpart of the Z toolkit, that offers all the needed constructions on sets.

2 A Primer on Transition Specifications

Transition specifications extend algebraic specifications by references and dereferencing functions, thus modeling states and state transformations on top of a purely functional abstract data type. A model of a transition specification consists of two components. The first one is a partial algebra A that models the functional (static) data type. It includes the references, which are nothing but names. The second component is a transition system whose states are extensions of A by partial dereferencing functions — corresponding to environments — and whose transitions are redefinitions of these functions. Accordingly, a transition specification consists of a partial equational specification for the static data type and a method specification for the transitions.

Let's first recall the main definitions of partial algebras and their specification.

A **signature** *SIG = (S,OP)* is given by a set S of sort names and an $S^* \times S^*$–indexed family $OP = (OP_{w,v})_{w,v \in S^*}$ of operation symbols, in the same way as for total algebras. Note, however, that the empty string λ and strings of sort names are allowed as codomains of operations. This allows to specify predicates, modeled by partial functions into the one point set $\{*\}$, and operations into product sorts.

The sorted set *Term = Term(SIG)* of **terms** corresponding to a signature is accordingly defined as follows.

- variables
  $x_i : s_1 \dots s_n \to s_i \in Term \quad (s \in S, i = 1, \dots, n)$
- tuples
  If $\quad t_1 : v \to s_1, \dots, t_n : v \to s_n \in Term$
  then $\quad \langle t_1, \dots, t_n \rangle : v \to s_1 \dots s_n \in Term$
- operation application
  If $\quad \langle t_1, \dots, t_n \rangle : v \to w \in Term$
  and $\quad \sigma : w \to s'_1 \dots s'_k \in OP$
  then $\quad \sigma_i(t_1, \dots t_n) : v \to s'_i \in Term \quad$ for $i = 1, \dots k$
  resp. $\quad \sigma(t_1, \dots t_n) : v \to \lambda \in Term \quad$ if $k = 0$

In a more traditional notation $t : w \to v \in Term$ corresponds to $t \in T_{SIG,s'_1}(X) \times \dots \times T_{SIG,s'_k}(X)$, where $v = s'_1 \dots s'_k$, $w = s_1 \dots s_n$ and $X = \{x_1 : s_1, \dots, x_n : s_n\}$.

A **partial *SIG*–algebra** $A = ((A_s)_{s \in S}, (\widehat{A_\sigma})_{\sigma \in OP})$ is given by

- a family of sets $(A_s)_{s \in S}$, the *carrier sets* of A
- a family of partial functions $(\widehat{A_\sigma})_{\sigma \in OP}$, the *operations* of A
  where
  - $\widehat{A_\sigma} : A_w \multimap A_v \quad$ if $\sigma : w \to v \in OP$

  that is $\widehat{A_\sigma}$ is given by
  - a subset $dom(A_\sigma) \subseteq A_w$, the *domain* of A_σ
  - a total function $A_\sigma : dom(A_\sigma) \to A_v$

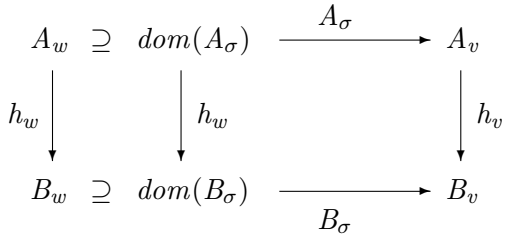
(With $A_\lambda = \{*\}$ and $A_u = A_{s_1} \times \dots \times A_{s_m}$ for $u = s_1 \dots s_m \in S^*$.)

Thus operations $\sigma : w \to s'_1 \dots s'_k$ are interpreted as tuples of partial functions, and operations $\rho : w \to \lambda$ correspond to predicates $dom(A_\rho) \subseteq A_w$; the total function part $A_\rho : dom(A_\rho) \to A_\lambda = \{*\}$ is redundant.

Homomorphisms of partial algebras preserve domains and are compatible with the operations on their domains. That is, a ***SIG*–homomorphism** $h : A \to B$, $h = (h_s)_{s \in S}$, is given by a family of total functions $h_s : A_s \to B_s \quad (s \in S)$, such that for all $\sigma : w \to v \in OP$

- $h_w(dom(A_\sigma)) \subseteq dom(B_\sigma)$
- $h_v(A_\sigma(a)) = B_\sigma(h_w(a)) \quad$ for all $a \in dom(A_\sigma)$

(where $h_\lambda = id : \{*\} \to \{*\}$ and $h_u = h_{s_1} \times \cdots \times h_{s_m}$ for $u = s_1 \ldots s_m \in S^*$)



The **evaluation** of a term $t : w \to v \in Term$ in a partial SIG–algebra A is the partial function $A_t : A_w \multimap A_v$ defined as follows.

- variables
  $dom(A_{x_i}) = A_{s_1} \times \ldots \times A_{s_n}$
  $A_{x_i}(\langle a_1, \ldots, a_n \rangle) = a_i$
- tuples
  $dom(A_{\langle t_1,\ldots,t_n \rangle}) = \bigcap_{1 \le i \le n} dom(A_{t_i})$
  $A_{\langle t_1,\ldots,t_n \rangle}(a) = \langle A_{t_1}(a), \ldots, A_{t_n}(a) \rangle$
- operation application
  $dom(A_{\sigma(t_1,\ldots,t_n)}) =$
  $\quad = \{a \in dom(A_{\langle t_1,\ldots,t_n \rangle}) \mid \langle A_{t_1}(a), \ldots, A_{t_n}(a) \rangle \in dom(A_\sigma)\}$
  $A_{\sigma_i(t_1,\ldots,t_n)}(a) = \pi_i(A_\sigma(A_{t_1}(a), \ldots, A_{t_n}(a)))$,
  $\quad$ the i'th component of $A_\sigma(A_{t_1}(a), \ldots, A_{t_n}(a))$
  resp. $A_{\sigma(t_1,\ldots,t_n)}(a) = *$, if $\sigma : w \to \lambda$

In terms of variable assignments $asg : X \to A$ and term evaluation $asg^\sharp : T_{SIG}(X) \to A$ this corresponds to $asg^\sharp(t) = A_t(a)$, where $X = \{x_1 : s_1, \ldots, x_n : s_n\}$, $a = \langle a_1, \ldots, a_n \rangle$ and $asg(x_i) = a_i$ $(i = 1, \ldots, n)$.

An **existence equation** e w.r.t a signature SIG is given by a pair of parallel sorted terms $t, t' : w \to v \in Term$, denoted $e = (w : t \stackrel{e}{=} t')$. $w = s_1 \ldots s_n$ represents the declaration $x_1 : s_1, \ldots, x_n : s_n$ of the variables that may occur in t or t' as above. A **conditional existence equation** ce w.r.t. SIG is given by two pairs of parallel sorted terms $r, r' : w \to v'$, and $t, t' : w \to v \in Term$ with common source w, denoted $ce = (w : r \stackrel{e}{=} r' \to t \stackrel{e}{=} t')$.

The **solution set** A_e of an existence equation $e = (w : t \stackrel{e}{=} t')$ in a partial SIG–algebra A is defined by

$$A_e = \{a \in dom(A_t) \cap dom(A_{t'}) \mid A_t(a) = A_{t'}(a)\}$$

and A **satisfies the conditional existence equation** $ce = (w : r \stackrel{e}{=} r' \to t \stackrel{e}{=} t')$, denoted $A \models ce$, if

$$A_{(w:r \stackrel{e}{=} r')} \subseteq A_{(w:t \stackrel{e}{=} t')}$$

Using an existence equation that always holds, like $(w : \langle\rangle \stackrel{e}{=} \langle\rangle)$, each existence equation $e = (w : t \stackrel{e}{=} t')$ can be considered as a conditional existence equation $ce = (w : \langle\rangle \stackrel{e}{=} \langle\rangle \to t \stackrel{e}{=} t')$. Since $A_{(w:\langle\rangle \stackrel{e}{=} \langle\rangle)} = A_w$ we have $A \models ce$ iff $A_w \subseteq A_{(w:t \stackrel{e}{=} t')}$, that is, iff each element $a \in A_w$ is a solution of e. In this sense $A \models e$ will be used as short notation for $A \models ce$. According to the definition of a solution set and the above convention $A \models (w : t \stackrel{e}{=} t)$ iff $A_t : A_w \multimap A_v$ is a total function. In particular $A \models (\lambda : t \stackrel{e}{=} t)$ for a ground term $t : \lambda \to v$ iff the element $A_t \in A_v$ is defined. In the following $(w : t \stackrel{e}{=} t)$ will be denoted $(w : t \uparrow)$. Note, furthermore, that terms comprise tuples of single sorted terms, $t = \langle t_1, \ldots, t_n \rangle$, whence existence equations comprise conjunctions of single sorted existence equations, $(w : t \stackrel{e}{=} t') \hat{=} (w : t_1 \stackrel{e}{=} t'_1 \wedge \ldots \wedge t_n \stackrel{e}{=} t'_n)$.

Applying partial equational specifications it is often convenient to use the following notion of *strong equality* $(t \stackrel{s}{=} t')$, that is satisfied if either both terms t and t' are undefined, or both are defined and equal. Since $t \stackrel{s}{=} t'$ iff $(t \uparrow \Rightarrow t = t') \wedge (t' \uparrow \Rightarrow t = t')$, strong equality can be reduced to existential equality.

Now a **partial equational specification** *SPEC = (S,OP,CE)* is given by

- a signature *SIG = (S,OP)*, and
- a set *CE* of conditional existence equations w.r.t. *SIG = (S,OP)*.

A partial *SIG*–algebra A is a **partial *SPEC*–algebra**, if A satisfies all conditional existence equations in *CE*. A ***SPEC*–homomorphism** $h : A \to B$ is the same as a *SIG*–homomorphism if A and B are partial *SPEC*–algebras.

For a partial specification *SPEC* the category *PAlg(SPEC)* is given by partial *SPEC*–algebras as objects and *SPEC*–homomorphisms as morphisms.

Partial equational specifications with designated reference sorts and dereferencing functions are the basis of transition specifications. Further restrictions on the operations and axioms ensure that equality of references is decidable, a property that will be used later to define the transitions. So a **base specification** *BS = (S,OP,CE,ref)* is given by a partial equational specification *SPEC = (S,OP,CE)* and a partial function $ref : S \to S$ satisfying the following conditions.

1. There are designated operations symbols $!_s : ref(s) \to s \in OP$ for all $s \in dom(ref)$.
2. Each operation symbol $\sigma : w \to v \in OP$ is either a constant of a reference sort (a *reference* for short), $\sigma : \lambda \to ref(s)$, or its target $v = s_1 \ldots s_n$ does not contain a reference sort, that is $v \in (S - ref(S))^*$.

3. Each reference is defined, that is
   $(\lambda : \delta \stackrel{e}{=} \delta) \in CE$ for all $\delta : \lambda \to ref(s) \in OP$.

4. There are no equations between references, that is
   $tar(t_1) \in (S - ref(S))^*$ for all $ce \in CE$.

Models of a base specification are partial algebras with distinct elements for syntactically different reference constants. Thus the **category of base models** *Mod(BS)* of a base specification *BS = (SPEC,ref)* is given by partial *SPEC*–algebras A with $A_\delta \neq A_{\delta'}$ for all $\delta \neq \delta' : \lambda \to ref(s) \in OP$ as objects and *SPEC*–homomorphisms $h : A \to B$ such that $h_{ref(s)} : A_{ref(s)} \to B_{ref(s)}$ is injective for all $s \in dom(ref)$ as morphisms.

States on a base model A are extensions of A by partial dereferencing functions $A_! : A_{ref(s)} \multimap A_s$. Syntactically such a partial function can be described by a list $\langle d_1, \ldots, d_n \rangle$ of (distinct) references and a list $\langle a_1, \ldots, a_n \rangle$ of values. Its semantics is the free extension of A by the existence equations $A_!(d_i) = a_i$.

Let *BS = (SPEC,ref)* be a base specification, A a *BS*–model. For each list of references $d_w = \langle d_1, \ldots, d_n \rangle \quad (d_i \in A_{ref(s_i)}, i = 1, \ldots n)$ with $d_i \neq d_j$ for $i \neq j$, and corresponding list of elements $a_w = \langle a_1, \ldots, a_n \rangle \quad (a_i \in A_{s_n}, i = 1, \ldots n)$, the **state** $A^{[d_w := a_w]}$ on A is the free extension of A by the A–equation $(\lambda :!(d_1) \stackrel{e}{=} a_1 \wedge \ldots \wedge !(d_n) \stackrel{e}{=} a_n)$, that is $A^{st} = A^{[d_w := a_w]}$ is defined by the following universal property.

- There is a *SPEC*–homomorphism $\eta^{st} : A \to A^{st}$.
- $A^{st}_!(d_i) = a_i \quad (i = 1, \ldots n)$
- For each partial *SPEC*–algebra B and *SPEC*–homomorphism $h : A \to B$ that satisfy $B_!(h(d_i)) = h(a_i) \quad (i = 1, \ldots n)$ there is a unique *SPEC*–homomorphism $h^* : A^{st} \to B$ with $h^* \circ \eta^{st} = h$.

State transitions are induced by methods, whose effect is to change the dereferencing functions. The references that are to be update or created, and further values that may influence the effect are given to the method as parameters. A **method signature** *MetSIG = (M,C)* w.r.t. a given base specification *BS = (S,OP,CE,ref)* is given by a family of sets $M = (M_{w,p})_{w \in dom(ref)^*, p \in S^*}$, the *set of methods*, and a family of subsets $C = (C_{w,p})_{w \in dom(ref)^*, p \in S^*}$, with $C_{w,p} \subseteq M_{w,p}$ for all $w \in dom(ref)^*, p \in S^*$, the *set of create methods*. Methods $m \in M_{w,p}$ are denoted $m : ref(w);\ p$. If $m \in M_{w,p} - C_{w,p}$, m is called a *transformation method*.

Methods are specified by conditional parallel assignments. The corresponding definitions are similar to conditional equations where the conclusion is replaced by an assignment to indicate the semantic difference. A **method definition** $def = (v : r_1 \stackrel{e}{=} r_2 \to m(\Delta;\ t) := r)$ w.r.t. a base specification *BS* and a corresponding method signature *MetSIG = (M,C)* is given by

- a *variable declaration* $v = s'_1 \ldots s'_k$, that represents $x_1 : s'_1, \ldots, x_k : s'_k$,

- a *condition* $r_1 \stackrel{e}{=} r_2$, given by lists of terms r_1, r_2 of the base specification with variables declared by v, that specifies whether the method can be applied,
- a *syntactic method expression* $m(\Delta;\ t)$, given by a method name m, a list of reference constants or variables $\Delta = \langle \Delta_1, \ldots, \Delta_n \rangle$ that are to be updated or created, and parameter terms $t = \langle t_1, \ldots, t_l \rangle$, and
- a *right hand side* term $r = \langle r_1, \ldots, r_n \rangle$, that specifies the (new) contents for $\Delta = \langle \Delta_1, \ldots, \Delta_n \rangle$.

A **transition specification** *TS = (BS,MetSPEC)* is given by a base specification *BS* and a **method specification** *MetSPEC = (MetSIG,DEF)*, where *DEF* is a set of method definitions w.r.t. *MetSIG* and *BS*.

Each method m with parameters $d_w = \langle d_1, \ldots, d_n \rangle \in A_{ref(w)}$ and $a_p = \langle a_1, \ldots, a_n \rangle \in A_p$ defines a relation $A^{m(d_w;\ a_p)}$ on the set of states of a base model A. To obtain this relation, i.e. the semantics of the method, we have to match the left hand side $m(\Delta;\ t)$ of a definition $def = (v : r_1 \stackrel{e}{=} r_2 \rightarrow m(\Delta;\ t) := r)$ against a suitable syntactic expression of a current state A^{st}. If m is a transformation method the contents of d_w must be defined in A^{st}, i.e. $A^{st} = A^{[\langle d_w, d_u \rangle := \langle a_w, a_u \rangle]}$. If A^{st} satisfies the condition $r_1 = r_2$ of def under a variable assignment $x_v \mapsto a_v$ that also maps $m(\Delta;\ t)$ to $m(d_w;\ a_p)$, then we can look for an element $b_w \in A_w$ from the base model whose image $\eta^{st}(b_w) = b_w^{st}$ in A^{st} is the evaluation of the right hand side r of def in A^{st}. This value b_w is then used to define the successor state $A^{st'} = A^{[\langle d_w, d_u \rangle := \langle b_w, a_u \rangle]}$.

In the following definition that rephrases this explanation formally we use the notation $x^{st} = \eta^{st}(x)$ for any element $x \in A$, where $\eta^{st} : A \rightarrow A^{st}$ is the universal morphism of A^{st}.

Let be given

- a transformation method expression $m(d_w;\ a_p)$ with $m \in M_{w;\,p} - C_{w;\,p}$, $d_w \in A_{ref(w)}$, $a_p \in A_p$
- a state $A^{st} = A^{[\langle d_w, d_u \rangle := \langle a_w, a_u \rangle]}$
- a method definition $def = (v : r_1 \stackrel{e}{=} r_2 \rightarrow m(\Delta;\ t) := r)$
- an element $a_v \in A_v$ (that yields a variable assignment $x_v \mapsto a_v^{st} = asg(x_v) \in A^{st}$) and an element $b_w \in A_w$, such that
  - Δ and t evaluate to d_w^{st} and a_p^{st} resp. in A^{st}:
    $A_\Delta^{st}(a_v^{st}) = d_w^{st}$ and $A_t^{st}(a_v^{st}) = a_p^{st}$
    (resp. $asg^\sharp(\Delta) = d_w^{st}$ and $asg^\sharp(t) = a_p^{st}$)
  - A^{st} satisfies the condition of def:
    $A_{r_1}^{st}(a_v^{st}) = A_{r_2}^{st}(a_v^{st})$
    (resp. $asg^\sharp(r_1) = asg^\sharp(r_1)$)

– the right hand side r of def evaluates to b_w^{st} in A^{st}:
$A_r^{st}(a_v^{st}) = b_w^{st}$
(resp. $asg^\sharp(r) = b_w^{st}$)

then

- $m(d_w;\ a_p) : A^{[\langle d_w, d_u\rangle := \langle a_w, a_u\rangle]} \ \Rightarrow\ A^{[\langle d_w, d_u\rangle := \langle b_w, a_u\rangle]}$

is a transition.

For a create method expression $c(d_w;\ a_p),\quad c \in C_{w;\,p}$ the first three points have to be replaced by

- a create method expression $c(d_w;\ a_p)$ with $c \in C_{w;\,p}$, $d_w \in A_{ref(w)}$, $a_p \in A_p$
- a state $A^{st} = A^{[\langle d_u\rangle := \langle a_u\rangle]}$
with $\{d'_1, \ldots, d'_m\} \cap \{d_1, \ldots, d_n\} = \varnothing$
for $d_w = \langle d_1, \ldots, d_n\rangle$ and $d_u = \langle d'_1, \ldots, d'_m\rangle$,
i.e. the references d_w must not be defined in A^{st}
- a method definition $def = (v : r_1 \stackrel{e}{=} r_2 \rightarrow c(\Delta;\ t) := r)$

then

- $c(d_w;\ a_p) : A^{[\langle d_u\rangle := \langle a_u\rangle]} \ \Rightarrow\ A^{[\langle d_w, d_u\rangle := \langle b_w, a_u\rangle]}$

is a transition.

A model of a transition specification *TS = (BS,MS)* is the transitive reflexive closure of the transition relation on the states of a base model defined above. Formally this yields a category, the **transition category** of A. Transition functors between transition categories of base models A and B can be defined as homomorphisms $h : A \rightarrow B$ that are compatible with the state transitions. One of the central results in [4] is that each base-homomorphism extends to a transition functor and that the categories of transition categories, the loose semantics of a transition specification, and the category of base models are equivalent. This allows to carry over all results concerning the semantics and compositionality of specifications from the algebraic case to transition specifications.

3 Transition Specifications as "models" for Z

In this section we discuss how to construct a transition specification from a given one written in the Z notation. Key issues are represented by a detailed discussion on how Z-schemas

translate into algebraic transition specification types, by a (informal) comparison of the formal semantics of both specifications, and the implementation of the Z toolkit library by parameterized algebraic data types.

3.1 Z by the numbers – main ideas

The Z notation is a language and a style for expressing formal specification of computing systems[1]. It is based on a typed set theory and first order logic, and the notion of a "schema" is one of its key features. A schema consist of a collection of named objects (variables) with a relationship specified by some axioms. Schemas are used to record both the static and dynamic aspects aspects of the system.

The static aspects include:

- the states it can occupy;
- the invariant relationships that are maintained as the system moves from state to state;

while the dynamic ones include:

- the operations that are possible;
- the relationships between their inputs and outputs, and
- the changes of states that can happen.

As a very simple example to illustrate this construct look at the following schema called *Alef* :

Alef

$$x, y : \mathbb{Z}$$

$$x < y$$

The part above the central dividing line consists of the *declaration* part and the one below, corresponds to the *predicate* part, where invariant assertions (if any) about the declared variables are given. It's appropriate to mention that the *Z* type system requires that any variable must be declared along with information about its type, where the type determines a set over which the variable ranges. More precisely, in Z certain basic types like $\mathbb{Z}$ (the integers) and $\mathbb{N}$ (natural numbers) and sets constructed recursively from these ones by cartesian products, powersets, etc., are called types. Further type constructors, like taking all partial relations from a type A to a type B are considered as subtypes of the powerset of $A \times B$, constrained by a predicate that describes, e.g., single-valuedness for partial functions.

[1] By computing systems we mean here the declaration of the system's state space together with a collection of operations that might access or change the information content of this state space.

Coming back to our example, the part above the horizontal line declares the variables x, y as being of type $\mathbb{Z}$ and the property $x < y$ relating the values of both variables asserts that the value of x should be less than of y.

Roughly speaking, the mathematical meaning of each schema can be viewed as the collection of all bindings of "suitable values" to the variables such that the axioms hold. In fact, the underlying technicalities needed to speak about such "bindings" are a bit more involved than the preceding declaration might imply, and to speak more precisely about it, we need first to introduce concepts like *signature*, *schema type*, and *assignments*.

A *signature* is a collection of variables, each with a type. Signatures are created by declarations and they provide a vocabulary for making mathematical statements which are expressed by predicates (thus, a schema can also be seen as a signature together with a property over the signature). For instance, looking at the schema *Alef*, we have that the declaration $x, y : \mathbb{Z}$ creates a signature with two variables x and y, both of type $\mathbb{Z}$. Each signature is naturally associated with a schema type; for example, the signature created by declaring $x, y : \mathbb{Z}$ is associated with the schema type $[\![x, y : \mathbb{Z}]\!]$. Note that two schema types are regarded as identical if they differ only in the order in which the components are listed.

The values in this type are bindings in which the variables take different values drawn from their types. The bindings are called *assignments*. A *property* over the signature is characterized by the set of bindings under which is true. For instance, the property expressed by $x < y$ is true under the binding $\langle x \Rrightarrow 3, y \Rrightarrow 5 \rangle$ and false under the binding $\langle x \Rrightarrow 6, y \Rrightarrow 4 \rangle$. Considering this example, the collection of all bindings which make $x < y$ a true statement is called the collection of *models* (or *variety*) – after Spivey [7] – of the schema *Alef*. More generally, if $x_1, \ldots, x_n$ are distinct identifiers and $p_1, \ldots, p_n$ are objects of type $t_1, \ldots, t_n$ respectively, then there is a binding $z = \langle x_1 \Rrightarrow p_1, \ldots, x_n \Rrightarrow tpn \rangle$ with components $z.x_i = p_i$ for $1 \leq i \leq n$.

Without entering into further details of Z, it is to worth mention that a Z specification consists of interleaved passages of formal, mathematical text and informal prose explanation. The formal text consists of a sequence of *paragraphs* which gradually introduce the schemas, global variables (if any), and basic types of the specification, each paragraph building on the ones that come before it. What exactly these paragraphs are is a task for our next discussion, namely, the introduction of two simple Z specifications and their corresponding presentation as transition specifications.

3.2 Constructing the translation

Going directly to a simple, yet still clarifying example of a Z specification, consider the specification of a "Birthday Book" (taken from [8]). The Birthday Book is a system which records people's birthdays. We shall need to deal with people's names and with dates. The Z specification, declaring the basic types, as well as the schemas for the abstract state space and the operations are given below. We adopt the Z style of writing specifications and provide a brief informal explanation of each paragraph right after its declaration. A brief discussion about the mathematical meaning of each construct will also take place.

$[NAME, DATE]$

This first paragraph is a declaration of the basic types of the the specification, i.e., *NAME* and *DATE* are the indivisible, atomic objects of our specification. Moreover, these basic types are to be understood as the basic *given* sets of this specification, whose internal structure we don't care about. A model of $[NAME, DATE]$ is a pair of sets, and the semantics of $[NAME, DATE]$ is the class of all such pairs.

BirthdayBook

$$known : \mathbb{P}\,(NAME)$$
$$birthday : NAME \pfun DATE$$
$$known = dom(birthday)$$

This schema serves the purpose of describing the abstract state space of the Birthday Book. In this specific case, we have a declaration of a variable *known* which has type $\mathbb{P}\,(NAME)$ and a variable *birthday* denoting a partial function from the set of names to the set of dates, which here is indicated by the symbol $\pfun$. The symbol $\pfun$ stands for a a so-called generic constant, a Z construct than can be viewed as a type constructor parameterized over sets. It is important here to emphasize that in Z, functions are identified with their graphs, and therefore they are just special kinds of relations. Since in Z a relation, say from X to Y has type (is a member of the set) $\mathbb{P}(X \times Y)$, our variable *birthday* should be understood has having type $\mathbb{P}(NAME \times DATE)$. Declarations in *Z* are relying heavily in the use of these generic constants (as defined in the *Z* toolkit library [8]). Hence, the slogan is: *given a declaration $y : S$ is always possible to discover the type of y by looking at the definition of S (in this case, the definition of $\pfun$) and seeing what type its members must be. Now, y is constrained to satisfy the defining properties of S.*

Therefore, the defining properties from an element (relation) of the set $\mathbb{P}(NAME \times DATE)$ which make it single-valued, and thus a partial function, should be understood as *contributing to the property* in the predicate part of the above schema. So this additional information, property, expressed in the declaration of a variable is called the *constraint* of the variable. As an illustration to this discussion, *BirthdayBook* could be declared with all the implicit information added, as

BirthdayBook$*$

$$known : \mathbb{P}\,(NAME)$$
$$birthday : \mathbb{P}(NAME \times DATE)$$
$$known = dom(birthday)$$
$$\forall\, birthday \in \mathbb{P}(NAME \times DATE);\ n : NAME;\ d_1, d_2 : DATE \bullet$$
$$(n \mapsto d_1) \in birthday \land (n \mapsto d_2) \in birthday \Rightarrow d_1 = d_2$$

On the other hand, the equation in the predicate part of the schema just says that in all possible states of the system the value of *known* should be equal to the domain of the partial function *birthday*. However, note that the invariant allows the value of the variable *known*

to be derived from that of *birthday*. The variable *known* is here a so-called (after Spivey [8]) *derived component* of the state, and this means that it would be possible to specify the system without mentioning it at all. The use of derived components has the sole purpose of making the specifications more readable, especially when such derived components are important abstract concepts in the system's specification. An important note here: in the discussions we shall go through in this section, we will abstract from such derived components, that is to say, we will assume that each variable declaration in the schema describing the abstract state is meaningful on its own. Later, in section 3.3, we shall have a detailed discussion on how to treat them and the immediate implications in the construction of the transition specification. We consider treating the case of derived components here only an unessential idiosyncrasy, since the main ideas concerning the construction of a transition specification from a Z one are completely independent from this observation.

Note that, according to our previous discussion, the schema type associated with *Birthday Book* is

$$(\!| \mathit{known} : \mathbb{P}(\mathit{NAME}), \mathit{birthday} : \mathbb{P}(\mathit{NAME} \times \mathit{DATE}) |\!).$$

Let be given, in the following discussion, two arbitrary but fixed sets for *NAME* and *DATE*. Now, the semantics of *BirthdayBook* w.r.t. to the above schema type can be understood as the set of all type-compatible assignments to the two variables such that the value of *known* is equal to the domain of the variable birthday. As an example, observe that the binding

$$\begin{array}{ll} \langle & \mathit{known} \Rrightarrow \{\mathit{Alfio}, \mathit{Martin}, \mathit{Uwe}\}, \\ & \mathit{birthday} \Rrightarrow \{\mathit{Alfio} \mapsto 26 - \mathit{Jun}, \mathit{Martin} \mapsto 11 - \mathit{May}, \mathit{Uwe} \mapsto 19 - \mathit{Oct}\} \quad \rangle \end{array}$$

is a model of *BirthdayBook* whereas

$$\begin{array}{ll} \langle & \mathit{known} \Rrightarrow \{\mathit{Alfio}, \mathit{Martin}, \mathit{Uwe}, \mathit{Magdalena}\}, \\ & \mathit{birthday} \Rrightarrow \{\mathit{Alfio} \mapsto 26 - \mathit{Jun}, \mathit{Martin} \mapsto 11 - \mathit{May}, \mathit{Uwe} \mapsto 19 - \mathit{Oct}\} \quad \rangle \end{array}$$

is not. Models of *BirthdayBook* can also be understood as "the possible" states of the system.

AddBirthday

$\Delta \mathit{BirthdayBook}$
$\mathit{name}? : \mathit{NAME}$
$\mathit{date}? : \mathit{DATE}$

$\mathit{name}? \notin \mathit{known}$
$\mathit{birthday}' = \mathit{birthday} \cup \{\mathit{name}? \mapsto \mathit{date}?\}$

This schema describes an operation that increases the database with a new birthday. The declaration part of any "operation schema" in Z consists of two copies of declarations of the state space: an undecorated set corresponding to the state of the system before the operation, and a dashed set corresponding to the state after the operation together with the declaration of the needed input and output variables for the specification of this operation. To make it

more convenient to denote these two copies of the abstract state space, there is an (officially) accepted convention in Z that whenever a schema *State* is introduced as a declaration of the system's state space, the schema $\Delta State$ is implicitly defined as the combination of *State* and $State'$, unless a different definition is made explicitly:

$$\begin{array}{|l} \Delta State \\ \hline State \\ State' \\ \hline \end{array}$$

which is to say that we have all declarations of *State*, and the predicate relating the variables declared therein, together with another set of declarations for a corresponding set of dashed variables, and the equivalent predicate relating those variables.

Thus, declaration $\Delta BirthdayBook$ says that the schema is describing a *state change*: it introduces four variables $known, birthday, known'$, and $birthday'$. The first two are observations of the state before the change, and the last two are observations of the state after the change. Each pair of variables is implicitly constrained to satisfy the invariant, so it must hold both before and after the operation. Note that this means that the equation $known' = dom(birthday')$ that specifies the "after" value of the variable $known'$ is, by the above discussion, implicitly present in the predicate part.

Next come the declarations of the two inputs to the operation. By convention, the names of inputs end in a question mark "?". The part of the schema below the line gives first a pre-condition for the realization of the operation. In this case, *name* must not belong to the domain of the database. Once this requirement is satisfied, the new entry is put into the database.

The schema type associated with *AddBirthday* is

$$\begin{array}{ll} (\!| & known, known' : \mathbb{P}(NAME), \\ & birthday, birthday' : \mathbb{P}(NAME \times DATE), \\ & name? : NAME, date? : DATE \qquad\qquad |\!) \end{array}$$

The semantics associated with this schema type consists of the set of all type-compatible assignments to the above variables such that both the axioms listed in the predicate part of *BirthdayBook* and *AddBirthday* as well as the following one should be satisfied:

$$known' = dom(birthday')$$

(see discussion about the Δ-convention). Moreover, it can be shown that the equation

$$known' = known \cup \{name\}$$

holds (as expected) after the execution of *AddBirthday*. This can be proved as a theorem $AddBirthday \vdash known' = known \cup \{name\}$ using the specification *AddBirthday*, the invariant, and some elementary algebraic laws associated with the generic constants *dom* and $\subseteq$.

Adding Magdalena's birthday (17-Nov) to the database implies that the following assignment is a model of the schema *AddBirthday*.

$$\begin{array}{ll}\langle & known \Rrightarrow \{Alfio, Martin, Uwe\}, known' \Rrightarrow \{Alfio, Martin, Uwe, Magdalena\}, \\ & birthday \Rrightarrow \{Alfio \mapsto 26-Jun, Martin \mapsto 11-May, Uwe \mapsto 19-Oct\}, \\ & birthday' \Rrightarrow \{Alfio \mapsto 26-Jun, Martin \mapsto 11-May, Uwe \mapsto 19-Oct, \\ & Magdalena \mapsto 17-Nov\}, \\ & name? \Rrightarrow Magdalena, date! \Rrightarrow 17-Nov \qquad \rangle\end{array}$$

Hence, semantics of Z schemas that stands for operations emphasizes two objects, the "before" and the "after" state.

FindBirthday

$\Xi BirthdayBook$
$name? : NAME$
$date! : DATE$

$name? \in known$
$date! = birthday(name?)$

Here we have an operation which, given a name known to the database, returns the corresponding birthday's date, where in Z's convention, variables for output are decorated with the exclamation mark "!". Analogously to the Δ-convention, the declaration $\Xi BirthdayBook$ corresponds to two copies of the state space *BirthdayBook*, the decorated and the undecorated set, but now, with the essential difference that for each variable in the declaration part of *BirthdayBook*, the additional equation $variable' = variable$ is implicitly assumed after the post-condition $date! = birthday(name?)$. This means essentially that the state does not change. Thereby, from now on, we consider the Δ and Ξ conventions understood, and besides, assume this convention as "mandatory" in schemas describing operations that might either change or just read the state, respectively.

The schema type associated with *FindBirthday* is

$$\begin{array}{ll}\langle\!| & known, known' : \mathbb{P}(NAME), \\ & birthday, birthday' : \mathbb{P}(NAME \times DATE), \\ & name?, date! : DATE \qquad |\!\rangle\end{array}$$

The semantics associated with this schema type consists of all type-compatible assignments to the above variables such that, besides the axioms listed in the schema, the following ones should hold:

$$\begin{array}{l} known' = known \\ birthday' = birthday \\ known' = dom(birthday') \end{array}$$

i.e, the assignment of type-compatible values for the state variables and its corresponding dashed copies should be equal. Moreover, *name* should be bound to one of the names of

the set *known* and *date* to the corresponding birthday, as dictated by the partial function *birthday*.

Remind

$$\Xi BirthdayBook$$
$$today? : DATE$$
$$cards! : \mathbb{P}\,(NAME)$$

$$cards! = \{n : known \mid birthday(n) = today?\}$$

This schema is again a reading operation (this time without any pre-condition) that specifies an output variable *cards*! as being equal to the set of all values n drawn from the set *known* such that the value of the birthday function at n is equal to the input variable *today*?.

The schema type associated with *Remind* is the following one:

$$\langle\!\!\langle \quad known, known' : \mathbb{P}(NAME),\ birthday, birthday' : \mathbb{P}(NAME \times DATE),\ today? : DATE, cards! : \mathbb{P}(NAME) \quad \rangle\!\!\rangle$$

The semantics here is, up to change of the axioms in the predicate part, the same to the one explained above.

InitBirthday

$$BirthdayBook$$

$$birthday = \varnothing$$

The above schema simply states what the system is when it is first started. The declaration *BirthdayBook* means here the inclusion of the schema *BirthdayBook*, so as the above one is just an extension of *BirthdayBook* by the predicate $birthday = \varnothing$. It describes a birthday book in which the partial function *birthday* is the empty function. Note that this implies that $known = \varnothing$ too!

The schema type associated with *InitBirthday* is the same as the one associated with *BirthdayBook*. The difference here is that the the set of models is restricted to one and only one, i.e., to the one which assigns the empty set to both variables.

We shall now begin the discussion of how to express the above specification in transition specifications. We will make it component-wise, that is to say, we will examine each of the component schemas of the specification and discuss in detail how a corresponding transition specification might be derived.

We point out again that in Z, every object is a typed set, where functions are notably just special kinds of relations. Since we are interested, as far as possible, in translating Z-signatures into transition specification-signatures and Z-axioms into algebraic ones, we must think in a way to express the variables declarations *known* and *birthday* directly into the signature of a transition specification.

First note that *birthday*, being a partial function (and hence a relation), has type $\mathbb{P}(NAME \times DATE)$. Our idea is to make every variable in the abstract state a reference constant to its corresponding type (sort) in the signature of the transition specification. But this immediately implies that we have to declare a sort that stands for the set of partial functions from $NAME$ to $DATE$. To implement this idea, we have specified an algebraic type for relations mimicking the Z-toolkit library. The idea is that this specification is parameterized over basic parameter types and extends two other parameterized specifications for sets and product of specifications. The complete specifications themselves are presented in the appendix. We list here just the signature part of them, where we adopt the syntax of the algebraic specification language ACT ONE [1] to write our specifications.

The parameterized specification for sets of data is realized as follows (where DATAEQ is a parameterized specification extending a basic specification BOOL for boolean values, declaring a sort *Data*, and an operation symbol to test equality of elements of the sort *Data*):

type SET[DATAEQ] $=_{def}$
  extend
    INIT(NAT)
  by
    sorts
      $Set(Data)$
    constructs
      $emptyset :\rightarrow Set(Data)$
      $\{\_\} : Data \rightarrow Set(Data)$
      $\_ \cup \_ : Set(Data), Set(Data) \rightarrow Set(Data)$
    axioms
      $\vdots$
    functions :
      $\_\ eq\_ : Set(Data), Set(Data) \rightarrow Bool$
      $\_ \in \_ : Data, Set(Data)$
      $isin : Data, Set(Data) \rightarrow Bool$
      $\_ \subseteq \_ : Set(Data), Set(Data) \rightarrow Bool$
      $notin : Data, Set(Data) \rightarrow Bool$
      $isempty : Set(Data) \rightarrow Bool$
      $\vdots$
  endext
endtype

Two notes about the semantics of our parameterized specifications for sets (and also to the ones for relations and functions below):

- Although the meaning of a transition specification *takes place in the world of partial algebras*, we consider here the operations from our (auxiliary) algebraic types (sets, relations and functions) as *being total*. Formally, we would need to add the so-called "totalization axioms" for each one of them. To avoid such despairing enterprise, we will

use the arrow $\multimap$ to denote explicitly when the declaration of an operation symbol stands for a partial function.

- We assume neither a loose semantics nor an initial one for this specification. Loose semantics would allow models which might not have the intended interpretation of the set operations. On the other hand, a usual initial semantics for parameterized specifications, i.e., by means of a free functor, might only be able to denote finitely generated sets. However, in Z, any set $S' \subseteq S$ is understood as being member of $\mathbb{P}\, S$, i.e., S' might also denote an (non-constructed) infinite object. We solve this problem by using a suitable datatype constructor (see discussion below). We make exceptions for the specifications BOOL for boolean values, and NAT of natural numbers, which should be interpreted *initially* ([2, 3]), a fact that for the case of natural numbers is denoted above by the keyword INIT. Initial interpretation for NAT for instance, means that its class of models (algebras) is restricted to the algebras isomorphic to the natural numbers $(\mathbb{N}, 0, +1, \ldots)$. The main fact underlying an initial constraint on booleans is that in this way we are able to test both for identity and non-identity of elements, the latter an essential requirement in any reasonably relevant specification problem.

Besides, note that, in the above parameterized specification, we have two operation symbols to test the membership of an element, namely $\_ \in \_ : Data, Set(Data)$ as a predicate (see discussion on transition specification, in the previous section) and $isin : Data, Set(Data) \rightarrow Bool$ as a boolean-valued function. The need for the second arises mainly by the need (and desire) to specify its negation (see appendix), i.e., the operation $notin : Data, Set(Data) \rightarrow Bool$. However, apart from this, we shall continue to use just the predicate version, since in this way we won't have to worry about a possible "destruction" of the sort *Bool*, allowing us to decrease substantially the number of equations needed to specify operations requiring this operation symbol.

For a type (specification) SPEC, we denote by $Mod(SPEC)$ the category of all SPEC-algebras together with all homomorphisms between them. $\mid Mod(SPEC) \mid$ denotes the associated class of algebras.

The powerset datatype constructor $\wp : \mid Mod(DATAEQ) \mid \rightarrow \mid Mod(SET) \mid$ is defined for each DATAEQ-algebra $\mathcal{A} = (A_{Data}, \ldots) \in \mid Mod(DATAEQ) \mid$ (i.e., essentially a set with an equality operation defined on it) by

$$\begin{array}{lll} \wp(A)_{Data} & =_{def} & A_{Data} \\ \wp(A)_{Set(Data)} & =_{def} & \mathbb{P}(A_{Data}) \end{array}$$

where the operations are defined as expected. This means that the class of SET-models contains algebras

$$\mathcal{A} = (A_{Data}, A_{Set(Data)}, \ldots)$$

where $A_{Set(Data)}$ denotes the set of all subsets of A_{Data}, and dots stand for the expected functions. In the sequel we consider these algebras as the intended models of the specification. The translation is then correct w.r.t. the intended model.

Now, assuming specifications A, B – renamings of DATAEQ –, parameterized specifications $\times[\mathsf{A},\mathsf{B}], \times[\mathsf{B},\mathsf{A}]$ (see appendix) for declaration of products of sorts, a parameterized algebraic specification for relations of type $A \times B$ can be given as follows (to improve readability, we provide alongside with each signature's function symbol the corresponding operation in the Z notation):

type $\mathsf{RELATION}[\mathsf{A},\mathsf{B}] =_{def}$
  extend
    union $\mathsf{SET}[\mathsf{A}], \mathsf{SET}[\mathsf{B}], \mathsf{SET}[\times[\mathsf{A},\mathsf{B}]], \mathsf{SET}[\times[\mathsf{B},\mathsf{A}]]$ **endunion**
    rename using
      sortnames
        $Rel(A \times B)$ **for** $Set(A \times B)$
        $Rel(B \times A)$ **for** $Set(B \times A)$
    endren
  by
    functions
      $rel\_image : Set(A), Rel(A \times B) \to Set(B)$ $(\llparenthesis\ \rrparenthesis)$
      $dom : Rel(A \times B) \to Set(A)$ (dom)
      $ran : Rel(A \times B) \to Set(B)$ (ran)
      $inv\_R : Rel(A \times B) \to Rel(B \times A)$ (R^{-1})
      $dom\_restr : Set(A), Rel(A \times B) \to Rel(A \times B)$ $(\lhd)$
      $ran\_restr : Rel(A \times B), Set(B) \to Rel(A \times B)$ $(\rhd)$
      $dom\_antirestr : Set(A), Rel(A \times B) \to Rel(A \times B)$ $(\lhd\!\!\!-)$
      $image\_antirestr : Set(A), Rel(A \times B) \to Rel(A \times B)$ $(\rhd\!\!\!-)$
      $override : Rel(A \times B), Rel(A \times B) \to Rel(A \times B)$ $(\oplus)$
  endext
endtype

The class of $\mathsf{RELATION}$-models contains algebras

$$\mathcal{A} = (A_A, A_B, A_{Set(A)}, A_{Set(B)}, A_{A\times B}, A_{B\times A}, A_{Rel(A\times B)}, \ldots)$$

where for instance, $A_{Rel(A\times B)}$ denotes the set of all subsets of $A_{A\times B}$.

Now, the specification of an algebraic type for partial functions becomes a trivial task. We just have to extend the type for relations with an axiom to express single-valuedness. This leads to the following specification:

type $\mathsf{PFN}[\mathsf{A},\mathsf{B}] =_{def}$
  rename
    extend *RELATION* **by**
      axioms {functions are singled-valued relations}
        $\forall\, a : A, b, b' : B, f : Rel(A \times B)$
        $\langle a, b\rangle \in f \wedge \langle a, b'\rangle \in f \Rightarrow b = b'$
      functions

$eval : A, Rel(A \times B) \to B$
$\forall\, a : A, b : B, f : Rel(A \times B)$
$\langle a, b \rangle \in f \Rightarrow b = eval(a, f)$

endext

using

sortnames

$Pfn(A \times B)$ **for** $Rel(A \times B)$

$Pfn(B \times A)$ **for** $Rel(B \times A)$

endren

endtype

The class of PFN-models are characterized by all RELATION-models where the elements of $A_{Rel(A \times B)}$ are single-valued relations, i.e., essentially partial functions.

Having done this, we may now start to construct the corresponding transition specification from the Birthday Book example. For each basic type, e.g., *NAME*, we generate a corresponding parameter algebraic type with equality. This is obtained straightforwardly by renaming the parameter type DATAEQ as follows:

parameter $=_{def}$

rename DATAEQ **using**

sortnames *Name* **for** *Data*

endren

endpar

The semantics of NAME is the collection of all NAME-algebras, i.e., the collection of all sets which have an equality operation defined on it. This contradicts a bit with the meaning of the declaration [*NAME*] in Z, which denotes any set, simply because in *Z* the whole universe of predicate logic is "always" available! Thus, the restriction of NAME-algebras to the sort *Name* coincides with the Z-semantics of [*NAME*].

The algebraic type corresponding to *BirthdayBook* may now be suitably specified as a parameterized type over the basic types *NAME*, *DATE*, where the parameterized type PFN[A, B] is actualized by these basic types. Now *birthday* becomes a reference to $Pfn(Name \times Date)$ and *known* a reference to $Set(Name)$. The corresponding specification for the type BIRTHDAYBOOK is then realized as follows:

type BIRTHDAYBOOK[NAME, DATE] $=_{def}$

extend

actualize

PFN **by union** NAME, DATE **endunion**

using

sortnames

Name **for** *A*

Date **for** *B*

endact

by

sorts

$ref(Pfn(Name \times Date))$

constructs

$birthday : \rightarrow ref(Pfn(Name \times Date))$

$known : \rightarrow ref(Set(Name))$

functions

$!_{Set(Name)} : ref(Set(Name)) \multimap Set(Name)$

$!_{Pfn(Name \times Date)} : ref(Pfn(Name \times Date)) \multimap Set(Name \times Date)$

axioms

$!known \stackrel{s}{=} dom(!birthday)$

endext

endtype

Note that the invariant of the Z schema meets a natural translation to an axiom in the basic type of the transition specification, with the only care needed being an appropriate enquiry about the definedness of the references *birthday* and *known* before accessing its content (recall discussion about strong equality in section 2).

A base model for BIRTHDAYBOOK is a partial algebra

$$\mathcal{A} = (A_{Name}, A_{Date}, A_{Set(Name)}, A_{Pfn(Name \times Date)}, A_{ref(Pfn(Name \times Date))}, \ldots)$$

which satisfies the axiom in the above specification as well as all the equations in the specification PFN. The "dots" stand for the remaining base sets and expected functions. Note that $A_{known}, A_{birthday}$, are reference constants, $A_{Pfn(Name \times Date)}$ is a set of partial functions, and $A_{!Set(Name)}, A_{!Pfn(Name \times Date)}$ are partial functions which are everywhere undefined. Note also, for instance, that in Z the values of variables range over sets, whilst modeled as references in transition specifications, they "point" to these sets. Moreover, since we are only interested in manipulating references that represent the original variables from *BirthdayBook*, we also assume that in our base models, the base sets $A_{ref(Set(Name))}, A_{ref(Pfn(Name \times Date))}$ are singletons, i.e., their unique elements are, respectively, the constants A_{known} and $A_{birthday}$.

The discussion we had before about constraints of declarations and generic constants is the very right argument here to explain this somehow "bursting effect" on the information needed to describe the meaning of the state in the transition specification. That is to say, all the information implicitly assumed in the Z schema, namely, the constraint of the declaration of *birthday* (the defining properties of $\rightarrow\!\!\!\!\!\!+$) together with defining properties of the generic constant *dom* are made syntactically visible in the corresponding transition specification. Note that, given a base model $\mathcal{A}$ for the above specification, we have that the partial contents functions are totally undefined. Therefore, in order to make a comparison with the semantics of the previous Z-construct, we need to consider any possible state $\mathcal{A}^{st}$. Now, each state algebra $\mathcal{A}^{st}$ with respect to a given base model $\mathcal{A}$, when restricted to its contents partial functions symbols and the obvious needed sorts, is essentially a model of *BirthdayBook*. To see it, just assign the contents of the references to the variables in *BirthdayBook*.

Operation	Inputs/Outputs	Pre-conditions
FindBirthday	$name? : NAME$ $date! : DATE$	$name \in known$
Remind	$today? : DATE$ $cards! : \mathbb{P}(Name)$	

Table 1: *BirthdayBook*: – I/O information for the reading operations

The next schema corresponds to the reading operation *FindBirthday*. This will become an operation on the basic type. The idea is quite simple: by looking at the declaration part of *FindBirthday* we get information about its input/output signature. Since we have another reading operation, namely *Remind*, we summarize in Table 1 the input, output, and pre-condition from each of these operations.

From this table we get immediately the signature of the corresponding operation symbols, i.e., $Findbirthday : Name \multimap Date$ and $Remind : Date \to Set(Name)$. We avoid here unnecessary discussions on how to translate the axioms in the Z specification to the algebraic one, since the axioms present in the examples treated here are essentially (conditional) equational ones. Since we have at our disposal the whole toolkit for relations specified as algebraic types, their formulation in the corresponding transition specification becomes a straightforward task. The corresponding transition specification type, BIRTHDAYBOOK02 is just an extension of BIRTHDAYBOOK by these two operations. It is realized as follows:

type BIRTHDAYBOOK02[NAME, DATE] $=_{def}$
  extend BIRTHDAYBOOK
    by
    functions
      $FindBirthday : Name \multimap Date$
        $\forall\, name : Name : !birthday \uparrow \wedge name \in !birthday$
        $\Rightarrow FindBirthday(name) = eval(name, !birthday)$
      $Remind : Date \to Set(Name)$
        $\forall\, today : Date, n : Name :$
        $!birthday \uparrow \Rightarrow Remind(today) = rel\_image(today, inv\_R(!birthday))$
  endext
endtype

An algebra of the specification BIRTHDAYBOOK02 is just like a BIRTHDAYBOOK-algebra, but now with two additional functions $A_{FindBirthday}$ and A_{Remind}. Although reading operations don't affect the state, the semantics of the corresponding Z-schemas emphasizes "well-shaped" structures that account for the state before and after the execution of the operation. This leads to a redundant model that describes two times the same object. In transition specifications this operations are to be seen as genuine functions, local to the base model, i.e., they are operations which can access the state and deliver results when given suitable arguments. The meaning of such operations simply vanishes in the semantics of the corresponding schemas. There, the "state" is the essential object of interest. Besides, note that *FindBirthday* is a

Operation	Updated variables	Inputs
AddBirthday	$birthday : \mathbb{P}(NAME \times DATE)$	$name : NAME, date : DATE$

Table 2: *BirthdayBook*: – signature/parameter information for dynamic methods

partial function which, by its definition, depends essentially on the contents referenced by *birthday*. Now, since *birthday* is redefined in every new state, it follows that *FindBirthday* is also a new function in every such state. Note also that this is only possible because the meaning of all these algebraic types lie in the "world of partial algebras".

We now come to the point where we really can use the full power of transition specification by translating a state change operation into a dynamic (transformation) method. The signature of each dynamic method is obtained as follows:

- the list of references to be updated is in one-to-one correspondence with the variables that have their contents rewritten according to the specification of the operation (since the contents of references which are not affected by the execution of a dynamic method have, by the semantics of a transition specification, their contents unchanged). This means that this set of variables is equal to the difference between the set of variables from the basic state and the set of primed variables which are equal to their respective unprimed ones;
- the list of parameters are in one-to-one correspondence with the list of declared input variables.

In table 2 we list, for *AddBirthday*, the corresponding updated variables and list of parameters. The types of the variables in both columns deliver the signature of the method, while their names the corresponding list of arguments to it.

Having said this, the corresponding signature for the dynamic type ADDBIRTHDAY is $ref(Pfn(Name \times Date));\ Name, Date$, and the complete specifications is as follows:

dyntype ADDBIRTHDAY $=_{def}$
  extend BIRTHDAYBOOK **by**
    method
      $AddBirthday : ref(Pfn(Name \times Date));\ Name, Date$
    defn
      $\forall\, name : Name, date : Date :$
      $notin(name, !known) = true \Rightarrow$
      $AddBirthday(birthday;\ name, date) :=!birthday \cup \{\langle name, date\rangle\}$
  endext
enddyntype

Recall from the discussion in section 2, that the semantics of dynamic transformation method is essentially a state transition which is achieved by redefinition of the partial contents functions, i.e., an assignment of new contents to a given list of references. Recall also

that the current state of the system must always be accessed from the base model by means of the universal morphism, since in the base model the contents partial functions are everywhere undefined. To ease understanding with the notation, we will follow here all the steps which describe the semantics of transformation methods as presented in 2. Now, let the method expression $AddBirthday(A_{birthday}, Magdalena, 17-Nov)$ stand for the introduction of Magdalena's birthday in the database, and let $\mathcal{A}^{st}$ be the current state of BIRTHDAYBOOK.

First, note that $\Delta = birthday$, $a_p = \langle Magdalena, 17-Nov\rangle$, and $X = \{name, date\}$. Let also $asg : X \to \mathcal{A}^{st}$ be such that $\{name \mapsto Magdalena^{st}, date \mapsto 17-Nov^{st}\}$. Now we have:

- $asg^\sharp(birthday) = A^{st}_{birthday}$, $asg^\sharp(\langle name, date\rangle) = \langle Magdalena, 17-Nov\rangle^{st}$, i.e., $birthday$ and $\langle name, date$ are evaluated in the current state w.r.t. to the variable assignment $asg : X \to \mathcal{A}^{st}$, and thus, $birthday$ and $\langle name, date\rangle$ match w.r.t. to the variable assignment, $A_{birthday}$ and $\langle Magdalena, 17-Nov\rangle$;
- $asg^\sharp(notin(name, !known)) = asg^\sharp(true)$, i.e., the current state $\mathcal{A}^{st}$ must be such that Magdalena's name does not belong to the set of known names;
- Let $b \in A_{Pfn(Name\times Date)}$ be the current value of $A_{birthday} \in \mathcal{A}^{st}$, i.e.,

  $$A^{st}_{!Pfn(Name\times Date)}(A^{st}_{birthday}) = b^{st}.$$

  Then we have:

  $$\begin{array}{rcl} asg^\sharp(!birthday \cup \{\langle name, date\rangle\}) & = & b^{st} \cup \{\langle Magdalena, 17-Nov\rangle^{st}\} \\ & = & (b \cup \{\langle Magdalena, 17-Nov\rangle\})^{st} \end{array}$$

  and $b \cup \{\langle Magdalena, 17-Nov\rangle\} \in A_{Pfn(Name\times Date)}$ is the new contents of $A_{birthday}$.
- now, the successor state $A^{st'}$ is given, after execution of

  $$AddBirthday(A_{birthday};\ \langle Magdalena, 17-Nov\rangle)$$

  by

  $$\mathcal{A}^{[\langle A_{birthday},\ldots\rangle := \langle b,\ldots\rangle]} \Longrightarrow \mathcal{A}^{[\langle A_{birthday},\ldots\rangle := \langle b\cup\{\langle Magdalena,17-Nov\rangle\},\ldots\rangle]}$$

Now, given a fixed base model $\mathcal{A}$, the transition model – where the state algebras are restricted to the sorts $Set(Name)$ and $Pfn(Name \times Date)$ to throw away explicit information about sets, constraints and so on (as already discussed in BIRTHDAYBOOK) – coincides with the reflexive transitive closure induced by the schema $AddBirthday$. Note that we assume here, that the given base sets of the Z-specification and the corresponding base sets A_{Name}, A_{Date} of the given base model $\mathcal{A}$ of the transition specification are the same.

Methods which initialize the system's state space are here considered as create methods. By the same token, the dynamic method is given as follows (where we now have no input variables):

dyntype INITBIRTHDAY $=_{def}$
  extend BIRTHDAYBOOK **by**
    createmeth
      $InitBirthday : ref(Pfn(Name \times Date))$
    defn
      $InitBirthday(birthday) := emptyset$
  endext
enddyntype

Recall that create methods increase the domain of definition of the partial contents functions. Since, as discussed before, the base set $A_{ref(Pfn(Name \times Date)}$ is a singleton, i.e., its unique element is the reference constant $A_{birthday}$, the execution of the above method should be intuitively understood as generating the initial state of the corresponding transition system. Thus, let $InitBirthday(A_{birthday})$ be the unique method expression w.r.t. to the above method definition. Since the method definition does not contain neither variable declarations nor any pre-condition, we proceed as follows:

- $InitBirthday(A_{birthday})$ is the only instance (method expression);
- $asg^\sharp(birthday) = A_{birthday}$;
- $asg^\sharp(emptyset) = \varnothing$, where $\varnothing \in Set(Name)$;
- $InitBirthday(A_{birthday})$ can only be applied when $A^{st}_{birthday} \notin dom(A^{st}_{!Pfn(Name \times Date)})$;
- now, the initial state $\mathcal{A}^{st}$, given after execution of $InitBirthday(A_{birthday})$, is the following one:

$$\mathcal{A}^{[\langle\,\rangle := \langle\,\rangle]} \cong (\mathcal{A})$$
$$\Longrightarrow$$
$$\mathcal{A}^{[\langle A_{birthday}\rangle := \langle\varnothing\rangle]} \cong (\mathcal{A}^{[\langle A_{birthday}, A_{known}\rangle := \langle\varnothing,\varnothing\rangle]})$$

where the second isomorphism follows directly from the axiom in the base specification BIRTHDAYBOOK02.

We shall now discuss a further, more involving example, taken from [6], for the specification of a library system.

The library comprises a stock of books and a community of registered readers. Each copy of a book is assigned a unique copy identifier. At any time, a certain number of copies of books are on loan to readers; the remainder of the stock is on the shelves of the library and available for loan. There is a maximum number of books which any reader may have on loan at any one time. Some of the desired operations are the ones for issuing a copy of a book to a registered reader, returning of a book to the library from a reader, adding and removing a copy of a book to the stock, enquiring which reader has a particular copy of a book, and registering a new reader.

We present here (a part of) the Z specification and, as before, after each paragraph, a brief explanation of what it is meant to specify. Since we have already made a succinct discussion about the meaning of Z specification constructs in the previous example, we'll take the freedom to be more succinct here.

$$[COPY, BOOK, READER]$$

These are our given sets, or basic types, whose inner components we don't care about.

$$maxloans : \mathbb{N}$$
$$\hline$$
$$maxloans = 10$$

The above construct is something new for us. It is called, within Z terminology, an *axiomatic description* (something like an "unnamed schema"), whose scope is from the point where it is declared to the end of the specification. It is used to introduce variables along with predicates giving further information. For us, the above declaration means that *maxloans* is a global variable which denotes the maximum number of books which a reader may have on loan at any given time. Here this maximum is taken to be (equal to) ten. This axiom contributes to a global property of the specification.

More generally, in order to understand a specification with global variables, it is helpful trying to imagine fixing one binding for them, so that they take fixed values that satisfy the global property. The property of each schema then restricts the components of the schema to take values that bear a certain relationship to the values of the global variables.

Understood in this way, a specification describes a family with one member for each binding that satisfies the global property. If this family has more than one member, we say that the specification is *loose*.

Library
$$stock : COPY \pfun BOOK$$
$$issued : COPY \pfun READER$$
$$shelved : \mathbb{F}\, COPY$$
$$readers : \mathbb{F}\, READER$$
$$\hline$$
$$shelved \cup \operatorname{dom} issued = \operatorname{dom} stock$$
$$shelved \cap \operatorname{dom} issued = \varnothing$$
$$\operatorname{ran} issued \subseteq readers$$
$$\forall\, r : readers \bullet \#(issued \rhd \{r\}) \leq maxloans$$

This schema describes the state space of our library system. The declaration part says that *stock*, *issued*, *readers*, *shelves* are the variables which constitute the state space. In more detail, *stock* is a partial function that records which books are associated with each copy identifier currently in use; *issue* is also a partial function that records which copies are on loan and to whom; *readers* is the set of registered readers, while *shelved* is the subset of copies on the shelves and available for issue. The declaration $\mathbb{F}\, COPY$ stands for all finite subsets of $COPY$ (and similarly for *readers*).

Again, $\mathbb{F}$ stands here for a generic constant which defined in the Z toolkit library as a parameterized construct, that given a set X, returns the set of all finite subsets of X. Therefore, according to our previous discussion about types and sets in Z, the maximal set here is $\mathbb{P}(X)$, and hence the type of *shelved* and *readers*.

The predicate part of *Library* may be briefly explained like this:

- the first two conjuncts state that all copies are either on loan or on the shelves but not both, i.e., *shelved* and $dom(issued)$ partition the domain of *stock*;
- the third conjunct says that loans are only made to registered readers;
- the final conjunct says that the number of loans to any registered reader must be no greater than the maximum numbers of loans allowed.

The schema type associated with *Library* records information for the name and type of the above four variables and also for the global variable *maxloans*. The set of type-compatible assignments which are models of this schema are all the ones which both set the value 10 for *maxloans* and satisfy the axioms in the predicate part of the schema.

Issue

$$\Delta Library$$
$$c? : COPY;\ r? : READER$$

$$c? \in shelved;\ r? \in readers;\ \#(issued \rhd \{r\}) < maxloans$$
$$issued' = issued \oplus \{c? \mapsto r?\}$$
$$stock' = stock;\ readers' = readers$$

Here we have an operation for issuing a copy of a book to a registered reader. The declaration $\Delta Library$, as already discussed, is saying to us that this is a state changing operation. Besides, it also declares input variables for a copy identifier and for a reader. The pre-condition of the operation says that the copy to be issued must be initially on the shelves, that the reader must be registered, and finally that the reader must have less than the maximum number of books allowed before this copy may be issued. Once the pre-condition holds, the copy is recorded as issued to the reader. The remaining equations say only that neither *stock* nor the set of *readers* are changed.

The schema type associated with *Issue* should record information about the names and types of the the global variable *maxloans*, the four variables of *Library*, their corresponding dashed copies, and also for the variables listed in the declaration of *Issue*. Again, the semantics of this schema should be understood as the set of all type-compatible assignments to the above mentioned variables such that the value of *maxloans* equals 10, both the axioms listed in *Library*, and their corresponding dashed ones, as well as the ones from *Issue* are satisfied.

WhoHasCopy

$\Xi Library$
$c? : COPY;\ r! : READER$

$c? \in \mathrm{dom}\ issued$
$r! = issued\ c?$

This schema describes a simple "reading" operation that, given an input variable for a copy identifier, delivers the corresponding associated reader.

In this schema there is no reference to the global variable *maxloans*. Therefore, the schema type associated with *WhoHasCopy* should just record information about the names and types of the variables declared in *Library*, their corresponding dashed copies. as well as the ones declared in the above schema. The corresponding semantics consists of all type-compatible assignments to the above mentioned variables such that besides the axioms listed in the predicate part of *WhoHasCopy* and *Library* also the following (trivial) dashed ones should hold:

$stock' = stock$
$issued' = issued$
$shelved' = shelved$
$readers' = readers$

AddKnownTitle

$\Delta Library$
$b? : BOOK$

$b? \in \mathrm{ran}\ stock$
$\exists\, c : COPY \mid c \notin \mathrm{dom}\ stock \bullet stock' = stock \oplus \{c \mapsto b?\} \land$
$\quad shelved' = shelved \cup \{c\}$
$issued' = issued;\ readers' = readers$

Here we have again a state changing operation which adds a new copy of a book already existent to the library. The pre-condition says essentially that the input variable b must be a book already existent in the library, and additionally, if there is a free identifier for a copy of a book available, then this operation records that this identifier refers to the book of which this is a new copy. Besides, this new copy is put on the shelves. The remaining equations assert that the loans and the set of readers are unchanged.

The above schema again makes no reference to global variable *maxloans*. We leave to the reader to think about the models of this one!

We begin now our discussion on how to transform the above specification to an equivalent one in the transition specification framework. We won't make a detailed discussion here, since up to one construct (the axiomatic description), the procedure is analogous to the one done in the *BirthdayBook* example.

For the basic types, we generate the corresponding algebraic parameters, as for instance:

parameter BOOK $=_{def}$
  rename DATAEQ **using**
    sortnames *Book* **for** *Data*
  endren
endpar

The semantics here, as discussed before, is the collection of all DATAEQ-algebras.

In Z, an axiomatic description is a global declaration. Since in algebraic specifications in general, we don't have such kind of construct, we generate a corresponding algebraic type MAXLOANS which extends an assumed given type for natural numbers. The specification for this is straightforward:

type MAXLOANS $=_{def}$
  extend INIT(NAT) **by**
    constructs
      $maxloans :\rightarrow Nat$
    axioms
      $maxloans = succ^{10}(0)$
  endext
endtype

where the semantics here is the collection of all algebras isomorphic to the algebra of natural numbers with an additional constant that stands for the number 10.

Again, the basic state is transformed into a basic type in the transition specification which is parameterized over BOOK, COPY, READER. It is extended by MAXLOANS, so as to make its declarations available, and by two actualizations of PFN; one to deliver a partial function from *Copy* to *Book* and another one from *Copy* to *Reader*. The four variables of the state space are generating references to their respective sorts. The translation of the invariants are self-explanatory. We also make a one-step construction to include also the reading operation *WhohasCopy*. Note that the partial contents functions are implicitly assumed.

type LIBRARY[COPY, BOOK, READER] $=_{def}$
  extend
    union
      MAXLOANS,
      actualize
        FINSET[A] **by** COPY
        using
          sortnames
            Copy **for** *A*
      endact,
      actualize
        FINSET[A] **by** READER

```
            using
                sortnames
                    Reader for A
            endact,
            actualize
                PFN by union COPY, BOOK endunion
                using
                    sortnames
                        Copy for A
                        Book for B
            endact,
            actualize
                PFN by union COPY, READER endunion
                using
                    sortnames
                        Copy for A
                        Reader for B
            endact
        endunion
    by
        sorts
            ref(Pfn(Copy × Book)),
            ref(Pfn(Copy × Reader)),
            ref(FinSet(Reader)), ref(FinSet(Copy))
        constructs
            stock : → ref(Pfn(Copy × Book))
            issued : → ref(Pfn(Copy × Reader))
            readers : → ref(FinSet(Reader)), shelved : → ref(FinSet(Copy))
        axioms
            !shelved ∪ dom(!issued) =^s dom(!stock)
            !shelved ∩ dom(!issued) =^s ∅
            ran(!issued) ⊆!readers =^s true
            ∀ r : Reader :!issued ↑⇒ card(ran_restr(!issued, {r})) ≤ maxloans = true
        functions
            WhoHasCopy : Copy → Reader
                ∀ c : Copy :!issued ↑∧ c ∈ dom(!issued) ⇒
                WhoHasCopy(c) = eval(c, !issued)
    endext
endtype
```

The semantics of partial base specifications, dynamic types as well as their relation to the Z semantics have been already discussed at length in the previous example. A complete analogous discussion would also be valid here.

In Table 3 we list, for *Issue* and *AddKnownTitle*, the corresponding updated variables and list of parameters. The types of the variables in both columns deliver the signature of

Operation	Updated variables	Inputs
Issue	$issued : \mathbb{P}(COPY \times READER)$	$c : BOOK, r : READER$
AddKnownTitle	$stock : \mathbb{P}(COPY \times BOOK)$ $shelved : \mathbb{P}(COPY)$	$b : BOOK$

Table 3: Library's system – signature/parameter information for dynamic methods

the method, while their names the corresponding list of arguments to it.

By looking at the above table and taking into account our previous discussion, the specification of the following dynamic methods is an easy task:

dyntype ISSUE $=_{def}$
  extend LIBRARY **by**
    method
      $Issue : ref(Pfn(Copy \times Reader));\ Copy, Reader$
    defn
      $\forall\, c : Copy, r : Reader :$
      $c \in !shelved \wedge r \in !readers \wedge card(range\_restr(!issued, \{r\})) < maxloans$
      $\Rightarrow Issue(issued;\ c, r) := override(!issued, \{\langle c, r\rangle\})$
  endext
enddyntype

dyntype ADDKNOWNTITLE $=_{def}$
  extend LIBRARY **by**
    method
      $AddKnownTitle :$
      $ref(Pfn(Copy \times Book)), ref(FinSet(Copy));\ Copy, Book$
    defn
      $\forall\, b : Book, c : Copy : b \in ran(!stock) \wedge notin(c, dom(!stock)) \Rightarrow$
      $AddKnownTitle(stock, issued, c, b) :=$
      $\langle override(!stock, \{\langle c, b\rangle\}), !shelved \cup \{c\}\rangle$
  endext
enddyntype

3.3 Compositionality and derived components

The reader may have observed how nice it was to be able to construct a transition specification by going from schema to schema. However, this procedure was only made feasible because we have abstracted from the so-called derived components.

For instance, in the *BirthdayBook* example, we have declared the variable *known* as a reference to $Set(Name)$, even though this reference was never explicitly updated in any of the dynamic methods of the corresponding transition specification. This might be already

expected by the corresponding Z specification, where the invariant $known = dom(birthday)$ express clearly that the contents of *known* are directly depending on *birthday*. Therefore, our decision of considering every declared variable in the state space of a given Z specification as a reference in the corresponding transition specification is prone to (some) criticism.

So we might refine our procedure a little bit, and set that only the variables in the state space which are not derived components generate reference constants in the transition specification. The derived component themselves, which can be deduced by looking at the invariants, are declared in the basic state.

Once we proceed like this in our first example, we eliminate the declaration of the sort $Set(Name)$ and $known : \rightarrow ref(Set(Name))$. Now *known* becomes a predicate and is declared as $known : Set(Name)$ in the **functions** part of the basic transition specification. Besides, since *known* is not a reference anymore, the invariant has to be rewritten as

$$known \stackrel{s}{=} dom(!birthday).$$

These changes put now the transition specification in complete accordance with the Z specification, since the previous declaration of *known* as a reference was in some sense a little redundant.

So far so good. Once we apply this very same principle to the second example, the library system, we run into difficulties. Let us see why. Firstly, as the attent reader may have noticed, *shelved* is, by the first two invariants, essentially a derived component, a fact which might be more explicitly stated by the equation

$$shelved = dom(stock) \setminus dom(issued).$$

Eliminating the declaration of *shelved* as a reference to $FinSet(Copy)$ and declaring it as a predicate would not work. The problem would arise later, in the construction of the dynamic method ADDKNOWNTITLE, since in the corresponding Z schema, an explicit actualization of *shelved* is performed by this operation, although it is clear that actualizations in the contents of derived components is a redundant task. Once it is clear that in transition specifications such actualization is only possible once *shelved* is a reference, we reach the undesirable conclusion that even a careful analysis of the state space of a Z specification may be insufficient in order to judge which variables are indeed relevant to the specification as a whole, i.e., which variables are indeed generating references in the corresponding transition specification. Worse than this, such information might only be able to be established after inspection of all schemas describing operations that change the state. Note also, that the library's system Z specification is somehow very subtle, since in the specification of the operation *Issue*, the equation $shelved' = shelved$ is not explicitly stated, and this means that the operation is only well-defined once one admits that this equation follows from the invariants, attesting after all, that *shelved* must indeed be a derived component.

Using this procedure, we may conclude that in the first example, only *birthday* becomes a reference, and in the second, all of them, as it was the case.

Summarizing, we have three possibilities:

1. We proceed as in the two discussed examples and consider, without further concern, that every variable declared in the state space of the Z specification as a meaningful reference. As it was shown in the corresponding discussions, the approach is both "compositional" and simple.

2. We analyze the state space by looking at the invariant relations in order to determine which of the declared variables are derived components. These are declared as operations (predicates) in the basic specification of the transition specification, and the remaining ones are generating references. Later, when translating operations which change the state, redundant actualizations of derived components are simply ignored. The advantage of this approach is that we keep the "compositionality" of our procedure. The disadvantage is that, as far as syntax is concerned, we may loose a bit the "faithfulness" of the translated transition specification with respect to the original Z specification.

3. We look to the whole set of schemas which describe state change operations in order to determine which of the variables from the state space actually have their contents changed. These are the ones generating references while the remaining ones are declared as operations and thus considered as derived components[2]. The advantage of this approach is that it generates a transition specification which possesses a faithful syntactical resemblance with the original Z specification. The obvious disadvantage is that we loose the compositionality of our procedure.

Taking into account the above considerations, it is reasonable to accept that the first alternative it is the most adequate, not only because it is "compositional" and the simplest one, but also because it is not the role of this work to judge how far a so-called derived component must or must not explicitly "take part" in each operation that might read or change the state space of a Z specification. This decision is clearly a specifier's choice, and as long as the specification itself is well-defined piece of Z text (both syntactically and semantically), it does not matter from the point of view of a transition specification the existence of these derived components. Thus, each variable declared in the state space is considered as a meaningful piece of information about the system's abstraction.

4 Concluding Remarks & Further work

A translation like the one presented in this paper will always be more or less informal, as long as the aim is not to write a compiler. Since transition specifications are a specification theory, and (up to now) there is no tool support, we see no need in making the translation more formal than it is. The general idea, its range and limits should be clear by now.

The discussion of the examples should have made the following general pattern of the translation apparent.

basic types are translated as renamings of the parameter specification DATAEQ of sets with equality. Equality as boolean operation, as well as a boolean subtype, is required,

[2]Taking into account at least a "well-defined" Z specification

because there are no implicit pervasive types in a transition specification, and inequality is needed in most of the Z–specifications, e.g. to formulate preconditions for operations.

names (variables) of Z–specifications are mapped to references of the transition specification. Their (contents) sort is the counterpart of the Z–type in the algebraic specification (of sets etc.) of the Z–toolkit library, taking into account the implicit constraints given by Z's generic constants, i.e. the contents sort of a name $A \nrightarrow B$ is $Pfn(A \times B)$, not $\mathbb{P}(A \times B)$.

axiomatic descriptions are treated like named schemas, where a name has to be invented. The concept of pervasive or global types is not available in transition specifications. Every type that is needed has to be imported explicitly.

read operations i.e. operations that are indicated by a Ξ are translated as dependent functions (read functions) of the base specification of the transition specification. Their specification by axioms also becomes a part of the base specification. Since the behavior of a read operation usually depends on the current state these axioms effect only in the states, where the references have contents.

operations indicated by Δ, i.e. operations that indeed may change the state are mapped to methods. Their parameters and the references they update are deduced from the Z–specification as seen in the examples.

A problem arises here if an operation has a result variable *result!*. In a transition specification result variables for methods are not provided, but there are two ways to model them. Either a further reference is introduced and the value result of the operation is assigned to this reference in parallel with the method execution. Or a read function (a constant) is introduced in the base specification whose value in each state is the value result of the operation. This can be deduced from the Z–specification as in the examples. Transition specifications could, of course, also be extended by this feature, that allows value results of methods.

axioms are directly translated as in the examples, as long as they are algebraic (i.e. conditional existence equations). Further encodings, like Skolemization, or taking parameters whose existence is required but not further specified as actual parameters of the operations (cf. the *AddKnownTitle*–operation), also yield a translation.

The structuring mechanisms available for transition specifications mentioned in the introduction — *locality* and *compositionality* — now can be transported backwards along the translation, and show how similar structurings can be obtained for Z–specifications. First of all, the locality principle for transition specifications means for a Z–specification, that names declared within one schema never have a meaning in another schema. Thus coincidence of names is treated as accidental, and identifications of names from different schemas, whether identical or not, have to be specified explicitly. The units of locality, specifications resp. schemas, are the same in Z and in transition specifications, as shown by the stepwise translation.

Composition of schemas could be defined in the same way as composition operations for (algebraic or) transition specifications, using specification morphisms to express the re-

lationship between schemas, diagrams for configurations, and their colimits as results of the composition operations.

Separation of concerns meant that the distinction between static and dynamic parts, i.e. functions without side effect and operations or methods that change the state, is already indicated syntactically and cannot be overwritten in any further extension or development of the same specification. If a function in a later specification phase turns out to be a state changing operation a genuine re–design of the concerned part would be required then.

One major drawback of the translation presented in this paper is, that a lot of the argumentation about a Z–specification depends on the underlying set theory. Since this is not made explicit in Z, via a proof system or axiomatization, many hand made proofs are possible in Z. In the translation, however, everything is explicit, and sets are simply inadequate as a data type for an algebraic specification. It is not possible to specify exactly the models, where $A_{Set(Data)}$ is the power set of A_{Data}. In the free generated models the infinite subsets are missing, and arbitrary models may have not intended interpretations of the set operations.

However, it has not been the purpose of this paper to discuss algebraic specifications of sets. The conclusion that we draw from this obstacle is to take a more direct approach, that doesn't take the Z–semantics as literally as in the translation presented here. The idea is to consider arbitrary partial functions directly as dynamic, and let state changes be defined by the way the dynamic functions are redefined. In the current approach only the dereferencing function (partial contents functions) are dynamic, and their definition determines the state. A more general approach, where arbitrary functions can be declared as dynamic, has been presented meanwhile ([5]). Thus e.g. a partial function like *birthday* need not be encoded as a relation with certain constraints, whose redefinition is obtained by set operations. It can directly be considered as a dynamic function, that is redefined directly by function updates whose semantics is defined uniformly in the semantics of transition specifications in the extended sense. The details of this translation are the subject of a subsequent paper.

Acknowledgments: We are very grateful to Uwe Wolter who carefully read this work, spotted mistakes here and there, and made very useful suggestions that improved significantly the presentation of this paper.

A Algebraic types for specification of sets, relations, and functions

In this appendix we present the complete underlying algebraic specification which was mostly assumed in the discussion in this report. Its intent is to show a possible way in which the Z toolkit library for sets, relations and functions might be specified in this framework. This specification is here written according to syntax of ACT-ONE [1]. Readers with a reading knowledge of algebraic specification should find ACT-ONE syntax self-explanatory. The key words $\mathsf{INIT(BOOL)}, \mathsf{INIT(NAT)}, \mathsf{INIT(SET)}$, wherever they appear, stand, respectively, for a initial interpretation of the specifications for boolean values, natural numbers, and sets.

parameter DATAEQ $=_{def}$
  extend INIT(BOOL)
    sorts
      $Data$
    operations
      $\_\ eq\ \_ : Data, Data \rightarrow Bool$
    axioms
      $\forall\, a, b : Data :$
      $a\ eq\ b = true \Rightarrow a = b$
      $a = b \Rightarrow a\ eq\ b = true$
  endext
endpar

type SET[DATAEQ] $=_{def}$
  extend
    INIT(NAT)
  by
    sorts
      $Set(Data)$
    constructs
      $emptyset :\rightarrow Set(Data)$
      $\{\_\} : Data \rightarrow Set(Data)$
      $\_ \cup \_ : Set(Data), Set(Data) \rightarrow Set(Data)$
    axioms
      $\forall\, A, B, C : Set(Data) :$
      $A \cup (B \cup C) = (A \cup B) \cup C$
      $\forall\, A, B : Set(Data) :$
      $A \cup B = B \cup A$
      $\forall\, A : Set(Data) :$
      $emptyset \cup A = A$
      $\forall\, A : Set(Data) :$
      $A \cup A = A$
    functions :
      $\_\ eq\ \_ : Set(Data), Set(Data) \rightarrow Bool :$

$\forall A, B : Set(Data) :$
$(A \subseteq B) = true \ and \ (B \subseteq A) = true \Leftrightarrow A \ eq \ B = true$

$isin : Data, Set(Data) \rightarrow Bool$
$\forall a : Data, A, B : Set(Data) :$
$isin(a, (A \cup B)) = isin(a, A) \ or \ isin(a, B)$
$isin(a, \{b\}) = a \ eq \ b$
$isin(a, emptyset) = false$

$\_ \in \_ : Data, Set(Data) \rightarrow Bool$
$\forall a : Data, A : Set(Data) :$
$isin(a, A) = true \Rightarrow a \in A$
$a \in A \Rightarrow isin(a, A) = true$

$\_ \subseteq \_ : Set(Data), Set(Data) \rightarrow Bool$
$\forall a : Data, A, B, C : Set(Data) :$
$emptyset \subseteq A = true$
$\{a\} \subseteq A = isin(a, A)$
$(A \cup B) \subseteq C = (A \subseteq C) \ and \ (B \subseteq C)$

$notin : Data, Set(Data) \rightarrow Bool$
$\forall a : Data, A : Set(Data) :$
$\{a\} \subseteq A = false \Rightarrow notin(a, A) = true$
$\{a\} \subseteq A = true \Rightarrow notin(a, A) = false$

$isempty : Set(Data) \rightarrow Bool$
$\forall A : Set(Data) :$
$isempty(A) = A \ eq \ emptyset$

$\_ \cap \_ : Set(Data), Set(Data) \rightarrow Set(Data)$
$\forall a : Data, A, B, C : Set(Data) :$
$emptyset \cap C = emptyset$
$isin(a, C) = true \Rightarrow \{a\} \cap C = \{a\}$
$isin(a, C) = false \Rightarrow \{a\} \cap C = emptyset$
$(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$

$insert : Data, Set(Data) \rightarrow Set(Data)$
$\forall a : Data, A : Set(Data)$
$insert(a, A) = \{a\} \cup A$

$- : Set(Data), Set(Data) \rightarrow Set(Data)$
$\forall a : Data, A, B : Set(Data) :$
$isin(a, A) = true \wedge isnot(a, B) = true \Rightarrow isin(a, A - B) = true$
$isin(a, A - B) = true \Rightarrow isin(a, A) = true \wedge isnot(a, B) = true$

endext
endtype

type FINSET[DATAEQ] $=_{def}$
rename
extend INIT(SET) *by*
functions
$card : Set(Data) \to Nat$
$\forall\, A : Set(Data) :$
$card(emptyset) = 0$
$card(\{a\}) = 1$
$card(A \cup B) = (card(A) + card(B)) - card(A \cap B)$
endext
using
sortnames $FinSet(Data)$ **for** $Set(Data)$
endren
endtype

parameter A $=_{def}$
rename DATAEQ
using
sortnames A **for** $Data$
endren
endpar

parameter B $=_{def}$
rename DATAEQ
using
sortnames B **for** $Data$
endren
endpar

type $\times$[A, B] $=_{def}$
sorts
$A \times B$
constructs
$\langle \_ , \_ \rangle : A, B \to A \times B$
$\pi_A : A \times B \to A$
$\pi_B : A \times B \to B$
axioms
$\forall\, a : A, b : B : \pi_A(\langle a, b \rangle) = a$
$\forall\, a : A, b : B : \pi_B(\langle a, b \rangle) = b$
$\forall\, c : A \times B : \langle \pi_A(c), \pi_B(c) \rangle = c$
functions
$\_\ eq\ \_ : A \times B, A \times B \to Bool$
$\forall\, a, a' : A, b, b' : B :$
$eq(\langle a, b \rangle, \langle a', b' \rangle) = a\ eq\ a'\ and\ b\ eq\ b'$
endtype

type SET[A] $=_{def}$
actualize SET **by** A
using
sortnames A **for** $data$
endact
endtype

type SET[×[A, B]] $=_{def}$
actualize SET **by** ×[A, B]
using
sortnames $A \times B$ **for** $data$
endact
endtype

In the following we assume to have parameter types SET[B], SET[×[B, A]] by suitable actualizations just as before.

type RELATION[A, B] $=_{def}$
extend
union SET[A], SET[B], SET[×[A, B]], SET[×[B, A]] **endunion**
rename using
sortnames
$Rel(A \times B)$ **for** $Set(A \times B)$
$Rel(B \times A)$ **for** $Set(B \times A)$
endren
by
functions
$dom : Rel(A \times B) \to Set(A)$
$\forall\, a : A, b : B, R : Rel(A \times B) :$
$\langle a, b\rangle \in R \Rightarrow a \in dom(R)$
$a \in dom(R) \Rightarrow \langle a, b\rangle \in R$

$ran : Rel(A \times B) \to Set(B)$
$\forall\, a : A, b : B, R : Rel(A \times B) :$
$\langle a, b\rangle \in R \Rightarrow b \in ran(R)$
$b \in ran(R) \Rightarrow \langle a, b\rangle \in R$

$inv\_R : Rel(A \times B) \to Rel(B \times A)$
$\forall\, a : A, b : B, R : Rel(A \times B)$
$\langle a, b\rangle \in R \Rightarrow \langle b, a\rangle \in inv\_R(R)$
$\langle b, a\rangle \in inv\_R(R) \Rightarrow \langle a, b\rangle \in R$

$rel\_image : Set(A), Rel(A \times B) \to Set(B)$

$\forall\, a : A, b : B, S : Set(A), R : Rel(A \times B) :$
$a \in S \wedge \langle a, b\rangle \in R \Rightarrow b \in rel\_image(S, R)$
$b \in rel\_image(S, R) \Rightarrow a \in S \wedge \langle a, b\rangle \in R$

$dom\_restr : Set(A), Rel(A \times B) \rightarrow Rel(A \times B)$
$\forall\, a : A, b : B, S : Set(A), R : Rel(A \times B) :$
$a \in S \wedge \langle a, b\rangle \in R \Rightarrow \langle a, b\rangle \in dom\_restr(S, R)$
$\langle a, b\rangle \in dom\_restr(S, R) \Rightarrow a \in S \wedge \langle a, b\rangle \in R$

$ran\_restr : Rel(A \times B), Set(B) \rightarrow Rel(A \times B)$
$\forall\, a : A, b : B, R : Rel(A \times B), S : Set(B) :$
$b \in S \wedge \langle a, b\rangle \in R \Rightarrow \langle a, b\rangle \in ran\_restr(R, S)$
$\langle a, b\rangle \in ran\_restr(R, S) \Rightarrow b \in S \wedge \langle a, b\rangle \in R$

$dom\_antirestr : Set(A), Rel(A \times B) \rightarrow Rel(A \times B)$
$\forall\, a : A, b : B, S : Set(A), R : Rel(A \times B) :$
$isnot(a, S) = true \wedge \langle a, b\rangle \in R \Rightarrow \langle a, b\rangle \in dom\_antirestr(S, R)$
$\langle a, b\rangle \in dom\_antirestr(S, R) \Rightarrow isnot(a, S) = true \wedge \langle a, b\rangle \in R$

$image\_antirestr : Set(A), Rel(A \times B) \rightarrow Rel(A \times B)$
$\forall\, a : A, b : B, R : Rel(A \times B), S : Set(B) :$
$isnot(b, S) = true \wedge \langle a, b\rangle \in R \Rightarrow \langle a, b\rangle \in ran\_antirestr(R, S)$
$\langle a, b\rangle, ran\_antirestr(R, S) \Rightarrow isnot(b, S) = true \wedge \langle a, b\rangle \in R$

$override : Rel(A \times B) \rightarrow Rel(A \times B) \rightarrow Rel(A \times B)$
$\forall\, f, g : Rel(A \times B) :$
$override(f, g) = dom\_restr(dom(f) - dom(g), f) \cup g$

endext
endtype

type TRANSRELATION[A, B] $=_{def}$
extend
actualize SET **by** $\times$[A, A]
using
sortnames $A \times A$ **for** *data*
endact
by
functions
$refl\_cl : Rel(A \times A) \rightarrow Rel(A \times A)$
$\forall\, a : A, R : Rel(A \times A) :$
$\langle a, a\rangle \in refl\_cl(R)$
$tran\_cl : Rel(A \times A) \rightarrow Rel(A \times A)$
$\forall\, a, b, c : A, R : Rel(A \times A) :$
$R \subseteq tran\_cl(R) = true$
$\langle a, b\rangle \in tran\_cl(R)$ *and* $\langle b, c\rangle \in tran\_cl(R)$
$\Rightarrow \langle a, c\rangle \in tran\_cl(R)$

$refl\_trans\_cl : Rel(A \times A) \to Rel(A \times A)$
$\forall\, R : Rel(A \times A)$
$refl\_tran\_cl(R) = refl\_cl(R) \cup tran\_cl(R)$

endext

endtype

type COMPRELATION[A, B] $=_{def}$

extend

union RELATION[A, B], SET[×[B, C]], SET[×[A, C]] **endunion**

rename using

sortnames

$Rel(B \times C)$ **for** $Set(B \times C)$

$Rel(A \times C)$ **for** $Set(A \times C)$

endren

by

functions

$\_ \circ \_ : Rel(B \times C), Rel(A \times B) \to Rel(A \times C)$
$\forall\, a : A, b : B, c : C, R : Rel(A \times B), R' : Rel(B \times C) :$
$\langle a, b\rangle \in R \wedge\ \langle b, c\rangle \in R' \Rightarrow \langle a, c\rangle \in R' \circ R$

endext

endtype

type PFN[A, B] $=_{def}$

rename

extend RELATION **by**

axioms {functions are singled-valued relations}

$\forall\, a : A, b, b' : B, f : Rel(A \times B)$
$\langle a, b\rangle \in f\ and\ \langle a, b'\rangle \in f \Rightarrow b = b'$

functions

$eval : A, Rel(A \times B) \to B$
$\forall\, a : A, b : B, f : Rel(A \times B)$
$\langle a, b\rangle \in f \Rightarrow b = eval(a, f)$

endext

using

sortnames

$Pfn(A \times B)$ **for** $Rel(A \times B)$

$Pfn(B \times A)$ **for** $Rel(B \times A)$

endren

endtype

type TFN[A, B] $=_{def}$

rename

extend PFN **by**

functions

$f : A, Pfn(A \times B) \to B$ {skolem fun}

axioms
$\{\forall\, a : A, g : Pfn(A \times B) \exists\, b : B.\langle a, b\rangle \in g\}$
$\forall\, a : A, g : Pfn(A \times B) : \langle a, f(a, g)\rangle \in g$
endext
using
sortnames
$Tfn(A \times B)$ **for** $Pfn(A \times B)$
$Tfn(B \times A)$ **for** $Pfn(B \times A)$
endren
endtype

type INJPFN[A, B] $=_{def}$
rename
extend PFN **by**
axioms {axiom for injectivity }
$\forall\, a, a' : A, b : B, f : Pfn(A \times B) :$
$\langle a, b\rangle \in f \wedge \langle a', b\rangle \in f \Rightarrow a = a'$
endext
using
sortnames
$InjPfn(A \times B)$ **for** $Pfn(A \times B)$
$InjPfn(B \times A)$ **for** $Pfn(B \times A)$
endren
endtype

type INJTFN[A, B] $=_{def}$
rename
extend TFN **by**
axioms {axiom for injectivity }
$\forall\, a, a' : A, b : B, f : Pfn(A \times B) :$
$\langle a, b\rangle \in f \wedge \langle a', b\rangle \in f \Rightarrow a = a'$
endext
using
sortnames
$InjTfn(A \times B)$ **for** $Tfn(A \times B)$
$InjTfn(B \times A)$ **for** $Tfn(B \times A)$
endren
endtype

References

[1] I. Claßen, H. Ehrig, and D. Wolz. *Algberaic Specification Techniques and Tools for Software Development - The ACT Approach.* AMAST Series in Computing Vol. 1. World Scientific, 1993.

[2] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Berlin, 1985.

[3] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science.* Berlin, 1990.

[4] M. Große-Rhode. *Specification of Transition Categories — An Approach to Dynamic Abstract Data Types —.* PhD thesis, TU Berlin, 1995.

[5] M. Große-Rhode. Concurrent State Transformations on Abstract Data Types. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specification*, volume 1130, pages 221–236. Springer Verlag, 1996.

[6] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall International, 1991.

[7] J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge University Press, 1988.

[8] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1989.