State-Based Extension of CASL*/**

H. Baumeister[1] and A. Zamulin[2]

[1] Institute of Computer Science, University of Munich
baumeist@informatik.uni-muenchen.de
[2] Instiute of Informatics Systems
Siberian Division of Russian Academy of Sciences
zam@iis.nsk.su

Abstract. A state-based extension of the algebraic specification language CASL is presented. It permits the specification of the static part of a complex dynamic system by means of CASL and the dynamic part by means of the facilities described in the paper. The dynamic system is defined as possessing a number of states and a number of operations (procedures) for transforming one state into another. Each state possesses one and the same static part specified by CASL and a varying part specified by additional tools. The varying part includes dynamic sorts/functions/predicates and dependent functions/predicates. The dependent functions/predicates are specified by formulae using the names of the dynamic functions/predicates so that each time one of the last ones is updated the corresponding former ones are also updated. The updates of the dynamic entities are produced by procedures which are specified by means of preconditions, postconditions, and dynamic equations.

1 Introduction

The Common Framework Initiative (CoFI) [18] is an open collaborative effort to design a common framework for algebraic specifications. The rational behind CoFI is that the lack of such a framework greatly hinders the dissemination and application of research results in algebraic specification. The aim is to base the common framework as much as possible on a critical selection of features that have already been explored in various contexts. The common framework will provide a family of languages centered around a single, reasonably expressive common specification language called CASL [17]. Some of these languages will be extensions of CASL, e.g. oriented to particular programming paradigms, while others will be sub-languages of CASL, e.g. executable.

In this paper we define SB-CASL, a state-based extension of CASL [17] which is based on algebraic specifications and the concept of implicit state à la Z, B, or VDM, also known as the state-as-algebra approach. In contrast to Z, VDM, and B, this approach does not constrain a specifier by a fixed number of basic types and type constructors used for the representation of application data, and

* This research has been partially supported by ESPRIT working group 29432 (CoFI WG) and the Russian Foundation for Basic Research under Grant 98-01-00682

** to appear in proceedings of IFM 2000, Springer, LNCS.

gives a formal semantics for all notions used in the method. SB-CASL brings together ideas from Typed Gurevich Machines of Zamulin [19], based on the original work of Gurevich [13, 14], Algebraic Specifications with Implicit State of Dauchy and Gaudel, first presented in [8] and further developed in [15, 9], D-oids by Astesiano and Zucca [2, 20, 21], and the work of Baumeister [3–5]. The formalism serves for the specification of *dynamic systems* possessing a state and a number of operations for accessing and updating the state.

The novelty of SB-CASL is that it combines the operational style for the specification of state transformations with the declarative style in a practical specification language. In the operational style one defines how one state is transformed into another; in contrast, in the declarative style only the properties that the successor state has to posses are specified and not how the state is constructed. Up to now either only the operational style was used, like in ASM's and the Implicit State approach, or only the declarative style was used as in the approach by Baumeister. A notable exception is the approach by Zucca [21] which also allows both styles of specifications; however her intention was not to provide a specification language.

The paper is organized as follows. The CASL institution is briefly described in Sec. 2. States and state updates are defined in Sec. 3. Dynamic systems are introduced in Sec. 4. Transition terms, serving for the construction of dynamic formulae, are described in Sec. 5 and dynamic formulae in Sec. 6. The structure of a dynamic system specification and supporting examples are given in Sec. 7. Some related work is discussed in Sec. 8, and in Sec. 9 some conclusions are drawn.

2 The CASL Institution

A *basic specification* in CASL consists of a many-sorted signature Σ together with a set of sentences. The (loose) *semantics* of a basic specification is the class of those models in $\mathbf{Mod}(\Sigma)$ which satisfy all the specified sentences. For reasons of simplicity we restrict ourselves to the many-sorted part of CASL and leave out the order-sorted part. However, all the subsequent constructions in this paper can also be performed in the presence of a subsorting relationship.

A *many-sorted signature* $\Sigma = (S, TF, PF, P)$ consists of:

- a set S of *sorts*;
- sets $TF_{w,s}$, $PF_{w,s}$, of *total function symbols*, respectively *partial function symbols*, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$, for each *function profile* (w, s) consisting of a sequence of *argument sorts* $w \in S^*$ and a *result sort* $s \in S$ (*constants* are treated as functions with no arguments);
- sets P_w of *predicate* symbols, for each *predicate profile* consisting of a sequence of argument sorts $w \in S^*$.

Here and in the sequel a function (predicate) symbol is a name accompanied with a profile. Names may be *overloaded*, occurring in more than one of the above sets.

In this paper we write a total function symbol as $f : s_1, \ldots, s_n \to s$ and a partial function symbol as $f : s_1, \ldots, s_n \to ?\, s$. When the list of argument values is empty, we write s and $?s$, respectively.

For a many-sorted signature $\Sigma = (S, TF, PF, P)$ a *many-sorted model* $A \in \mathbf{Mod}(\Sigma)$ is a *many-sorted first-order structure* consisting of a *many-sorted partial algebra*:

- a non-empty *carrier set* $|A|_s$ for each sort $s \in S$ (let $|A|_w$ denote the cartesian product $|A|_{s_1} \times \cdots \times |A|_{s_n}$ when $w = s_1 \ldots s_n$),
- a *partial function* f^A from $|A|_w$ to $|A|_s$ for each function symbol $f \in TF_{w,s}$ or $f \in PF_{w,s}$, the function being required to be total in the former case,
- together with a *predicate* $p^A \subseteq |A|_w$ for each predicate symbol $p \in P_w$.

Many-sorted terms on a signature $\Sigma = (S, TF, PF, P)$ and a set of sorted, non-overloaded variables X are built from:

- universally quantified variables from X, introduced by

$$\mathbf{var}\ v_{11}, \ldots, v_{1k} : s_1, \ldots, v_{n1}, \ldots, v_{nm} : s_n;$$

- applications of function symbols in $TF \cup PF$ to argument terms of appropriate sorts.

For a many-sorted signature $\Sigma = (S, TF, PF, P)$, the set of Σ-sentences consists of *sort-generation constraints* and the usual closed many-sorted first-order logic formulae, built from atomic formulae using quantification (over sorted variables) and logical connectives. The *atomic formulae* are:

- applications of predicate symbols $p \in P$ to argument terms of appropriate sorts;
- assertions about the definedness of terms, written *def* t;
- existential and strong equations between terms of the same sort, written $t_1 \stackrel{e}{=} t_2$ and $t_1 = t_2$, respectively.

The satisfaction of a sentence in a structure A is determined as usual by the holding of its atomic formulae w.r.t. assignments of (defined) values to all the variables that occur in them. The value of a term w.r.t. a variable assignment may be undefined due to the application of a partial function during the evaluation of the term, or because some arguments of a function application are undefined. The satisfaction of sentences, however, is 2-valued.

The application of a predicate symbol p to a sequence of argument terms holds in A iff the values of all the terms are defined and give a tuple belonging to p^A. A definedness assertion concerning a term holds iff the value of the term is defined. An existential equation holds iff the values of both terms are defined and identical, whereas a strong equation holds also when the values of both terms are undefined. A sort-generation constraint (S', F') is satisfied in a Σ-model A if the carriers of the sorts in S' are generated by the function symbols in F'.

3 States and State Updates

The signature of a system defined by SB-CASL includes a part

$$\Sigma_{stat} = (S_{stat}, TF_{stat}, PF_{stat}, P_{stat})$$

which defines some data types (sorts and operations) using the standard CASL facilities. These data types are used for the specification of system's states and the description of possible state updates. A Σ_{stat}-structure is called a *static structure* in the sequel.

The system's states are defined by *dynamic sorts*, *dynamic functions*, and *dynamic predicates*. The names and profiles of these sorts/functions/predicates, $\Delta_{dyn} = (S_{dyn}, TF_{dyn}, PF_{dyn}, P_{dyn})$, form a signature extension of the static signature Σ_{stat}.

We require that each dynamic function $f : w \to s$ where s is a dynamic sort from S_{dyn} or w contains a dynamic sort from S_{dyn} is in PF_{dyn}. The reason is that a function having a dynamic sort in its profile may become partial if elements are added or removed from this sort.

In the rest of this paper we denote by $\Sigma_{dyn} = (S', TF', PF', P')$ the union of Σ_{stat} with Δ_{dyn}.

Example 1. The first example is a specification of an identifier table. The identifier table stores some data for each identifier definition. It can be block-structured according to block nesting. Typical functions are creating an empty identifier table, inserting identifier data in the current block, checking whether an identifier is defined in the current block, checking whether an identifier is defined in the program, fetching identifier data, and deleting all identifier definitions of the current block.

The following specification defines the state of the identifier table. The static signature is given by the union of the signatures of NAT, NAME, and DEFDATA; and id_table and cur_level are dynamic functions:

System ID_TABLE
use NAT, NAME, DEFDATA ** *The specifications used*
dynamic
  func id_table: Name, Pos $\longrightarrow$? Defdata;
  func cur_level: Pos; ** *the current level of block nesting*

Example 2. The second example is taken from one of the latest work of Zucca [21]. The procedures (dynamic operations in her paper), whose "intended interpretation" is described at the model level in that paper, will be formally specified here.

System CIRCLES
use REAL, COLOUR ** *The spec. COLOUR has only two constants*
  ** *"green" and "red" of sort "Colour"*
dynamic
  sort Circle;
  func X, Y: Circle $\longrightarrow$ Real;
  func radius: Circle $\longrightarrow$ Real;
  func col: Circle $\longrightarrow$ Colour;

A dynamic sort/function/predicate can be different in different states.

Definition 1. *A Σ_{dyn}-state is a Σ_{dyn}-structure where $\Sigma_{dyn} = \Sigma_{stat} \cup \Delta_{dyn}$.*

The restriction of any Σ_{dyn}-state A to Σ_{stat}, $A|_{\Sigma_{stat}}$, is a static structure called the *base* of A. Several Σ_{dyn}-states can have the same base. Following [15], we denote the set of all Σ_{dyn}-states with the same base B by $state_B(\Sigma_{dyn})$ and mean by a Σ^B_{dyn}-state a Σ_{dyn}-state with the static structure B. Thus, the carrier of any static sort $s \in S_{stat}$ in a Σ_{dyn}-state A is the same as the carrier of s in B, that is $|A|_s = |B|_s$.

3.1 Update-Sets

One state can be transformed into another by a *state update* which is either a *function update* or a *predicate update* or a *sort update*.

Definition 2. *Let B be a static structure over Σ_{stat}. A* function update *in a Σ^B_{dyn}-state A is a triple $(f, \bar{a}, a)$ where f_{ws} is a dynamic function (constant) symbol in Δ_{dyn}, $w = s_1 \dots s_n$, $\bar{a} = <a_1, ..., a_n> \in |A|_w$ ($\bar{a}$ is the empty tuple $<>$ when n is equal to zero), and a is either an element of $|A|_s$ or the symbol $\perp$. A function update $(f, \bar{a}, \perp)$ is* valid *only if f is the symbol of a partial function.*

A function update $\alpha = (f, \bar{a}, a)$ serves for the transformation of a Σ^B_{dyn}-state A into a new Σ^B_{dyn}-state $A\alpha$ in the following way:

- $g^{A\alpha} = g^A$ for any $g_{ws} \in TF' \cup PF'$ different from f;
- $f^{A\alpha}(\bar{a}) = a$ if a is not $\perp$, $f^{A\alpha}(\bar{a})$ becomes udefined otherwise;
- $f^{A\alpha}(\bar{a}') = f^A(\bar{a}')$ for any tuple $\bar{a}' = <a'_1, \dots, a'_n>$ different from $\bar{a}$;
- $p^{A\alpha} = p^A$ for any $p_w \in P'$;
- $|A\alpha|_s = |A|_s$ for any $s \in S'$.

Following Gurevich [13], we say that $A\alpha$ is obtained by firing the update α on A. Roughly speaking, firing a function update either inserts an element in the definition domain of a dynamic function or modifies the value of such a function at one point in its domain or removes an element from the definition domain.

Definition 3. *Let B be a static structure over Σ_{stat}. A* predicate update *in a Σ^B_{dyn}-state A is a either a triple $(+, p, \bar{a})$ or a triple $(-, p, \bar{a})$ where p_w is a predicate symbol in Δ_{dyn}, $w = s_1 \dots s_n$, and $\bar{a} = <a_1, ..., a_n> \in |A|_w$.*

A predicate update $\beta = (\xi, p, \bar{a})$ serves for the transformation of a Σ^B_{dyn}-state A into a new Σ^B_{dyn}-state $A\beta$ in the following way:

- $p^{A\beta}(\bar{a})$ holds if ξ is "+" and $p^{A\beta}(\bar{a})$ does not hold if ξ is "−";
- $p^{A\beta}(\bar{a}')$ iff $p^A(\bar{a}')$ for any tuple $\bar{a}' = <a'_1, \dots, a'_n>$ different from $\bar{a}$;
- $q^{A\beta} = q^A$ for any $q_w \in P_{dyn}$ different from p;
- $f^{A\beta} = f^A$ for any $f_{ws} \in TF' \cup PF'$; and
- $|A\beta|_s = |A|_s$ for any $s \in S'$.

Definition 4. *Let s be the name of a dynamic sort. A sort-update δ in A is either a triple $(+, s, id)$ where id is an element such that $id \notin |A|_s$ or a triple $(-, s, id)$ where id is an element such that $id \in |A|_s$.*

A sort update $\delta = (+, s, id)$ transforms a Σ^B_{dyn}-state A into a new Σ^B_{dyn}-state $A\delta$ in the following way:

- $|A\delta|_s = |A|_s \cup \{id\}$,
- $|A\delta|_{s'} = |A|_{s'}$ for any $s' \in S'$ different from s,
- $f^{A\delta} = f^A$ for any $f_{ws} \in TF' \cup PF'$, and
- $p^{A\delta} = p^A$ for any $p_w \in P'$.

A sort update $\delta = (-, s, id)$ transforms a Σ^B_{dyn}-state A into a new Σ^B_{dyn}-state $A\delta$ in the following way:

- for any function $f^A : |A|_{s_1} \times ... \times |A|_{s_n} \to |A|_{s_{n+1}}$, if $|A|_{s_i}$, $i = 1, ..., n+1$, is a dynamic sort associated with the sort name s and $id \in |A|_{s_i}$, then if there is a maplet $<a_1, ..., a_n \mapsto a_{n+1}> \in f^A$ such that $a_i = id$, then $f^{A\delta} = f^A \setminus \{<a_1, ..., a_n \mapsto a_{n+1}>\}$; $f^{A\delta} = f^A$ otherwise;
- for any predicate $p^A : |A|_{s_1} \times ... \times |A|_{s_n}$, if $|A|_{s_i}$, $i = 1, ..., n$, is a dynamic sort associated with the sort name s and $id \in |A|_{s_i}$, then if there is a tuple $<a_1, ..., a_n> \in p^A$ such that $a_i = id$, then $p^{A\delta} = p^A \setminus \{<a_1, ..., a_n>\}$; $p^{A\delta} = p^A$ otherwise;
- $|A\delta|_s = A_s \setminus \{id\}$ and $|A\delta|_{s'} = |A|_{s'}$ for any s' different from s.

Thus, the sort update $\delta = (-, s, a)$ contracts the set of elements of a certain sort and deletes the corresponding entries from the graphs of all dynamic functions and predicates using and/or producing the element indicated.

Note that it is possible to create algebras with empty carrier sets using sort updates. For example, if A is an algebra with $|A|_s = \{a\}$ and $\delta = (-, s, a)$, then $|A\delta|_s = \{\}$. This is in contradiction with the requirement that within the CASL institution carrier sets are non-empty. To solve this problem we introduce the institution SB-CASL which is the same as the CASL institution, but allows their models to have empty carrier sets. Note that the introduction of empty carriers does not pose any problems with the satisfaction relation in case of partial logic (cf. [7]), and that any many-sorted model of the CASL institution is also a model of the SB-CASL institution.

Definition 5. *Let Γ be a set of function/predicate/sort updates. The set Γ is* inconsistent *if it contains either*

- *two contradictory function updates of the following kind: $\alpha_1 = (f, \bar{a}, a)$ and $\alpha_2 = (f, \bar{a}, a')$, where $a \neq a'$ (two contradictory function updates define the function differently at the same point), or*
- *two contradictory predicate updates of the following kind: $\beta_1 = (+, p, \bar{a})$ and $\beta_2 = (-, p, \bar{a})$ (two contradictory predicate updates define the predicate differently at the same tuple of arguments), or*
- *two sort updates of the following kind: $\delta_1 = (+, s, id)$ and $\delta_2 = (-, s, id)$ (two contradictory sort updates insert in the sort and delete from the sort the same element), or*
- *either an $\alpha = (f, <a_1, ..., a_n>, a_{n+1})$ for an $f^A : |A|_{s_1} \times ... \times |A|_{s_n} \to |A|_{s_{n+1}}$ or a $\beta = (\xi, p, <a_1, ..., a_n>)$ for a $p^A : |A|_{s_1} \times ... \times |A|_{s_n}$, and $\delta = (-, s, id)$ such that s is s_i for some $i = 1, ..., n+1$ and $id = a_i$ (a sort element is removed while a function/predicate is forced to use it);*

the update set is consistent *otherwise.*

A consistent update-set Γ applied to a Σ^B_{dyn}-state A transforms A into a new Σ^B_{dyn}-state A' by simultaneous firing all $\alpha \in \Gamma$, all $\beta \in \Gamma$, and all $\delta \in \Gamma$. If Γ is inconsistent, the new state in not defined. If Γ is empty, A' is the same as A. Following [16], we denote the application of Γ to a state A by $A\Gamma$. The set of all consistent sets of updates in $state_B(\Sigma)$ is denoted by $update_B(\Sigma_{dyn})$ in the sequel.

Definition 6. *Let Γ_1 and Γ_2 be two consistent update-sets in a Σ^B_{dyn}-state A, $\alpha_1 = (f, \langle a_1, \ldots, a_n\rangle, a)$, $\alpha_2 = (f, \langle a_1, \ldots, a_n\rangle, a')$, $\beta_1 = (\xi_1, p, \langle a_1, \ldots, a_n\rangle)$, $\beta_2 = (\xi_2, p, \langle a_1, \ldots, a_n\rangle)$, $\delta_1 = (+, s, id)$, and $\delta_2 = (-, s, id)$, where $a \neq a'$ and both ξ_1 and ξ_2 are either "+" or "−" such that ξ_1 is different from ξ_2. The* sequential union *of Γ_1 and Γ_2, denoted by $\Gamma_1 \ ; \Gamma_2$, is defined as follows: $u \in \Gamma_1 \ ; \Gamma_2$ iff $u \in \Gamma_1$ or $u \in \Gamma_2$, except the following cases:*

- *if $\alpha_1 \in \Gamma_1$ and $\alpha_2 \in \Gamma_2$, then $\alpha_2 \in \Gamma_1 \ ; \Gamma_2$ and $\alpha_1 \notin \Gamma_1 \ ; \Gamma_2$;*
- *if $\beta_1 \in \Gamma_1$ and $\beta_2 \in \Gamma_2$, then $\beta_2 \in \Gamma_1 \ ; \Gamma_2$ and $\beta_1 \notin \Gamma_1 \ ; \Gamma_2$;*
- *if $\delta_1 = (-, s, a) \in \Gamma_2$, then for any $\alpha = (g, \langle a_1, \ldots, a_n\rangle, a_{n+1}) \in \Gamma_1$, where $g^A : |A|_{s_1} \times \ldots \times |A|_{s_n} \to |A|_{s_{n+1}}$, and for any $\beta = (\xi, p, <a_1, \ldots, a_n>) \in \Gamma_1$, where $p^A : |A|_{s_1} \times \ldots \times |A|_{s_n}$: if s is s_i, $i = 1, \ldots, n+1$, and $a_i = id$, then $\alpha \notin \Gamma_1 \ ; \Gamma_2$ and $\beta \notin \Gamma_1 \ ; \Gamma_2$, and if $\delta_2 = (+, s, a) \in \Gamma_1$ then both $\delta_1 \notin \Gamma_1 \ ; \Gamma_2$ and $\delta_2 \notin \Gamma_1 \ ; \Gamma_2$.*

Thus, in a sequential union of update sets, each next update of a function/predicate at a certain point waives each preceding update of this function/predicate at the same point. If there are sequential creations of elements of the same sort, the set of elements of this sort will be expanded accordingly. If there is a deletion of an element, the corresponding sort will be contracted accordingly and all function/predicate updates involving this element will be ignored. Note that if an element is first created and then deleted, then the resulting update set will contain no trace of this element.

3.2 Dependent Functions and Predicates

A function update of the form $\alpha = (f, \bar{a}, a)$ applied to a Σ^B_{dyn}-state A does not change any other function g, only f at point $\bar{a}$ is changed. Consider now the following partial function $find$: Name $\to$? Defdata fetching data for a name in ID_TABLE. Clearly the result of *find* depends on the state of ID_TABLE; thus *find* has to be a dynamic function. Further, *find* depends on the dynamic function *id_table* because, starting from the current block level, *find* looks for the definition of an identifier at each block level. Thus we expect that whenever *id_table* is changed so is *find*. However, the semantics of update-sets and dynamic functions does not provide us with such an automatism.

To still allow such functions as *find*, we introduce *dependent functions* and *dependent predicates* as a second set of state components. They form a signature extension $\Delta_{dep} = (\emptyset, TF_{dep}, PF_{dep}, P_{dep})$ of Σ_{dyn} where $TF_{dep} \cap PF_{dep} = \emptyset$. Note that the set of sort names of Δ_{dep} is empty; we don't allow for dependent sorts. As with dynamic functions, we require that each dependent function $f : w \to s$

where s is a dynamic sort from S_{dyn} or w contains a dynamic sort from S_{dyn} is in PF_{dep}. For example, the following dependent functions/predicates can be defined in the system ID_TABLE:

depend
pred defined_current: Name; ** *checks whether an id is defined in the current block*
pred is_defined: Name; ** *checks whether an id is defined in the table*
func find: Name $\longrightarrow$? Defdata; ** *fetches data from the table*

Definition 7. *A Σ^B_{dep}-state is a Σ_{dep}-structure with the static structure B where $\Sigma_{dep} = \Sigma_{dyn} \cup \Delta_{dep}$.*

Thus, dependent functions/predicates extend a Σ^B_{dyn}-state to a Σ^B_{dep}-state. The set of all Σ^B_{dep}-states with the same static structure B is denoted by $state_B(\Sigma_{dep})$.

4 Dynamic Systems

A state update modifies the dynamic and dependent functions/predicates. Possible state updates are specified by *procedures* declared in the fourth part of the system's signature, Δ_{proc}, which consists of sets TP_w, PP_w of *total procedure symbols*, respectively *partial procedure symbols*, such that $TP_w \cap PP_w = \emptyset$, for each *procedure profile* w consisting of a sequence of *argument sorts* from Σ_{dep}.

Example 3. For example, the following procedures can be defined in the system ID_TABLE:

proc
initialize; ** *construction of an empty identifier table*
insert_entry: ? Name, Defdata; ** *insertion of a new entry in the identifier table*
new_level; ** *creation of a new level of nesting in the identifier table*
delete_level ?; ** *deletion of the last block in the identifier table*

Example 4. The following procedures can be declared in the system CIRCLES:

proc
start; ** *create one circle with default attributes*
move: Circle, Real, Real; ** *change the coordinates of a circle*
moveAll: Real, Real; ** *change the coordinates of all circles*
resize: Circle, Real; ** *change the radius of a circle*
changeCol: Circle; ** *change the colour of a circle*
copy: circle ** *create a new circle with the attributes of the argument circle*
delCreen; ** *delete all green circles*

Definition 8. *The signature $D\Sigma = (\Sigma_{stat}, \Delta_{dyn}, \Delta_{dep}, \Delta_{proc})$ of a dynamic system consists of*

- *a static signature Σ_{stat};*
- *a signature extension Δ_{dyn} of Σ_{stat} by symbols of dynamic sorts, functions, and predicates such that the profiles of total functions do not contain a dynamic sort;*

– *a signature extension* Δ_{dep} *of* $\Sigma_{stat} \cup \Delta_{dyn}$ *by symbols of dependent functions and predicates, but without dependent sorts; and*
– *two families of sets* $\Delta_{proc} = (TP, PP)$ *of total and partial procedure symbols.*

Definition 9. *A dynamic system,* $DS(B)$*, of signature* $D\Sigma$ *consists of*

– *a set of states* $|DS(B)| \subseteq state_B(\Sigma_{dep})$*, called the* carrier *of the system;*
– *a partial surjective function* $map^{DS(B)} : state_B(\Sigma_{dyn}) \to |DS(B)|$ *such that if* $map^{DS(B)}(A)$ *is defined, then* $map^{DS(B)}(A)|_{\Sigma_{dyn}} = A$ *for each* $A \in state_B(\Sigma_{dyn})$*;*
– *for each procedure symbol* $p : s_1, \ldots, s_n$*, a (partial) map* $p^{DS(B)}$ *associating an update set* $\Gamma \in update_B(\Sigma_{dyn})$ *with a state* A *of* $DS(B)$ *and a tuple* $<a_1, \ldots, a_n>$ *where* $a_i \in |A|_{s_i}, i = 1, \ldots, n$*.*

Given a dynamic system $DS(B)$, we call a Σ_{dep}-structure A a *state* of $DS(B)$ if $A \in |DS(B)|$.

We write $p^{DS(B)}(A, \bar{a})$ for the application of a procedure $p^{DS(B)}$ to a pair consisting of a state A and a tuple $\bar{a}$ where $A \in |DS(B)|$ and $\bar{a} = <a_1, ..., a_n>$.

For a procedure p, we say that p is a *constant procedure* if the result of $p^{DS(B)}(A, \bar{a})$ does not depend on A. This kind of procedure can be used for the initialization of a dynamic system.

5 Transition terms

State updates are specified by means of special *transition terms*. The interpretation of a transition term TT in a dynamic system $DS(B)$ at a state A w.r.t. a variable assignment $\sigma : X \to |A|$ produces an update set Γ or is undefined. The corresponding state A' after firing the update set can be obtained by

$$A' = map^{DS(B)}((A|_{\Sigma_{dyn}})\Gamma)$$

for which we will simply write, in abuse of notation, $A' = A\Gamma$.

5.1 Basic transition terms

Basic transition terms are *update instructions*, *procedure call*, *sort contraction instruction*, and the *skip instruction*.

Update instructions Let f be the name of a dynamic function with the profile $s_1, \ldots, s_n \to s$, g the name of a partial dynamic function with the profile $s_1, \ldots, s_n \to s$, p the name of a dynamic predicate with the profile $s_1, \ldots, s_n$, X a set of sorted variables, t_i a term of sort s_i over signature Σ_{dep} with variables X for $i = 1, ..., n$. Then

$$\begin{aligned} f(t_1, ..., t_n) &:= t, \\ g(t_1, ..., t_n) &:= undef, \\ p(t_1, ..., t_n) &:= true, \\ p(t_1, ..., t_n) &:= false \end{aligned}$$

are transition terms called *update instructions*.

Interpretation If A is a state of $DS(B)$, σ a variable assignment, t and $t_i, i = 1, \ldots, n$, are defined terms in A under σ, then

$$\begin{aligned}
[\![f(t_1, ..., t_n) := t]\!]^{A,\sigma} &= \{(f, <t_1^{A,\sigma}, \ldots, t_n^{A,\sigma}>, t^{A,\sigma})\}, \\
[\![g(t_1, ..., t_n) := \mathit{undef}]\!]^{A,\sigma} &= \{(g, <t_1^{A,\sigma}, \ldots, t_n^{A,\sigma}>, \perp)\} \\
[\![p(t_1, ..., t_n) := \mathit{true}]\!]^{A,\sigma} &= \{(+, p, <t_1^{A,\sigma}, \ldots, t_n^{A,\sigma}>)\}, \\
[\![p(t_1, ..., t_n) := \mathit{false}]\!]^{A,\sigma} &= \{(-, p, <t_1^{A,\sigma}, \ldots, t_n^{A,\sigma}>)\},
\end{aligned}$$

If at least one of $t, t_i, i = 1, \ldots, n$, is not defined in A under σ, then the interpretation of the above transition terms is undefined.

Example 5. Let x be a variable of sort *Nat* and f be a dynamic function from Nat to Nat. The execution of the transition term $f(x) := f(x) + 1$ under the variable assignment $\sigma = \{x \mapsto a\}$ transforms a state A into a state A' so that $f^{A'}(a) = f^A(a) + 1$ and $f^{A'}(n) = f^A(n)$ for all $n \neq a$.

If c is a partial dynamic constant, a transition term $c := \mathit{undef}$ will make c undefined in the new state.

Procedure call If $p : s_1, ..., s_n$ is a procedure symbol and $t_1, ..., t_n$ are terms of sorts $s_1, ..., s_n$ over Σ_{dep} with variables from X, then $p(t_1, ..., t_n)$ is a transition term called a *procedure call*.

Interpretation Let $DS(B)$ be a dynamic system of signature $D\Sigma$, A a state in $|DS(B)|$, and σ a variable assignment. The interpretation of a procedure call is defined as follows:

$$[\![p(t_1, ..., t_n))]\!]^{A,\sigma} = p^{DS(B)}(A, <t_1^{A,\sigma}, ..., t_n^{A,\sigma}>)$$

if each $t_i^{A,\sigma}$, $i = 1, ..., n$, is defined and $p^{DS(B)}$ is defined for the state A and the tuple $<t_1^{A,\sigma}, ..., t_n^{A,\sigma}>$; $[\![p(t_1, ..., t_n))]\!]^{A,\sigma}$ is undefined otherwise.

Sort contraction If t is a term of a dynamic sort s, then **drop** t is a transition term called a *sort contraction*.

Interpretation: $[\![\textbf{drop}\ t]\!]^{A,\sigma} = \delta$ where $\delta = (-, s, t^{A,\sigma})$.

Skip The transition term **skip** causes no state update, i.e. $[\![\textbf{skip}]\!]^{A,\sigma} = \emptyset$.

5.2 Transition term constructors

Complex transition terms are constructed recursively from basic transition terms by means of several term constructors, e.g., *sequence constructor*, *set constructor*, *condition constructor*, *guarded update*, *loop constructor*, *import constructor*, and *massive update*.

Sequence constructor If $TT_1, TT_2, \ldots, TT_n$ are transition terms, then

$$\textbf{seq}\ TT_1, TT_2, \ldots, TT_n\ \textbf{end}$$

is a transition term called a *sequence of transition terms*.

Interpretation Let A be a state, $\Gamma_1 = [\![TT_1]\!]^{A,\sigma}$, $A_1 = A\Gamma_1$, $\Gamma_2 = [\![TT_2]\!]^{A_1,\sigma}$, $A_2 = A_1\Gamma_2$, $\ldots$, $\Gamma_n = [\![TT_n]\!]^{A_{n-1},\sigma}$. Then

$$[\![\textbf{seq}\ TT_1, TT_2, \ldots, TT_n\ \textbf{end}]\!]^{A,\sigma} = \Gamma,$$

where $\Gamma = \Gamma_1\ \textbf{;}\ \Gamma_2\ \textbf{;}\ \ldots\ \textbf{;}\ \Gamma_n$ and each $[\![TT_i]\!]^{A_{i-1},\sigma}$ is defined.

Thus, to execute a sequence of transition terms starting with a state A, it is sufficient to create the sequential union of their update sets and use it for the transformation of A (which is equivalent to the sequential execution of the terms one after another).

Set constructor If $TT_1, \ldots, TT_n$ are transition terms, then

$$\textbf{set}\ TT_1, \ldots, TT_n\ \textbf{end}$$

is a transition term called a *set of transition terms*.

Interpretation Let A be a state and $\Gamma_1 = [\![TT_1]\!]^{A,\sigma}, \ldots, \Gamma_n = [\![TT_n]\!]^{A,\sigma}$. Then

$$[\![\textbf{set}\ TT_1, \ldots, TT_n\ \textbf{end}]\!]^{A,\sigma} = \Gamma_1 \cup \ldots \cup \Gamma_n$$

if each $[\![TT_i]\!]^{A,\sigma}$ is defined and $\Gamma_1 \cup \ldots \cup \Gamma_n$ is consistent; the semantics is not defined otherwise.

In other words, to execute a set of transition terms, execute all of them in parallel and unite the results if they are consistent.

Condition constructor If k is a natural number, $g_0, ..., g_k$ are formulae, and $TT_0, ..., TT_k$ are transition terms, then the following expression is a transition term called a *conditional transition term*:

$$\begin{array}{l}
\textbf{if}\ g_0\ \textbf{then}\ TT_0 \\
\textbf{elseif}\ g_1\ \textbf{then}\ TT_1 \\
\ldots \\
\textbf{elseif}\ g_k\ \textbf{then}\ TT_k \\
\textbf{endif}
\end{array}$$

If g_k is the formula *true*, then the last **elseif** clause can be replaced with "**else** TT_k". We also write **if** g **then** TT for **if** g **then** TT **else skip endif**.

Interpretation Let A be a state, σ a variable assignment, and TT a conditional transition term, then

$$[\![TT]\!]^{A,\sigma} = [\![TT_i]\!]^{A,\sigma}$$

if g_i holds in A w.r.t. σ, but every g_j with $j < i$ fails in A w.r.t. σ. $[\![TT]\!]^{A,\sigma} = \emptyset$ if every g_i fails in A w.r.t. σ.

Loop constructors The condition constructor together with the sequence constructor gives us a possibility to define some loop constructors. If TT is a transition term and g is a formula, then **while** g **do** TT and **do** TT **until** g are transition terms.

Interpretation

$$[\![\textbf{while } g \textbf{ do } TT]\!]^{A,\sigma} = [\![\textbf{if } g \textbf{ then seq } TT,\ \textbf{while } g \textbf{ do } TT \textbf{ end}]\!]^{A,\sigma};$$
$$[\![\textbf{do } TT \textbf{ until } g]\!]^{A,\sigma} = [\![\textbf{seq } TT,\ \textbf{if } \neg g \textbf{ then do } TT \textbf{ until } g]\!]^{A,\sigma}.$$

Import constructor If x is a variable, s is a dynamic sort name and TT is a transition term, then **import** $x : s$ **in** TT is a transition term called an *import term.*

Interpretation Let $A' = A\delta$, $\delta = (+, s, a)$ for some $a \notin A_s$, and $\sigma' = \sigma \oplus \{x \mapsto a\}$ where "$\oplus$" is the overriding union, i.e. $(\sigma \oplus \sigma'')(x) = \sigma''(x)$ if x is in the domain of σ'' and $(\sigma \oplus \sigma'')(x) = \sigma(x)$ otherwise, then

$$[\![\textbf{import } x : s \textbf{ in } TT]\!]^{A,\sigma} = \{\delta\}; [\![TT]\!]^{A',\sigma'}$$

Massive update Let x be a variable of sort s and TT a transition term. A massive update

$$\textbf{forall } x : s \,.\, TT$$

permits the specification of a parallel update of one or more sorts/functions/ predicates at several points.

Interpretation Let A be a state such that $|A|_s$ is not empty, let σ and $\sigma' = \{x\} \to |A|_s$ be variable assignment, and let $\sigma'' = \sigma \oplus \sigma'$. Then

- if $[\![TT]\!]^{A,\sigma''}$ is defined and $\Gamma = \bigcup\{[\![TT]\!]^{A,\sigma''}\}$ is consistent, then
$$[\![\textbf{forall } x : s \,.\, TT]\!]^{A,\sigma} = \Gamma;$$
- $[\![\textbf{forall } x : s \,.\, TT]\!]^{A,\sigma}$ is not defined otherwise.

If $|A|_s = \emptyset$, then $[\![\textbf{forall } x : s \,.\, TT]\!]^{A,\sigma} = \emptyset$. That is, the massive update over the empty sort produces nothing.

Example 6. Let f be a dynamic function from Nat to Nat. A transition term

$$\textbf{forall } x : Nat \,.\, f(x) := f(x) + 1$$

interpreted in a state A yields the update-set

$$\{(f, n, f^A(n) + 1) \mid n \in |A|_{Nat}\}$$

if $f^A(n)$ is defined for all n.

Example 7. The execution of the update set produced by the transition term

$$\textbf{forall } x : s \,.\, \textbf{drop } x$$

at the current state A will remove all elements from $|A|_s$ and will make empty all functions and predicates using s in their profiles.

6 Dynamic Formulae

For the specification of dynamic systems we introduce dynamic formulae which can be *dynamic equations*, *precondition* formulae, or *postcondition* formulae. A dynamic equation serves for the specification of a behaviour of a procedure in terms of a transition rule. A precondition formula allows us to define the domain of a procedure. Finally, a postcondition formula is used to specify the behaviour of a procedure similar to VDM or Z.

6.1 Dynamic Equations

A *dynamic equation* is of the form $TT_1 = TT_2$ where TT_1 and TT_2 are transition terms over variables from X. A dynamic system $DS(B)$ satisfies a dynamic equation if for all states A of $DS(B)$ and variable assignments $\sigma : X \to |A|$:

$$A[\![TT_1]\!]^{A,\sigma} = A[\![TT_2]\!]^{A,\sigma} \text{ and } A[\![TT_{1/2}]\!]^{A,\sigma} \in |DS(B)|.$$

The most common use of dynamic equations is in the form:

$$p(x_1, \dots, x_n) = TT$$

where TT does not contain a direct or indirect call of p. This defines the semantics of p in a dynamic system to be a function mapping a state A and a tuple $<a_1, \dots, a_n>$ to the update set given by the interpretation of TT w.r.t. A and the variable assignment mapping each x_i to a_i.

Example 8. For a simple example consider a dynamic constant *counter* : *Int* and procedures *Inc* and *Dec*. The procedure *Inc* can be defined by the following dynamic equation:

$$\mathit{Inc} = \mathit{counter} := \mathit{counter} + 1.$$

Similarly, *Dec* can be defined using the dynamic equation

$$\mathit{Dec} = \mathit{counter} := \mathit{counter} - 1.$$

However, dynamic equations need not follow the pattern $p(x_1, \dots, x_n) = TT$. For example, an alternative way to define *Dec* from the previous example is by the following dynamic equation:

$$\textbf{seq}\ \mathit{Dec}, \mathit{Inc}\ \textbf{end} = \textbf{skip}.$$

This means that whenever a *Dec* procedure is followed by an *Inc* procedure, the state of the system should not change.

6.2 Precondition Formulae

Let p be an element of $PP_{s_1,\dots,s_n}$, i.e. a partial procedure symbol, and $t_1, \dots, t_n$ are terms over Σ_{dep} with variables from X. A *precondition formula* of the form

pre $p(t_1, \ldots, t_n) : \varphi$ can be used to state under which conditions a partial procedure p is guaranteed to be defined.

A dynamic system $DS(B)$ satisfies a precondition formula iff the value of $p^{DS(B)}(A, <[\![t_1]\!]^{A,\sigma}, \ldots, [\![t_n]\!]^{A,\sigma}>)$ is defined in exactly those states A and for those variable assignments σ for which φ holds.

The following precondition formula

$$\textbf{pre } \mathit{insert\_entry}(id, d) : \neg \mathit{defined\_current}(id)$$

states that the partial procedure *insert_entry* declared in Ex. 3 for the dynamic system ID_TABLE must be defined only in those states A and only for those arguments id for which the interpretation of *defined_current* is false.

6.3 Postcondition Formulae

Dynamic equations of the form $p(x_1, \ldots, x_n) = TT$ can be used to specify dynamic systems in an operational style similiar to Gurevich's ASMs.

However, sometimes it is convenient to use a declarative style similar to the one used by the specification languages Z and VDM. In the declarative style, the values of a dynamic/dependent component before and after the execution of a procedure are related by a first-order formula. Usually this formula only defines the relationship between the values and does not provide an algorithm how to change the value. For example, the *Dec* operation of Ex. 8 could be defined by a *postcondition formula*

$$\textbf{post } \mathit{Dec} : \mathit{counter} = \mathit{counter}' + 1$$

where *counter* refers to the value of *counter* in the state before executing the *Dec* operation, and *counter'* refers to the value of *counter* after executing *Dec*. Note that this formula does not prescribe how the value of *counter* is computed after performing the *Dec* operation. In contrast, a dynamic equation of the form

$$\mathit{Dec} = \mathit{counter} := \mathit{counter} - 1$$

defines an update-set used for changing the value of *counter*.

To be more precise, let A be a state of a dynamic system $DS(B)$ containing the dynamic constant *counter* and a procedure *Dec*. Then the interpretation of *Dec* in $DS(B)$ yields the update-set $\mathit{Dec}^{DS(B),\{\}} = \Gamma$ which applied to a state A yields the state $A\Gamma$. Then $A\Gamma$ has the following property: the interpretation of *counter* in A has to be the same as the interpretation of *counter* in $A\Gamma$ plus 1:

$$\mathit{counter}^{A,\{\}} = \mathit{counter}^{A\Gamma,\{\}} + 1.$$

The general syntax of a postcondition formula is:

$$\textbf{post } p(t_1, \ldots, t_n) : \varphi$$

where p is a (partial) procedure, $t_1, \ldots, t_n$ are terms over Σ_{dep}, and φ is a formula over the signature $\bar{\Sigma}$ which is constructed as follows. First consider the case where there are no dynamic sorts, then the sorts of $\bar{\Sigma}$ are the static sorts of Σ_{stat}. The operation symbols (total or partial) of $\bar{\Sigma}$ are those defined in the static signature plus two copies of the dynamic and dependent function symbols: one copy denoting the function before the execution of a procedure and one copy, decorated by a prime ($\prime$), denoting the function after the execution of a procedure. Similar for the predicate symbols of $\bar{\Sigma}$.

To define whether φ is satisfied by a state A and an update-set Γ, we construct from A and $A\Gamma$ a $\bar{\Sigma}$-model A^{Γ} as follows. The interpretation of a static signature component, i.e. a sort, operation, or predicate symbol from Σ_{stat}, is the same as the interpretation of that component in A. Note that this is the same as the interpretation of that symbol in $A\Gamma$ and B, because $A|_{\Sigma_{stat}} = B = A\Gamma|_{\Sigma_{stat}}$. The interpretation of a dynamic or dependent signature component in A^{Γ} is either the interpretation of that component in A, or, if this is a component with a prime in $\bar{\Sigma}$, the interpretation of the corresponding unprimed component in $A\Gamma$. Then a state A and an update-set Γ satisfy a formula φ iff $A^{\Gamma} \models \varphi$.

In the case of dynamic sorts the construction is similar. That is, $\bar{\Sigma}$ contains the static sorts plus two copies of the dynamic sorts, one copy decorated with a prime. However, the profile of a primed dynamic or dependent function/predicate in $\bar{\Sigma}$ having a dynamic sort in its profile changes. Each dynamic sort in the profile has to be replaced by its primed version. Note that the profiles of unprimed function/predicate symbols remain the same.

As an example consider the dynamic system CIRCLES, which has a dynamic sort *Circle* and a dynamic function $X : \mathit{Circle} \rightarrow ?\ \mathit{Real}$ (among others). Now the signature $\bar{\Sigma}$ contains two sorts *Circle* and *Circle*$'$ and two function symbols $X : \mathit{Circle} \rightarrow ?\ \mathit{Real}$ and $X' : \mathit{Circle}' \rightarrow ?\ \mathit{Real}$. Note that in the profile of X' the dynamic sort *Circle* is also decorated with a prime.

There is still another problem with dynamic sorts. Consider the following specification of the *move* operation in the dynamic system CIRCLES:

$$\textbf{post}\ \mathit{move}(c, x, y) : X'(c) = X(c) + x \wedge Y'(c) = Y(c) + y$$

Note that the formula above is not well-formed w.r.t. $\bar{\Sigma}$ because X' is a function from *Circle*$'$ to *Real* while c is of sort *Circle*, and similar for Y' and c. The solution is to introduce in $\bar{\Sigma}$ a partial function $tm_s : s \rightarrow s'$ called a *tracking map* for each dynamic sort s. The notion of tracking map was first introduced with d-oids (cf. [1, 2, 20, 21]). In our example we use the function $tm_{Circle} : \mathit{Circle} \rightarrow ?\ \mathit{Circle}'$ and write

$$\textbf{post}\ \mathit{move}(c, x, y) : X'(tm_{Circle}(c)) = X(c) + x \wedge Y'(tm_{Circle}(c)) = Y(c) + y$$

In the sequel we leave the application of the tracking map implicit whenever possible. If $t[r]$ is a term with a subterm r and r is required to be of dynamic sort s', then we allow r to be of dynamic sort s and understand this as an abbreviation for $t[tm_s(r)]$. This allows us to write the above postcondition formula as:

$$\textbf{post}\ \mathit{move}(c, x, y) : X'(c) = X(c) + x \wedge Y'(c) = Y(c) + y$$

For the interpretation of the tracking maps in A^{Γ} we have to define the tracking map associated with a dynamic sort and an update-set. This tracking map is undefined for elements that are removed and is the identity otherwise.

$$tm_s^{A,\Gamma} : |A|_s \rightarrow |A\Gamma|_s$$

$$tm_s^{A,\Gamma}(a) = \begin{cases} \perp \text{ if } \delta = (-, s, a) \in \Gamma \\ a \text{ else} \end{cases}$$

The definition of satisfaction remains the same, i.e. A and Γ satisfy φ if $A^\Gamma \models \varphi$. The formal definition of $\bar{\Sigma}$ and A^Γ is given in App. A.

7 Specification of Dynamic Systems

Definition 10. *A dynamic system specification*

$$DSS = (SPEC, \Delta_{dyn}, (\Delta_{dep}, Ax), (\Delta_{proc}, Ax_{proc}))$$

has four levels:

- *The first level is a CASL specification SPEC with semantics (Σ_{stat}, M). SPEC defines the data types used in the system.*
- *The second level defines those aspects of the system's state which are likely to change. It includes a signature extension, Δ_{dyn}, which declares some* dynamic *sorts/functions/predicates.*
- *The third level defines some dependent functions/predicates and indicates state invariants. It does not introduce new sorts and uses the names of dynamic functions/predicates from Δ_{dyn}, the names of dependent function/predicates from Δ_{dep}, and the operations of Σ_{stat}. The formulae in Ax restrict the set of possible states of a dynamic system satisfying DSS.*
- *The fourth level, $(\Delta_{proc}, Ax_{proc})$, defines some* procedures. *Ax_{proc} is a set of dynamic formulae.*

A dynamic system specification DSS defines a dynamic signature

$$D\Sigma = (\Sigma_{stat}, \Delta_{dyn}, \Delta_{dep}, \Sigma_{proc}),$$

and a dynamic system $DS(B)$ over signature $D\Sigma$ satisfies a dynamic specifiction DSS iff

- B is a model of $SPEC$,
- $|DS(B)|$ is the set $\{A \mid A \in state_B(\Sigma_{dep}) \wedge A \models Ax\}$,
- $DS(B)$ satisfies each dynamic formula in Ax_{proc}.

Example 9.

***specification of the "ID_TABLE" system*
System ID_TABLE
  use NAT, NAME, DEFDATA ** *The specifications used*
  dynamic
    id_table: Name, Pos $\longrightarrow$? Defdata;
    cur_level: Pos; - *the current level of block nesting*
  depend
    pred defined_current, is_defined: Name;
    pred local_defined: Name, Pos;
    func local_find: Name, Pos $\longrightarrow$? Defdata;

func find: Name ⟶? Defdata;
var id: Name, k: Pos
• defined_current(id) ⇔ def id_table(id, cur_level));
• local_defined(id, 0) ⇔ false;
• local_defined(id, k) ⇔ def id_table(id, k)∨ local_defined(id, k-1);
• is_defined(id) ⇔ local_defined(id, cur_level);
• local_find(id, k) = id_table(id, k) **when** def id_table(id, k)
    else local_find (id, k-1)};
• find(id) = local_find(id, cur_level) **if** is_defined(id);
proc
initialize; ** *construction of an empty identifier table*
insert_entry: ? Name, Defdata;
new_level;
delete_level?;
var id: Name, k: Pos, d: Defdata
• **pre** delete_level: cur_level > 1;
• **pre** insert_entry(id, d): ¬ defined_current(id);
• initialize = **set** cur_level := 1,
    forall id: Name, x: Pos. id_table(id, x) := undef **end**;
• **post** insert_entry(id, d): id_table'(id, cur_level) = d;
• **post** new_level: cur_level' = cur_level + 1;
• delete_level = **set** cur_level := cur_level − 1,
    forall id: Name. id_table(id, cur_level) := undef **end**;

Example 10.
System CIRCLES
use REAL, COLOUR ** *The spec. COLOUR has only two constants*
** *"green" and "red" of sort "Colour"*
dynamic
sort Circle;
func X, Y: Circle ⟶? Real;
func radius: Circle ⟶? Real;
func col: Circle ⟶? Colour
proc
start; ** *creation of one circle with default attributes*
move: Circle, Real, Real; ** *change the coordinates of a circle*
moveAll: Real, Real; ** *change the coordinates of all circles*
resize: Circle, Real; ** *change the radius of a circle*
changeCol: Circle; ** *change the colour of a circle*
copy: circle ** *create a new circle with the attributes of the argument circle*
delCreen; ** *delete all green circles*
var x, y, r: Real, cir: Circle
• start = **seq forall** c: Circle. **drop** c,
    import c: Circle **in**
        set X(c) := 0, Y(c) := 0, radius(c) := 1, col(c) := green **end**
    end;

- **post** move(cir, x, y): X'(cir) = X'(cir) + x $\wedge$ Y'(cir) = Y(cir) + y;
- moveAll(x, y) = **forall** c: Circle.
  set X(c) := X(c) + x, Y(c) := Y(c) + y **end**;
- **post** resize(cir, r): radius'(cir) = r;
- changeCol(cir) = **if** colour(cir) = green **then** colour(cir) := red
  else colour(cir) := green **endif**;
- copy(cir) = **import** c: Circle **in set** X(c) := X(cir),
  Y(c) := Y(cir), radius(c) := radius(cir), col(c) := col(cir) **end**;
- delGreen = **forall** c: Circle. **if** colour(c) = green **then drop** c;

end

8 Related Work

As mentioned in the introduction, this extension of CASL is based on several works using the concept of implicit state. Currently none of them offers the full set of tools needed for the specification of a broad range of dynamic systems. Therefore the natural combination of the facilities offered by these works and their adaptation to the CASL institution has been one of our main goals.

We have liked very much the concept of update set introduced by Gurevich in [13] for the explanation of state transitions. It was used later by him in [14] for giving denotational semantics of transition rules of Abstract State Machines (ASMs). We also give the semantics of our transition terms in terms of update sets. However, ASMs are based on total universal algebras treating predicates as Boolean functions. Since our states are partial many-sorted structures, we have extended α-updates of ASMs with β- and δ-updates representing, respectively, the updates over predicates and sorts. Our notion of transition term is an extension and generalization of the ASM notion of transition rule. The amendments are the procedure call, the **drop** rule allowing a sort to shrink, the sequence constructor and based on it the loop constructors (see also [6] for another proposition of sequence and loop constructors). ASMs do not have such features as dependent functions and procedures, nor is the notion of dynamic system defined for them.

Dependent functions and procedures are borrowed from [8, 15] through intermediate steps of [19, 9]. However, their semantics is different. In [19, 9] dependent functions are not part of the state; they belong to the dynamic system, and this caused some problems with the use of their names in terms. In [8, 15] dependent functions are part of the state with a very complex semantics of their redefinition in different states. The introduction of the function *map* has allowed us to treat dependent functions as part of the state with very simple semantics of their redefinition. The semantics of modifiers (analogue of our procedures) and update expressions (analogue of our transition terms) is given in [8, 15] operationally (there is no notion of update set in this approach), while we have done it denotationally. We have also provided the means for working with partial structures (only total many-sorted algebras are used in [8, 15]).

The notion of dynamic system, as it is defined in our paper, stems from the notion of *d-oid* introduced in [1, 2] and further developed in [20, 21]. A d-oid is a set of instance structures (e.g., algebras), a set of dynamic functions resembling our dependent functions and procedures and a tracking map indicating relationships between instance structures. The approach deals with models and does not address the issue of *d-oid* specification. Therefore, we had to borrow and adapt for our purpose a specification technique of another source.

Another approach are "Transition Categories" presented in [10, 11]. Here the algebra of data types is equipped with sorts called *reference sorts* in addition to ordinary sorts, and the state extends this algebra by partial functions called *contents functions* for each reference sort which map elements of reference sorts to their contents. State transformations are defined by *conditional parallel assignment* redefining the contents functions. Though based on different semantical foundations, Große-Rhode's approach appears as a special class of dynamic system defined in this paper where the static part contains the reference sorts and the only components of the dynamic part are the contents functions. Then the effect of conditional parallel assignment can be achieved using dynamic equations.

In a later work [12], Große-Rhode defines *Algebra Rewrite Systems*. A rewrite rule r is of the form $P_l \longleftrightarrow P_r$ where $P_l = (X_l, E_l)$ and $P_r = (X_r, E_r)$ are presentations consisting of sets of generators X_l and X_r and sets of equations E_l and E_r, respectively. A rule is applied to a partial algebra A by first removing the elements of X_l from the carrier sets of A together with the equalities in E_l, and then adding the elements in X_r and the equalities in E_r. This allows the modelling of update instructions $f(t_1) := t_2$ by first undefining a function entry and then adding a new function entry by a rule $(\{x : s\}, \{f(t_1) = x\}) \longleftrightarrow (\{x : s\}, \{f(t_1) = t_2\})$. Deletion of an element of sort s is modeled by a rule $(\{x : s\}, \emptyset) \longleftrightarrow (\emptyset, \emptyset)$, and creation of an element by a rule $(\emptyset, \emptyset) \longleftrightarrow (\{x : s\}, \emptyset)$ However, the approach by Große-Rhode is restricted to partial algebras and thus cannot be used in the context of CASL, which is order-sorted and in addition to partial function contains total functions and predicates. Further, the application of rewrite rules and also the interaction between axioms defining the static part with rewrite rules may yield unexpected results, like the identification of elements in the result state.

While all the above approaches favour an operational style of writing specifications, with the exception of Zucca [21], that is, they specify how a state is transformed into another state, the approach by Baumeister [3–5] uses a declarative approach, defining how the states before and after the execution of a state transformation are related. This allows writing specifications which are similar to those written in specification languages like Z or VDM. Baumeister's approach does not require states to be modeled as algebras; states can be structures from any suitable institution. From this approach we have used the idea how to interpret postcondition formulae.

9 Conclusion

In this paper we have defined an extension of CASL for the specification of state-based software systems intended to be part of the common framework defined by the Common Framework Initiative. It permits the specification of the static part of a complex dynamic system by means of CASL and the dynamic part by means of the facilities described in the paper. This is one level of integration of two different specification paradigms. The specification of the dynamic part can also be done either by means of transition rules or by means of postconditions (where it is appropriate). This provides the second level of integration. Moreover, the use of update sets for describing the semantics of both transition rules and postconditions has permitted us to define the semantics in a simple way and easily solve the well-known frame problem.

The next step in the development of the described specification technique is the introduction of structuring facilities permitting the specifications to be united, localized, parameterized, etc.

References

1. E. Astesiano and E. Zucca. A semantic model for dynamic systems. In *Modeling Database Dynamics*, Workshops in Computing, pages 63–83, Volkse 1992, 1993. Springer.
2. E. Astesiano and E. Zucca. D-oids: a model for dynamic data types. *Mathematical Structures in Computer Science*, 5(2):257–282, June 1995.
3. H. Baumeister. Relations as abstract datatypes: An institution to specify relations between algebras. In *TAPSOFT 95*, volume 915 of *LNCS*, pages 756–771. Springer, 1995.
4. H. Baumeister. Using algebraic specification languages for model-oriented specifications. Tech. report MPI-I-96-2-003, Max-Planck-Institut für Informatik, Saarbrücken, February 1996.
5. H. Baumeister. *Relations between Abstract Datatypes modeled as Abstract Datatypes*. PhD thesis, Universität des Saarlandes, Saarbrücken, November 1998.
6. E. Börger. Composition and structuring principles for ASMs. In *Abstract State Machines — ASM 2000 (Proc. Int. Workshop on Abstract State Machines, Monta Verita, Switzerland, March 2000)*, LNCS. Springer, 2000. to appear.
7. M. Cerioli, T. Mossakowski, and H. Reichel. From total equational to partial first-order logic. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, pages 31–104. Springer, 1999.
8. P. Dauchy and M.-C. Gaudel. Algebraic specifications with implicit state. Tech. report No 887, Laboratoire de Recherche en Informatique, Univ. Paris-Sud, 1994.
9. M.-C. Gaudel, C. Khoury, and A. Zamulin. Dynamic systems with implicit state. In *Fundamental Approaches to Software Engineering*, volume 1577 of *LNCS*, pages 114–128. Springer, 1999.
10. M Große-Rhode. Concurrent state transformation on abstract data types. In *Recent Trends in Data Type Specifications*, volume 1130 of *LNCS*, pages 222–236. Springer, 1995.
11. M. Große-Rhode. Transition specifications for dynamic abstract data types. *Applied Categorical Structures*, pages 265–308, 1997.
12. M. Große-Rhode. Specification of state based systems by algebra rewrite systems and refinements. Tech. report 99-04, TU Berlin, FB Informatik, March 1999.
13. Y. Gurevich. Evolving Algebras 1993: Lipary guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
14. Y. Gurevich. ASM guide 1997. EECS Departmental Technical Report CSE-TR-336-97, University of Michigan, 1997.
15. C. Khoury, M.-C. Gaudel, and P. Dauchy. AS-IS. Tech. report No 1119, Univ. Paris-Sud, 1997.
16. K. Lellahi and A. Zamulin. Dynamic systems based on update sets. Tech. report 99-03, LIPN, Univ. Paris 13 (France), 1999.

17. P. Mosses. CASL: a guided tour of its design. In *Recent Trends in Algebraic Development Techniques: Selected Papers from WADT'98*, volume 1589 of *LNCS*, Lisbon, 1999. Springer.
18. The Common Framework Initiative. `http://www.brics.dk/Projects/CoFI/`.
19. A. Zamulin. Dynamic system specification by Typed Gurevich Machines. In *Proc. Int. Conf. on Systems Science*, Wroclaw, Poland, September 15–18 1998.
20. E. Zucca. From static to dynamic data types. In *Mathematical Foundations of Computer Science*, volume 1113 of *LNCS*, pages 579–590, 1996.
21. E. Zucca. From static to dynamic abstract data types: an institution transformation. *Theorical Computer science*, 216:109–157, 1999.

A Postcondition Formulae

In this section we give a formal definition of $\bar{\Sigma}$, A^{Γ}, and postcondition formulae. Let

$$\Delta_{dyn} = (S_{dyn}, TF_{dyn}, PF_{dyn}, P_{dyn})$$

be a signature extension w.r.t. $\Sigma_{stat} = (S_{stat}, TF_{stat}, PF_{stat}, P_{stat})$,

$$\Delta_{dep} = (\emptyset, TF_{dep}, PF_{dep}, P_{dep})$$

be a signature extension w.r.t. $\Sigma_{stat} \cup \Delta_{dyn}$, then

$$\Delta = (S, TF, PF, P) = (S_{dyn}, TF_{dyn} \cup TF_{dep}, PF_{dyn} \cup PF_{dep}, P_{dyn} \cup P_{dep})$$

is a signature extension w.r.t. Σ_{stat}. We define $\bar{\Sigma} = (\bar{S}, \bar{TF}, \bar{PF}, \bar{P})$ as follows:

$$\bar{S} = S_{stat} \cup S_{dyn} \cup \{s' \mid s \in S_{dyn}\}$$

$$\begin{aligned}\bar{TF} = \ & \{f : w \to s_0 \mid f : w \to s_0 \in (TF_{stat} \cup TF)\} \cup \\ & \{f' : \bar{s}_1, \ldots, \bar{s}_n \to \bar{s}_0 \mid f : s_1, \ldots, s_n \to s_0 \in TF, \\ & \qquad \bar{s}_i = s'_i \text{ if } s_i \in S_{dyn},\ \bar{s}_i = s_i \text{ else},\ 0 \le i \le n\}\end{aligned}$$

$$\begin{aligned}\bar{PF} = \ & \{f : w \to s_0 \mid f : w \to s_0 \in (PF_{stat} \cup PF)\} \cup \\ & \{f' : \bar{s}_1, \ldots, \bar{s}_n \to \bar{s}_0 \mid f : s_1, \ldots, s_n \to s_0 \in PF, \\ & \qquad \bar{s}_i = s'_i \text{ if } s_i \in S_{dyn},\ \bar{s}_i = s_i \text{ else},\ 0 \le i \le n\} \cup \\ & \{tm_s : s \to s' \mid s \in S_{dyn}\}\end{aligned}$$

$$\begin{aligned}\bar{P} = \ & \{p : w \mid p : w \in (P_{stat} \cup P)\} \cup \\ & \{p' : \bar{s}_1, \ldots, \bar{s}_n \mid p : s_1, \ldots, s_n \in P, \\ & \qquad \bar{s}_i = s'_i \text{ if } s_i \in S_{dyn},\ \bar{s}_i = s_i \text{ else},\ 0 \le i \le n\}\end{aligned}$$

Definition 11. *A* postcondition formula *is a formula of the form*

$$\mathbf{post}\ p(t_1, \ldots, t_n) : \varphi$$

where $p : s_1, \ldots, s_n$ is a procedure symbol, $s_1, \ldots, s_n$ are sorts in $S_{stat} \cup S_{dyn}$, $t_1, \ldots, t_n$ are terms over variables X of sorts $s_1, \ldots, s_n$, and φ is a formula over $\bar{\Sigma}$ with variables X.

Given a state A in a dynamic system $DS(B)$ and an update-set Γ, we define a $\bar{\Sigma}$-structure A^{Γ} by:

$$\begin{aligned} |A^{\Gamma}|_s &= |A|_s \quad \text{if } s \in S_{stat} \cup S_{dyn} \\ |A^{\Gamma}|_{s'} &= |A\Gamma|_s \quad \text{if } s' \in \bar{S} \setminus (S_{stat} \cup S_{dyn}) \end{aligned}$$

for all sort symbols in $\bar{S}$,

$$\begin{aligned} f^{A^{\Gamma}} &= f^A \qquad \text{if } f \in TF_{stat} \cup TF \\ f'^{A^{\Gamma}} &= f^{A\Gamma} \quad \text{if } f' \in \bar{TF} \setminus (TF_{stat} \cup TF) \end{aligned}$$

for all total function symbols in $\bar{PF}$,

$$\begin{aligned} f^{A^{\Gamma}} &= f^A \qquad \text{if } f \in PF_{stat} \cup PF \\ f^{A^{\Gamma}} &= tm_s^{A,\Gamma} \quad \text{if } f = tm_s \\ f'^{A^{\Gamma}} &= f^{A\Gamma} \qquad \text{if } f' \in \bar{PF} \setminus (PF_{stat} \cup PF \cup \{tm_s \mid s \in S_{dyn}\}) \end{aligned}$$

for all partial function symbols in $\bar{PF}$, and

$$\begin{aligned} p^{A^{\Gamma}} &= p^A \qquad \text{if } p \in P_{stat} \cup P \\ p'^{A^{\Gamma}} &= p^{A\Gamma} \quad \text{if } p' \in \bar{P} \setminus (P_{stat} \cup P) \end{aligned}$$

for all predicate symbols in $\bar{P}$.

Note that one can write $f'^{A^{\Gamma}} = f^{A\Gamma}$ and $p'^{A^{\Gamma}} = p^{A\Gamma}$, though different sort symbols are used in the profiles of f and f' (p and p'), since the carrier-set of each s' in A^{Γ} is the same as the carrier set of the corresponding s in $A\Gamma$.

A dynamic system $DS(B)$ satisfies a postcondition formula

$$\mathbf{post}\ p(t_1, \ldots, t_n) : \varphi$$

iff for any state A and variable assignment $\sigma : X \to A$: if the update-set

$$\Gamma = p^{DS(B)}(A, <[\![t_1]\!]^{A,\sigma}, \ldots, [\![t_n]\!]^{A,\sigma}>)$$

exists, then

$$\varphi^{A^{\Gamma},\sigma} = true.$$