Joint NCC & IIS Bull., Comp. Science, 15 (2001), 137–157
© 2001 NCC Publisher

Typed ASMs with updateable locations as values*

A. V. Zamulin

A formal model of the state of a dynamic system with updateable locations as values is presented. A mechanism of dynamic function declaration resembling that of variable declaration in programming languages is suggested. With each of these functions a dynamic access function is associated. An access function can be used either explicitly or implicitly like in programming languages (explicit or implicit dereferencing). An update of an access function causes the update of the corresponding location. A procedure parameter declared as a reference parameter accepts locations as arguments so that a value associated with the location can be updated by the procedure.

Keywords: dynamic systems, formal methods, implicit state, updateable locations.

1. Introduction

States of dynamic systems as *algebras* are widely investigated in recent years (see the list of references). State components that can be different in different states are normally represented by so-called *dynamic functions* updateable by special operations, *modifiers*. There are elementary modifiers updating a function at a single point and user-defined modifiers constructed as combinations of modifiers and updating one or several functions at several points. Different techniques for the specification of modifiers are proposed. Pre- and post-conditions are used in [2, 7], conditional replacement rules in [4, 5], transition rules in [6, 11], dynamic terms in [12], update expressions in [3, 8].

A common feature of all approaches listed above is the absence of a built-in notion of a *location* value. Any function produces a value of a definite sort. No function can produce a location which can be further dereferenced if needed. For this reason, a notion like l-value of C^{++} cannot naturally be modeled. In [4] location (reference) types can be declared explicitly and then implicitly supplied with content functions extracting values associated with locations. However, these functions must always be used explicitly, thus preventing the passing of a location instead of a value if needed. At the same time we know that passing a location instead of a value is a normal

*Partially supported by the Russian Foundation for Basic Research under Grant 01-01-00787.

practice of imperative programming languages. Moreover, declaring a function parameter as a location (l-value), one can implement the mechanism of *call-by-reference* permitting the update of an argument location. All present models of states provide only the mechanism of call-by-value.

The purpose of this work is the suggestion of a state model with *location values*. The main task is the provision of a mechanism of dynamic function declaration resembling that of variable declaration in programming languages. With each of these functions (called *location functions* in this paper) a dynamic *access function* should be associated. An access function can be used either explicitly or implicitly like in programming languages (explicit or implicit dereferencing). An update of an access function must lead to the update of the corresponding location. A modifier (*procedure* in this paper) parameter declared as a reference parameter must serve as an alias of an argument location so that a value associated with the location can be updated by the procedure. In this way a suitable model of an imperative language can be developed.

An ASM is usually a transition rule recursively built from function updates and the skip rule by a number of rule constructors. The semantics of the transition rule is defined in [6] in terms of *update sets*. Each update in the set is a pair (loc, val), where loc is a location and val a value to be associated with it. A location is a pair $(f, < a_1, ..., a_n >)$ representing an n-ary function f applied to a tuple of elements $< a_1, ..., a_n >$, the value val is then the value to be produced by $f(a_1, ..., a_n)$ in the new state. In this approach a location can never be a value used and/or produced by another function. It can only be considered as the name of a value.

We should introduce another kind of update if locations are considered as values produced by functions. Thus, if $name$ is a pair $(f, < a_1, ..., a_n >)$, where f is a function producing locations, and loc is a location, then the above update is transformed in the pair $(name, loc)$ associating a location with a certain location name. In addition to this, an update (loc, val) associates the value val with the location loc. Another kind of state update occurs when a location is created or destroyed. For this reason the set of update instructions should be expanded accordingly. These extensions are described in this paper which is organized as follows.

A notion of the static part of the state of a dynamic system is given in Section 2. Components of the state including location sorts and location and access functions are formally described in Section 3. Primitives for state updates are introduced in Section 4. Dependant functions defined in terms of location and access functions and serving for observing the state are defined in Section 5. Procedures serving for updating the state are introduced in Section 6. A formal definition of a dynamic system with implicit state is given in Section 7. Transition rules as a means of defining state transformations are described in Section 8. Dynamic formulae serving for procedure specification are defined in Section 9, and the whole structure of a dynamic system specification is given in Section 10. Some conclusions and directions of further work are outlined in Section 11.

2. Static structure

A notion of dynamic system and its signature will be introduced in this paper. It is sufficient to indicate for the beginning that the signature of a dynamic system includes a part $\Sigma_{dat} = (S_{dat}, F_{dat})$ which defines some data types (sorts and operations), using the facilities of an algebraic specification language. These data types are used for the specification of system states and the description of possible state updates. We do not suggest a mechanism for the specification of this part of the system. Any specification language whose semantics is given as a class of many-sorted algebras is suitable for this purpose.

A Σ_{dat}-algebra is called a *data algebra* in the sequel.

3. State

The system states are defined by *location sorts*, *location functions* and *access functions*. The names and profiles of location functions F_{loc} are introduced in the second part of the system signature Δ_{dyn} with the use of sort terms defined in the following way.

Definition 1. Sort terms:

1) given $\Sigma_{dat} = (S_{dat}, F_{dat})$, if $\mathsf{s} \in S_{dat}$, then s is a sort term;
2) if s is a sort term, then $loc(s)$ is a sort term.

An element of F_{loc} is called a *function symbol.* It has the following form: f_{ws}, where $w = s_1...s_n$ is a sequence of sort terms called *domain units* and s is a sort term called a *codomain unit*; ws is a *function profile.*

The set F_{loc} consists of three subsets, C_{loc}, UF_{loc} and SF_{loc}, called *constants*, *unique functions* and *shared functions*, respectively. We require that $UF_{loc} \cap SF_{loc} = \emptyset$, i. e., there cannot exist a unique function and a shared function with the same name and profile. As it will be defined in the sequel, the range of a unique function does not intersect with the range of any other function while the range of a shared function can intersect with the range of another shared function. A constant, i. e., a function symbol without domain units, can also be shared (i. e., a constant may be in the range of a shared function). In the examples which follow, a constant symbol c_s is preceded with a keyword **const**, a unique function symbol f_{ws} is preceded with a keyword **ufunc**, and a shared function symbol f_{ws} is preceded with a keyword **sfunc**.

Example. Suppose we wish to maintain a double-linked list of locations storing data of type `Node` with a distinguished node `Header` and an array of nodes. In this case, we can do the following declarations:

System LINKED_LIST ** *a list of locations containing nodes*
use NAT, NODE; ** *the specifications used*
location
  const head: Node;
  sfunc next: Node $\longrightarrow$ Node;
  sfunc prev: Node $\longrightarrow$ Node;
  ufunc array: Nat $\longrightarrow$ Node;

In this example,

$$\begin{array}{lcl} F_{loc} & = & \{head_{Node}, next_{Node,Node}, prev_{Node,Node}, array_{Nat,Node}\}, \\ SF_{loc} & = & \{head_{Node}, next_{Node,Node}, prev_{Node,Node}\}, \\ UF_{loc} & = & \{array_{Nat,Node}\}. \end{array}$$

Given F_{loc}, let S' be the set of sort terms constructed as follows:

- if s is a domain unit in the profile of some $f_{ws'} \in F_{loc}$, then $s \in S'$;
- if s is a codomain unit in the profile of some $f_{ws} \in F_{loc}$, then both $s \in S'$ and $loc(s) \in S'$.

The set $S_{loc} = S' \backslash S$ is called the set of *location sorts*. Note that the set is finite for a finite F_{loc}.

Example. For the system LINKED_LIST we have the following sets S' and S_{loc}:
  S': $\{Nat, Node, loc(Node)\}$;
  S_{loc}: $\{loc(Node)\}$;

F_{loc} is now extended by the set F_{acc} of *access function* symbols $\_!_{loc(s),s}$ for each $loc(s) \in S_{loc}$. For example, we have the following access function for the system LINKED_LIST:

$$\_!_{loc(Node),Node};$$

If l and a are elements of respective sorts $loc(s)$ and s such that $l! = a$, then

a is the *content* of the location l. The application of the function "!" to its argument l is called the *dereferencing* of l.

A signature extension $\Delta_{dyn} = \langle S_{loc}, F_{dyn} \rangle$, where $F_{dyn} = F_{loc} \cup F_{acc}$, is called a *dynamic signature extension*.

An algebra $\mathtt{A}$ of the signature $\Sigma = \Sigma_{dat} \cup \Delta_{dyn}$ is created by extending a Σ_{dat}-algebra $\mathtt{A}_{\mathtt{dat}}$ with:

- a set of elements (called *locations*) $|\mathtt{A}|_{\mathtt{loc(s)}}$ for each $loc(s) \in S_{loc}$;
- an element $\mathtt{c}^{\mathtt{A}} \in |\mathtt{A}|_{\mathtt{loc(s)}}$ for each constant symbol c_s from F_{loc};
- a partial function $\mathtt{f}^{\mathtt{A}} : |\mathtt{A}|_{\mathtt{s_1}} \times ... \times |\mathtt{A}|_{\mathtt{s_n}} \longrightarrow |\mathtt{A}|_{\mathtt{loc(s)}}$ for each $f_{s_1 \ldots s_n, s} \in F_{loc}$;
- a partial function $\_!^{\mathtt{A}} : |\mathtt{A}|_{\mathtt{loc(s)}} \longrightarrow |\mathtt{A}|_{\mathtt{s}}$ for each $\_!_{loc(s),s} \in F_{acc}$.

Each Σ-algebra must satisfy the following *state invariant*:

- each constant is supplied with a different location,
- unique functions are injective, and
- the range of a unique function does not intersect with both the range of any other function from F_{loc} and the set of locations assigned to constant symbols.

A location constant can be considered as a counterpart of a program variable having an address different from the addresses of all other variables, and a unique location function can be considered as a counterpart of a program array where elements have different addresses and there is a unique address for each index value (note that a unique function is injective).

The set of all locations in an algebra $\mathtt{A}$ is denoted by $|\mathtt{A}|_{\mathtt{loc}}$ in the sequel. The range of a function f in $\mathtt{A}$ is denoted by $\mathtt{Ran(f^A)}$, the range of a constant c in $\mathtt{A}$ is a singleton set $\{\mathtt{c}^{\mathtt{A}}\}$.

The set of Σ-terms is created by extending the usual way of term construction with the following rules:

- if $f_{s_1 \ldots s_n, s} \in F_{dat}$ and t_i, $i = 1, ..., n$, is a term either of sort s_i or $loc(s_i)$, then $f(t_1, ..., t_n)$ is a term of sort s;
- if $f_{s_1 \ldots s_n, s} \in F_{loc}$ and t_i, $i = 1, ..., n$, is a term either of sort s_i or $loc(s_i)$, then $f(t_1, ..., t_n)$ is a term of sort $loc(s)$;
- if l is a term of sort $loc(s)$, then $l!$ is a term of sort s.

Thus, a term of sort $loc(s)$ can be used where a term of sort s is needed. Note that a term constructed with the use of the name of a location function is always a location sort term.

A term constructed in this way is interpreted in a Σ-algebra $\mathtt{A}$ as follows:

- if c is a constant name, then it is interpreted by $\mathtt{c}^{\mathtt{A}}$;
- if f is a (partial) function symbol with the profile $s_1 ... s_n, s$ and t_i, $i = 1, ..., n$, is a term either of sort s_i or $loc(s_i)$, then $f(t_1, ..., t_n)^A = \mathtt{f}^{\mathtt{A}}(\mathtt{v}_1^{\mathtt{A}}, ..., \mathtt{v}_n^{\mathtt{A}})$ where

$$v_i = \begin{cases} t_i & \text{if } t_i \text{ is a term of sort } s_i, \\ t_i! & \text{if } t_i \text{ is a term of sort } loc(s_i); \end{cases}$$

- $f(t_1, ..., t_n)^A$ is undefined if at least one of $\mathtt{v}_i^{\mathtt{A}}$ is undefined or $\mathtt{f}^{\mathtt{A}}$ is undefined for $(\mathtt{v}_1^{\mathtt{A}}, ..., \mathtt{v}_n^{\mathtt{A}})$.

Note that a term of sort $loc(s)$ is dereferenced when it is substituted for a term of sort s. This rule may lead to ambiguities if there are overloaded function names in the signature (the model allows for function overloading). For instance, there could be the following function declaration in the system LINKED_LIST:

next: loc(Node) $\longrightarrow$ Node;

In this case, if *ln* is a term of type Node, the function call *next*(*ln*) may refer to both *next* functions. The mechanism of best matching like that in C++ may be used in such a case.

A location sort, location function, and access function can be different in different states.

Definition 2. A Σ-state is a Σ-algebra where $\Sigma = \Sigma_{dat} \cup \Delta_{dyn}$. ■

Let $\Sigma_{stat} = \Sigma_{dat} \cup C_{loc}$. The restriction of any Σ-state $\mathtt{A}$ to Σ_{stat}, $\mathtt{A}|_{\Sigma_{\mathtt{stat}}}$, is a static algebra called the *base* of $\mathtt{A}$. Several Σ-states can have the same base. Following [8], we denote the set of all Σ-states with the same base $\mathtt{B}$ by $\mathtt{state}_{\mathtt{B}}(\Sigma)$ and mean by a Σ_B-state a Σ-state with the static algebra $\mathtt{B}$. Thus, the carrier of any sort $s \in S_{dat}$ in a Σ-state $\mathtt{A}$ is the same as the carrier of s in $\mathtt{B}$, that is $\mathtt{s}^{\mathtt{A}} = \mathtt{s}^{\mathtt{B}}$ for every $s \in S_{dat}$. Note that the set of locations associated with constant symbols is the same in all states (however, the content of the locations can be different).

4. State updates

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

Definition 3. A *function update* in a Σ_B-state $\mathtt{A}$ is a triple $(\mathtt{f}, \bar{\mathtt{a}}, \mathtt{l})$ where $\mathtt{f}$ is a function symbol in $UF_{loc} \cup SF_{loc}$ with a profile ws, $w = s_1 \dots s_n$, $\bar{\mathtt{a}} = <\mathtt{a}_1, ..., \mathtt{a}_n> \in |\mathtt{A}|_{\mathtt{w}}$, and $\mathtt{l}$ is either an element of $|\mathtt{A}|_{\mathtt{loc(s)}}$ or the symbol $\perp$. ■

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

- $\mathtt{g}^{\mathtt{A}\alpha} = \mathtt{g}^{\mathtt{A}}$ for any g different from f in F_{dyn};
- $\mathtt{f}^{\mathtt{A}\alpha}(\bar{\mathtt{a}}) = \mathtt{l}$ if $\mathtt{l}$ is not $\perp$, $\mathtt{f}^{\mathtt{A}\alpha}(\bar{\mathtt{a}})$ becomes undefined otherwise;
- $\mathtt{f}^{\mathtt{A}\alpha}(\bar{\mathtt{a}'}) = \mathtt{f}^{\mathtt{A}}(\bar{\mathtt{a}'})$ for any tuple $\bar{\mathtt{a}'} = <\mathtt{a}'_1, \dots, \mathtt{a}'_n>$ different from $\bar{\mathtt{a}}$;
- $|\mathtt{A}\alpha|_{\mathtt{s}} = |\mathtt{A}|_{\mathtt{s}}$ for any $s \in S_{loc}$.

Following Gurevich [6], we say that $\mathtt{A}\alpha$ is obtained by firing the update α on $\mathtt{A}$. Roughly speaking, firing a function update either inserts an element to the definition domain of a location function or modifies the value of such a function at one point in its definition domain or removes an element from the definition domain.

Note that $\mathtt{f}$ in a triple $(\mathtt{f}, \bar{\mathtt{a}}, \mathtt{l})$ is actually a function symbol, i. e., a function name qualified by its profile, which helps us to avoid ambiguity when function names are overloaded.

Fact 1. *A function update* $(\mathtt{f}, \bar{\mathtt{a}}, \mathtt{l})$ *is legal in* $\mathtt{A}$ *if:*

- $\mathtt{l} \notin \mathtt{Ran}(\mathtt{g}^{\mathtt{A}})$ *for any* $g \in F_{loc}$ *if* $f \in UF_{loc}$;
- $\mathtt{l} \notin \mathtt{Ran}(\mathtt{h}^{\mathtt{A}})$ *for any* $h \in UF_{loc}$ *if* $f \in SF_{loc}$. ■

Indeed, the first case guarantees that an update of a unique function does not violate the state invariant, and the second case guarantees that an update of a shared function does not violate the state invariant.

Definition 4. A *location update* in a Σ_B-state $\mathtt{A}$ is a triple $(\mathtt{s},\ \mathtt{l},\ \mathtt{a})$, where $\mathtt{s}$ is a sort name, $\mathtt{l}$ an element of sort $|\mathtt{A}|_{\mathtt{loc(s)}}$, and $\mathtt{a}$ either an element of sort $|\mathtt{A}|_{\mathtt{s}}$ or the symbol $\perp$. ■

A location update $\beta = (\mathtt{s}, \mathtt{l}, \mathtt{a})$ serves for the transformation of a Σ_B-state $\mathtt{A}$ into a new Σ_B-state $\mathtt{A}\beta$ in the following way:

- $\mathtt{g}^{\mathtt{A}\beta} = \mathtt{g}^{\mathtt{A}}$ for any g in F_{dyn} different from $\_!_{loc(s),s}$;
- $(\mathtt{l}!)^{\mathtt{A}\beta} = \mathtt{a}$ if $\mathtt{a}$ is not $\perp$, $(\mathtt{l}!)^{\mathtt{A}\beta}$ becomes undefined otherwise;
- $(\mathtt{l}'!)^{\mathtt{A}\beta} = (\mathtt{l}'!)^{\mathtt{A}}$ for any $\mathtt{l}' \in |\mathtt{A}|_{\mathtt{loc(s)}}$ different from $\mathtt{l}$;
- $|\mathtt{A}\beta|_{\mathtt{s}} = |\mathtt{A}|_{\mathtt{s}}$ for any $s \in S_{loc}$.

We say that $\mathtt{A}\beta$ is obtained by applying (or firing) the update β on $\mathtt{A}$. Roughly speaking, firing a location update either initializes a location or updates its content or makes its content undefined. The following fact is self-evident:

Fact 2. *A location update is legal, i. e., it does not violate the state invariant.* ■

Definition 5. Let $\mathtt{s}$ be the name of a location sort from S_{loc}. A *sort update* δ in $\mathtt{A}$ is either a triple $(+, \mathtt{s}, \mathtt{l})$ where $\mathtt{l}$ is an element such that $\mathtt{l} \notin |\mathtt{A}|_{\mathtt{loc}}$, or a triple $(-, \mathtt{s}, \mathtt{l})$ where $\mathtt{l}$ is an element of $|\mathtt{A}|_{\mathtt{s}}$ that is neither used nor produced by constants or functions associated with symbols from F_{loc}. ■

A sort update $\delta = (+, \mathtt{s}, \mathtt{l})$ transforms a Σ_B-state $\mathtt{A}$ into a new Σ_B-state $\mathtt{A}\delta$ in the following way:

- $|\mathtt{A}\delta|_{\mathtt{s}} = |\mathtt{A}|_{\mathtt{s}} \cup \{\mathtt{l}\}$;
- $|\mathtt{A}\delta|_{\mathtt{s}'} = |\mathtt{A}|_{\mathtt{s}'}$ for any $s' \in S_{loc}$ different from s;
- $\mathtt{f}^{\mathtt{A}\delta} = \mathtt{f}^{\mathtt{A}}$ for any $f_{ws} \in F_{dyn}$.

Thus, the sort update $\delta = (+, \mathtt{s}, \mathtt{l})$ extends the set of elements of a certain location sort by a new element different from any location existing in $\mathtt{A}$.

A sort update $\delta = (-, \mathtt{s}, \mathtt{l})$ transforms a Σ_B-state $\mathtt{A}$ into a new Σ_B-state $\mathtt{A}\delta$ in the following way:

- $|\mathtt{A}\delta|_{\mathtt{s}} = |\mathtt{A}|_{\mathtt{s}} \setminus \{\mathtt{l}\}$;
- $|\mathtt{A}\delta|_{\mathtt{s}'} = |\mathtt{A}|_{\mathtt{s}'}$ for any s' different from s;
- $\mathtt{f}^{\mathtt{A}\delta} = \mathtt{f}^{\mathtt{A}}$ for any $f_{ws} \in F_{dyn}$.

Thus, the sort update $\delta = (-, \mathtt{s}, \mathtt{l})$ contracts the set of elements of a certain location sort by an element that is not associated with a constant symbol and not in the graph of any location function.

Fact 3. *Both kinds of sort updates are legal, i. e., none of them violates the state invariant.* ■

Indeed, in a sort update $\delta = (+, \mathtt{s}, \mathtt{l})$ the location $\mathtt{l}$ is not yet in the range of any function, and in a sort update $\delta = (-, \mathtt{s}, \mathtt{l})$ the location $\mathtt{l}$ is not in the range of any function by definition.

We will consider only legal state updates in the sequel. A set of legal function/location/sort updates is called an *update set.*

Definition 6. Let Γ be a set of function/location/sort updates. The set Γ is *inconsistent* if it contains:

- two contradictory function updates of the following kind: $\alpha_1 = (\mathtt{f}, \bar{\mathtt{a}}, \mathtt{a})$ and $\alpha_2 = (\mathtt{f}, \bar{\mathtt{a}}, \mathtt{a}')$, where $\mathtt{a} \neq \mathtt{a}'$ (two contradictory function updates define the function differently at the same point), or
- two contradictory location updates of the following kind: β_1 = (s, l, a) and $\beta_2 = (\mathtt{s}, \mathtt{l}, \mathtt{a}')$, where $\mathtt{a} \neq \mathtt{a}'$ (two contradictory location updates define an access function differently at the same point);

the update set is *consistent* otherwise. ■

A consistent update set $\mathtt{\Gamma}$ applied to a Σ_B-state $\mathtt{A}$ transforms $\mathtt{A}$ into a new Σ_B-state $\mathtt{A}'$ by the simultaneous firing of all $\alpha \in \mathtt{\Gamma}$, $\beta \in \mathtt{\Gamma}$, and $\delta \in \mathtt{\Gamma}$. If $\mathtt{\Gamma}$ is inconsistent, the new state in not defined. If $\mathtt{\Gamma}$ is empty, $\mathtt{A}'$ is the same as $\mathtt{A}$. Following [9], we denote the application of $\mathtt{\Gamma}$ to a state $\mathtt{A}$ by $\mathtt{A\Gamma}$. The set of all consistent sets of updates in $\mathtt{state_B(\Sigma)}$ is denoted by $\overline{\mathtt{update}}_{\mathtt{B}}(\Sigma)$ in the sequel.

Definition 7. Let $\mathtt{\Gamma_1}$ and $\mathtt{\Gamma_2}$ be two consistent update sets in a Σ_B-state $\mathtt{A}$, $\alpha_1 = (\mathtt{f}, \langle \mathtt{a_1}, \dots, \mathtt{a_n} \rangle, \mathtt{a})$, $\alpha_2 = (\mathtt{f}, \langle \mathtt{a_1}, \dots, \mathtt{a_n} \rangle, \mathtt{a}')$, $\beta_1 = (\mathtt{s}, \mathtt{l}, \mathtt{a})$, and $\beta_2 = (\mathtt{s}, \mathtt{l}, \mathtt{a}')$ where $\mathtt{a} \neq \mathtt{a}'$. The *sequential union* of $\mathtt{\Gamma_1}$ and $\mathtt{\Gamma_2}$, denoted by $\mathtt{\Gamma_1;\Gamma_2}$, is defined as follows: $\mathtt{u} \in \mathtt{\Gamma_1;\Gamma_2}$ iff $\mathtt{u} \in \mathtt{\Gamma_1}$ or $\mathtt{u} \in \mathtt{\Gamma_2}$ except the following cases:

- if $\alpha_1 \in \mathtt{\Gamma_1}$ and $\alpha_2 \in \mathtt{\Gamma_2}$, then $\alpha_2 \in \mathtt{\Gamma_1;\Gamma_2}$ and $\alpha_1 \notin \mathtt{\Gamma_1;\Gamma_2}$;
- if $\beta_1 \in \mathtt{\Gamma_1}$ and $\beta_2 \in \mathtt{\Gamma_2}$, then $\beta_2 \in \mathtt{\Gamma_1;\Gamma_2}$ and $\beta_1 \notin \mathtt{\Gamma_1;\Gamma_2}$. ■

Thus, in a sequential union of update sets, each next update of a location/access function at a certain point waives each preceding update of this function at the same point. It is not difficult to prove the following:

Fact 4. *If* $\mathtt{\Gamma_1} \cup \mathtt{\Gamma_2}$ *is consistent, then for any consistent* $\mathtt{\Gamma}$*:*

$$\mathtt{\Gamma;\Gamma_1} \cup \mathtt{\Gamma;\Gamma_2} = \mathtt{\Gamma;(\Gamma_1} \cup \mathtt{\Gamma_2)}.$$

Fact 5. *If* $\mathtt{\Gamma}$, $\mathtt{\Gamma_1}$ *and* $\mathtt{\Gamma_2}$ *are consistent update sets, then*

$$\mathtt{\Gamma;\Gamma_1} = \mathtt{\Gamma;\Gamma_2} \textit{ if } \mathtt{\Gamma_1} = \mathtt{\Gamma_2}.$$

Fact 6. *For any* Σ_B*-state* $\mathtt{A}$ *and all consistent update sets* $\mathtt{\Gamma_1}$ *and* $\mathtt{\Gamma_2}$ *in* Σ_B*-states,* $\mathtt{A(\Gamma_1;\Gamma_2)} = \mathtt{(A\Gamma_1)\Gamma_2}$.

As an immediate consequence, for any sequence of update sets $\mathtt{\Gamma_1}, \cdots, \mathtt{\Gamma_n}$ and any algebra $\mathtt{A}$, the algebra $\mathtt{A\Gamma}$, where $\mathtt{\Gamma} = \mathtt{\Gamma_1}; \cdots ;\mathtt{\Gamma_n}$, is the algebra produced by the application of $\mathtt{\Gamma}$ to $\mathtt{A}$.

5. Dependant functions

A number of *dependant (derived) functions* can be defined on the $state_B(\Sigma)$ using the functions from Δ_{dyn} and the operations from Σ_{dat}. The values produced by these functions depend both on the particular state and the values of arguments if any. The names and profiles of these functions, $\Delta_{dep} = (F_{dep})$, are introduced in the third part of the system signature with the use of sorts of Σ. For example, the following dependant functions can be defined in the system LINKED_LIST:

depend
func has: Node $\longrightarrow$ Boolean; ** *checks whether a node is in the list*
func find: Node $\longrightarrow$ loc(Node); ** *fetches a location with a given node*

Definition 8. A Σ'_B-state is a Σ'-algebra with the static algebra $\mathtt{B}$ where $\Sigma' = \Sigma \cup \Delta_{dep}$. ■

A Σ'-algebra $\mathtt{A}'$ is created by extending a Σ-algebra $\mathtt{A}$ with a (partial) function $\mathtt{f}^{\mathtt{A}} : |\mathtt{A}|_{\mathtt{s_1}} \times ... \times |\mathtt{A}|_{\mathtt{s_n}} \longrightarrow |\mathtt{A}|_{\mathtt{s}}$ for each $f_{s_1 \ldots s_n, s} \in F_{dep}$.

Thus, dependant functions extend a Σ_B-state to a Σ'_B-state. The set of all Σ'_B-states with the same static algebra $\mathtt{B}$ is denoted by $\mathtt{state}_{\mathtt{B}}(\Sigma')$. Σ'-terms are constructed in the same way as Σ-terms using function symbols from F_{dep} in addition.

6. Procedures

A state update modifies location sorts and/or location and access functions. Several state updates can be performed by a *procedure* defined in the fourth part of the system signature Δ_{proc} that consists of a set P_w of *procedure names* where w is a sequence of *argument sorts.* An argument sort is either s or *ref* s where $s \in S_{dat} \cup S_{loc}$. An argument sort of the form s indicates that an element of the corresponding sort is passed to the procedure when it is invoked (*call-by-value*) while an argument sort of the form *ref* s indicates that an element of sort $loc(s)$ is passed to the procedure when it is invoked (*call-by-reference*). For example, the following procedures can be defined in the system LINKED_LIST:

proc
initialize; ** *construction of the empty list*
insert: loc(Node), Node; ** *insertion of a node in the list*
update: ref Node, Node; ** *update of the node data*

7. Dynamic system

Definition 9. A dynamic system $\mathtt{DS(B)}$ of signature $D\Sigma = \Sigma' \cup \Delta_{proc}$ consists of

- a set of states $|\mathtt{DS(B)}| = \mathtt{state}_{\mathtt{B}}(\Sigma')$ with the same static algebra $\mathtt{B}$ called the *carrier* of the system;
- a partial surjective function $\mathtt{map}^{\mathtt{DS(B)}} : \mathtt{state}_{\mathtt{B}}(\Sigma) \longrightarrow |\mathtt{DS(B)}|$ such that, for each $\mathtt{A} \in \mathtt{state}_{\mathtt{B}}(\Sigma)$, $\mathtt{map}^{\mathtt{DS(B)}}(\mathtt{A})|_{\Sigma} = \mathtt{A}$ if $\mathtt{map}^{\mathtt{DS(B)}}(\mathtt{A})$ is defined;

- a partial map $\mathtt{p}^{\mathtt{DS(B)}}$, for each procedure symbol $p_{u_1 \cdots u_n}$, where u_i is either s_i or *ref* s_i, associating an update set $\Gamma \in \overline{\mathtt{update}_{\mathtt{B}}}(\Sigma)$ with each pair $\langle \mathtt{A}, \langle \mathtt{a_1}, \cdots, \mathtt{a_n} \rangle\rangle$, where $\mathtt{A} \in |\mathtt{DS(B)}|$ and $\mathtt{a_i} \in |\mathtt{A}|_{\mathtt{loc(s_i)}}$ if u_i is *ref* s_i and $\mathtt{a_i} \in |\mathtt{A}|_{\mathtt{s_i}}$ in the opposite case. ■

Note that the procedure produces an update set.

We write $\mathtt{p}^{\mathtt{DS(B)}}(\mathtt{A}, \bar{\mathtt{a}})$ for the application of a procedure $\mathtt{p}^{\mathtt{DS(B)}}$ to $\langle \mathtt{A}, \bar{\mathtt{a}} \rangle$, where $\mathtt{A}$ is a state, and $\bar{\mathtt{a}} = \langle \mathtt{a_1}, ..., \mathtt{a_n} \rangle$. When $n = 0$, the tuple $\bar{\mathtt{a}}$ is the empty tuple $\langle\rangle$.

A new kind of term, *transition term*, is created with the use of procedure names.

Definition 10. If $p_{u_1 \ldots u_n}$ is a procedure symbol, where u_i, $i = 1, ..., n$, is either s_i or *ref* s_i, and

$$t_i \text{ is } \begin{cases} \text{a } \Sigma'\text{-}\textit{term} \text{ of sort } loc(s_i) \text{ when } u_i \text{ is } \textit{ref}\ s_i, \\ \text{a term either of sort } s_i \text{ or } loc(s_i) \text{ in the opposite case,} \end{cases}$$

then $p(t_1, ..., t_n)$ is a *transition term* called a *procedure call.* ■

Interpretation. Let $\mathtt{DS(B)}$ be a dynamic system of signature $D\Sigma = \Sigma' \cup \Delta_{proc}$ and $p(t_1, ..., t_n)$ a transition term constructed as above. We denote by t^A the interpretation of a Σ-term t in a Σ_B-state $\mathtt{A}$ and by $[\![t]\!]^{A(B)}$ the interpretation of a transition term t in $\mathtt{DS(B)}$ in the current Σ_B-state $\mathtt{A}$. The interpretation of a procedure call $p(t_1, ..., t_n)$ is defined as follows:

$[\![p(t_1, ..., t_n))]\!]^{A(B)} = \mathtt{p}^{\mathtt{DS(B)}}(\mathtt{A}, \langle \mathtt{v_1^A}, ..., \mathtt{v_n^A} \rangle)$, where

$$v_i = \begin{cases} t_i & \text{if either } u_i \text{ is } \textit{ref}\ s_i \text{ and } t_i \text{ is a term of sort } loc(s_i) \\ & \text{or } u_i \text{ is } s_i \text{ and } t_i \text{ is a term of sort } s_i, \\ t_i! & \text{if } u_i \text{ is } s_i \text{ and } t_i \text{ is a term of sort } loc(s_i), \end{cases}$$

if $\mathtt{p}^{\mathtt{DS(B)}}$ is defined for the tuple $\langle \mathtt{v_1^A}, ..., \mathtt{v_n^A} \rangle$ in the state $\mathtt{A}$; $[\![p(t_1, ..., t_n))]\!]^{A(B)}$ is undefined otherwise.

Thus, an argument is dereferenced when a location is substituted where its content is needed, and it is directly substituted in all other cases.

8. Transition rules

State updates are specified by means of *transition rules.* A transition rule is a special kind of transition term. It is applicable either to a location function or to an access function from F_{dyn}. The successful interpretation of a transition term R in a dynamic system $\mathtt{DS(B)}$ in a state $\mathtt{A}$ produces an update set Γ. The resulting state A' can be obtained as $A' = \mathtt{map}^{\mathtt{DS(B)}}(\mathtt{A}|_{\Sigma}\Gamma)$. That is, the current Σ'_B-state is reduced to a Σ_B-state, updated by Γ, and converted into the corresponding Σ'_B-state. This two-way transformation of a state is needed for providing the appropriate dependant functions in the resulting state.

8.1. Basic transition rules

Update instructions. Let f be the name of a partial location function with profile $s_1 \ldots s_n s$, t_i, $i = 1, ..., n$, a term either of sort s_i or $loc(s_i)$, t a term either of sort s or $loc(s)$, and lt a term of sort $loc(s)$ over signature Σ. Then

$$\begin{aligned}&f(t_1, ..., t_n) \leftarrow lt,\\ &f(t_1, ..., t_n) \leftarrow undef,\\ &lt := t,\\ &lt := undef\end{aligned}$$

are transition rules called *update instructions*. The first two instructions serve for updating a partial location function while the second two serve for updating an access function.

Interpretation. Let $\mathtt{A}$ be the current state of $\mathtt{DS(B)}$. If lt, t and t_i are defined in $\mathtt{A}$, then

$$\begin{aligned}&[\![f(t_1, ..., t_n) \leftarrow lt]\!]^{A(B)} = \{(\mathtt{f}, \langle \mathtt{v}_1^{\mathtt{A}}, \ldots, \mathtt{v}_n^{\mathtt{A}}\rangle, \mathtt{lt}^{\mathtt{A}})\},\\ &[\![f(t_1, ..., t_n) \leftarrow undef]\!]^{A(B)} = \{(\mathtt{g}, \langle \mathtt{v}_1^{\mathtt{A}}, \ldots, \mathtt{v}_n^{\mathtt{A}}\rangle, \bot)\},\\ &[\![lt := t]\!]^{A(B)} = \{(\mathtt{s}, \mathtt{lt}^{\mathtt{A}}, \mathtt{v}^{\mathtt{A}})\},\\ &[\![lt := undef]\!]^{A(B)} = \{(\mathtt{s}, \mathtt{lt}^{\mathtt{A}}, \bot)\},\end{aligned}$$

where $v = t$ and $v_i = t_i$ if t and t_i are terms of sorts s and s_i, respectively, and $v = t!$ and $v_i = t_i!$ if t and t_i are terms of sorts $loc(s)$ and $loc(s_i)$, respectively. If at least one of v_i, lt, v is not defined in $\mathtt{A}$, then the semantics of a rule including this term is undefined.

Examples. Let x_{Nat} be a location constant and $f_{Nat,Nat}$ a partial location function symbol from F_{loc}. The execution of the transition rule

$$f(x) := f(x) + 1$$

will transform a state $\mathtt{A}$ into a state $\mathtt{A}'$ so that $f^{A'}(x!^{A})! = f^{A}(x!^{A})! + 1$. The execution of the transition rule

$$x := undef$$

will make $x!$ undefined in the new state. The execution of the transition rule

$$f(x) \leftarrow l,$$

where l is a term of sort $loc(Nat)$, will transform a state $\mathtt{A}$ into a state $\mathtt{A}'$ so that $f^{A'}(x!^A) = l^A$.

Fact 7. *An update instruction $f(t_1, ..., t_n) \leftarrow lt$ evaluates in $\mathtt{A}$ to a legal function update if:*

- $\mathtt{lt}^{\mathtt{A}} \notin \mathtt{Ran}(\mathtt{g}^{\mathtt{A}})$ *for any $g \in F_{loc}$ if $f \in UF_{loc}$;*
- $\mathtt{lt}^{\mathtt{A}} \notin \mathtt{Ran}(\mathtt{h}^{\mathtt{A}})$ *for any $h \in UF_{loc}$ if $f \in SF_{loc}$;*

all other kinds of update instruction always evaluate to a legal function update. ■

The fact immediately follows from facts 1 and 2.

Sort contraction instruction. If x is a variable of a location sort s, then **drop** x is a transition term called a *sort contraction instruction.*

Interpretation. Let $\mathtt{A}$ be a state and $\sigma : \{x\} \longrightarrow |\mathtt{A}|$ a variable assignment. Then

$[\![\textbf{drop } x]\!]^{A(B),\sigma} = \{\delta\}$ where $\delta = (-, \mathtt{s}, \mathtt{x}^{\mathtt{A},\sigma})$ if there is no constant c such that $\mathtt{c}^{\mathtt{A}} = \mathtt{x}^{\mathtt{A},\sigma}$ and no location function $\mathtt{f}^{\mathtt{A}} : |\mathtt{A}|_{\mathtt{s_1}} \times ... \times |\mathtt{A}|_{\mathtt{s_n}} \longrightarrow |\mathtt{A}|_{\mathtt{s_{n+1}}}$ with a maplet $\langle \mathtt{a_1}, ..., \mathtt{a_n} \mapsto \mathtt{a_{n+1}} \rangle$ such that $\mathtt{x}^{\mathtt{A},\sigma} = \mathtt{a_i}$, $i = 1, ..., n+1$;

$[\![\textbf{drop } t]\!]^{A(B),\sigma} = \emptyset$ otherwise.

Fact 8. *The interpretation of the sort contraction instruction does not violate the state invariant.* ■

Indeed, the interpretation of the sort contraction instruction produces either a legal sort update or the empty update set. Therefore, an element of a location sort is not deleted if it is used in this state.

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

8.2. Rule constructors

Complex transition terms are recursively constructed from update instructions and procedure calls by means of several rule constructors, e.g., *sequence constructor*, *set constructor*, *condition constructor*, *guarded update*, and *loop constructors.*

Sequence constructor. If $R_1, \dots, R_n$ are transition terms, then **seq** $R_1, \dots, R_n$ **end** is a transition term called a *sequence of transition rules.*

Interpretation. Let $\mathtt{A}$ be a state, $\Gamma_1 = [\![R_1]\!]^{A(B)}$, $\mathtt{A}_1 = \mathtt{A}\Gamma_1$, $\Gamma_2 = [\![R_2]\!]^{A_1(B)}$, $\mathtt{A}_2 = \mathtt{A}_1\Gamma_2$, ..., $\Gamma_n = [\![R_n]\!]^{A_{n-1}(B)}$. Then

$$[\![\textbf{seq } R_1, R_2, \ldots, R_n \textbf{ end}]\!]^{A(B)} = \Gamma,$$

where $\Gamma = \Gamma_1;\Gamma_2;...;\Gamma_n$ and each $[\![R_i]\!]^{A_{i-1}(B)}$ is defined.

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

Set constructor. If $R_1, \ldots, R_n$ are transition terms, then **set** $R_1, \ldots, R_n$ **end** is a transition term called a *set of transition rules.*

Interpretation. Let $\mathtt{A}$ be a state and $\Gamma_1 = [\![R_1]\!]^{A(B)}, \ldots, \Gamma_n = [\![R_n]\!]^{A(B)}$. Then

$$[\![\textbf{set } R_1, \ldots, R_n \textbf{ end}]\!]^{A(B)} = \Gamma_1 \cup \ldots \cup \Gamma_n$$

if each $[\![R_i]\!]^{A(B)}$ 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 rules, execute all of them in parallel and unite the results if they are defined and consistent.

Example. Let $x_{Nat}, y_{Nat}, z_{Nat}, f_{Nat,Nat}$ be location function symbols from F_{loc}. Then the execution of a set of rules:

$$\textbf{set } f(x) := y, y := x, x := z \textbf{ end}$$

will produce a new state $\mathtt{A}'$ where $f^{A'}(x!^{A})! = y!^{A}$; $y!^{A'} = x!^{A}$; $x!^{A'} = z!^{A}$.

Conditional constructor. If k is a natural number, $g_0, ..., g_k$ are Boolean terms, and $R_0, ..., R_k$ are transition terms, then the following expression is a transition term called a *conditional transition rule*:

```
if $g_0$ then $R_0$
elseif $g_1$ then $R_1$
    .
    .
elseif $g_k$ then $R_k$
endif
```

If g_k is the constant *true*, then the last **elseif** clause can be replaced with "**else** R_k".

Interpretation. Let $\mathtt{A}$ be a state and R a conditional transition rule, then

$$[\![R]\!]^{A(B)} = [\![R_i]\!]^{A(B)}$$

if g_i holds in $\mathtt{A}$, but every g_j with $j < i$ fails in $\mathtt{A}$. $[\![R]\!]^{A(B)} = \emptyset$ if every g_i fails in $\mathtt{A}$.

A **guarded update** instruction is a rule of the form **if** g **then** R where g is a Boolean term and R is a transition term.

Interpretation. Let $R1 =$ **if** g **then** R. Then $[\![R1]\!]^{A(B)} = [\![R]\!]^{A(B)}$ if g holds in $\mathtt{A}$; $[\![R1]\!]^{A(B)} = \emptyset$ otherwise. In other words, perform the transition if the condition holds and do nothing in the opposite case.

Loop constructors. The guarded update together with the sequence constructor gives us a possibility to define some loop constructors. If R is a transition term and g is a Boolean term, then

while g **do** R and
do R **until** g

are transition terms called *loops*.

Interpretation.

$$[\![\textbf{while } g \textbf{ do } R]\!]^{A(B)} = [\![\textbf{if } g \textbf{ then seq } R, \textbf{ while } g \textbf{ do } R \textbf{ end}]\!]^{A(B)};$$
$$[\![\textbf{do } R \textbf{ until } g]\!]^{A(B)} = [\![\textbf{seq } R, \textbf{ if} \neg g \textbf{ then do } R \textbf{ until } g]\!]^{A(B)}.$$

Import constructor. If x is a variable, s is a location sort name, R is a transition term using x as a term of type $loc(s)$ and not having x as a fresh variable, and there is no total location function symbol $f_{ws'}$ such that s is in w, then

import x: s **in** R

is a transition term with a fresh variable x called an *import term*.

Interpretation.

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

where $\mathtt{A}' = \mathtt{A}\delta$, $\delta = (+, \mathtt{s}, \mathtt{a})$ for some $\mathtt{a} \notin |\mathtt{A}|_{\mathtt{loc}}$, and $\sigma = \{x \mapsto \mathtt{a}\}$ is a variable assignment.

Note that one cannot insert a new element into a location sort if the sort is used in the domain of a total function (otherwise the function becomes partial).

8.3. Massive update

A massive update permits the specification of a parallel update of one or more sorts/functions at several points. The corresponding transition term has the following form:

forall $x_1 : s_1, ..., x_n : s_n.R,$

where $x_1, ..., x_n$ are variables of sorts $s_1, ..., s_n$, respectively, and R is a transition term having no free variables.

Interpretation. Let $\mathtt{A}$ be a state. Then for all $\sigma : \{x_1, ..., x_n\} \longrightarrow |\mathtt{A}|$:

- if $\Gamma = \bigcup\{[\![R]\!]^{A(B),\sigma}\}$ is defined and consistent, then
$$[\![\mathbf{forall}\ x_1 : s_1, ..., x_n : s_n.R]\!]^{A(B)} = \Gamma;$$
- $[\![\mathbf{forall}\ x_1 : s_1, ..., x_n : s_n.R]\!]^{A(B)}$ is not defined otherwise.

Example. Let $f_{Nat,Nat}$ be a location function. A transition rule

forall x: Nat. $f(x) := f(x) + 1$

is equivalent to the set of rules

$\{(f(t) := f(t) + 1) : \ t \in T(\Sigma')_{Nat}\}$.

This means that $f^{A'}(t^A)! = f^A(t^A)! + 1$ if $f^A(t^A)!$ is defined for all t.

9. Dynamic formulae

For the specification of dynamic systems we introduce dynamic formulae which can be either *procedure definition* or *precondition* formula.

$$F \quad ::= \quad p(x_1, ..., x_n) = tt \mid \mathbf{pre}\ p(t_1, \ldots, t_n) : \varphi$$

A procedure definition serves for the specification of the behavior of a procedure in terms of a transition rule, and a precondition formula allows us to define the domain of a partial procedure.

9.1. Procedure definition

A procedure definition has the following form:

$p(x_1, ..., x_n) == tt$

where $p_{u_1 \ldots u_n}$ is a procedure symbol such that u_i is either s_i or *ref* s_i, $X = \{x_1, ..., x_n\}$ a set of variables such that

$$x_i \text{ is of sort } \begin{cases} s_i \text{ if } u_i \text{ is } s_i, \\ loc(s_i) \text{ if } u_i \text{ is } ref\ s_i, \end{cases}$$

and tt a transition term over X.

The variables $x_1, ...x_n$ are called *procedure parameters*, and the term tt is called a *procedure body*. A procedure parameter x_i is called a *value parameter* if u_i is s_i, and it is called a *reference parameter* if u_i is *ref* s_i.

A dynamic system $\mathtt{DS(B)}$ satisfies a procedure definition $p(x_1, ..., x_n) == tt$ iff for all states $\mathtt{A}$ of $\mathtt{DS(B)}$ and variable assignments $\sigma : X \to |\mathtt{A}|$:

$$\mathtt{A}|_{\Sigma}[\![p(x_1,...,x_n)]\!]^{A(B),\sigma} = \mathtt{A}_2$$

where the algebra $\mathtt{A}_2$ is created as follows.

We know that in a programming language a value parameter is typically regarded in the procedure body as a local variable (location) whose update does not influence the environment in which the procedure is called. We will follow the same line in giving semantics for our procedure definition. Let $\bar{\Sigma}$ be the extension of Σ' by symbols $y_i : loc(s_i)$ for those $1 \leq i \leq n$ for which u_i is s_i in the profile of p, such that no y_i is an operation (function) symbol in Σ'. A variable x_i is said to *correspond* to y_i in the sequel. Given a Σ'-algebra $\mathtt{A}$, a $\bar{\Sigma}$-algebra $\bar{\mathtt{A}}$ is created as an extension of $\mathtt{A}$ in the following way:

- initially $|\bar{\mathtt{A}}| = |\mathtt{A}|$; then, for each u_i that is s_i in the profile of p, the sort $\bar{\mathtt{A}}_{\mathtt{loc(s_i)}}$ is extended by an element $\mathtt{y}_\mathtt{i}^{\bar{\mathtt{A}}} \notin \mathtt{loc(A)}$,
- each y_i is mapped to the element $\mathtt{y}_\mathtt{i}^{\bar{\mathtt{A}}}$.

Note that a sort $\bar{\mathtt{A}}_{\mathtt{loc(s)}}$will be expanded several times if s is used several times in the profile of p as a value parameter sort.

Now $\mathtt{A}_2 = (\bar{\mathtt{A}}\bar{\Gamma})|_{\Sigma}$ where

$$\bar{\Gamma} = [\![\textbf{seq set } y_i := x_i \textbf{ end},\ tt' \textbf{ end}]\!]^{\bar{A},\sigma}$$

and **set** $y_i := x_i$ **end** means the parallel update of each y_i by the corresponding x_i, the term tt' is obtained from tt by replacing with y_i each x_i corresponding to y_i (this means that the term tt is type-checked only when it is converted into tt', so the specifier, when writing the term tt, may consider x_i as a term of sort $loc(s_i)$, i. e., may assign to it).

Thus, a value parameter $x_i : s_i$ is a term of sort $loc(s_i)$ in the procedure body, like a location constant $c : s_i$ is a term of sort $loc(s_i)$. This corresponds one-to-one to the practice of imperative programming languages.

Example. Let

p: int, loc(int);

be a procedure declaration and $x : int,\ l : loc(int)$ be variables. Then the right-hand side of procedure definition

p(x, l) == l := x;

will be converted into the transition term

seq set *xy := x, ly := l* **end**, *ly := xy* **end**

where ly and xy are local location constants of sorts $loc(loc(int))$ and $loc(x)$, respectively, initialized by procedure arguments. Therefore, the interpretation of the instruction $ly := xy$ will actually put the location xy into the location ly without any effect on the state in which p is invoked. However, if the right-hand side of procedure definition were the instruction $l! := x$, then the value residing in the location xy (i. e., x) would be put in the location l passed as argument, and the state in which p is invoked would be updated.

If the procedure is declared as follows:

p: int, ref int;

and defined as follows:

p(x, l) == l := x;

then the right-hand side of the definition will be converted into the transition term

seq set *xy := x* **end**, *l := xy* **end**.

Its interpretation will put x in the location l, thus updating the state in which the procedure is invoked.

9.2. Precondition formula

A *precondition formula* of the form

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

where p is a procedure name, $t_1, \ldots, t_n$ terms over Σ' with variables from a set X, and φ a Boolean term, can be used to state under what conditions a partial procedure p is guaranteed to produce a result.

A dynamic system DS(B) satisfies a precondition formula

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

iff $\mathtt{p}^{\mathtt{DS(B)}}(\mathtt{A}, <\mathtt{t}_1^{\mathtt{A},\sigma}, \ldots, \mathtt{t}_n^{\mathtt{A},\sigma}>)$ is defined only in those states A and only for those variable assignments $\sigma : X \to |\mathtt{A}|$ for which φ holds.

10. Specification of a dynamic system

Let $DSS = \langle(\Sigma_{dat}, Ax_{dat}), (\Delta_{dyn}), (\Delta_{dep}, Ax_{dep}), (\Delta_{proc}, Ax_{proc})\rangle$ be a dynamic system specification. It has four levels:

- The first level is an algebraic specification $\langle\Sigma_{dat}, Ax_{dat}\rangle$ which defines the data types used in the system.
- The second level defines those aspects of the system state which are likely to change. It includes a signature extension F_{loc} that declares some *location* functions. F_{loc} is extended to Δ_{dyn} as described in Section 3.

  A model of the $\langle\Sigma_{dat} \cup \Delta_{dyn}, Ax_{dat}\rangle$ specification is a Σ-algebra where $\Sigma = \langle\Sigma_{dat} \cup \Delta_{dyn}\rangle$.

- The third level defines some dependant functions. It uses the sort terms from Σ and Δ_{dyn}, the names of locaton/access functions from Δ_{dyn}, the names of dependant functions from Δ_{dep}, and the operations of Σ_{dat}. The specification $\langle \Delta_{dep}, Ax_{dep} \rangle$ must be sufficiently complete and hierarchically consistent with respect to $\langle \Sigma, Ax_{dat} \rangle$. This reflects the fact that a Σ-state is a Σ-algebra where $\Sigma = \langle \Sigma_{dat} \cup \Delta_{dyn} \rangle$. A model of the $\langle \Sigma \cup \Delta_{dep}, Ax_{dep} \rangle$ specification is a Σ'-algebra where Σ' $= \langle \Sigma \cup \Delta_{dep} \rangle$. Thus, a formula from Ax_{dep} must hold in any Σ'-state of the dynamic system. Note that the function $map^{DS(B)}$ maps Σ-states to those Σ'-states which satisfy all formulae from Ax_{dep}.

  Since a dependant function can be partial, a definedness predicate **D** is used in specifications. That is, if t is a term, then the predication $\mathbf{D}(t)$ holds in an algebra A iff t is defined in it. A precondition formula of the form **dom** $t : bt$ states in the specification that the term t is defined iff the Boolean term bt evaluates to true.

- The fourth level, $\langle \Delta_{proc}, Ax_{proc} \rangle$, defines some *procedures* by means of *dynamic formulae.* If a dynamic formula *de* holds in DS(B), we say that DS(B) is a *model* of *de*. A dynamic formula *de* is *consistent* if there is at least one model of it.

  If *DE* is a set of consistent dynamic formulae, then DS(B) is a model of *DE* if each $de \in DE$ holds in DS(B).

Example. Specification of the "LINKED_LIST" system.

```
System LINKED_LIST ** we ignore the function prev in this specification
  use BOOLEAN, NAT, NODE; ** the specifications used
  location
    const head: Node;
    sfunc next: Node ⟶ Node;
    ufunc array: Nat ⟶ Node;
  depend
    func local_has: loc(Node), Node ⟶ Boolean;
             ** auxiliary function needed for the specification of the next one
    func has: Node ⟶ Boolean;
    func local_find: loc(Node), Node ⟶ loc(Node);
             ** auxiliary function needed for the specification of the next one
    func find: Node ⟶ loc(Node);
    var ln: loc(Node), n: Node.
    • dom find(n): has(n);
    • local_has(ln, n) ==
             if ¬D(ln!) then false
```

```
            elseif ln! = n then true else local_has(next(ln), n);
    • has(n) == local_has(next(head), n);
    • local_find(ln, n) == if ln! = n then ln else local_find(next(ln), n);
    • find(n) == local_find(next(head), n);
  proc
    initialize; ** construction of the empty list
    insert: loc(Node), Node; ** insertion of a node in the middle of the list
    update: ref Node, Node; ** update of the node data
    local_delete: ref Node, Node;
            ** auxiliary procedure needed for the specification of the next one
    delete: Node;
    var ln: loc(Node), n: Node.
    • pre delete(n): has(n);
    • collect_garbage == forall l: loc(Node). drop l;
    • initialize == seq forall l: loc(Node). set l := undef, next(l) ← undef end,
                       collect_garbage end;
            ** parallel deletion of all elements of the sort loc(Node)
    • insert(ln, n) == import new_loc: loc(Node) in
            set next(ln) ← new_loc, next(n) ← next(ln), update(new_loc, n) end;
    • update(ln, n) == ln := n;
    • local_delete(ln, n) == ** it is assumed that the list contains n
            if next(ln) = n
            then next(ln) ← next(next(ln))
            else local_delete(next(ln), n);
    • delete(n) == local_delete(head, n);
end.
```

11. Conclusion

A model of the state of a dynamic system as a many-sorted algebra with updateable locations as values is given in the paper. The model provides both the mechanism of call-by-value and the mechanism of call-by-reference typical of imperative programming languages. A specification technique for this model is suggested. The use of this technique for program specification permits one to write specifications naturally refined to practical programs. On the other hand, the model can serve as a high-level representation of the semantics of an imperative language.

In the present form, the model still lacks many means needed for a suitable modeling of programming languages, e.g., local objects in procedures and function parameters. All of this is a subject of further research.

References

[1] Astesiano E., Zucca E. D-oids: a model for dynamic data types // Mathematical Structures Comput. Sci. — 1995. — Vol. 5, № 2. — P. 257–282.

[2] Baumeister H. Relations as abstract data types: an institution to specify relations between algebras // Lect. Notes Comput. Sci. — 1995. — Vol. 915. — P. 756–771.

[3] Dauchy P., Gaudel M. C. Algebraic Specifications with Implicit State. — Paris, 1994. — (Tech. rep. / Laboratoire de Recherche en Informatique, Univ. Paris-Sud; № 887).

[4] Grosse-Rhode M. Concurrent state transformation on abstract data types // Lect. Notes Comput. Sci. — 1995. — Vol. 1130. — P. 222–236.

[5] Grosse-Rhode M. Algebra transformation systems and their composition // Lect. Notes Comput. Sci. — 1998. — Vol. 1382. — P. 107–122.

[6] Gurevich Y. Evolving algebras 1993: lipary guide // Specification and Validation Methods. — Oxford University Press, 1995. — P. 9–36.

[7] Jonkers H. B. M. An introduction to COLD-K // Lect. Notes Comput. Sci. — 1989. — Vol. 394. — P. 139–205.

[8] Gaudel M.-C., Khoury C., Zamulin A. Dynamic systems with implicit state // Lect. Notes Comput. Sci. — 1999. — Vol. 1577. — P.114–128.

[9] Lellahi K., Zamulin A. Dynamic Systems Based On Update Sets. — Paris, 1999. — (Tech. rep. / LIPN, Univ. Paris 13; № 99-03)

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

[11] Zamulin A. V. Dynamic system specification by typed Gurevich machines // Proc. Intern. Conf. on Systems Science. — Wroclaw, Poland, September 15–18, 1998.

[12] Zucca E. Fundamental Study. From static to dynamic abstract data-types: an institution transformation // Theor. Comput. Sci. — 1999. — Vol. 216. — P. 109–157.