Modularization of typed Gurevich machines*

A. V. Zamulin

An important problem of the representation of a big dynamic system as a number of interrelating typed Gurevich Machines (Abstract State Machines or just ASMs in the sequel) and the subsequent combination of the specifications of individual ASMs into the specification of the whole system is investigated in the paper. The structure of such a system is formally defined and a notion of external signature of a typed ASM is introduced. Two main operations for combining existing specifications (and their implementing ASMs) are suggested: import of existing typed ASMs by a new one and union of several typed ASMs into a new one. The syntax and semantics of the operations are formally defined.

Keywords: abstract state machines, specification-in-the-large, modular design, dynamic system

1. Introduction

A dynamic system was defined in [1] as a typed ASM (TASM or simply machine in the sequel) consisting of the following components:

- set of states,
- set of dependent (derived) functions for observing the current system’s state,
- set of procedures for updating the state.

Formally, a TASM is represented by the following tuple:

\[
\langle (\Sigma_0, Ax), \Sigma_{\text{din}}, (\Sigma_{\text{dep}}, Ax_{\text{dep}}), (\Sigma_{\text{proc}}, Ax_{\text{proc}}) \rangle
\]

called a TASM specification. Its first component is the specification (the signature and axioms) of a number of data types and related functions, the second component is the signature of a set of dynamic functions, the third component is the signature and axioms of dependent functions, and the fourth one is the signature and axioms of procedures. A TASM state corresponding to such a specification is a \(\Sigma\)-algebra \(A\), where \(\Sigma = \Sigma_0 \cup \Sigma_{\text{din}}\). Such an algebra consists of a static part \(A_0\) which is a \(\Sigma_0\)-algebra and a dynamic part \(\Delta(\Sigma_{\text{din}})\) mapping each name in \(\Sigma_{\text{din}}\) to a corresponding function so that \(A = A_0 \cup \Delta(\Sigma_{\text{din}})\).

The class of all possible states (\(\Sigma\)-algebras) is divided into subclasses, \(\text{state}_A(\Sigma_0, \Sigma_{\text{din}})\), which share the same (static) \(\Sigma_0\)-algebra \(A_0\). If \(\{\Delta(\Sigma_{\text{din}})\}\) is the set of all possible dynamic parts, then \(\text{state}_A(\Sigma_0, \Sigma_{\text{din}}) = \{A_0\} \times \{\Delta(\Sigma_{\text{din}})\}\).

The specification of a TASM can be regarded as a specification in-the-small. Such a TASM is suitable for modeling a small dynamic system whose states are centralized. However, a sufficiently big dynamic system can consist of several TASMs communicating with each other and thus representing the modular structure of the system. The states of such a system are decentralized, and the overall current state of the system is the union of the current states of the component TASMs. The specification of such a system is a kind of specification in-the-large. Special means like those existing in some programming languages (e.g., Oberon [4] and Ada [5]) and specification languages (e.g., Extended ML [6] and CASL [3]) are to be elaborated for creating bigger TASMs from smaller ones.

The following structural mechanisms are investigated in this paper:

1. Use of existing TASMs when constructing a new one. The components of an existing TASM should only be referenced in some way in the specification of the new TASM.
2. Composition of existing TASMs into a bigger one.

---

*This research is supported in part by Russian Foundation for Basic Research under Grant 98-01-00682.
The paper is organized in the following way. The specification technique is briefly described in Section 2. The structure of a complex dynamic system is defined in Section 3. The notion of external signature of a TASM is introduced in Section 4. The facilities for importing existing TASMs by a new one are developed in Section 5. The operation of union of several TASMs into a new one is described in Section 6. Some related work is discussed in Section 7, and some conclusions and directions of further work are given in Section 8.

2. Specification conventions

In this section we consider that a dynamic system is a TASM. The first component of its specification is a classic algebraic specification defining a number of data types and related functions representing the static part of the system and used for the possible state observations/updates, its semantics is given by the specification language used. For simplicity, we consider in the sequel that data type specifications are collected in a special module with the signature $\Sigma_0$. The rules for creating such a module are not discussed here. Possible conflicts between overloaded operation names are resolved according to the underlying specification language (for example, prefixing with the type name if a language like Ruslan [2] is used or qualifying the operation name with its profile if a language like CASL [3] is used). In this case the specification of a dynamic system is a triple $(\Sigma_{din}, (\Sigma_{dep}, Ax_{dep}), (\Sigma_{proc}, Ax_{proc}))$, its signature is $(\Sigma_{din}, \Sigma_{dep} \Sigma_{proc})$, and its state is a $\Sigma$-algebra where $\Sigma = (\Sigma_0, \Sigma_{dep})$. We denote by $DS(A_0)$ a dynamic system with the set of states sharing the same $\Sigma_0$-algebra $A_0$.

In the general case, a function name in $\Sigma_{din}$ is declared as follows:

$$\text{dynamic function function-name: } T_1, \ldots, T_n \rightarrow T;$$

where $T_1, \ldots, T_n$ are the types of the function arguments and $T$ is the type of result. A function without arguments is called a constant and declared as follows:

$$\text{dynamic const constant-name: } T;$$

where $T$ is the type of the constant. The updates of the functions and constants declared in $\Sigma_{din}$ cause the transformation of the state.

The function names in $\Sigma_{dep}$ are declared similarly to the function names in $\Sigma_{din}$ with the use of the keyword depend. However, the declaration of a dependent function is accompanied with one or more data equations, $Ax_{dep}$, thus building a function specification. A data equation is a pair $t_1 = t_2$ where $t_1$ and $t_2$ are two terms of the same type. The terms are composed of universally quantified variables, operation names from $\Sigma_0$, and function names from $\Sigma_{din}$ and $\Sigma_{dep}$. The interpretation of such a term produces an algebra element. A data equation $t_1 = t_2$ is satisfied in a dynamic system iff the interpretation of both $t_1$ and $t_2$ produces the same algebra element in any its state. Since a term containing the name of a dynamic function can evaluate differently in different states, the function whose specification contains such a term generally produces different results in different states (i.e., the result depends on the state).

For this reason, a dependent function name $df : T_1, \ldots, T_n \rightarrow T$ is interpreted in a dynamic system $DS(A_0)$ by a map $df^{DS(A_0)}$ associating a value of type $T$ with each pair $\langle A, \langle \nu_1, \ldots, \nu_n \rangle \rangle$, where $A$ is a state of $DS(A_0)$ and $\nu_i$ is an element of type $T_i$; this map must satisfy the corresponding dynamic equations from $Ax_{dep}$.

A procedure name in $\Sigma_{proc}$ is declared as follows:

$$\text{proc procedure-name: } T_1, \ldots, T_n;$$

where $T_1, \ldots, T_n$ are the types of the procedure arguments ($n$ can be zero, i.e., a procedure can have no arguments). The declaration of a procedure is accompanied with one or more dynamic equations, $Ax_{proc}$, thus building a procedure specification. A dynamic equation is a pair $t_1 = t_2$ where $t_1$ and $t_2$ are two transition terms. There are two kinds of transition term, a procedure call and a transition rule. The interpretation of either of them produces an update set. A procedure call is created by
the application of a procedure name to a list of argument terms. Thus, if \( p : T_1, \ldots, T_n \) is a procedure
declaration and \( t_1, \ldots, t_n \) are terms of types \( T_1, \ldots, T_n \), respectively, then \( p(t_1, \ldots, t_n) \) is a transition term.
In a dynamic equation \( t_1 = t_2 \), the first term is normally a procedure call and the second one is
a transition rule. A dynamic equation is satisfied in a dynamic system iff the interpretation of both
terms in any its state produces the same update set.

Transition rules are conventionally created like in traditional ASMs [7] with an additional possibility of using procedure calls in rule constructors. There is, however, an important difference in the treatment of the assignment of an undefined value to a location. There can be no single \texttt{undef} value for all data types. To simplify the specification of data types, no one of them is equipped with its
own \texttt{undef} value. Partial functions are used instead, and a definedness predicate, \( \texttt{D} \), is introduced.
For each term \( t \), the predication \( \texttt{D}(t) \) holds in a given algebra \( A \) if \( t \) is defined in it and does not hold otherwise. In an update rule

\[
f(t_1, \ldots, t_n) := \texttt{undef}
\]

\( \texttt{undef} \) is just a keyword indicating that \( f(t_1, \ldots, t_n) \) becomes undefined.

A procedure declaration \( p : T_1, \ldots, T_n \) from \( \Sigma_{\text{proc}} \) is interpreted in a dynamic system \( \text{DS}(A_0) \) by a
map \( p^D(A_0) \) associating an update set with each pair \( \langle A, \langle v_1, \ldots, v_n \rangle \rangle \), where \( A \) is a state of \( \text{DS}(A_0) \) and
\( v_i \) is an element of type \( T_i \); this map must satisfy the corresponding dynamic equations from \( A \Sigma_{\text{proc}} \).

Both a dependent function and a procedure can be partial. In this case, the function (procedure)
specification is augmented with a special clause \texttt{dom} defining the domain of the function or procedure:
for a domain definition \( \texttt{dom} \ t : b \), the predication \( \texttt{D}(t) \) must hold in a given state \( A \) iff \( b \) evaluates to
\texttt{true} in this state.

A dynamic system is a model of a specification \( \text{DSS} = \langle \Sigma_{\text{din}}, (\Sigma_{\text{dep}}, \Sigma_{\text{dep}}, \Sigma_{\text{proc}}), (\Sigma_{\text{proc}}, \Sigma_{\text{proc}}, \Sigma_{\text{proc}}) \rangle \) iff all
equations and domain definitions of \( \text{DSS} \) are satisfied in it. A specification is consistent if there is at
least one its model. A counterexample: consider a specification with the procedure declaration \texttt{proc}
\( R : \text{Integer} \) and the dynamic equation \( R(x \times x) = f := x \) where \( f : \text{Integer} \) is a dynamic constant.
It is clear that no model can satisfy the equation under both evaluations, \( x = 2 \) and \( x = -2 \). Only
consistent specifications are considered in the sequel.

The set of models of \( \text{DSS} \) is denoted by \( \text{Mod}(\text{DSS}) \). Any model contains the state where all
dynamic functions/constants are undefined. This state is called the undefined state of the system. A
procedure can be used for setting an appropriate initial state of the system.

3. Structure of the state of a complex dynamic system

In a dynamic system consisting of several communicating TASM\( s, a \) procedure in one TASM can
generally update the state of some other TASM\( s. \) Therefore, one cannot use only the state of a
particular TASM as an argument and/or result of a procedure of this TASM. For this reason, we
define the system state on the base of individual TASM states. Let a dynamic system \( \text{DS}(A_0) \) consist of
\( n \) TASM\( s.

\textbf{Definition 1.} If \( \Sigma_0 \) is the signature of data types and \( \langle \Sigma_{\text{din}}, \Sigma_{\text{dep}}, \Sigma_{\text{proc}} \rangle \) is the signature of \( \text{TASM}_i \),
then \( \langle \Sigma_0, \Sigma_D \rangle \), where

\[
\Sigma_D = \bigcup_{i=1}^{n}(\Sigma_{\text{din}}, \Sigma_{\text{dep}}, \Sigma_{\text{proc}}),
\]

is the signature of \( \text{DS}(A_0) \). That is, the signature of a dynamic system is the discriminated union of
the signatures of the component TASM\( s \) extending the signature of the data type module.

\textbf{Fact 1.} If \( \text{state}_{A_0}(\Sigma_0, \Sigma_{\text{din}}) \) is the set of possible states of \( \text{TASM}_i \), then \( \text{state}_{A_0}(\Sigma_0, \Sigma_{\text{din}}) \),
where \( \Sigma_{\text{DIN}} = \bigcup_{i=1}^{n}(\Sigma_{\text{din}}) \), is the set of states of \( \text{DS}(A_0) \).

Thus, the state of a dynamic system is the concatenation of the states of the component machines.
The change of the state of one of them causes the change of the state of the dynamic system.
4. External signature of a TASM

A TASM specification with import and export clauses introduced below is called a module. \((\Sigma_{\text{din}}, \Sigma_{\text{dep}}, \Sigma_{\text{proc}})\) and \((\Sigma_{\text{dim}}, (\Sigma_{\text{dep}}, A_{\text{dep}}), (\Sigma_{\text{proc}}, A_{\text{proc}}))\) of a module are called, respectively, the own signature and own specification of the module. A particular model of a module is called a TASM or, simply, machine. The class of all models of a module \(M\) is denoted by \(\text{Mod}(M)\).

In the specification of a big dynamic system, some names declared in the own signature of a module can be seen from the outside, some others can be seen only inside the module. We consider that no name from \(\Sigma_{\text{din}}\) is seen from the outside. Special specification language conventions (export clause listing exported symbols, keyword \text{export} in front of a symbol, etc.) can be used for indicating which names from \(\Sigma_{\text{dep}}\) and \(\Sigma_{\text{proc}}\) are seen from the outside. These names define the external signature of the module.

**Definition 2.** Let a module \(M\) have the signature \((\Sigma_{\text{din}}, \Sigma_{\text{dep}}, \Sigma_{\text{proc}})\) and the set of exported names \(E_{M}\). Then the external signature of \(M\) is \((\Sigma_{\text{dep}}^M, \Sigma_{\text{proc}}^M)\) composed in the following way:

- if \((f : T_1, \ldots, T_n \rightarrow T) \in \Sigma_{\text{dep}}\) and \(f \in E_{M}\), then \((M.f : T_1, \ldots, T_n \rightarrow T) \in \Sigma_{\text{dep}}^M\);
- if \((p : T_1, \ldots, T_n) \in \Sigma_{\text{proc}}\) and \(p \in E_{M}\), then \((M.p : T_1, \ldots, T_n) \in \Sigma_{\text{proc}}^M\).

**Example 1.** Specification of a stack machine (the types \text{Boolean}, \text{Nat} and \text{Oper} used in the specification are defined in the data type module).

```plaintext
tasm StackOfOper = spec
  export initialize, push, pop, top, is_empty;
  dynamic function cont: Nat → Oper; — content of the stack
  dynamic const size: Nat; — size of the stack
  proc initialize; — construction of an empty stack
    initialize == set size := 0, forall x: Nat. cont(x) := undef end;
  push: Oper; — pushing a stack with an element
    push(o) == set size := size + 1, cont(size+1) := o end;
  pop — deleting the top element of a stack
    dom pop: size > 0;
    pop == set size := size - 1, cont(size) := undef end;
  depend function top: Oper; — fetching the top element of a stack
    top == cont(size);
  depend function is_empty: Boolean; — checking whether a stack is empty
    is_empty == size = 0;
end;
```

In the above module all operations except dynamic functions are exported. Therefore, the external signature of the TASM is the following:

\[(\text{StackOfOper}.top : \text{Oper};\]
\[\text{StackOfOper}.is_empty : \text{Boolean},\]
\[\text{StackOfOper}.initialize;\]
\[\text{StackOfOper}.push : \text{Oper};\]
\[\text{StackOfOper}.pop) > .\]

**Example 2.** Specification of a block-structured identifier table (the types \text{Boolean}, \text{Nat}, \text{Name} and \text{Defdata} used in the specification are defined in the data type module).

```plaintext
tasm IdTable = spec
  export initialize, insert_entry, new_level, delete_level,
```
defined current, is defined, find;
dynamic function id table: Name, Nat \rightarrow Defdata;
dynamic const cur level: Nat; — the current level of block nesting
proc initialize; — construct an empty identifier table
   initialize \[\] \rightarrow cur level := 1,
   \forall id: Name, \times: Nat. id table(id, \times) := undefined end;
proc insert entry: Name, Defdata; — insert an entry in the current block
   insert entry(id, d) \rightarrow id table(id, cur level) := d;
proc new level; — create a new level
   new level \rightarrow cur level := cur level + 1;
proc delete level; — delete the innermost level
   delete level \rightarrow cur level := cur level - 1,
   \forall id: Name. id table(id, cur level) := undefined end;
depend function defined current: Name \rightarrow Boolean;
   — is the name defined in the current block?
   defined current(id) \rightarrow D(id table(id, cur level));
depend function local defined: Name, Nat \rightarrow Boolean;
   — local function used for the specification of the next one
   local defined(id, 0) \rightarrow false;
   local defined(id, k) \rightarrow D(id table(id, k)) \land local defined(id, k-1);
depend function is defined: Name \rightarrow Boolean;
   — is the name defined in some block?
   is defined(id) \rightarrow local defined(id, cur level);
depend function local find: Name, Nat \rightarrow Defdata;
   — local function used for the specification of the next one
   dom local find(id, k): k > 0;
   local find(id, k) \rightarrow if D(id table(id, k)) \then id table(id, k)
   \else local find(id, k-1)};
depend function find: Name \rightarrow Defdata; — find an entry in the table
   dom find(id) \rightarrow is defined(id);
   find(id) \rightarrow local find(id, cur level);
end

In the above specification the functions local defined and local find are auxiliary local functions which are not exported. Therefore, the external signature of the module is the following:

\plaintext{(IdTable.defined current : Name \rightarrow Boolean;
   IdTable.is defined : Name \rightarrow Boolean;
   IdTable.find : Name \rightarrow Defdata),
   (IdTable.initialize;
   IdTable.insert entry : Name, Defdata;
   IdTable.new level;
   IdTable.delete level)).

5. Import of TASMs

The first operation of the in-the-large level is the use of existing modules in a new one. This means that, when constructing a module, one can use exported names from other modules, they constitute the import of the module. Respectively, a model (TASM) of a given module is extended by the components of the models (TASMs) of the imported modules.

An imported name can be referenced in one of the following ways:
1) directly with a possible qualification in case the name is overloaded;
2) prefixed with the name of the module where it is originally defined as it is done for modules in Oberon [4] or some other programming languages.

The direct use of an imported name is not possible, however, when it is declared with the same profile in two or more modules. The name initialize in the modules StackOfOper and IdTable can serve as an example. Therefore, we prefer the second way, i.e., prefixing an imported name with a module name. A special clause import listing the names of imported modules is included in the TASM specification in this case. A prefixed name can be used in the specification for creating terms according to the following rule.

**Definition 3.** Let a module $M$ have the import $M_1, \ldots, M_k$ where $M_i$ is the name of an imported module. If $(M_i, f : T_1, \ldots, T_n \rightarrow T) \in \Sigma_{dep}^{M_i}$ and $t_1, \ldots, t_n$ are terms of types $T_1, \ldots, T_n$, respectively, then $M_i.f(t_1, \ldots, t_n)$ is a term of type $T$ in $M$. If $(M_i, f : T_1, \ldots, T_n) \in \Sigma_{proc}^{M_i}$ and $t_1, \ldots, t_n$ are terms of types $T_1, \ldots, T_n$, respectively, then $M_i.f(t_1, \ldots, t_n)$ is a transition term in $M$. We write $M_i.f$ if the profile of $f$ has no argument types.

**Example 3**

```
tasm BiggerTasm = spec
import StackOfOper, IdTable;
export initialize ...; list of exported names
  . . .
proc initialize;
  initialize == set ... StackOfOper.initialize, IdTable.initialize, ... end;
end
```

In the above example `StackOfOper.initialize` and `IdTable.initialize` are transition terms created with the use of imported names.

**Fact 2.** The state of a TASM which is a model of a module importing other modules is defined by its own signature and the signature of the module of data types. Indeed, since dynamic functions are not exported, $\Sigma_{din}$ of the importing module is not influenced by the import.

**Fact 3.** If $M_1, \ldots, M_n$ are the names of modules imported by the module $M$, then the own specification of $M$ generally defines a function $F : Mod(M_1), \ldots, Mod(M_n) \rightarrow Mod(M)$. This means that supplying different models of imported modules, we get different models of the importing module. In other words, a TASM which is a model of a given module depends on the imported TASM. If $Md_1, \ldots, Md_n$ are models of the modules $M_1, \ldots, M_n$, respectively, then $F(Md_1, \ldots, Md_n)$ is a model of $M$.

Consider the specification of the procedure `initialize` in the above example. Each time when the specification of one of the procedures, `IdTable.initialize` or `StackOfOper.initialize`, is changed so that the corresponding procedure must yield a different update set, the procedure `initialize` in the model of the module "BiggerTasm" must also produce a different update set. This does not mean, of course, that an importing TASM must be actually reconstructed each time one of the imported machines is reconstructed (the conventional mechanism of procedure calls helps to avoid it), but conceptually the TASM is changed.

Let now a module $M$ import modules $M_1, \ldots, M_n$, $Md_1, \ldots, Md_n$ be models of the modules $M_1, \ldots, M_n$, respectively, and $Md = F(Md_1, \ldots, Md_n)$ be a model of the module $M$. We denote by $[t]^{Md,A}$ the interpretation of a term $t$ in the model $Md$ at the state $A$. If $t$ is a dependent function name or a procedure name, we simply write $[t]^{Md}$ since the interpretation of this name does not depend on the state. In this case, the interpretation of a term $M_i.f(t_1, \ldots, t_n)$ in the model $Md$ at the state $A$ is defined in the
following way:

$$[[M, f(t_1, \ldots, t_n)]^{Md.A} = f^{Md}([[t_1]^{Md.A}, \ldots, [t_n]^{Md.A})].$$

6. Union of TASMs

The second operation of the *in-the-large* level is the *union* of several existing machines into a new one. This corresponds to the modular decomposition of a big dynamic system into several smaller dynamic systems developed independently. Two options of the specification of a big system are possible:

1. The specifications of the component machines are developed independently and then they are united into a single piece of specification. A machine corresponding to the resulting specification is just a model of this specification.

2. The specifications of the component machines are developed independently and each of them is provided with its own model. A machine corresponding to the resulting specification is the union of the component machines.

The first case corresponds to linking several pieces of the source text of a program and then compiling them into a single unit. The second case corresponds to the independent compilation of programs with the subsequent linkage of the object codes. Since the first case does not impose any structure on the set of models, its task can successfully be solved by modern text editing facilities. Therefore, we will discuss possible solutions of the second task which provide better modularization facilities. Recall that we have modules at the specification level and TASMs (machines) at the model level.

Thus, we assume that the module union operation, *union*, is supported at the model level by a TASM union operation, $union_{Md}$. It must actually use the component machines without their reconstruction. Therefore, the union operation is required to be *persistent*, i.e., the reduct of the result of $union_{Md}$ to the name of a component machine must yield exactly the component machine. Respectively, the reduct of the resulting module to the name of a component module must yield the component module.

The second requirement for the union operation is that it must be *constructive* in the following sense. Let $Md_1, \ldots, Md_n$ be models of modules $M_1, \ldots, M_n$, respectively. Then $union_{Md}(Md_1, \ldots, Md_n)$ must be a model of $union(M_1, \ldots, M_n)$. This might not be the case if the united machines have operations with the same names and profiles.

There are also problems with the creation of the external signature of the new module. The first of them concerns prefixing. Assume that we wish to unite the modules $StackOfOper$ and IdTable specified above to create the module *JointMachine*. In all likelihood, a user will not be happy if he now has to use operations with double prefixing, e.g., $JointMachine.StackOfOper.push(x)$, $JointMachine.IdTable.initialize$, etc. A better solution is to prefix operations only with the name of the new machine, for example: $JointMachine.push(x)$, $JointMachine.initialize$. Unfortunately, we have a problem with overloaded names in this case. For example, the operations $StackOfOper.initialize$ and $IdTable.initialize$ will become unrecognizable if their prefixes are replaced with $JointMachine$. Therefore, a mechanism of renaming the exported operations is needed.

Next problem concerns the volume of export of the new module. It might happen that the list of exported operations of the new module is shorter than the union of the lists of exported operations of the united modules. For example, if the modules $StackOfOper$, $IdTable$ and $BiggerTasm$ are united, there is no need to export the operations $StackOfOper.initialize$ and $IdTable.initialize$. In this case, a mechanism of defining a new export is needed.

Taking into account the above considerations, the following syntax of the union operation can be proposed:
union-operation ::= union list-of-modules [, export] end
list-of-modules ::= component-module {, component-module}
component-module ::= module-name [export-renaming]
export-renaming ::= (pair-of-names {, pair-of-names})
pair-of-names ::= new-name = old-name
new-name ::= name
old-name ::= name
export ::= export qualified-name {, qualified-name}
qualified-name ::= module-name-name

If there is no export clause, then all exported names of the component modules are exported. The absence of "export-renaming" for a particular component module means that no exported name of this module is renamed.

Example 4.

```
tasm Union2 = union StackOfOper (empty = initialize), IdTable end
```

```
tasm Union3 = union StackOfOper, IdTable, BiggerTasm
  export StackOfOper.push, StackOfOper.pop, StackOfOper.top,
    StackOfOper.is-empty, IdTable.insert-entry, IdTable.new-level,
    IdTable.delete-level, IdTable.defined-current,
    IdTable.is-defined, IdTable.find, BiggerTasm.initialize, ...
end
```

To define the requirements for the well-formedness of the union operation, we introduce several auxiliary notions. Let component modules $M_1, \ldots, M_n$ have the sets of exported names $E_{M_1}, \ldots, E_{M_n}$, respectively, and let $EL_M$ be the set of qualified names in the export clause of the union operation (the set is empty if there is no export clause). The set $EL_M$ is well-formed if, for any $M_i$, $\text{exported-name} \in EL_M$, $M_i$ is the name of a component module and $\text{exported-name}$ belongs either to the list of exported names of $M_i$ (if it is not renamed) or to the list of new names in the export renaming for $M_i$. That is, the new name must be used in the export clause if the corresponding exported name is renamed and the old one in the opposite case.

If $EL_M$ is well-formed, we construct, for each component module $M_i$ with the export set $E_{M_i}$, the renaming set, $ER_i$, as the set of pairs $\langle \text{new-name}, \text{old-name} \rangle$ in the following way:

1. if there is no export-renaming for $M_i$, then
   - if $EL_M$ is empty, then $ER_i$ is the set of all pairs $\langle \text{old-name}, \text{old-name} \rangle$, where $\text{old-name} \in E_{M_i}$;
   - if $EL_M$ is not empty, then $\langle \text{old-name}, \text{old-name} \rangle \in ER_i$ iff $M_i$ old-name $\in EL_M$;
2. if there is an export-renaming for $M_i$, then
   - if $EL_M$ is empty, then $\langle \text{new-name}, \text{old-name} \rangle \in ER_i$ if the pair $\langle \text{new-name} = \text{old-name} \rangle$ is part of the export-renaming, and $\langle \text{old-name1}, \text{old-name1} \rangle \in ER_i$ if $\text{old-name1} \in E_{M}$ and there is no $\text{new-name1}$ such that $\langle \text{new-name1} = \text{old-name1} \rangle$ is part of the export-renaming;
   - if $EL_M$ is not empty, then $\langle \text{new-name}, \text{old-name} \rangle \in ER_i$ if the pair $\langle \text{new-name} = \text{old-name} \rangle$ is part of the export-renaming and $M_i$ new-name $\in EL_M$, and $\langle \text{old-name1}, \text{old-name1} \rangle \in ER_i$ if $\text{old-name1} \in E_{M_i}$, there is no $\text{new-name1}$ such that $\langle \text{new-name1} = \text{old-name1} \rangle$ is part of the export-renaming and $M_i$ old-name1 $\in EL_M$;
Thus, the renaming sets for the module \textit{Union2} are the following:\n\textit{StackOfOper} : \{\langle empty, initialize \rangle, \langle push, push \rangle, \langle pop, pop \rangle, \langle top, top \rangle, \\
\langle is\_empty, is\_empty \rangle\},
\textit{IdTable} : \{\langle initialize, initialize \rangle, \langle insert\_entry, insert\_entry \rangle, \\
\langle new\_level, new\_level \rangle, \langle delete\_level, delete\_level \rangle, \\
\langle defined\_current, defined\_current \rangle, \langle is\_defined, is\_defined \rangle, \\
\langle find, find \rangle\} 
and the renaming sets for the module \textit{Union3} are the following:\n\textit{StackOfOper} : \{\langle push, push \rangle, \langle pop, pop \rangle, \langle top, top \rangle, \langle is\_empty, is\_empty \rangle\},
\textit{IdTable} : \{\langle initialize, initialize \rangle, \langle insert\_entry, insert\_entry \rangle, \\
\langle new\_level, new\_level \rangle, \langle delete\_level, delete\_level \rangle, \\
\langle is\_defined, is\_defined \rangle, \langle find, find \rangle\},
\textit{BiggerTasm} : \{\langle initialize, initialize, initialize \rangle, \ldots \}.

The set \( ER_i \) is consistent iff for any pair \( \langle new\_name, old\_name \rangle \in ER_i \), \( old\_name \in E_M \) and there is no \( old\_name 1 \) such that \( \langle new\_name, old\_name 1 \rangle \in ER_i \). This means that new names must be unique within the module. According to this, all renaming sets above are well-formed.

The set of renaming sets, \( ER_1, \ldots, ER_n \), is consistent if any \( ER_i \) is consistent and for any pair \( \langle new\_name, old\_name \rangle \in ER_i \), there is no \( old\_name 1 \) such that \( \langle new\_name, old\_name 1 \rangle \in ER_j \), \( j = 1, \ldots, n \) and \( j \neq i \). This means that new names must be unique in the family of renaming sets. For example, the set of renaming sets for the module \textit{Union3} would be inconsistent if there were no export clause (there would be pairs \( \langle initialize, initialize \rangle \) in three renaming sets).

Now we can construct the set of exported names, \( E_M \), of the resulting module: if a pair \( \langle new\_name, old\_name \rangle \in ER_i \), then \( new\_name \in E_M \).

Thus, the set of exported names of the module \textit{Union2} is \{\textit{push, pop, empty, top, is\_empty, initialize, insert\_entry, new\_level, delete\_level, defined\_current, is\_defined, find, initialize}\} and the set of exported names of the module \textit{Union3} is \{\textit{push, pop, top, is\_empty, insert\_entry, new\_level, delete\_level, defined\_current, is\_defined, find, initialize ... \}.

The external signature of the resulting module is constructed as follows:

- if \( f \in E_M \), \( f_i \) is a name such that the pair \( \langle f, f_i \rangle \in ER_i \) and \( (M_i, f_i : T_1, \ldots, T_n \rightarrow T) \in \Sigma_{dep}^{M_i} \) then \( (M, f : T_1, \ldots, T_n \rightarrow T) \in \Sigma_{dep}^M \);
- if \( p \in E_M \), \( p_i \) is a name such that the pair \( \langle p, p_i \rangle \in ER_i \) and \( (p_i : T_1, \ldots, T_n) \in \Sigma_{proc}^{M_i} \) then \( (M, p : T_1, \ldots, T_n) \in \Sigma_{proc}^M \).

Thus, the external signature of the module \textit{Union2} is the following:

\{\langle Union2, top, Oper \rangle; \\
\langle Union2, is\_empty, Boolean \rangle; \\
\langle Union2, is\_defined, Name \rightarrow Boolean \rangle; \\
\langle Union2, defined\_current, Name \rightarrow Boolean \rangle; \\
\langle Union2, find, Name \rightarrow Defdata \rangle; \\
\langle Union2, empty \rangle; \\
\langle Union2, push, Oper \rangle; \\
\langle Union2, pop \rangle; \\
\langle Union2, initialize \rangle; \\
\langle Union2, insert\_entry, Name, Defdata \rangle; \\
\langle Union2, new\_level \rangle; \\
\langle Union2, delete\_level \rangle\}.

and the external signature of the module \textit{Union3} differs from the previous one only by the absence of the procedure name \textit{empty}. 
The resulting module consists of the set of the names of the component modules with their renaming sets, the set of the names of the imported modules, and the set of the exported names. The own signature and own specification of the resulting module are empty. The module is consistent if the set of the renaming sets is consistent.

If the resulting module is consistent, then the components of the resulting machine itself are defined as follows:

- the set of states is the Cartesian product of the sets of the states of the component machines as stated by Fact 1;
- the set of dependent functions is the discriminated union of the sets of dependent functions of the component machines;
- the set of procedures is the discriminated union of the sets of procedures of the component machines;

Thus, the state of the resulting machine is a tuple \((A_0, A_1, \ldots, A_n)\) where \(A_0\) is the algebra of data types and \(A_1, \ldots, A_n\) are the states of the component machines.

**Fact 4.** The union operation as defined above is persistent. Indeed, the reduct of the resulting machine to the name of a component machine produces exactly that machine. This fact permits us to reconstruct a component machine if needed, without reconstructing the other component machines.

**Fact 5.** The union operation as defined above is constructive. Indeed, the resulting machine is a model of the resulting module since it provides a unique function for any function/procedure name defined in the component modules.

If a module \(M\) produced by the union of modules \(M_1, \ldots, M_n\) is imported in a module \(M'\), then the terms prefixed with \(M\) in \(M'\) are interpreted differently then the terms prefixed with ordinary modules as described in Section 5. Let \(f\) be a name from the set of exported names of \(M\), \(M.f(t_1, \ldots, t_n)\) a term constructed according to Definition 3, \(M_d'\) a model of \(M'\) and \(ER_i\) the renaming set for \(M_i\) in \(M\), then

\[
[M.f(t_1, \ldots, t_n)]^{M_d'.A} = [M_i.f1(t_1, \ldots, t_n)]^{M_d'.A}
\]

if the pair \(\langle f, f1 \rangle \in ER_i\). That is, such a term is interpreted as if \(f\) is imported directly from a component module.

7. Related work

Specification in-the-large is one of the main concerns of the traditional algebraic specification languages. The aim is the splitting of the specification and design of a single task into a number of well-defined modules so that each of them could be independently implemented. For example, a specification framework which allows the independent construction and implementation of specification modules and incorporates the separation of an implementation task into smaller units with the subsequent stepwise development of single "implementation pieces" is described in [8]. Two basic specification units are introduced: specification modules (classes of algebra-valued functors) and system specifications (classes of algebras constructed by successive functor applications according to the modular structure of the system specification). However, no language constructs implementing these theoretical notions are suggested.

The specification in-the-large in [9] deals with specification modules and their interconnections. A specification module consists of three parts: an export interface, an import interface and a body. Three operations for module interconnections are proposed: composition, union, and extension. The composition of two modules \(M1\) and \(M2\) connects the import of \(M2\) with the export interface of \(M1\). The operation roughly corresponds to the use of existing machines in the new one as described in Section 5. The union of two modules \(M1\) and \(M2\) is the disjoint union of \(M1\) and \(M2\). The constituent
parts of the resulting module are the union of the corresponding parts of the original modules. This operation corresponds to our union operations. Unfortunately, no formal semantics of the operation is given in [9]. The extension of a module $M$ is the result of extending some or all constituent parts of the module by additional items. We believe that this operation can be easily realized by modern text editing facilities and, therefore, do not include it in the list of specification-in-the-large operations.

In the most developed way the specification in-the-large is incorporated in the specification language CASL [3]. It is represented in the language in the form of so-called structural specifications and architectural specifications [10]. The first ones provide means for composing larger specifications from smaller ones (composing the source text of a program) while the second ones provide means for creating larger modules ("units" in the language) from smaller ones implementing the corresponding specifications (linkage of object codes). At both levels the means for specification reduction, translation, union and instantiation are provided. We have concentrated in this paper on some problems related to architectural specifications, paying the major attention to the union operation as most important in the modular design and specification of a big dynamic system. Our definition of the operation as discriminated union of the component specifications has allowed us to avoid many name sharing problems specific for the CASL architectural specifications.

Concrete constructs supporting the specification in-the-large in some earlier algebraic specification languages can be found in [11, 12, 13].

In traditional ASMs, a high-level concept of modularity is realized, according to [14], by function classification. This means that they distinguish between basic functions and derived ("dynamic" in this paper) functions. Within derived or basic functions they distinguish between static functions and dynamic functions; among dynamic functions they distinguish between controlled ones and monitored ones. As stated in [14], "Distinguishing between basic and derived, static and dynamic or controlled and monitored functions constitutes a rigorous high-level realization of Parnas' information hiding principle". Fully supporting this classification of functions and the facilities for information hiding provided by this variety of sorts of functions, we still believe that more powerful modularization facilities and means of their interconnection are needed.

8. Conclusion

The main contribution of this work is the elaboration of some formal mechanisms for combining individual TASM specifications in a big dynamic system. For this purpose, each TASM specification is provided with an export interface permitting us to hide TASM's features used exclusively for its internal needs. A TASM specification using exported operations of some other TASM is also provided with an import interface. The most important operation for combining existing TASM specifications is the union of several TASM specifications into a new one. The requirements for such an operation are stated in the paper, and syntax and semantics of the operation are formally defined.

The presented facilities still do not provide a possibility to define a generic TASM and instantiate it later for producing a number of "sibling" TASM specifications. This remains a subject of further research. A need for it is also indicated in [14].

References


