MAX-PLANCK-INSTITUT FÜR INFORMATIK

Using Algebraic Specification Languages for Model-Oriented Specifications

Hubert Baumeister

MPI–I–96–2-003 February 1996

INFORMATIK

Im Stadtwald
D 66123 Saarbrücken
Germany

Author's Address

Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken, Germany
hubert@mpi-sb.mpg.de

Acknowledgements

The research described in this paper was supported in part by the ESPRIT Basic Research Working Group 6112 COMPASS (A Comprehensive Algebraic Approach to System Specification and Development).

Abstract

It is common belief that there is a substantial difference between model-oriented (eg. Z and VDM) and algebraic specification languages (eg. LSL and ACT-ONE) wrt. their applicability to the specification of software systems. While model-oriented specification languages are assumed to be suited better for the description of state based systems (abstract machines), algebraic specification languages are assumed to be better for abstract datatype specifications. In this paper we shall demonstrate how an algebraic specification language (the Larch Shared Language) can be used to write specifications of abstract machines in the style of Z and how support tools for algebraic specification languages, eg. type checker and theorem provers, can be used to reason about abstract machines.

Keywords

abstract data type, algebraic specification, model-oriented specification, Z, Larch Shared Language, abstract machine, institution

1. Introduction

In the literature (eg. [16, 15]), there is a perceived difference between model-oriented and algebraic specification languages wrt. their applicability to the specification of software systems. Model-oriented specification languages are assumed to be better for the specification of state-based software systems (abstract machines) because of their implicit handling of the state, which may be called *implicit state* approach and is closely related to how imperative programming languages deal with state.

In contrast, algebraic specification languages are assumed to be better for the specification of (functional) abstract datatypes. When algebraic specification languages deal with abstract machines, they do so in a functional way by adding the state of the machine as an additional parameter to the operations of the machine (cf. [9]). This may be called *explicit state* approach.

In this paper we shall give a model for abstract machines parameterized by a model for the state. The state as algebra approach [8, 12, 7, 18, 4, 11, 1, 6] is used to model the state of an abstract machine as a Σ-algebra or, more general, as a Σ-structure in an arbitrary institution (cf. [10]). For the operations of the abstract machine we use the relations as abstract datatype approach [3] to model the operations as sets of Σ-algebras (-structures). This model of an abstract machine will allow us to specify abstract machines in the Z style but using the Larch Shared Language (LSL) [13] instead of Z. We shall also show how the support tools for LSL (the LSL type checker and the Larch Prover) can be used to reason about abstract machines.

Preliminaries. The Larch Shared Language is based on many sorted, first-order logic with equations. A many sorted *signature* Σ consists of a set of sorts S and a set O of sorted function symbols $f\colon w \to s$, where $w \in S^*$ and $s \in S$. A signature morphism $\sigma\colon \Sigma \to \Sigma'$ is a mapping from the sorts of Σ to the sorts of Σ' together with a mapping from the function symbols of Σ to the function symbols of Σ' respecting the sort mapping.

A *Σ-algebra* A consists of a family of sets s_A for each sort s in S and a total, deterministic function $f_A\colon w_A \to s_A$ for each function symbol $f\colon w \to s$ in O. For each signature morphism $\sigma\colon \Sigma \to \Sigma'$ there exists a function $\_|_\sigma$ mapping a Σ'-algebra A to a Σ-algebra $A|_\sigma$ by $s_{A|_\sigma} = \sigma(s)_A$ for each sort and function symbol s from Σ'.

Σ-formulas in LSL are universally quantified equations build from terms over Σ. Predicates are modeled by boolean functions. The

satisfaction relation $\models$ between Σ-algebras and Σ-formulas over the same signature is defined as usual.

An *abstract datatype* of signature Σ is a set of Σ-algebras M. Using loose semantics, a set of Σ-formulas F defines an abstract datatype by

$$M = \{ A \mid A \models F \}$$

A *Larch trait* is a signature together with a set of formulas.

The approach presented in this paper can be generalized from many sorted first-order logic with equations to an arbitrary logic formalizable as an institution. Here we will only recall the definition of institutions, for more details see the paper by Goguen and Burstall [10]. An institution $I = \langle SIGN, Str, Sen, \models \rangle$ consists of

- a category of *signatures* $SIGN$,
- a functor $Str\colon SIGN \to CAT^{op}$ assigning to each signature Σ the category of Σ-structures and to each signature morphism $\sigma\colon \Sigma \to \Sigma'$ a forgetful functor $\_|_\sigma\colon Str(\Sigma') \to Str(\Sigma)$,
- a functor $Sen\colon SIGN \to SET$, assigning to each signature Σ the set of *Σ-formulas* and to each signature morphism $\sigma\colon \Sigma \to \Sigma'$ a translation $\bar{\sigma}$ of Σ-formulas to Σ'-formulas, and
- a family of *satisfaction relations* $(\models_\Sigma \subseteq Str(\Sigma) \times Sen(\Sigma))_{\Sigma \in SIGN}$ indicating whether a Σ-formula f is valid in a Σ-structure m ($m \models_\Sigma f$ or for short $m \models f$)

where the *satisfaction condition* holds: for all signature morphisms $\sigma\colon \Sigma \to \Sigma'$, formulas $f \in Sen(\Sigma)$ and structures $m' \in Str(\Sigma')$

$$m'|_\sigma \models f \quad \text{iff} \quad m' \models \bar{\sigma}(f).$$

Implicit State Approach. An abstract machine consists of a set of states St, a set of initial states $I \subseteq St$ and a family of operations Op. Operations op from Op are relations between possible input values of the operation and the state before (pre-state) and the state after the operation (post-state) and possible output values. The view of operations as relations allows the modelling of non-deterministic state changes and partiality. Other names for abstract machines are abstract datatypes (with state) and dynamic abstract datatypes.

As an example of abstract machines and the way they are specified using a model-oriented language, we give part of the specification of an employment agency, from Middelburg [16], using Z. An employment agency keeps a list of associations of people who are looking for a job to their skills and a list of vacancies. A vacancy is modeled by a pair of the company looking for an employee and the required skills for the jobs. Only one person is allowed to apply for a job at any time whereas a company may offer several jobs simultaneously. Here we model only two operations, Apply and Subscribe. The Apply operation adds a person and his/her skills and the Subscribe operation adds vacancies to the agency database. The Subscribe operation returns a vacancy number to identify the job offer for later retrieval. The specification is generic with respect to the sets *Person*, *Skill* and *Company*.

$[Person, Skill, Company]$

The set *Skills* is the set of finite subsets of *Skill*, denoted by $\mathbb{F}\,Skill$, and *Vacno* is just another name for the set of natural numbers.

$$Skills == \mathbb{F}\,Skill$$
$$Vacno == \mathbb{N}$$

The state space of the agency is given by the schema *Agency*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Agency}} \\ \hline
cands : Person \pfun Skills \\
vacs : Vacno \pfun Vacdata \\ \hline
\end{array}$$

The components of the schema are two partial functions *cands* and *vacs*. The function *cands* maps a person looking for a job to the his/her skills and the function *vacs* maps a vacancy number to a job offer.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Vacdata}} \\ \hline
comp : Company \\
skills : Skills \\ \hline
\end{array}$$

The initial state of the agency is described by the schema *Empty*, stating that *cands* and *vacs*, viewed as sets of tuples, are the empty set.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Empty}} \\ \hline
Agency \\ \hline
cands = \emptyset \\
vacs = \emptyset \\ \hline
\end{array}$$

$\Delta Agency$ introduces two copies of the state of the agency, an unprimed one for the pre-state and a primed one for the post-state.

$$\Delta Agency == Agency \wedge Agency'$$

By convention, variable names decorated with a question mark are used as input parameters and variables decorated with an exclamation mark as output parameters of an operation. The operation Apply is applicable only if the person $p?$ is not already applying for a job. The function *vacs* does not change and *cands* is updated at $p?$ by $s?$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Apply \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta Agency \\
p? : Person \\
s? : Skills \\
\hline
p? \notin \mathrm{dom}\, cands \\
vacs' = vacs \\
cands' = cands \cup \{ p? \mapsto s? \} \\
\hline
\end{array}
$$

A job offer is introduced by using the *Subscr* operation. The parameters are the company who offers the job and the set of skills the job requires. It returns the vacancy number for this job offer. The vacancy number is chosen non-deterministically but is required to be unique.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Subscr \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta Agency \\
c? : Company \\
s? : Skills \\
n! : Vacno \\
\hline
cands' = cands \\
n! \notin \mathrm{dom}\, vacs \\
vacs' = vacs \cup \{ n! \mapsto \mu\ Vacdata \mid comp = c? \land skills = s? \} \\
\hline
\end{array}
$$

The function *cands* does not change and *vacs* is updated at $n!$ by an element of schema type *Vacdata* that has $c?$ as its *comp* component and $s?$ as its *skills* component. The μ-expression allows to refer to the components *comp* and *skills* of *Vacdata* within the predicate of the set comprehension.

Explicit State Approach. The usual way of modeling abstract machines using algebraic specifications is by providing a sort *St* for the state space and adding an additional argument of sort *St* to the functions representing the operations of the abstract machine. This approach is called explicit state approach because of the explicit representation of the state as a formal parameter to the operations. In contrast, the model-oriented approach is called implicit state approach because the state is not given as an argument to the operation but implicitly. As an example of the explicit state approach, we give the specification of the employment agency using the Larch Shared Language.

Again `Person`, `Skill` and `Company` are generic sorts. The trait `Set` for finite sets is included from the LSL library of traits to provide the sort `Skills` of finite sets over `Skill`. The sort `Vacdata` is defined as the sort of records with components `comp` and `skills`.

```
Agency (Person, Skill, Company) : trait
   includes Set(Skill, Skills), Natural(Vacno for N)
   Vacdata tuple of comp Company, skills: Skills
```

The state space is modeled by a sort `Agency` and the initial state empty by the constant `empty` of sort `Agency`. No explicit model for the elements of the sort `Agency` is given, therefore functions are needed to observe the state. These *observers* are `isCand`, `skillsFor`, `isVacNo` and `vacFor`. The function `isCand` checks whether a person has applied for a job with a given agency or not. If a person has applied for a job then `skillsFor` returns his/her skills. Functions `isVacNo` and `vacFor` do a similar job for vacancies.

```
introduces
   isCand: Agency, Person → Bool
   skillsFor: Agency, Person → Skills
   isVacNo: Agency, Vacno → Bool
   vacFor: Agency, Vacno → Vacdata

   empty: → Agency
   apply: Agency, Person, Skills → Agency
   subscr: Agency, Company, Skills → Agency
   newVNo: Agency, Company, Skills → Vacno
```

The operations Apply and Subscribe are modeled by the functions `apply`, `subscr` and `newVNo`. Note that `apply` and `subscr` have an additional argument of sort `Agency` and return a value of sort `Agency`. Since LSL does not allow a function to return two values, the operation `Subscr` is split into two functions `subscr` and `newVNo`. The first returns a new agency and the second the vacancy number.

The axioms of the Agency trait describe the values of the observer functions for a given state, which is constructed from the operations `empty`, `apply` and `subscr`. Here we only give the axioms for `isCand` and `skillsFor`, the others are similar.

```
asserts
   forall a: Agency, p, p1: Person, c: Company,
          s, s1: Skills, n, n1 : Vacno

      ¬ isCand(empty,p);
      isCand(apply(a,p,s),p1) ⇔ p = p1 ∨ isCand(a,p1);
      isCand(subscr(a,c,s1),p1) ⇔ isCand(a,p1);

      skillsFor(apply(a,p,s),p1) = (if p = p1 ∧ ¬ isCand(a,p1)
                                    then s
                                    else skillsFor(a, p1));
      skillsFor(subscr(a,c,s1), p1) = skillsFor(a, p1);
```

One problem with this approach is that the specifier has to make the state of the abstract machine explicit as an argument to the operations of the abstract machine. This can be cumbersome in real life examples as Dauchy and Gaudel observe in [4] and "is in conflict with the tradition of imperative programming, where the state always remains implicit" [7].

One has to be careful when adding new operations to the abstract machine because all possible interactions of the new operations with the old operations and the observers have to be specified, and therefore, the resulting specification is likely to be unclear.

2. Algebraic Specifications and the Implicit State Approach

Another way to model abstract machines is by using the state as algebra approach [8, 12, 7, 18, 4, 11, 1, 6]. In this approach the state of an abstract machine is an algebra over a signature containing the components of the state. The state space St is given by an abstract datatype, a set of algebras over the signature of the state. For example, the state of an employment agency is given by the following LSL trait:

```
Agency (Person, Skill, Company): trait
   includes
      Set(Skill, Skills),
      Natural(Vacno for N),
      FiniteMap(CandsMap,Person,Skills),
      FiniteMap(VacMap,Vacno,Vacdata)
   Vacdata tuple of company: Company, skills: Skills
   introduces
      cands : → CandsMap
      vacs : → VacMap
```

The components of the state are the constants `cands` of sort `CandsMap` and `vacs` of sort `VacMap`. `CandsMap` is the sort of finite maps from `Person` to `Skills` and `VacMap` the sort of finite maps from `Vacno` to `Vacdata`. `FiniteMap` is a trait from the LSL library introducing the sort of finite, partial maps from a domain sort to a range sort. The trait introduces the operations `dom` to test if an element is in the domain of a finite map f, `update` for changing f at c to d and `apply` to apply f to a value in its domain. The sorts `Skills`, `Vacno` and `Vacdata` are defined as in the explicit state approach.

The initial states of an abstract machine are given by an abstract datatype having fewer models than the abstract datatype for the state space.

```
Empty : trait
   includes Agency
   asserts equations
         cands = {};
         vacs = {}
```

Relations as Abstract Datatypes. How do we specify the operations of the abstract machine? The proposal of this paper is to use again abstract datatypes to describe the operations, according to the relation as abstract datatype approach of Baumeister [3].

We start from the observation that each pair of algebras A and B of signature Σ can be combined to one algebra $A + B$ over the disjoint union of Σ and Σ. The signature $\Sigma \uplus \Sigma$ contains each symbol from Σ twice, primed and unprimed. Each disjoint union of signatures has associated two signature morphisms $\iota_1 \colon \Sigma \to \Sigma \uplus \Sigma$ and $\iota_2 \colon \Sigma \to \Sigma \uplus \Sigma$. The first morphism ι_1 is the identity on Σ and the second maps a symbol s from Σ to a symbol s' in $\Sigma \uplus \Sigma$. The algebra $A+B$ is constructed from A and B such that the interpretation of s in $A + B$ is the interpretation of s in A and the interpretation of s' is the interpretation of s in B.

As a consequence any relation $R \subseteq Alg(\Sigma) \times Alg(\Sigma)$ can be viewed as a set $R' \subseteq Alg(\Sigma \uplus \Sigma)$. On the other hand, any set $S \subseteq Alg(\Sigma \uplus \Sigma)$ can be viewed as a relation $R \subseteq Alg(\Sigma) \times Alg(\Sigma)$ by defining R as

$$R = \{ (C|_{\iota_1}, C|_{\iota_2}) \mid C \in S \}$$

Thus, there is a one to one correspondence between relations $Alg(\Sigma) \times Alg(\Sigma)$ and sets $S \subseteq Alg(\Sigma \uplus \Sigma)$.

In practice, an algebra representing a state has two components, an environment component and a state component. This situation can be described by an inclusion of the environment signature Σ_{Env} into the state signature Σ by $\iota \colon \Sigma_{Env} \to \Sigma$. If A and B are algebras representing a state and R is a relation on states, we require that the environment components of A and B remain unchanged.

$$A \; R \; B \Rightarrow A|_{\iota} = B|_{\iota}$$

For example, the environment components of `Agency` are the sorts `Skills`, `CandsMap`, `VacMap`, `Vacno`, `Vacdata` and the operations on them. Their interpretation should not change when applying the operations of the abstract machine. The constants `cands` and `vacs` are the only state components of `Agency`. These are the only components allowed to change their interpretation by the operations of the abstract machine.

As above, we have the following property. For each pair of Σ-algebras A and B that share the interpretation of Σ_{Env} (eg. $A|_\iota = B|_\iota$), there exists a unique algebra $A +_\iota B$ over the pushout signature $\Sigma \uplus_\iota \Sigma$. $A +_\iota B$ is called the amalgamated sum of A and B (cf. [5]).

$$\begin{tikzcd}
& \Sigma \uplus_\iota \Sigma & \\
\Sigma \arrow[ur, dashed, "\iota_1"] & \text{po} & \Sigma \arrow[ul, dashed, "\iota_2"'] \\
& \Sigma_{Env} \arrow[ul, "\iota"] \arrow[ur, "\iota"'] &
\end{tikzcd}$$

The morphism ι_1 is again the identity on Σ and ι_2 leaves all symbols in Σ from Σ_{Env} unchanged and maps s to s' if s is in Σ but not in Σ_{Env}. $A +_\iota B$ uses the same construction as $A + B$; however, this time $A +_\iota B$ is well defined for symbols s from Σ_{Env} if and only if the interpretation of s in A and in B are the same, but this follows from the requirement that $A|_\iota = B|_\iota$.

Again, there is a one to one correspondence between relations $R \subseteq Alg(\Sigma) \times Alg(\Sigma)$ such that $A\ R\ B \Rightarrow A|_\iota = B|_\iota$, and sets $S \subseteq Alg(\Sigma \uplus_\iota \Sigma)$. Because sets $S \subseteq Alg(\Sigma \uplus_\iota \Sigma)$ are just abstract datatypes of signature $\Sigma \uplus_\iota \Sigma$, operations on abstract machines can be built using techniques for building abstract datatypes. For example, we can give a set of formulas F over $\Sigma \uplus_\iota \Sigma$ and define a relation R by:

$$A\ R\ B \quad \text{iff} \quad A + B \models F$$

Note that sharing is essential in this approach because formulas over the disjoint union $\Sigma \uplus \Sigma$ can contain either primed or unprimed symbols but not both kinds together. Consider a constant c of sort s in Σ. Then $\Sigma \uplus \Sigma$ contains also a constant c' of sort s'. Now it is not possible to write a formula $c = c'$ to relate the interpretation of c in a Σ algebra A to the interpretation of c in the Σ algebra B because c and c' are of different sorts. However, if s is defined in the environment and c is a state component then $\Sigma \uplus_\iota \Sigma$ contains the constants c and c' of the *same* sort s. This allows us to write the formula $c = c'$.

In the employment example the trait `DeltaAgency` represents the pushout `Agency` $\uplus_\iota$ `Agency`

```
DeltaAgency : trait
   includes Agency(cands' for cands, vacs' for vacs), Agency
```

To specify the Apply operation of the employment agency, we have to give a specification of a set of `Agency` $\uplus_\iota$ `Agency` algebras. For this we include the trait `DeltaAgency` into a trait `Apply`.

```
Apply : trait
```

```
   includes DeltaAgency
   introduces
      p_in : → Person
      s_in : → Skills
   asserts equations
      ¬defined(cands,p_in);
      vacs' = vacs;
      cands' =  update(cands,p_in,s_in)
```

Apply introduces two new constants, p_in and s_in, to represent the input values of the operation. Given an interpretation of p_in and s_in as p_{in} and s_{in}, Apply denotes a relation between Agency algebras A and B such that

$$\begin{aligned} \mathtt{cands}_B &= \mathtt{update}_B(\mathtt{cands}_A, p_{in}, s_{in}) \\ \mathtt{vacs}_B &= \mathtt{vacs}_A \end{aligned}$$

provided that $\mathtt{defined}_A(\mathtt{cands}_A, p_{in})$ does not hold. Note that the interpretation of update and defined in A is the same as in B.

The specification of the Subscribe operation is given as follows, where n_out represents the output value of the Subscribe operation.

```
Subscr : trait
   includes DeltaAgency
   introduces
      c_in : → Company
      s_in : → Skills
      n_out : → Vacno
   asserts equations
      cands' = cands;
      ¬defined(vacs,n_out);
      vacs' = update(vacs,n_out,[c_in,s_in])
```

Note the similiarity of this specification of an employment agency with the Z specification of the employment agency from section 1 although we have used a model-oriented specification language, based on set theory, for one specification and an algebraic language, based on many sorted equational logic for the other.

We have chosen to model cands and vacs as constants of sorts CandsMap and VacMap because we wanted to stress the similarity of the LSL specification with the Z specification. However, since the logics used by Z and LSL *are* different, we might prefer to model the state of an employment agency by two total functions cands, from Person to Skills, and vacs, from Vacno to Vacdata, together with two predicates knownCands ⊆ Person and knownVacs ⊆ Vacno. This gives the following traits for the state space and the Apply operation.

```
Agency (Person, Skill, Company): trait
   includes
      Set(Skill, Skills),
      Natural(Vacno for N)
   Vacdata tuple of company: Company, skills: Skills
   introduces
      cands: Person → Skills
      vacs: Vacno → Vacdata
      knownCands: Person → Bool
      knownVacs: Vacno → Bool
Apply : trait
   includes DeltaAgency
   introduces
      p_in : → Person
      s_in : → Skills
   asserts
      forall p : Person, n : Vacno
         ¬ knownCands(p);
         knownCands'(p) ⇔ p = p_in ∨ knownCands(p);
         cands'(p) = (if p = p_in then s_in else cands(p));
         knownVacs'(n) ⇔ knownVacs(n);
         vacs'(n) = vacs(n)
```

Institution Independence. The argumentation just presented holds for any institution where the category of signatures *SIGN* has all pushouts and the model functor *Str* maps pushouts in *SIGN* to pullbacks in *CAT* (eg. the institution has unique amalgamated sums [5, 2]). This allows the use of a model-oriented style for specifying abstract machines independent from the concrete logic used to specify the state space and the operations. We could take any of the variety of logics used for algebraic specifications, for example, total functions, partial functions, non-deterministic functions (multi-algebras), (conditional) equational logic, first-order (w/o constraints), higher-order logic or any other logic that can be given the form of an institution and has the above mentioned properties.

3. Using Operations on ADTs to define Abstract Machines

In the previous section we have seen that an abstract machine can be specified using abstract datatypes for describing the state space, the initial states and the operations of an abstract machine. We have defined these abstract datatypes by providing a signature and a set of formulas over this signature. Each algebra satisfying this set of formulas denotes either a possible state or a pair of states in an relation. However, this is not the only way to define an abstract datatype. Another way is with the help of operations on abstract datatypes as defined, for example, by Sannella and Tarlecki in [17]. As abstract datatypes correspond to schemata in Z, the use of operations on abstract datatypes corresponds to the schema calculus of Z. In the following we give an example on how to use these operators to define an abstract machine of a stack from an abstract machine of a counter and an array.

Counter. The state of the counter has one component `c`, which is set in the initial states to `zero`.

```
Counter : trait
   includes Natural
   introduces c : → N
```

```
Zero : trait
   includes Counter
   asserts equations c = zero
```

The trait `DeltaCounter` defines the pushout of `Counter` $\uplus_\iota$ `Counter`, where the definition of natural numbers is in the environment and `c` is the only state component.

```
DeltaCounter : trait
   includes Counter, Counter (c' for c)
```

The increment operation on the counter increments the state component `c` by one. The decrement operation decrements `c` only if `c` is not `zero`. Note that `c'` is not given explicitly in terms of `c` but only implicit by the solution of the equation `succ(c') = c`.

```
Inc : trait
   includes DeltaCounter
   asserts equations
      c' = succ(c)
```

```
Dec : trait
   includes DeltaCounter
   asserts equations
      c ≠ zero;
      succ(c') = c
```

Array. The abstract machine specification of an array shows that not only constants can be state components but also functions. The state of an array has two functions `map` and `dom` as components. If `dom(n)` is true, `map(n)` yields the value of the array at index `n`. In the empty array the domain function `dom` always yields false.

```
Array : trait
   includes
      Natural
   introduces
      map : N → E
      dom : N → Bool
```

```
EmptyArray : trait
   includes Array
   asserts
      forall n : N
         ¬ dom(n)
```

The update operations assigns the value `v_in` to the map at `i_in`.

```
Update : trait
   includes DeltaArray
   introduces
      i_in : → N
      v_in : → E
   asserts forall n : N
         map'(n) = (if n = i_in then v_in else map(n));
         dom'(n) ⇔ n = i_in ∨ dom(n)
```

The get operation returns the value `v_out` for an index value `i_in`.

```
Get : trait
   includes DeltaArray
   introduces
      i_in : → N
      v_out : → E
   asserts forall n : N
         dom(i_in);
         map'(n) = map(n);
         v_out = map(i_in)
```

Stack Implementation. The state space of an abstract machine stack is the union of the state spaces for the counter and the array.

$$\mathtt{StackState} = \mathtt{Counter} \uplus_\iota \mathtt{Array}$$

`StackState` is the pushout of `Counter` and `Array` wrt. a common environment containing the union of the environment components of `Counter` and `Array`. Since the only operations available on traits in LSL are the inclusion of traits and the renaming of symbols, `StackState` is written as the following trait.

```
StackState : trait
   includes  Array, Counter
```

The initial states of the stack are given by the union of the initial states of the counter and the array.

$$\mathtt{Empty} = \mathtt{Zero} \uplus_\iota \mathtt{EmptyArray}$$

```
Empty : trait
   includes Zero, EmptyArray
```

The push operation is

$$\mathtt{Push} = \textbf{impose}\{\,\mathtt{i\_in} = \mathtt{c}\,\}\,\textbf{on}\,(\mathtt{Update} \uplus_\iota \mathtt{Inc})$$

The input parameter `v_in` of the update operation of the array is also the input parameter of the push operation.

```
Push : trait
   includes Update, Inc
   asserts equations i_in = c
```

The pop operation pops the stack and returns the top of the stack in v_out.

$$\texttt{Pop} = \textbf{impose}\{\,\texttt{c'} = \texttt{i\_in}\,\}\,\textbf{on}\,(\texttt{Get} \uplus_{\iota} \texttt{Dec})$$

```
Pop : trait
   includes Dec, Get
   asserts equations i_in = c'
```

4. Example of Proofs

One advantage of representing the state space and the operations of an abstract machine as abstract datatypes is the possibility of using tools for the specification language to apply to the specification of abstract machines. For example, we have used the LSL checker to type check all the LSL traits defining abstract machines.

In this section we will use the Larch Prover to prove the correctness of the abstract machine for a stack defined in the previous section. First, we have to give a more abstract specification of an abstract machine for a stack. To do this, we first specify a conventional (functional) abstract datatype of a stack. The trait Stack defines the sort S of stacks over elements of sort E.

```
Stack : trait
   introduces
      empty : → S
      isEmpty : S → Bool
      pop : S → S
      top : S → E
      push : E, S → S
   asserts S generated by empty, push
      forall e : E, s, s1 : S
         push(e,s) = push(e,s1) ⇒ s = s1;
         top(push(e,s)) = e;
         pop(push(e,s)) = s;
         isEmpty(empty);
         ¬ isEmpty(push(e,s));
```

The state space and the operations of the abstract machine of the abstract version of a stack are now defined with the help of Stack. The only state component of AStackState is the constant stack of sort S. AStackState and the initial state is given by

```
AStackState : trait
   includes Stack
   introduces stack : → S
```

```
AEmpty : trait
   includes AStackState
   asserts equations
      stack = empty
```

and the operations *push* and *pop* by

```
APush : trait
   includes DeltaStack
   introduces e_in : → E
   asserts equations
      stack' = push(e_in,stack)
```

```
APop : trait
   includes DeltaStack
   introduces e_out : → E
   asserts equations
      ¬ isEmpty(stack);
      stack' = pop(stack);
      e_out = top(stack)
```

To show that this abstract machine is implemented by the abstract machine of the previous section we use an abstraction relation between the state space of the implementation `StackState` and `AStackState`. Similar to the operations of an abstract machine we can define the abstraction relation `Alpha` by providing an abstract datatype over the pushout signature `StackState` $\uplus_\iota$ `AStackState`.

```
Alpha : trait
   includes StackState, AStackState
   introduces alpha: N → S
   asserts forall n:N
      alpha(0) = empty;
      succ(n) ≤ c ⇒ alpha(succ(n)) = push(map(n),alpha(n));
      stack = alpha(c)
```

`Alpha` is defined with the help of a function `alpha` from natural numbers to `S`, which depends implicit on the value of `map` in `StackState`. `alpha` creates a stack from a number and `map`. The relation between `StackState` and `AStackState` is established by the equation `stack` = `alpha(c)`.

We then have to show that `AEmpty` equals `Empty` $⨾$ `Alpha` and that `Alpha` $⨾$ *AOp* is the same as *Op* $⨾$ `Alpha` for all operations *Op* of the abstract machine stack where $⨾$ denotes the composition of relations. Here we can only give the proof for the push operation, the proofs of the others are similar. We show that `Alpha` $⨾$ `APush` equals `Push` $⨾$ `Alpha`. First, the composition of relations is represented as an algebraic specification. For this we introduce new symbols for the intermediate states, `stack''` in the first and `map''`, `dom''` and `c''` in the second case.

```
APush_o_Alpha : trait
   includes
```

```
      Alpha (stack'' for stack),
      APush (stack'' for stack, stack for stack')
Alpha_o_Push : trait
   includes Push (dom'' for dom', map'' for map', c'' for c'),
            Alpha (dom'' for dom, map'' for map, c'' for c)
```

In the next step `Alpha ⨟ APush` and `Push ⨟ Alpha` are included into one trait. We have to be careful to avoid name clashes; therefore, we decorate the input states, output states and the functions `alpha` with 1 and 2.

```
All : trait
   includes APush_o_Alpha (dom1 for dom, map1 for map,
               c1 for c, stack1 for stack, alpha1 for alpha)
            Alpha_o_Push (dom2 for dom, map2 for map,
               c2 for c, stack2 for stack, alpha2 for alpha)
```

Now we have to prove that if `e_in` = `v_in`, `dom1` ≡ `dom2`, `map1` ≡ `map2` and `c1` = `c2` then `stack1` = `stack2`.

```
   prove e_in = v_in ∧ c1 = c2 ∧ ∀ n (dom1(n) = dom2(n))
      ∧ ∀ n (map1(n) = map2(n)) ⇒ stack1 = stack2
```

This can be proved using the Larch Prover without much difficulty given the lemma ∀ `n` (`n` ⩽ `c` ⇒ `alpha1(n)` = `alpha2(n)`, which itself is proved by induction over `n` by the Larch Prover.

5. Conclusion

The novelty of the approach presented in this paper is to separate the model-oriented style for the specification of abstract machines from the logic used to talk about the state components and the relation between the state components of the pre- and post-state.

State space and operations of abstract machines are modeled by abstract datatypes, which can be defined as sets of Σ-structures in any suitable logic (institution) (cf. section 1 and [17]). Thus our method is parameterized by the notion of state, which is a Σ-structure in an institution. If we provide a model for the state of our abstract machines we get a method for specifying the operations of the abstract machines over this state. Therefore, a state can be an association of names to values and sets, as for example in Z or an association of names to values (no sets) and (partial/total) functions on sets, as with algebraic specifications or more concrete state structures involving structured names, pointers, arrays etc. For example, if we model the notion of state of a programming language we get a specification language for abstract machines over this kind of state in the spirit of the family of Larch Interface Languages [13].

In Z the state space and the operations of an abstract machine are represented as schemata, which are sets definable and manipulatable within Z. Thus in Z the state space and the operations are modeled by sets, but also the values of state components are sets and elements of sets. The advantage of this approach is that we can reason within the logic of Z about abstract machines, the disadvantage is that this also fixes the logic for describing the state components and the relation between state components of the pre- and the post-state to set theory.

As in Z we are able to reason about the state space and the operations of abstract machines using the logic used for modeling the state (cf. the example in section 4). No additional logic is needed to reason about the operations of abstract machines. This is unlike, for example, COLD-K, where dynamic logic is used to specify the operations (see below).

All other implicit state as algebra approaches [8, 12, 7, 18, 4, 11, 1, 6] have in common that they use an algebraic logic to define the state space of an abstract machine and another logic, for example, local function updates, elementary modifiers, dynamic logic etc. to define the operations. Another difference is that most of these approaches fix the logics for defining the state space and the operations and are not formulated in a logic independent way.

For example, the evolving algebra approach by Gurevich [12] uses unsorted algebra for the state space and defines a transformation of an algebra A to an algebra B by a set of guarded local function updates of the form $\textbf{if}\ b\ \textbf{then}\ f(\vec{t}) := t_0$. The semantics is that if b evaluates to true in A then the interpretation of f in B is $f_B(x) = (t_0)_A$, if $x = t_B$ and $f_B(x) = f_A(x)$ otherwise. The interpretation of all other functions remains unchanged.

In the implicit state approach by Dauchy and Gaudel [4] the state space consists of many sorted algebras. The operations are functors on algebras built from elementary modifiers $\mu\text{-}ac(\vec{p}) = t$ by sequential and indifferent composition and using conditionals.

COLD-K [7] uses many sorted partial algebras for specifying the state space and Harel's dynamic logic [14] to specify the operations, while Wieringa [18] uses order sorted algebra defined by equations and equational dynamic logic for the operations.

References

[1] Egidio Astesiano and Elena Zucca. A semantic model for dynamic systems. In *Modelling Database Dynamics*, Workshops in Computing, pages 63–83. Springer, Volkse, 1992.

[2] Hubert Baumeister. Unifying initial and loose semantics of parameterized specifications in an arbitrary institution. In *TAPSOFT '91, Volume 1: CAAP*, volume 493 of *LNCS*, pages 103–120, Brighton, UK, April 1991. Springer.

[3] Hubert Baumeister. Relations as abstract datatypes: An institution to specify relations between algebras. In *TAPSOFT '95*, volume 915 of *LNCS*, pages 756–771, Århus, Denmark, May 1995. Springer.

[4] P. Dauchy and M.-C. Gaudel. Algebraic specifications with implicit state, February 1994.

[5] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and initial Semantics*. Number 6 in EATCS Monographs on Theoretical Computer Science. Springer, 1985.

[6] Hartmut Ehrig and Fernando Orejas. Dynamic abstract data types, an informal proposal. *Bulletin of the EATCS*, 53:162–169, June 1994.

[7] Loe M. Feijs and H. B. Jonkers. *Formal Specification and Design*, volume 35 of *Cambridge tracts in theoretical computer science*. Cambridge Univ. Press, Cambridge, 1992.

[8] Harald Ganzinger. Programs as transformations of algebraic theories (extended abstract). *Informatik Fachberichte*, 50:22–41, 1981.

[9] J. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In *ICALP*, volume 140 of *LNCS*, pages 265–281. Springer, 1982.

[10] J. A. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.

[11] Martin Große-Rhode. *Specification of Transition Categories, An Approach to Dynamic Abstract Data Types*. PhD thesis, Fachbereich 13 — Informatik, Technische Universtiät, Berlin, 1995.

[12] Yuri Gurevich. Evolving algebras: An attempt to discover semantics. *Bulletin of the EATCS*, 43:264–284, February 1991.

[13] John V. Guttag and J. Horning. *LARCH: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer, New York, 1993.

[14] David Harel. Dynamic logic. In Dov M. Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic: Vol. 2: Extensions of Classical Logic*, volume 165 of *Synthese library*, pages 497–604. Kluwer, Dordrecht, 1984.

[15] I. J. Hayes, C. B. Jones, and J. E. Nicholls. Understanding the differences between VDM and Z. Technical Report UMCS-93-8-1, Computer Science, University of Manchester, August 1993.

[16] Cornelis A. Middelburg. *Logic and Specification: Extending VDM-SL for Advanced Formal Specification*. Computer science research and practice. Chapman and Hall, London, 1993.

[17] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76(2/3):165–210, February/March 1988.

[18] Roel Wieringa. Equational specification of dynamic objects. Technical Report 91-1, Faculty of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1991.