Using Occurrence and Evolving Algebras for the Specification of Language–Based Programming Tools

Arnd Poetzsch-Heffter
Institut für Informatik
Technische Universität
D–80290 München
poetzsch@informatik.tu-muenchen.de

1 Introduction

The specification of realistic programming languages is difficult and expensive. That is why such specifications should be used not only as formal language documentation, but as well as a starting point for the development of language specific software (interpreters, compilers, verification editors, browsers, etc.). The MAX system supports such developments in two ways: 1. It provides a formal, algebra-based specification framework. 2. It generates prototyping tools from such specifications and enables stepwise refinement of such tools (cf. [PH94] for an example). In the framework, static language aspects are defined by a very general attribution technique enabling e.g. the formal specification of flow graphs. Dynamic aspects are defined by evolving algebra rules, a technique that has been successfully applied to several realistic programming languages.

This extended abstract sketches the supported specification methods, the underlying formal approach, and experiences made so far by the MAX system.

2 Specification Methods

The MAX system supports operational language specifications. Conceptually, a MAX specification consists of two parts: the static language aspects, specified in a declarative way, and the dynamic aspects specified by transistion rules. Usually, the static aspects comprise context-free and context-dependent syntax, context conditions, basic data types of the language, and possibly further aspects that can be statically determined, like e.g. control flow information. Dynamic aspects usually deal with the operational behaviour of programs. Of course, there is no principle border between static and dynamic aspects. E.g. the relation between jumps and corresponding labels can be specified statically by edges in control flow graphs or dynamically by using continuation techniques. Essentially there are two reasons why MAX supports different techniques for specifying static and dynamic aspects:

1. Adequacy: Whereas declarative techniques are usually more compact and lead to simpler proof techniques, operational techniques are better suited for the specification of nondeterminism and parallelism.

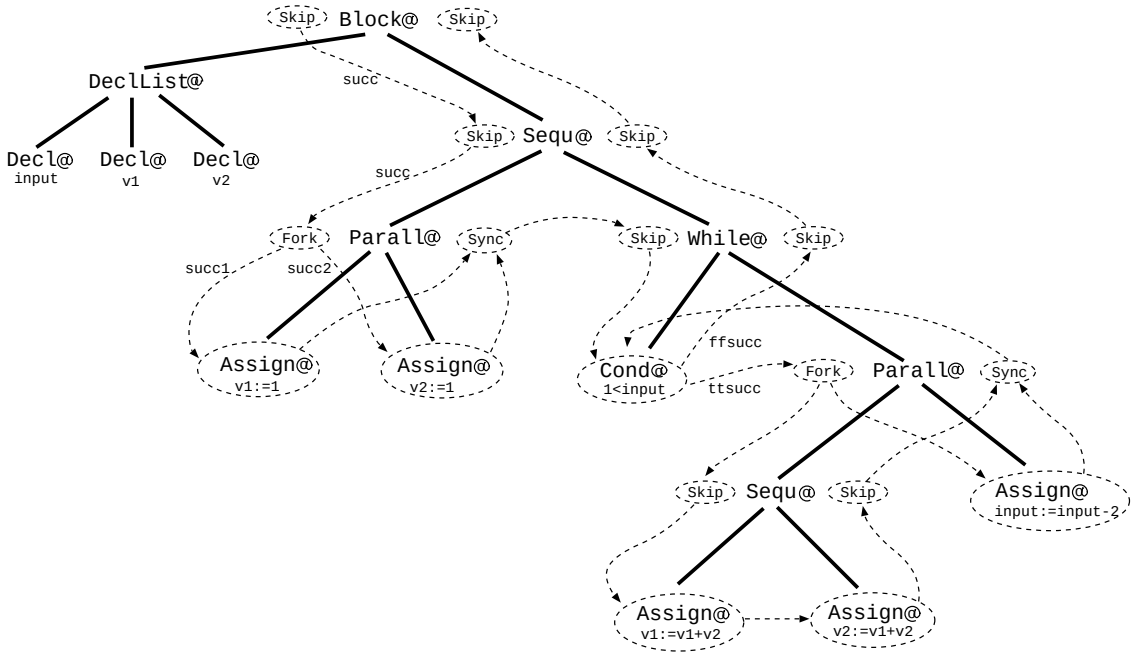
2. Efficiency: Distinguishing between static and dynamic aspects in the specification allows for the generation of more efficient programming tools.

To demonstrate the specification techniques, we roughly sketch the main concepts used in a specification of a **s**mall **im**perative, **p**arallel **l**anguage, called SIMPL. The following SIMPL program computes the Fibonacci function of the initial value of the variable *input*[1]:

[1] If *input* is initially even, the result is contained in *v1*, otherwise in *v2*.

```
declare
   var  input, v1, v2 : int
begin
   (  v1 := 1  ||  v2 := 1  );
   while 1 < input  do
      ( v1:=v1+v2 ; v2:=v1+v2 )  ||  input:=input-2
   end
end
```

Here, the operator `||` denotes parallel execution. The specification techniques for syntax and context conditions used in MAX are similar to techniques used with attribute grammars (cf. [DJL88]). The main difference is that in our approach attributes are ordinary functions of an algebra. This leads to a simpler formal model (cf. section 3) and provides more flexibility. In particular, it allows to specify control flow graphs in a declarative way as enrichment of syntax trees. The following figure visualizes how control flow graphs would look like for SIMPL programs using the example program from above. The nodes of the flow graph (dashed ellipses) correspond to the *atomic tasks* that have to be performed during program execution; `Skip`-tasks are only an auxiliary device to specify the control flow over the syntax tree:



The dynamic semantics of SIMPL specifies the signature of states, possible initial states, and for each type of atomic task the corresponding state transitions. States for SIMPL have four components: a) a function *cont* mapping variables to values; b) a variable *curr_task* holding the task to be executed next; c) a set of active tasks *sat*; d) a predicate *waiting* recording the status of `Sync`-tasks. Initially, *cont* may be arbitrary, *curr_task* is the `Skip`-task before the root, *sat* is empty, and *waiting* yields false for all `Sync`-tasks. Transition rules specify how the atomic tasks modify the state. This will be explained at the end of the next section.

3 Formal Approach

In order to realize the sketched specification method in a formal framework, the framework should support a powerful attribution mechanism and a technique for specifying transition rules. This attribution mechanism has to enable in particular the declarative specification

of flow information. For the specification of attributions, we developed occurrence algebras, an extension of order-sorted term algebras. For the specification of transition rules, we use evolving algebras developed by Gurevich (cf. [Bö95]). This section provides a first step introduction to both techniques. Beside these two techniques, MAX supports the specification of functions, i.e. in summary, a MAX specification consists of an occurrence algebra, a set of functions, and a set of evolving algebra rules.

3.1 Occurrence Algebras

Occurrence algebras can be considered as an extension of order–sorted term algebras. The universe of an occurrence algebra contains not only all terms that can be built by a set of constructors, but as well all occurrences[2] of subterms within these constructor terms. For the convenient specification of such an algebra, MAX provides a data construct similar to that of functional programming languages. As an example, let us consider part of the abstract syntax of SIMPL:

```
occdata
    Stm  =  Block  (  decls:   DeclList  body:    Stm  )
         |  While  (  cond:    Cond      lbody:   Stm  )
         |  Parall (  thread1: Stm       thread2: Stm  )
         |  Sequ   (  stm1:    Stm       stm2:    Stm  )
         |  Assign (  lhs:     Var       rhs:     Exp  )
end
```

This conctruct specifies sorts `Stm`, `Block`, `While`, `Parall`, `Sequ`, and `Assign`, specifies that `Block`, `While`, etc. are subsorts of `Stm`, specifies the constructors *Block*, *While*, *Parall*, *Sequ*, and *Assign*, and specifies selectors *decls*, *body*, etc.; i.e. we have axioms like:

$$body(Block(DL, S)) = S$$

In addition to that, the construct specifies for each sort a corresponding occurrence sort; e.g. `Block@` is the sort of all subterm occurrences of sort `Block` in some term. And, it specifies a constructor *occ* yielding for each term the corresponding root occurrence, as well as a unary constructor for each selector, e.g. *body*@: `Block@`→`Stm@`. If BO is an occurrence of sort `Block@`, i.e. BO represents a subterm B of sort `Block` in some term T, then $body@(BO)$ represents "the occurrence of $body(B)$ in T". Finally, it specifies a function *term* yielding for each subterm occurrence the corresponding subterm. Typical axioms are:

$$term(occ(T)) = T$$

$$term(BO) = Block(DL, S) \;\Leftrightarrow\; term(decls@(BO)) = DL \;\wedge\; term(body@(BO)) = S$$

Now, attributes in the sense of attribute grammars are simply unary functions having an occurrence sort as domain. Moreover, we can define new sorts based on occurrence sorts. E.g. the `Fork`- and `Sync`-tasks illustrated in the example above can be specified as follows:

```
occdata
    Task =  Fork(  nodef:  Parall@  )
         |  Sync(  nodes:  Parall@  )
            . . .
end
```

For a more detailed definition of occurrence algebras, we refer to [PH94].

[2] In some communities, subterm occurrences are called positions.

3.2 Evolving Algebras

Evolving algebras provide an elaborate specification framework used for a fairly broad range of applications (cf. [Bö95]). Essentially they allow to specify state transition systems where a state is modeled by an algebra. The MAX system uses the kernel notions of evolving algebras for the specification of transition rules. A rule essentially consists of a guard and a set of so-called updates. The meaning of a set of rules is as follows: Simultaneously perform all updates the guard of which evaluate to true in the current state. To give an idea of how such rules look like, we sketch the rules for SIMPL; for brevity, the rules for the `Cond`- and `Sikp`-tasks are omitted:

```
IF  isSchedule( curr_task )      THEN   curr_task :=  choose( sat )
IF  NOT  isSchedule( curr_task ) THEN   curr_task :=  Schedule
IF  isAssign( curr_task )
  THEN  sat  :=  sat\{curr_task} U { succ(curr_task) }
        cont( var(lhs(curr_task)) ) := eval( rhs(curr_task), cont )
IF  isFork( curr_task )
  THEN  sat :=  sat\{curr_task} U { succ1(curr_task), succ2(curr_task) }
IF  isSync( curr_task )  AND  waiting(curr_task) = false
  THEN   waiting( curr_task ) := true
         sat  :=  sat\{curr_task}
IF  isSync( curr_task )  AND  waiting(curr_task) = true
  THEN   waiting( curr_task ) := false
         sat  :=  sat\{curr_task} U { succ(curr_task) }
```

A task is either the task `Schedule` or one of the tasks shown in the above figure. If the current task is the `Schedule`-task, only the first guard is true and the next current task is chosen from the set of active tasks. Otherwise the guard of the second rule and of some other rule is true and the corresponding updates specify the state transition.

4 Pragmatic Aspects and Experiences

The MAX System is implemented in ANSI C and runs under different UNIX versions including Linux, HPUX, Solaris, and SunOS. An important design aspect of the MAX implementation was the development of a simple and flexible interface to C. Whereas this is of minor importance from a specification point of view, it is of great importance for the construction of realistic systems from specifications, in particular to allow flexible interaction with the environment (graphical user interfaces, etc.) and to support implementation refinements leading to efficient tools even for huge language specifications.

Until now, the MAX System has been used for several applications of different sizes and in compiler construction courses. In particular, it was used for the bootstrap of the system itself, for the specification of a subset of the object-oriented language Sather, and for an almost complete specification of ANSI C.

References

[Bö95] E. Börger, editor. *Specification and Validation Methods.* Clarendon Press, 1995.

[DJL88] P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars*, Springer–Verlag, 1988. (LNCS 323)

[PH94] A. Poetzsch-Heffter. Developing efficient interpreters based on formal language specifications. In P. Fritzson, editor, *Compiler Construction*, pages 233–247, Springer–Verlag, 1994. (LNCS 786)