Xasm– An Extensible, Component-Based Abstract State Machines Language

Matthias Anlauff

GMD FIRST, Kekuléstr. 7
D-12489 Berlin, Germany
ma@first.gmd.de

Abstract. The Abstract State Machine (ASM) [15] approach has already proven to be suitable for large-scale specifications of realistic systems [9, 8, 18, 20]. Due to the fact that the ASM approach defines a notion of *executing* specifications, it provides a perfect basis for a language, which can be used as a specification language as well as a high-level programming language. However, in order to upgrade to a realistic programming language, such a language must – besides other features – add a modularization concept to the core ASM constructs in order to provide the possibility to structure large-scale ASM-formalizations and to flexibly define reusable specification units. In this paper, the language Xasm, which stands for *Extensible ASM*, is presented. Xasm realizes a component-based modularization concept based on the notion of external functions as defined in ASMs. This paper also briefly describes the support environment of Xasm consisting of the Xasm-compiler translating Xasm programs to C source code, and the graphical debugging and animation tool.

1 Introduction

The Abstract State Machine approach has been and is successfully used to model a large number of case studies including industry-relevant ones. The simplicity of the basic data and execution model of ASMs makes them perfectly suitable as the basis for a language that on the one hand can be used as specification language and on the other hand as a high-level programming language. In this paper, the Xasm (*Extensible ASM*)[1] language is presented which aims at providing support for using ASMs as a programming language for producing efficient and reusable programs. There exists a number of other ASM implementations which all implement most of the ASM constructs as defined in the Lipari-Guide [15]. While the realization of the ASM constructs can be seen as the core functionality which must be present in each ASM support system, the difference of an ASM system compared to all others can be characterized by

- its efficiency,

[1] formerly known as "Aslan"; the name has been changed because of a name conflict with another tool.

– the functionality of its support environment,
– its rule abstraction concept, and
– its interoperability with other languages and systems.

For example, all ASM implementations – including XASM – define some macro structures on top of the core ASM language in order to provide some kind of rule abstraction concept. These additional features are indispensable for managing large formalizations. In the ASM-Workbench [11], for instance, the a special "Rule" construct is introduced being used to assemble ASM specifications from smaller pieces.

Concerning these features, XASM combines the advantages of using a formally defined method with the features of a full-scale, component-based programming language and its support environment.

The paper is organized as follows: In Section 2 an overview of XASM is given. Section 3 introduces the component-based module concept of XASM , in Section 4 the external language interface of XASM is described. In Section 6 the possibility to specify the syntax of input languages using context-free grammar definitions is presented, which is followed by the description of non-standard language constructs defined in XASM in Section 5. Section 7 sketches the support environment of XASM ; Section 8 contains concluding remarks and points out future work.

2 Overview of XASM

XASM is an implementation of sequential ASMs focusing on the generation of efficient executable programs simulating the run of the specified ASM. In general, the main design goals of XASM can be given as follows:

– full support of the ASM language as defined in the Lipari-Guide;
– efficient execution of generated executables;
– comfortable animation and debugging of ASM specifications;
– component-based library concept for managing large-scale specifications;
– external language interface for integrating ASM specifications in other systems.

The scenario of building ASM-based programs using XASM is depicted in Figure 1. XASM source files are translated into C source by the XASM-compiler. Additionally, the user can integrate C-sources and -libraries using the external language interface. As described below, XASM introduces a notion of components being stored in a special repository. During the translation process, the XASM-compiler retrieves registry information from the component in order to integrate pre-compiled XASM-components in the current build process. The result of such a build process is a binary being either an executable or a new element of the component library. In either case, the binary contains the ASM algorithms specified in the XASM source files.

Basically, XASM-programs are structured using "**asm** ... **endasm**" constructs each of which containing a list of local function and universe declarations

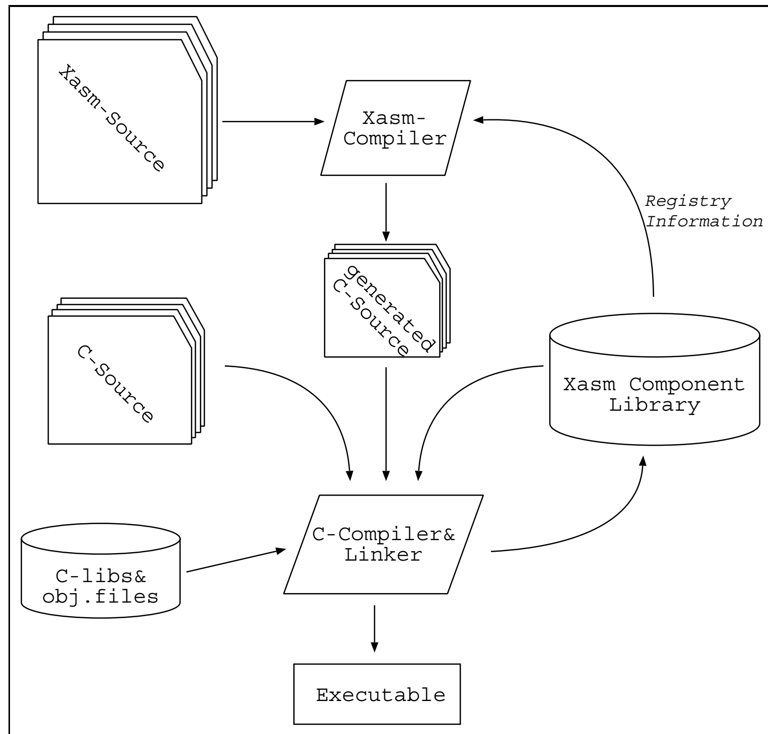


Fig. 1. Building XASM applications

and a list of ASM rules representing a certain part of the overall specification. In general, the structure of an XASM-**asm** is shown in Figure 2. The meta information part contains information concerning the role of the **asm** as a reusable component; this part is described in more detail below.

As defined in the Lipari-Guide, types are not part of the core ASM language. However, because typing has been proven to be very useful to avoid many kinds of errors, in XASM types can be supplied to the declaration of a function and are used to detect static semantics inconsistencies of the formalization.

3 The Basic Structure of XASM Programs: The XASM Component Model

In order to provide the full comfort of a modern programming language, pure ASMs lack a concept of modularization which is indispensable for structuring large-scale formalizations. Macros, which are normally used in the ASM literature to structure large ASM formalizations, only provide limited functionality with respect to the advantages one expects from a module concept. However, macros are a good means for "ASM-programming-in-the-small", but they fail

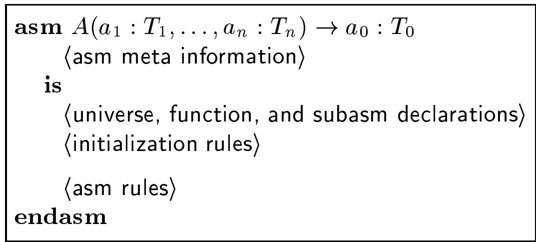


Fig. 2. The general structure of an XASM-**asm**

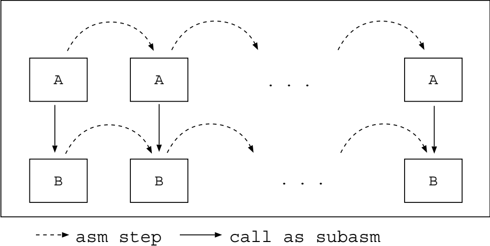
to provide a basis for writing ASM formalizations that can be re-used in other formalizations.

Therefore, XASM uses a more powerful modularization concept which is based on the notion of a *component* as it is used in component-based systems (e.g. [23, 25]).

3.1 The Use Modes of "asm"-Constructs

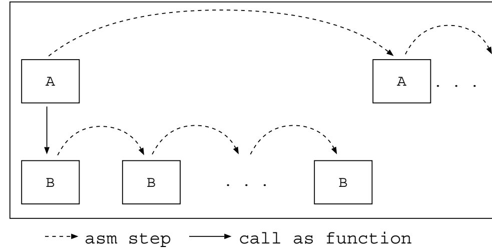
As mentioned above, an XASM formalization is structured using "**asm**. . . **endasm**" units. In order to explain the relationships that can exist between these units, we will first introduce the possible "use modes": An **asm** can be accessed by other **asms** in either of the following two ways:

If an asm A uses B as **sub-asm**, it means that B – possibly together with arguments, if the arity of $B > 0$ – is used as a *rule* in the body of A. If this rule fires, the rules of **asm** B fire, which may result in updating locations of functions declared in A. The sub-asm-use relation between **asms** may contain cycles; lazy evaluation techniques are used to avoid an infinite number of rules. The call as subasm is illustrated in the figure. The subasm B and its parent asm A step simultaneously; formally they can be seen as one single ASM.



Asm A uses B as a **function**, if B is defined as **external function** in A. In this case, B – possibly together with arguments, if the arity of $B > 0$ – is used as a

term in the body of A. Recursion is allowed, so that the function-use relation between **asms** may contain cycles. The call as function is illustrated in the figure. During the run of the function-asm B, its parent A doesn't make any step; from A's point of view B's run happens in zero time. As depicted in the figure, B behaves like a "normal" asm, the iterations shown here are caused by the steps of the B-asm itself.



In each of the above cases, we call A the *parent-asm* of B, if A uses B as sub-asm or as function. In any case, the **asm** must be declared in the parent **asm**. As part of its meta information, an **asm** can be marked as a function or as a sub-asm, so that it can only be used by other **asms** in the specified way. For example, if B and C are **asms** defined as follows

asm $B(x : Int) \rightarrow Int$
    used as function
  is
    ...
endasm
	asm $C(x : Int)$
	    used as subasm
	  is
	    ...
	endasm

then B can only be used as function and C as sub-asm in other **asms**. This is reflected by corresponding declarations of B and C:

asm A
  is
    subasm $C(x : Int)$
    external function $B(x : Int) \rightarrow Int$
    ...
endasm

Example

A typical situation for using sub-asms is given, when the specification can be split up naturally into several sub-specifications each of which modeling a certain aspect of the overall specification.

```
asm Robot is
    universe ModeValue = {standing, moving}
    subasms Robot_is_standing,
            Robot_is_moving
    function mode → ModeValue
    ...
    if mode = standing then
      Robot_is_standing
    elseif mode = moving then
      Robot_is_moving
    endif
    ...
endasm
```

```
asm Robot_is_standing
    used as subasm
  is
    ...
    mode := moving
    ...
endasm
asm Robot_is_moving
    used as subasm
  is
    ...
    mode := standing
    ...
endasm
```

In this case, the specification introduces the notion of a mode which can be used to structure the formalization. In the example, it is assumed, that the sub-asms update the value of the mode function to some value.

3.2 XASM Components

The declaration of sub-asms and external functions that refer to other **asms** in the specification requires that the existence and functionality of these **asms** is known at specification time. This is comparable to a static module concept and is useful for defining sub-parts of one specific formalization. In order to be "component-based" like announced above, the module concept must be enriched with some other features allowing a more flexible and comfortable definition of reusable units.

For example, consider the following **asm** that may be used in the context of a programming language semantics specification. It checks, whether a given variable is defined in the current block, or in one of the parent blocks. The information whether a variable is defined in a certain block is stored in the ASM function *DeclTable*; the block structure is stored in the function *ParentBlock* mapping blocks to its corresponding parent blocks:[2]

[2] the meaning of the **return** rule is explained later

```
asm check_blockvar(block : Str, var : Str) → Bool
      used as function
      accesses functions DeclTable(Block : Str, var : Str) → _ ,
                         ParentBlock(Block : Str) → Str
   is
      function current_block ← block
      if DeclTable(current_block, var) ≠ undef then
         return true
      else
         current_block := ParentBlock(current_block);
         if current_block = undef then
            return false
         endif
      endif
endasm
```

Note, that these function works correctly without recursively calling itself; it iterates until no update changes the internal state of the asm.

The "**accesses**" construct is used to specify the functions the **asm** expects from its parent **asm**. Now, with this additional meta information, the **asm** can be regarded as a *component*, because its provides information necessary to be processed as stand-alone unit. The **asm** can be separately compiled and put into the XASM component library; other formalization can reuse it provided that they declare the required functions.

Besides the "**accesses**" construct, which allows to *read* the locations of the corresponding functions provided by the parent **asm**, the XASM "**updates**" construct marks the corresponding function as readable *and* writable for the sub-asm or ASM function. In the previous example, the *mode* function must be marked as "updated" the two sub-asms, because it is updated in the body of each of them:

```
asm Robot_is_standing
      used as subasm in Robot
      updates function
         mode → ModeValue
   is
      ...
      mode := moving
      ...
endasm
```

```
asm Robot_is_moving
      used as subasm in Robot
      updates function
         mode → ModeValue
   is
      ...
      mode := standing
      ...
endasm
```

Like the accessed functions, the updated functions must be declared in the parent **asm**. In order to avoid repetitions in the source code, the notation "**used as subasm in** A" can be used as an abbreviation of accessing all functions and universes declared in A except those that are explicitly marked as "updated" by the sub-asm (analogously for asms that are used as functions).

Besides functions, sub-asms can also be contained in the "accesses"-list of an asm-component. The accessed sub-asms are used in the rule section of the asm as if they have been declared locally.

3.3 Gluing of XASM components

In order to provide a high degree of flexibility in interconnecting XASM-components, it is possible to define local derived functions using so-called "**with**" definitions. For example, a **asm** A wants to use the *check_blockvar* as introduced above, but A doesn't declare a function named "$DeclTable$" as it is required by the *check_blockvar* **asm**. However, the $DeclTable$ must be somehow expressible using existing functions in A. The "with"-statement can be used to provide the called **asm** with the necessary function declaration, as illustrated in the following example:

```
asm A is
    ...
    function currentmodule → Str
    function SymTable(mod : Str, block : Str, v : Str) → Int
    external function check_blockvar(b : String, v : Str) → Bool
            with DeclTable(b : Str, v : Str) ==
                SymTable(currentmodule, b, v)
    ...
endasm
```

In a similar way, accessed sub-asms can be specified in the context of a "with" statement.

Using XASM components together with this kind of gluing mechanism provides a powerful means to structure large specifications using smaller and reusable units.

3.4 Informal Semantics of "accesses" and "updates" Declarations

The semantics of the "accesses" and "updates" declaration in **asms** depends on whether the **asm** is used as a sub-asm or as a function. In the following, the semantics of these constructs in each of these cases is explained briefly.

Accessed and updated functions in Sub-Asms If an **asm** B is used in A as a sub-asm, the accessed and updated functions in B are directly linked to the corresponding functions in A. That means that if B updates a function declared in A, the update is visible for both A and B in the subsequent step. This can be done in this way, because A and B step simultaneously; the rules of B are regarded as part of the rules of A. Similarly, if B accesses a sub-asm C, then firing C in B has the same effect than firing C in A.

Accessed and updated functions in External Functions The more complicated case is given when an **asm** B being used as an external function in A accesses and updates functions declared in A. Due to the fact that A doesn't make any step during the run of B, rules in B updating dynamic functions declared in A are actually not performed from B's point of view. Therefore, in XASM the semantics of updated functions in external asm functions is defined in a way, that these kind of unintuituive behavior is avoided:

- For each function f being marked as "updated" a local function with the same name is (internally) declared in B;
- this local function is initialized with the values of the original function in A;
- during the run of B, the functions marked as "updated" can be accessed like any other local function in B;
- on termination of B the updated locations of these function are propagated to the original function declared in the parent **asm** A.

This ensures, that *all updates* of an "updates" function are accessible in B, and that only the *last updates* are forwarded to the parent A. In general, the updates of functions declared in A and updated in B are treated as being part of the update set of A's current step. As a consequence, multiple invocations of B in the same step of A do not influence each other.

Consider the following – somewhat artificial – example:

```
asm A is
    function f(x : Int) → Int
    function v → Int
    external function B → Int
    ...
    v := B
    ...
endasm
```

```
asm B → Int
    used as function
    updates function f(x : Int) → Int
  is
    function i ← 0
    function r
    if i < 3 then
      r := f(0)
      f(0) := i
      i := i + 1
    else
      return r
    endif
endasm
```

In each step of the run of **asm** B the value of the updated function $f(0)$ is updated with a new value. The semantics of the "updates" declaration in XASM ensures that *all* updates of f are accessible in B and that only the *last* update of f in B is propagated to the parent **asm** A. In this example, the update $f(0) := 2$ is propagated to A, while all other updates occurring in the "internal" steps in B[3] have only local effects in B.[4]

[3] in step 1: $f(0) := 0$; in step 2: $f(0) := 1$

[4] that means that in the second step the update $r := 0$ is performed in B, in the third step $r := 1$

3.5 Access Modes of External Functions

In order to allow different kinds of accesses to external entities, external functions can be declared either as "monitored" or as "output" functions. In the first case, the external function can be read, but not written (e.g user input), in the second case, the external function can be written, but not read (e.g. output channels like stdout and stderr).

As a restriction, an external function can be either a monitored *or* an output function, not both. If a function would have both modes, reasoning about the values of that function would require special case distinctions: If a location of such a function is updated in one step of the ASM by means of an update rule, it cannot be guaranteed that the location has the updated value in the next step, because the environment might have changed it in the meantime.

In the following example, the 0-ary function *error* is defined as an external function with "output" access mode; it is used in the parent **asm** for displaying an error message on "stderr" and for setting an ok_flag to false.

```
asm A is
    relation checkok
    external [output] function error → Str
            with ok_flag == checkok
    ...
    error := "..."
    ...
endasm

asm error → msg : Str
    used as function
    updates relation ok_flag
  is
    use stdio
    stderr := msg
    ok_flag := false
endasm
```

The value that is used for updating the external function can be accessed using the named result parameter msg. The **use** construct includes pre-defined header files containing function declaration that are in this case used to declare the external functions *stdout*, *stdin*, and *stderr*.

If no access mode is specified in the declaration of an external function, the mode "monitored" is assumed.

3.6 Scoping

In the context of use-relations between **asms**, XASM distinguishes between the *parent-asm* and the *caller-asm*:

> The **parent-asm** of an **asm** B is the **asm** where B is declared (either as sub-asm or as external function), while the **caller-asm** is the **asm** where the call actually takes place.

In the easiest case, parent and caller are the same, as in the above example: **asm** *error* is declared in and called by **asm** A. In the following example, this is not the case:

```
asm A is                                           asm error → msg : Str
    relation checkok                                   used as function
    external [output] function error → Str             updates
            with ok_flag == checkok                      relation ok_flag
    subasm B                                         is
    ...                                                use stdio
    B                                                  stderr := msg
    ...                                                ok_flag := false
endasm                                             endasm

asm B
    updates function error → Str
  is
    error := "..."
endasm
```

Here, A is the parent-asm and B the caller-asm of **asm** *error*. As a consequence, the exported relation *ok_flag* is taken from the parent-asm, rather than from the caller-asm. That means, that the update of *error* in B has the consequence that the *checkok* relation is updated in A. This distinction has been made, in order to completely abstract from the actual realization of exported and accessed functions and sub-asms. In this case, B doesn't need to "know" that *error* is an external function.

The scoping rule for XASM-**asms** is similar to static scoping in programming languages and can be summarized as follows:

> *Exported and accessed functions and sub-asms of an* **asm** B *are always taken from the* **asm** *where* B *has been declared either as external function or as sub-asm.*

3.7 Returning Values From External Functions

As already frequently used in the examples in this document, the **return** construct is used to specify the return value of a monitored external function. In terms of ASMs, "**return** " is realized as follows: In each **asm** B that is used as monitored function, a 0-ary dynamic function *B_result* is declared and initialized with *no_result*, a specific element of the superuniverse. Let R be the rule of B as defined by the user, then the internally used rule representing the body of B is given by the following conditional:

```
if B_result = no_result then
  R
endif
```

In other words, updating *B_result* with value different from the special *no_result* element directly forces the **asm** to terminate. The notation "**return** t" in an **asm** B is then simply an abbreviation for the update "$B_result := t$".

4 The XASM External Language Interface

In order to integrate ASM algorithms into other applications, XASM defines an external language interface. In the current version, this interface is implemented for the connection of XASM programs with programs written in the C language. Interfaces to other language, like Java, are in preparation.

In principle, there are two alternatives how the interconnection to the external application can be realized:

– C-functions are used to implement external ASM functions, or
– XASM-**asms** are called from the C main program.

In the first case, the main control of the application is handled by the XASM-part of the system, while in the second case the C-application has the main control. This is also reflected by the definition of the "main" C function: in the first case it is contained in the XASM-part, in the second case, the C-part must provide it. The corresponding interfaces of XASM for these two alternatives are explained in the following.

4.1 External C-functions

In the previous section we have shown, that external function can be specified in XASM using the **asm** construct. Alternatively, external functions can be implemented in C. The corresponding XASM-declaration is given as follows:

external "C:*c_name*" [*access_mode*] **function**
$xasm_name(a_1 : T_1, \ldots, a_n : T_n) \rightarrow T$

The *c_name* specifies the name of the C function; it can be omitted, if it is equal to the XASM-name of the function. Depending on the access mode of the XASM-function, the corresponding C-function prototypes differ slightly:

– If the access mode is "monitored", the C-functions are defined as follows:
  `ASMOBJ c_name(ASM a, int argc, ASMOBJ* argv);`
  where "ASMOBJ" is the C-type representing elements of the superuniverse; "ASM" is a C-struct containing information related to parent **asm**. These types are specified in the header file "xasm.h" which must be included in those files containing external function implementations. The arguments of the function call can be accessed via the "argv" field using "argc" as argument count. The first argument, argv[0], always contains the name of the

corresponding XASM-function as string element. The result of the external function that is accessible in the calling **asm** is returned by the C-function as "ASMOBJ".

– If the access mode is "output", the C-functions are defined as follows:
  `void` *cname*`(ASM a, int argc, ASMOBJ* argv, ASMOBJ val);`
  In this case, additionally the value that is used in the update representing the call of the external function can be accessed using the "val" parameter.

As an example, the following C-code contains the implementation of the "stderr" function previously used in one of the examples:

```
void xasm_stderr (ASM a, int argc, ASMOBJ* argv, ASMOBJ val) {
  if (argc != 1) {
    error("wrong # args for external function '%s'.\n",
          c_stringvalue(argv[0]));
    return;
  }
  fprintf(stderr,"%s",str_obj(val));
}
```

The XASM-library function "str_obj" returns the string representation of an ASMOBJ. The corresponding declaration of the external C-function in an asm has the following format:

external `"C:xasm_stderr" [output]` **function** *stderr* → *String*

4.2 Embedding XASM-programs in C-applications

If the XASM-part of a system should provide services for a C-based application, the main **asm** and all sub-asms of it can be called from the C-code. In this case, the XASM-compiler must be invoked with a special option that prevents the generation of the "main"-function.

Before any of the **asms** can be invoked, the XASM-part must be initialized. For this purpose, the generated C-code defines the function "asm_main" as follows:

```
int asm_main(int argc, char **argv);
```

The arguments to this function are given as strings which are parsed and transformed to corresponding "ASMOBJs". The actual invocation of the main **asm** can be made using the C-function "run_mainasm":

```
ASMOBJ run_mainasm();
```

This kind of embedding is actually used in the implementation of the XASM-compiler itself: the static semantics check is carried out by an algorithm specified in XASM.

A number of external C-functions are already integrated into the runtime system. For example, external functions to communicate using UNIX-Sockets, string manipulation functions, file access functions etc. .

5 Non-Standard Language Constructs of XASM

Besides the implementation of the ASM core constructs, XASM provides a number of useful extensions that can all be directly mapped to the original ASM constructs. In the following, some of these extensions are described briefly; a full version of the language specification is in preparation and will be available shortly.

5.1 Constructor Terms

XASM provides the possibility to define and use *constructor terms*. The concept of constructor terms can be mapped to the ASM core language as follows: According to [16] each of the function names contained in the vocabulary V of an ASM may be marked as *relational* or *static*, or both. In addition, we allow static functions to be marked as *constructive*. Let F_c be the set containing all functions in V marked as constructive, $F_c \subseteq V$. Let $f_c \in F_c$, arity of $f_c = n$, then the following conditions hold for all states A of the ASM:

(i) $\forall t_1, \ldots, t_n, \mathrm{V}al_A(t_i) \neq$ **undef** $, 1 \leq i \leq n \bullet$
$f_c(t_1, \ldots, t_n) \neq$ **undef**
(ii) $\forall g_c \in F_c,$ arity of g_c is $m; \quad t_1, \ldots, t_n, \mathrm{V}al_A(t_i) \neq$ **undef** $\bullet$
$f_c(t_1, \ldots, t_n) = g_c(s_1, \ldots, s_m) \Leftrightarrow$
$f_c = g_c \wedge n = m$
$\wedge \mathrm{V}al_A(t_i) = \mathrm{V}al_A(s_i), 1 \leq i \leq n$

where $\mathrm{Val}_A(t)$ stands for the evaluation of term t in state A of the ASM. Informally speaking that means that each constructive function is (i) defined at all locations and that (ii) the content of each location is a unique element of the superuniverse w.r.t. the set of locations of all constructive functions. If $f_c \in F_c$, then f_c is called a *constructor*, and the terms $f_c(t_1, \ldots, t_n)$ are called *constructor terms*. In XASM, the declaration of a constructor is part of the function declarations, for example

constructor $nil, cons(_,_)$
universe $BinTree = \{empty, children(l : BinTree, r : BinTree)\}$

introduces the constructors nil, $cons$, $empty$, and $children$, where terms constructed using the latter two constructors are elements of the universe $BinTree$.

XASM also provides pattern matching functionality like it is used in many other languages. Syntactically, pattern matching terms are used as condition terms in conditionals, e. g.: [5]

[5] XASM uses a special syntax for pattern matching variables and equality symbol

```
if b =~ children(&l, &r) then
   R(&l, &r)
else
   ...
endif
```

There are three kinds of pre-defined, commonly-used constructors in XASM: sets, sequences, and tuples. These constructors are specified using their usual representation: $\{x_1, \ldots, x_n\}$ for sets, $[x_1, \ldots, x_n]$ for sequence, and $(x_1, \ldots, x_n)$ for n-tuples. For sequences, the notation $[H|T]$ can be used in pattern matching terms for accessing head and tail of a sequence.

5.2 Regular Expressions

In practice, strings are widely used as a common data format for exchanging information between different systems. XASM therefore provides a special kind of pattern matching based on *regular expressions* as they are used in UNIX (e.g. in the "sed" program) as well as in many scripting languages like Perl [27] and Tcl [21]. The regular expression pattern matching is invoked using the =~ operator like for pattern matching with constructor terms. If both operands of the =~ operator are strings, then the right operand is interpreted as regular expression and the left operand as string being match against the regular expression. For example, the regular expression pattern matching expression

```
s =~ '^[A-Z]'
```

evaluates to true, if s is a string starting with a capital letter.

In regular expressions, parenthesis "\(..\)" can be used to mark certain parts of the expression that correspond to sub-strings of the left-hand-side string, if the pattern matching has been successful. For that, Xasm provide a special form of regular expression pattern matching: If the left-hand-term of a pattern matching expression evaluates to a string object s and the right-hand-term evaluates to a tuple the first argument of which represents a string object r and the remaining arguments are pattern matching variables $\&v_1, \ldots, \&v_n$, the string s is matched against the regular expression r and the sub-matches are put into the pattern matching variables $\&v_1, \ldots, \&v_n$, if the match has been successful.

```
if "AnyString" =~ ('^\(.\)\(.*\)$', &hd, &tl) then
   ...
endif
```

In this example, the regular expression contains two sub-matches, the first one matches the string `"A"`, the second one the string `"nyString"`.[6] The submatches can be accessed in the then-part of the conditional rules as values of the pattern matching variables $\&hd$ and $\&tl$.

[6] Single quotes are used for the regular expressions in order to prevent interpretation of special symbols (like "\") in the string. To understand the example: ^ stands for

5.3 The "Once"-Rule

A common situation occuring in ASM formalizations is that certain rules should fire only once during the run of an ASM. Normally, one has to introduce extra functions in order to ensure this behaviour. XASM introduces a "once" rule for these situations:

if once R_1
else R_2
endif

which is – in terms of ASMs – equivalent to the conditional rule

if $once(n)$ **then**
  R_1
  $once(n) := false$
else
  R_2
endif

where n is a unique number representing the n'th occurence of a "once"-rule in the ASM, and *once* is a one-ary dynamic relation initialized with **true** for all these numbers n. The notation "**once** R" is an abbreviation for

if once R **else skip endif**

6 Grammar Definitions in XASM

Historically, XASM has been developed as underlying ASM implementation for *Montages*, a semi-visual method for specifying the syntax and semantics of programming languages, see [1–4, 19] and [12, 17] in this proceedings. As a consequence, the support for programming language related features has been integrated into the XASM language as a means to extend the original syntax with domain-specific constructs. The syntax and semantics of these extensions can be specified using the Montages method together with tool support environment Gem-Mex which translates user-defined language definitions into XASM-code.

The grammar definitions used for the translation of Montages-specification can also be used directly in XASM. For that purpose, **nonterm** and **token**-declarations can be given in XASM, resulting in the generation of a parser for the specified language. As an example for using grammar definitions in XASM, Figure 3 contains the specification of a parser that accepts empty XML tags. The generated C-function can be accessed using an external function returning the root node of the parse tree being constructed during parsing.

7 The XASM Support Environment

The support environment of XASM consists of the XASM-compiler, the runtime system and the graphical debugging and animation interface. In this paper, these

the beginning, $ for the end of a string, the dot represents any character, and the asteriks stand for zero or more occurences of the preceeding expression.

```
asm xmlparser(inpfile) is
    use syntax
    token Ident = "[A-Za-z][A-Za-z0-9_]*"
    startnonterm Elements
    nonterm Elements[Element]
      base /* empty */
      cont Elements Element
    endnonterm
    nonterm Element = ElementEmpty endnonterm
    nonterm ElementEmpty ::= "<"Ident Attributes"/ >";
    endnonterm
    nonterm Attributes[Attribute];
      base /* empty */
      cont Attributes Attribute
    endnonterm
    nonterm Attribute ::= Ident"="String_token;
        lhs ← (Ident.Name, String_token.Name)
    endnonterm
    external "C" function parse_Elements(filename : String) → Elements
    function RootNode → Elements
    RootNode := parse_Elements(inpfile);
endasm
```

Fig. 3. A grammar specification in XASM for parsing empty XML Elements

tools are only sketched briefly, a more detailed description will be contained in the XASM user manual which is currently under development. A description of the graphical animation tool is also contained in [1, 2].

7.1 The XASM-compiler and runtime system

As already mentioned in Section 2, the XASM-compiler `xasmc` translates XASM source code into C-code implementing the executable version of the ASMs specified in the XASM source files. The syntax analysis part is implemented in C using "lex" and "yacc" for generating scanner and parser code. The type checking part is implemented in XASM itself using the C-interface as introduced above.

The XASM runtime system implements the core functionality of the XASM language. In here, all algorithms and data structures are realized being used to transform an ASM into an executable program. At the heart of the runtime system is the implementation of update and access functionality for ASM functions. For that, a hashing mechanism is used to provide optimized access to values of ASM functions. The runtime system also contains garbage collection facilities, which is indispensable, if ASM algorithms are used for continuous control systems, as described in [5].

7.2 The XASM graphical animation and debugging interface

In order to be able to animate and/or debug the XASM program, a graphical animation and debugging tool has been realized that enables to stepwisely execute the ASM, to trace updates that has been performed in each step, and to view function values in each step. In case a grammar has been specified as input format for the XASM program, a special kind of graphical animation window can be used to display function values that refer to node in the parse tree. Figure 4 shows a screendump of a debugging session. Additionally, an integrated design environment, incorporating the graphical user interface is currently under development.

7.3 The XASM-LaTeX Package

As an additional support feature, an LaTeX-package "asm.sty" is defined for typesetting XASM specifications. The LaTeX-files can directly be used as input file to the XASM-compiler, so that no additional work is necessary to produce a high-quality documentation from a running XASM specification. The XASM code parts in this document are produced using the asm style being realized based on the LaTeX "program" style as defined in [13].

8 Conclusion

In this paper, the ASM based language XASM has been presented focussing on the additional features provided by the language with respect to the ASM core

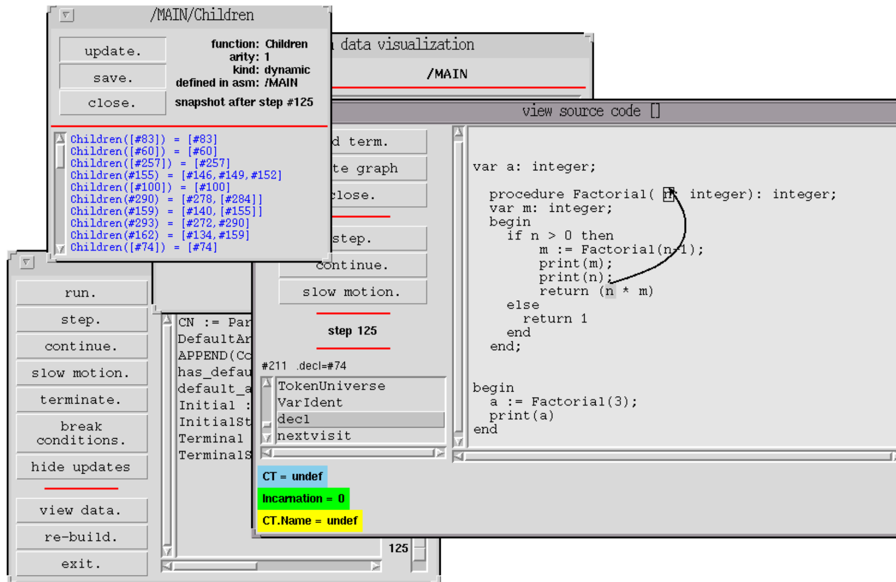


Fig. 4. Snapshot of a XASM-debugger session

concepts as defined in the Lipari Guide. A novel concept for structuring ASM specifications based on the notion of components has been presented. This concept perfectly fits into the basic model of the ASM approach, because it allows to choose the level of abstraction for describing that fits best to a given problem without regarding technical constraints. Furthermore, this concept allows efficient development cycles, because asm's can be designed as reusable components by exactly specifying what is expected from the environment.

The language presented in this paper is fully implemented. The system is used as the basis for the Montages/Gem-Mex, where generated XASM code is translated into an interpreter for the language specified using Montages. Other case studies are currently under development, see for example [26] in this proceedings.

As next steps, the XASM compiler, runtime system and graphical support environment will be further optimized. Also, a concept how tools for the (automatic) verification of the ASM formalizations can be integrated into the system is currently under development and will be part of the support system in future versions of the tool. Furthermore, the connection to repository systems will be subject to future considerations concerning the XASM-language. For example, certain ASM functions may be marked as "persistent", meaning that the values of the locations of these functions are stored in the repository system, so that they can be access the next time the XASM is executed. This kind of extension is currently part of work carried out in an industry-based research project running

at GMD In this project, it is currently considered to use XASM for formulating certain consitency check algorithms occuring in the context of this project.

Acknowledgements

I very much thank Philipp Kutter and Alfonso Pierantonio for their collaboration and fruitful discussions which had a great influence in the design and implementation of XASM. I also like to thank Yuri Gurevich and Egon Börger for their interest in my work presented in this paper and for always helping me solving the right problems. My dearest special thanks go to Asuman Sünbül. Her work in the field of component-based software engineering has very much influenced the component model presented in this paper.

References

1. M. Anlauff, P. Kutter, and A. Pierantonio. Formal Aspects of and Development Environments for Montages. In M. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing, Amsterdam, 1997. Springer.
2. M. Anlauff, P. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In A. D.Bjoerner, M.Broy, editor, *Perspective of System Informatics*, LNCS, 1999. to appear.
3. M. Anlauff, P. Kutter, and A. Pierantonio. Tool Support for Language Design and Prototyping with Montages. In *Proceedings of Compiler Construction (CC'99)*, Lecture Notes in Computer Science. Springer, 1999.
4. M. Anlauff, P. Kutter, A. Pierantonio, and L. Thiele. Generating an Action Notation Environment from Montages Descriptions. In P. Mosses and D. Watt, editors, *Proceedings of the 2nd International Workshop on Action Semantics (AS'99)*, number NS-99-3 in BRICS Notes Series, pages 1–42. University of Aarhus, Department of Computer Science, Mar. 1999.
5. M. Anlauff and A. Sünbül. An ASM specification of an elevator control system. 1999.
6. M. Anlauff and A. Sünbül. Software architecture based composition of components. In *GI-Workshop Sicherheit und Zuverlässigkeit software-basierter Systeme*, 1999.
7. J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In Ciancarini and Hankin [10], pages 75–88.
8. E. Börger and W. Schulte. A Modular Design for the Java VM architecture. In E. Börger, editor, *Architecture Design and Validation Methods*. Springer, 1998.
9. E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.
10. P. Ciancarini and C. Hankin, editors. *Coordination and models, Proceedings of the first international conference, Cesena, Italy*, number 1061 in LNCS. Springer Verlag, 1996.
11. G. Del Castillo. The ASM Workbench: an Open and Extensible Tool Environment for Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.

12. G.Goos, A. Heberle, W. Löwe, and W. Zimmermann. On modular definitions and implementations of programming languages using order-sorted partial abstract state machines. In *Proceedings of the ASM2000 Conference*, 2000.
13. M. Goossens, F. Mittelbach, and A. Samarin. *The LaTeX Companion*. Tools and Techniques for Computer Typesetting. Addison-Wesley, Reading, MA, USA, second edition, 1994.
14. F. Griffel. *Componentware*. dpunkt.verlag, 1998.
15. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
16. Y. Gurevich. May 1997 Draft of the ASM Guide. Department Technical Report CSE-TR-336-97, University of Michigan, 1997.
17. J. Huggins and W. Shen. The static and dynamic semantics of C. In *Proceedings of the ASM2000 Conference*, 2000.
18. J. K. Huggins and W. Shen. The static and dynamic semantics of C: Preliminary version. Technical Report Technical Report CPSC-1999-1, Computer Science Program, Kettering University, 1999.
19. P. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.
20. P. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.
21. J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, Mar. 1998.
22. J.-G. Schneider and O. Nierstrasz. Scripting: Higher-level programming for component-based systems. In *OOPSLA 1998*, 1998. Tutorial.
23. J.-G. Schneider and O. Nierstrasz. Components, scripts and glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures – Advances and Applications*, pages 13–25. Springer, 1999.
24. A. Sünbül. *Architectural Design of Evolutionary Software Systems*. PhD thesis, Technical University Berlin, 1999. in preparation.
25. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, N.Y., 1998.
26. J. Teich, P. Kutter, and R. Weper. Description and simulation of microprocessor instruction sets using asms. In *Proceedings of the ASM2000 Conference*, 2000.
27. L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly Associates, Inc., Sebastopol, CA, 1990.