A Component Concept for Abstract State Machines

Matthias Anlauff

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

Abstract. The Abstract State Machine (ASM) [8] approach has already proven to be suitable for large-scale specifications of realistic systems [5, 4, 9, 10]. 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*)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 [8]. 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,
- 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 [6], 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 , Section 4 sketches the support environment of XASM ; Section 5 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 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

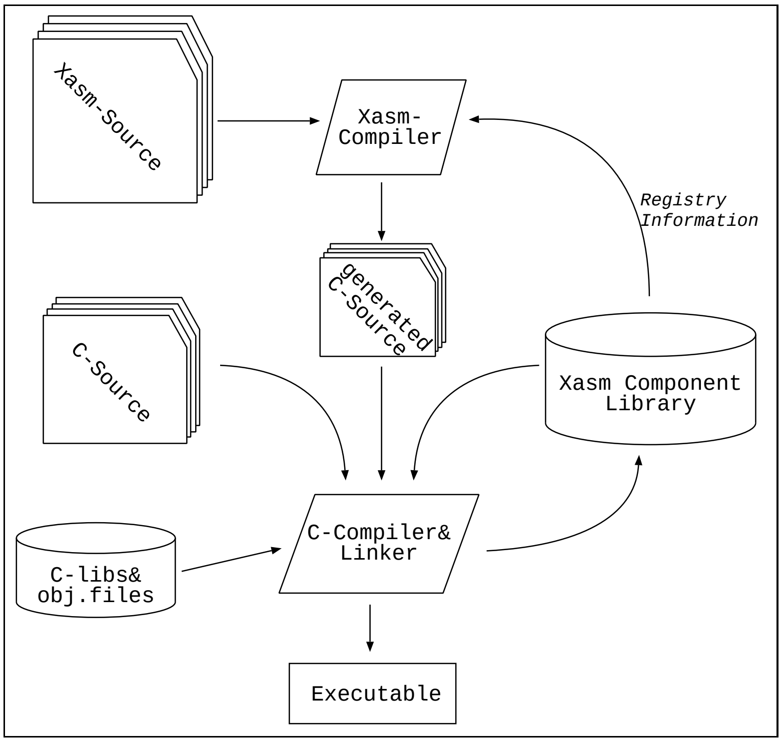


Fig. 1. Building XASM applications

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 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. [11, 13]).

asm $A(a_1 : T_1, \ldots, a_n : T_n) \rightarrow a_0 : T_0$
$\langle$asm meta information$\rangle$
is
$\langle$universe, function, and subasm declarations$\rangle$
$\langle$initialization rules$\rangle$

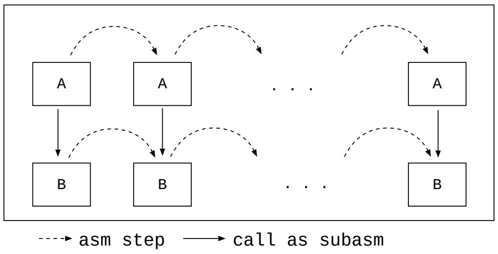
$\langle$asm rules$\rangle$
endasm

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

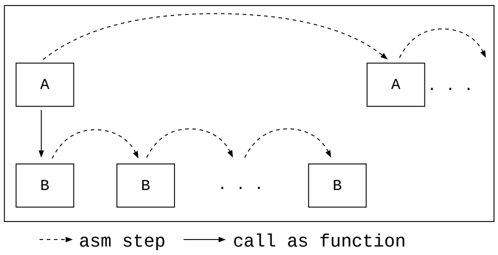
3.1 The Use Modes of "asm"-Constructs

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 sub-asm 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) → 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) → 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:[1]

```
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 this 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:

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

```
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.

4 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 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].

4.1 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 [3].

4.2 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 3 shows a screendump of a debugging session. Additionally, an integrated design environment, incorporating the graphical user interface is currently under development.

4.3 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 [7].

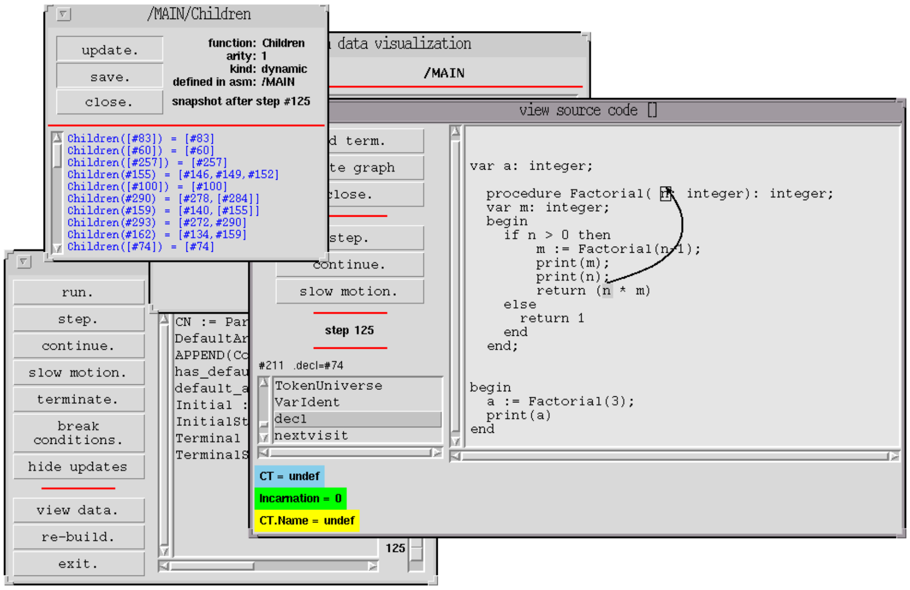


Fig. 3. Snapshot of a XASM-debugger session

5 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 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 [14] 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.

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 and A. Sünbül. An ASM specification of an elevator control system. 1999.
4. M. Anlauff and A. Sünbül. Software architecture based composition of components. In *GI-Workshop Sicherheit und Zuverlässigkeit software-basierter Systeme*, 1999.
5. E. Börger and W. Schulte. A Modular Design for the Java VM architecture. In E. Börger, editor, *Architecture Design and Validation Methods*. Springer, 1998.
6. 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.
7. 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.
8. M. Goossens, F. Mittelbach, and A. Samarin. *The LaTeX Companion*. Tools and Techniques for Computer Typesetting. Addison-Wesley, Reading, MA, USA, second edition, 1994.
9. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
10. 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.
11. P. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.
12. 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.
13. A. Sünbül. *Architectural Design of Evolutionary Software Systems*. PhD thesis, Technical University Berlin, 1999. in preparation.
14. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, N.Y., 1998.
15. J. Teich, P. Kutter, and R. Weper. Description and simulation of microprocessor instruction sets using asms. In *Proceedings of the ASM2000 Conference*, 2000.