Compiling Abstract State Machines to C++
(Extended Abstract)

Joachim Schmid
Siemens Corporate Technology, D-81730 Munich
`joachim.schmid@mchp.siemens.de`

Abstract State Machines (ASMs) [Gur95] have been widely used to specify software and hardware systems. Only a few of these specifications are executable, although there are several interpreters and some compilers for ASMs.

At Siemens Corporate Technology a part of the FALKO project [BPS00] was designed with ASMs. The specification for that part was written in ASM-SL [Cas99] which can be interpreted by the ASM Workbench [Cas00]. The Workbench was useful to debug the specification, but too slow for full test cases of FALKO. The question arose whether the code for the final product release had to be coded by hand or if it was possible to generate the C++ code from the specification automatically. For an automatic code generation, the following constraints had to be fulfilled by a compiler:

1. The specification was already written in ASM-SL and the compiler should use the same input for code generation. Otherwise the specification would have to be rewritten in a different syntax, maybe with a slightly different semantics.

2. The part designed with ASMs was one component in the FALKO project and not a standalone application. Thus, the generated code had to interact with other components of FALKO. The other components were written in C++ and therefore the compiler had to generate C++ code, too.

3. The generated code had to be fast enough for the product release. It should also be possible to debug the generated code, because otherwise it would be nearly impossible to locate errors in large runs.

There was no compiler which fulfilled these constraints. Therefore, we decided to build a new compiler and see whether the generated code could be used. And in fact, the generated code is used since two years and no bugs in the compilation scheme from ASM-SL to C++ have been detected up to now. This code generation is subject of the following paragraphs.

The task of the compiler is to translate an ASM specification written in ASM-SL to C++ code such that the semantics is preserved. This translation is complicated, because the two languages are at some points very different. We now briefly describe both languages to show the differences and explain how the translation works.

A specification in ASM-SL can be divided into a dynamic part and a static part. The dynamic part of the language is built up from ASM rules whose semantics is described in the Lipari Guide [Gur95]. Roughly speaking, a rule consists of guarded function updates which are executed in parallel. A function update assigns a new value for a given function symbol at the given arguments. For example, $f(x, y) := 5$ is a function update for the function symbol f with arity 2 at the arguments x and y.

The static part of the specification can be defined in terms of functions and the syntax used in ASM-SL is very similar to Standard ML, a functional programming language with abstract data types, pattern matching, overloading, let-expressions, garbage collection, etc. ASM-SL is a strongly typed language, but each expression may also evaluate to the special value `undef`, even if the expression is of basic type *integer*.

C++ is an object oriented programming language where all statements are executed sequentially. C++ supports classes, which group code (methods) and data (variables) to one structure. There are a lot of efficient libraries for C++, like the Standard Template Library (STL) which contains data structures for trees, sets, lists, ... and algorithms based on these data structures for sorting, iterating, etc. In C++, memory has to be allocated and deallocated manually, i.e. there is no garbage collection like in functional programming languages. There is also no pattern matching and variables can be introduced only at the level of statements. In a functional programming language new variables can be introduced by *let* in every expression.

Due to the differences of both languages, the compiler has to perform three transformations, namely parallel assignments to sequential assignments, functional expressions to imperative statements, and dynamic functions to data structures in C++. In the remaining paragraphs we describe these translations.

One of the main advantages of ASMs is the parallel execution of all rules where the updates are applied simultaneously. In C++, all statements are executed sequentially. There are several possibilities to simulate parallel assignments. For example, execute each rule, gather the updates and after execution of all rules, check consistency of the update set and apply it in case it is consistent. Such a solution is not very efficient if there are many updates. A better solution is to duplicate each variable. For a nullary dynamic function f in the ASM specification, we introduce variables f_{old} and f_{new} in the C++ program. When reading the function f, we take the variable f_{old}. On the other hand, updates are applied to f_{new}. Thus the sequential execution of the rules (methods) in C++ has the same result as if they would be executed in parallel. This works fine for one execution step, but for the next execution step the new value f_{new} has somehow to be moved to f_{old}. This could be done by looping through all variables, but it is also possible without moving anything, when f_{new} and f_{old} are implemented in a clever way.

The translation from functional expressions to imperative statements is more complicated. For example, consider the following functional case expression with pattern matching

```
case $(xs, ys)$ of
    $([a], [b, 5]) \to [a, b]$
    $([a, b], [c]) \to \ldots$
```

where the first case from top to bottom has to be selected which matches the case expression. We do not want to go into details, but let us point out some of the difficulties.

The example above is an expression and can occur everywhere where an expression of the corresponding type is accepted. The above case expression introduces new variables which have to be assigned according to the pattern. In C++, neither can variables be introduced in expressions, nor can they be assigned to a data structure. For example, $(a, b) = (3, 5)$ is not a valid assignment.

Another problem concerns garbage collection. In a functional language, data structures are created on the fly and are available as long as they are needed. For example, the expression $[a, b]$ creates a list of two elements. This list object will be removed by the garbage collector if the expression is no longer needed. C++ has no garbage collection and a program which allocates memory but does not deallocate it, would need a huge amount of memory.

The translation from dynamic functions to data structures in C++ is simple. Nullary dynamic functions are variables. Dynamic functions with arity 1 can be implemented by arrays, hash arrays, trees, etc. Functions with arity $n > 1$ can be implemented like functions with arity 1 where we pack the n arguments to one type. In functional languages this is called *currying*.

The described transformations are implemented in our compiler which was successfully used to automatically generate C++ code from the ASM specification for the FALKO project. The compiler itself is written in Haskell.

References

[BPS00] E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of ASMs in software design. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines*, number 1912 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[Cas99] Giuseppe Del Castillo. *ASM-SL, a Specification Language based on Gurevich's Abstract State Machines*, 1999.

[Cas00] Giuseppe Del Castillo. *The ASM-Workbench – A tool environment for computer aided analysis and validation of ASM models.* PhD thesis, University of Paderborn, 2000.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.