Journal of Universal Computer Science, vol. 7, no. 11 (2001), 1068-1087
submitted: 7/6/01, accepted: 1/10/01, appeared: 28/11/01 © Springer Pub. Co.

Compiling Abstract State Machines to *C++*

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

Abstract: Abstract State Machines (ASMs) 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. This paper introduces a compilation scheme to transform an ASM specification in the syntax of the ASM-Workbench into *C++*. In particular, we transform algebraic types, pattern matching, functional expressions, dynamic functions, and simultaneous updates to *C++* code. The main aim of this compilation scheme is to preserve the specification structure in the generated code without generating inefficient code. The implemented compiler was used successfully in the industrial FALKO application at Siemens Corporate Technology.

Keywords: Abstract State Machines, ASM, Compilation, *C++*

Category: D.1.1, D.3.3

1 Introduction

At Siemens Corporate Technology a part of the software produced in the FALKO project (a tool for railway simulation) [Börger et al. 2000] was developed with ASMs [Gurevich 1995]. The specification for this part was written in ASM-SL (a specification language for ASMs) [Del Castillo 1998] which can be interpreted by the ASM-Workbench (a tool to execute ASM-SL) [Del Castillo 2000]. 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 automatic code generation in FALKO the following constraints had to be fulfilled by a compiler:

1. The specification had been 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 is 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 had been written in *C++* and therefore the compiler had to generate *C++* code for seamless integration.

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.

At that time there was no compiler fulfilling these constraints and we decided to build a new compiler. This compiler was used to translate the given ASM specification into *C++*. The generated code is used successfully in the FALKO project [Börger et al. 2000] and the compilation scheme itself is implemented in the functional programming language Haskell [Thompson 1999].

In this paper we describe informally the basic concepts of the compilation scheme fulfilling the above listed constraints. The compilation scheme is applicable for all ASMs formalizable in ASM-SL. The aim is to present an overall solution for human readable compiled code and not a proposal for new compilation techniques. In fact some of the used techniques are almost well-known but not in this combination and not with the introduced optimizations.

In [Section 2] we briefly describe the specification language ASM-SL in order to give the necessary background for the succeeding sections. [Section 3] and [Section 4] introduce the compilation scheme for features related to the dynamic semantics and static semantics of ASMs. [Section 5] describes the FALKO application and the *Production Cell* case study [FZI 1998] where the compiler was applied. Finally, [Section 6] concludes this paper and discusses related work.

2 The Source Language

The source language for our compiler is ASM-SL [Del Castillo 1998]; a typed specification language for Abstract State Machines inspired by the functional programming language ML [Paulson 1996]. This specification language was designed for the ASM-Workbench [Del Castillo 2000] which is an interpreter and a debugger for the language. In ASM-SL the static and the dynamic semantics of an ASM can be defined using function definitions and transition rules. A function definition in ASM-SL is similar to a function definition in ML except that there are no higher-order functions, no lambda expressions, and no side-effects. A transition rule in ASM-SL corresponds to an ASM update rule.

Our compiler takes a specification in ASM-SL as input. More precisely we use the ASM-Workbench to generate a textual representation of a typed abstract syntax tree for a given specification. This typed abstract syntax tree and an additional configuration file containing compilation options is the input for the compiler.

We do not introduce the ASM-SL syntax in this paper; the syntax is defined completely in [Del Castillo 2000]. Moreover we describe some features of ASM-SL by an example which is used especially in [Section 4] to illustrate our compilation scheme for functional types and expressions.

```
freetype Nat == { zero, succ : Nat }

static function one == succ(zero)

dynamic function fib : Nat -> Nat
initially MAP_TO_FUN { zero -> zero, one -> one }

dynamic function n : Nat
initially zero

static function add : Nat * Nat -> Nat ==
fn(a,b) -> case b of
             zero    : a;
             succ(n) : succ(add(a,n))
           endcase

transition main ==
 block
   fib(succ(succ(n))) := add(fib(n),fib(succ(n)))
   n                  := succ(n)
 endblock
```

Figure 1: Computation of Fibonacci numbers in ASM-SL

We show the idea how the transformation works but we do not define formally the compilation scheme. In particular we consider simultaneous updates, dynamic functions, algebraic types, garbage collection, and pattern matching. The first two are related to the dynamic semantics of ASMs and the last three are well-known from functional programming (used for the static semantics).

The specification in [Fig. 1] shows an example for an incremental computation of Fibonacci numbers in ASM-SL. In the example we first define a new algebraic type `Nat` containing the type constructors `zero` and `succ`. We use this type to represent natural numbers (0 is represented by `zero` and if n is a natural number, then $n + 1$ is represented by `succ`(n)). Although there is an integer type in ASM-SL we use this inductive definition to illustrate pattern matching.

The definition below the *freetype declaration* introduces a static nullary function `one` which is an abbreviation for the natural number 1. The next two items declare two dynamic functions namely `fib` and `n` (dynamic functions are updatable functions). The dynamic function `fib` from type `Nat` to type `Nat` is initialized using a finite mapping ($\texttt{fib}(\texttt{zero}) = \texttt{zero}, \texttt{fib}(\texttt{one}) = \texttt{one}$) and the dynamic function `n` of type `Nat` is initialized to `zero`.

For two natural numbers the function `add` computes the sum of the function arguments `a` and `b` using a case expression with pattern matching; the function is defined recursively. The symbols `zero` and `succ` in the two cases are the type constructors of type `Nat` and the argument `n` of `succ` in the second case is a pattern variable and not the global dynamic function `n`.

The last definition defines a *transition rule*, i.e., the dynamic behavior in our example. This rule computes in each step the next Fibonacci number based on the previous two and increments the dynamic function `n`.

In the following two sections we will translate some ASM features and some functional features of ASM-SL into *C++* such that the specification structure (functions, variable names, updates, etc.) is preserved in the compiled code.

3 Dynamic Semantics

In this section we describe the compilation scheme for features in ASM-SL which are related to the dynamic semantics of ASMs. This includes simultaneous updates in contrast to sequential execution in *C++* and the treatment of dynamic functions.

3.1 Simultaneous Updates

One of the main advantages of ASMs is the parallel execution of rules. This means that all rules are executed with respect to the same global state. The execution of rules in a given state yields an update set. An update set consists of updates and an update is a location (dynamic function symbol with function arguments) together with a value (See [Del Castillo 2000] for a formal definition of update sets for ASM-SL.). In case the update set is consistent (there are no contradicting updates for a location) we apply the update set to the current state and obtain the next state. This is defined formally in [Gurevich 1995].

Consider the following two updates whose parallel execution swaps the values of a and b where we assume that a and b are defined as nullary dynamic functions:

$$a := b$$
$$b := a$$

In *C++* the same example does not swap the values, because the statements are executed sequentially and therefore the variables a and b would contain the value of b after executing both assignments. Thus, the semantics in *C++* would be different from the one in ASMs.

One solution to this problem is to implement the above described algorithm namely to collect the updates, check if they are consistent, and then apply them to the current state. For our example, the update set is $\{(a, \overline{b}), (b, \overline{a})\}$ where $\overline{a}$ and $\overline{b}$ denote the evaluated values of a and b respectively. If we apply this update set, then the values of a and b are swapped as expected.

This algorithm works fine and for instance the ASM-Workbench computes the next state in this way. On the other hand the algorithm is inefficient especially when there are many execution steps. The problem is that we first have to collect all updates in a corresponding data structure and after all rules are executed we must loop through these collected updates to apply them to the current state. Another problem with this algorithm is how to check consistency of the update set? Hence, this algorithm has several disadvantages and we propose another solution where we can execute the rules sequentially without collecting updates and where sequential execution is equivalent to parallel execution. We are now going to describe this algorithm based on double buffering—a technique well-known from applications where images have to be displayed.

In graphical applications an image is drawn on an invisible buffer and this buffer is made visible when the drawing process is finished. We adapt this technique and use for each location two buffers—one for reading and one for writing. However, we do not explicitly swap the two buffers when switching to the next state, because this would also lead to performance problems, because each dynamic function might consist of many locations and usually only some locations are updated in a step. Moreover, when reading a value we decide which buffer to take. We are now going to explain this in detail.

Double buffering. We uniquely tag each state with a natural number and assume that there are not too many execution steps. For the current state tag we use a global variable *cstate*. Below (paragraph rebasing) we describe what we are doing if there are too many execution steps, i.e. *cstate* overflows. For the time being let us assume that this is not the case.

For each location in the ASM specification we introduce in *C*++ two variables to store the value of the location. We call these variables *newVal* and *savedVal*. Additionally, for each location we use an integer variable called *stateno*. In this variable we store the state tag of the last assignment to the location; this tag will be used to determine which of both variables has to be used for reading.

We store the initial value of a location in the variable *newVal* and set *stateno* of the location to 0. Furthermore, the global state counter *cstate* is initialized with 1. We now distinguish between write access and read access of a location [see Fig. 2].

When writing a location we first check whether the location was already updated in the same state. This is the case if *cstate* is equal to *stateno* and then we have to check whether the value stored in *newVal* and the value which we want to write are equal. If they are not equal then we have an inconsistent update. Otherwise nothing has to be done, because the location (*newVal*) already contains the right value.

If *cstate* is different from *stateno* then the location was updated in a previous

```
write(cstate, val) {
  if (cstate == stateno)
    consistent(val, newVal);
  else {
    savedVal = newVal;
    newVal   = val;
    stateno  = cstate;
  }
}
```

```
read(cstate) {
  if (cstate == stateno)
    return savedVal;
  else
    return newVal;
}
```

Figure 2: Writing and reading of values

state. The value of that update is currently stored in *newVal* and we first copy it to *savedVal*. Then we copy the value of the current update to *newVal* and set *stateno* to *cstate*. Since we execute the rules sequentially there might be a rule which is executed after the current update and wants to read the location. Such a read access must not get the value of the current update and this is the reason why we first copied the value from *newVal* to *savedVal*.

The read access in [Fig. 2] is similar. We first check whether *cstate* is equal to *stateno*. If they are equal we know that *newVal* contains a value written in the same state and therefore we take the old value in *savedVal*. If *cstate* is different from *stateno*, then the location was written in a previous state and we have to take the value in *newVal*.

If we implement the write access and read access in this way, then we can execute all rules sequentially and afterwards it is sufficient to increment the global state counter *cstate*. Additionally the consistency of the updates is checked on the fly during the assignments.

Rebasing. The problem is how we can ensure that *cstate* does not overflow. Obviously we can not limit the number of execution steps otherwise we could not apply this compilation scheme for long running applications. However, in the definition of *read* and *write* we use only the equality function to compare *cstate* and *stateno*. And in fact it is sufficient to know whether *cstate* and *stateno* are equal and therefore, we can rebase the whole system where we reset the *stateno* variable for each location and the global state counter *cstate*. Rebasing works as follows:

1. We loop through all locations. If *stateno* of the location is equal to *cstate*, then we set *stateno* to 1 otherwise to 0.
2. We set the global state counter back to its initial value 1.

```
write(cstate,val) {
  if (cstate == stateno) {
    if (newValIsA)
      consistent(valA, val);
    else
      consistent(valB, val);
  }
  else {
    if (newValIsA)
      valB = val;
    else
      valA = val;
    newValIsA = not newValIsA;
    stateno   = cstate;
  }
}
```

```
read(cstate) {
  if (cstate == stateno) {
    if (newValIsA)
      return valB;
    else
      return valA;
  }
  else {
    if (newValIsA)
      return valA;
    else
      return valB;
  }
}
```

Figure 3: Optimized writing and reading of values

This rebasing implies that if *stateno* and *cstate* are equal before rebasing then they are equal after rebasing; analogously for unequal. Note that rebasing is necessary only when *cstate* reaches its maximal value.

The definition of *write* in [Fig. 2] has a disadvantage with respect to efficiency when copying a value from *newVal* to *savedVal* is an expensive operation. In our compilation scheme copying a value is done by copying a pointer which is described in [Section 4.1]. However we can improve the definition of *write* such that copying is not necessary and we are now going to explain this in detail.

Improving write access. The definitions for *write* and *read* ensure that always the newest value is stored in *newVal*. This is the reason why we have to copy a value from *newVal* to *savedVal*. To prevent copying we use for each location an additional boolean variable *newValIsA* to denote which of both variables contain the newest value. Since the newest value is no longer always stored in *newVal* we use the neutral names *valA* and *valB* instead of *newVal* and *savedVal*.

If the boolean variable *newValIsA* is true then the newest value is stored in *valA* and otherwise in *valB*. [Fig. 3] shows the modified *write* and *read* methods. The locations are initialized similar to the old solution in [Fig. 2], i.e. *valA* contains the initial value for the location and *newValIsA* is set to *true*. For the improved definitions in [Fig. 3] we can use the same rebasing algorithm. Writing is more efficient in this version. However, we have to pay a price in making reading a bit less efficient. So it depends on the context which of both solutions should be preferred.

We encapsulate the *write* and *read* access in a template class *AsmValue* where we also include rebasing of a location. We denote $AsmValue{<}T{>}$ as the instantiation of *AsmValue* for a location of type T. Hence, to obtain parallel update semantics in *C++* it is sufficient to use $AsmValue{<}T{>}$ instead of T.

3.2 Dynamic Functions

Dynamic functions in ASMs are functions which can be updated at run-time. Nullary dynamic functions are like variables in *C++* except that the assignments are executed in parallel. Here are some examples for function updates:

```
a          := b
f(g(a),a) := g(a)
g(a)       := f(g(a),b)
```

Nullary functions. We first consider nullary functions. Since they are similar to variables in *C++* we implement them as global variables with the simultaneous update technique of [Section 3.1]. Therefore, for each nullary dynamic function `f` of type `T` we define a global variable f of type $AsmValue{<}T{>}$. The translation of types will be discussed in [Section 4.1].

Unary functions. Unary functions can be implemented similarly. The Standard Template Library [Robson 2000] supports several container classes. For example, the *map* class is implemented as an AVL tree. There are methods for inserting, modifying, and deleting elements. Hence a unary function $f\colon A \to B$ can be implemented by using the template instance $map{<}A, AsmValue{<}B{>}\,{>}$, i.e., a mapping from type A to type $AsmValue{<}B{>}$. This implies that all updates to the function are executed according to the ASM semantics. This *map* instance can also be defined as a global variable. Note that we can use the lexicographical order on (evaluated) terms as a total order for the AVL tree.

N-ary functions. Dynamic functions with arity $n > 1$ can be implemented like unary functions, because we can put the n arguments together to one by tupling. We can define tuples in *C++* similarly to other functional types as will be introduced in [Section 4.1].

The suggested implementation for dynamic functions with arity greater than zero is suboptimal. Each time we need $O(n \cdot \log n)$ steps to add, delete, or modify an element if the map is implemented as an AVL tree containing n elements. The idea here is to reduce the function arity by one and to put the dynamic function into the argument type which is eliminated instead of using a global variable.

For example, if f is a dynamic function of type $A \to B$, then instead of defining a global *map* from A to $AsmValue{<}B{>}$ we put a variable f of type $AsmValue{<}B{>}$ into the type definition of class A. When accessing $f(a)$ we translate it to $a.f$. Similarly, if f is a dynamic function of type $A_1 \times \ldots \times A_n \to B$, then we define a *map* from $(A_2, \ldots, A_n)$ to $AsmValue{<}B{>}$ in class A_1 and translate $f(a_1, \ldots, a_n)$ into $a_1.f(a_2, \ldots, a_n)$. For the implementation it does not matter whether A_1 or another A_i is taken; this is the freedom of the compiler.

Expression identity. In ASM-SL (and in functional programming in general) two expressions are equal if they represent the same semantical value. This is not true for object instances (the correspondence for functional terms) in *C++* and leads to problems for dynamic functions defined as instance variables in type definitions as suggested above.

For instance, when we put the dynamic function `fib` into the class definition of type `Nat`, then $o_1.fib$ and $o_2.fib$ (corresponds to `fib`(o_1) and `fib`(o_2) in ASM-SL) might be different even if the instances o_1 and o_2 of type `Nat` represent the same natural number.

There are at least two possibilities to deal with this problem. In both cases for type `Nat` e.g., we need a variable *repr* of type set of `Nat` as a class variable (static variable in *C++*) in the class definition of `Nat`. In this set we store the representants for instances which have the same content.

For the first solution when creating a new instance of type `Nat` we look into the set *repr* to find the representant. In case there is no one we insert the current instance; otherwise we take the representant in the set and throw away the currently created instance. Therefore we always work with the representant and we use always the same instance for dynamic functions.

In the second version we look into the set *repr* only before we access a dynamic function in the instance.

It depends on the context which solution should be preferred. The first is better for many function accesses while the second is better when creating many instances.

4 Static Semantics

In this section we describe the compilation scheme for the features of ASM-SL which are related to functional programming. This includes the definition of *C++* classes according to type definitions in the ASM specification, the problem that *C++* supports no garbage collection, the transformation from pattern matching to imperative statements, and lifting of *let*-expressions since in *C++* no variables can be declared inside expressions. By using these transformations static function definitions in ASM-SL can be translated to methods in *C++*.

All these problems have already been solved (see [Jones 1987, Wilson 1992, Boehm 1993, Barnard 1994, Maranget 1994, Papaspyrou 1996], e.g.) since there has been extensive research in this area of compiling functional languages. Probably the most work has been done for the Glasgow Haskell Compiler [GHC 2001] which compiles Haskell-code to *C*-code. The functional part of ASM-SL can be viewed as a subset of Haskell, because there are no higher-order functions, no type classes, no constructor classes, no lambda expressions, and no lazy evaluation. However, in ASM-SL there is a special element `undef` which will be discussed later.

Although there has been quite a lot of work in this area we present in this section the mentioned transformations to illustrate that those techniques together with the techniques from the previous section are sufficient to generate reliable, human readable, efficient, and integratable code for a specification in ASM-SL.

4.1 Types

The specification language ASM-SL has several predefined types like *boolean*, *integer*, *float*, *string*, lists, sets, and tuples. Additionally, one can define new algebraic types with a *freetype* declaration. Consider the following declaration:

```
freetype Nat == { zero, succ : Nat }
```

As already explained the declaration defines a new type `Nat` where elements of `Nat` can be created using the constructors `zero` and `succ`. Both can be viewed as abstract functions generating elements of `Nat`:

$$zero : Nat$$
$$succ : Nat \rightarrow Nat$$

Instead of a fixed type as the argument for a constructor we can also use type variables. For instance, the declaration below defines a binary tree of any type `'a` where the type constructor `Node` takes two binary trees as arguments (see [Del Castillo 2000] for a detailed discussion about such polymorphic type definitions).

```
freetype BTree('a) == { Leaf : 'a,
                        Node : BTree('a) * BTree('a) }
```

Our task here is to compile such functional types into *C++*. Care has to be taken, because there is a distinguishable element `undef` polymorph in all types in ASM-SL. This value may be used like other ordinary values in computations and here ASM-SL differs from other functional languages. Despite of a uniform interface common to all types this is the main reason why we do not use the basic types of *C++* ($bool, int, \ldots$) to implement the basic ASM-SL types; for instance, the *C++* type *int* has no *undefined* element.

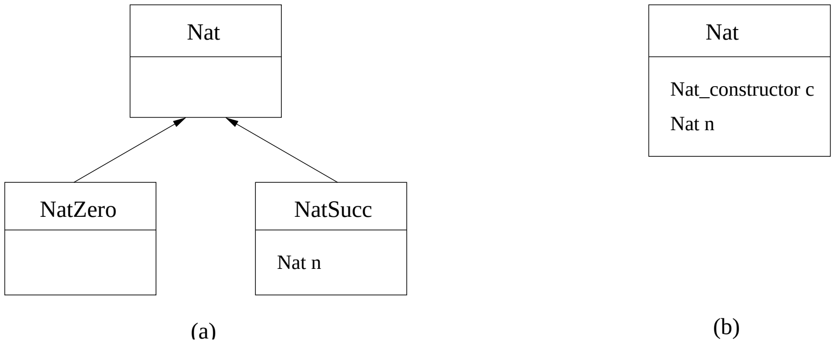

Figure 4: *C++* classes for type `Nat`

For the algebraic types there are at least two possibilities to define them in *C++*. An obvious solution is to define for each construction a separate class; each of them as a sub-class of the same base class for the corresponding type. Consider the above example for `Nat`. We can define the classes *Nat*, *NatZero*, and *NatSucc* as illustrated in [Fig. 4(a)] where *Nat* is the superclass of *NatZero* and *NatSucc*. If an expression is equal to `zero`, then it is an instance of *NatZero* otherwise an instance of *NatSucc*. The field n of type *Nat* in the definition of *NatSucc* contains the argument for the constructor `succ`. In the signature declaration for methods and functions we use *Nat*, but at run-time an object is either an instance of *NatZero* or *NatSucc*.

The other possibility is to include all type information in one class as shown in [Fig. 4(b)]. For the type `Nat` this implies that we define the class *Nat* containing two fields namely the constructor information and the possible argument for `succ`. The field c is the constructor information of type *Nat_constructor* and its value may be *zero* or *succ*. The field n is of type *Nat* and contains a value only if c is equal to *succ*.

Both solutions have advantages and disadvantages. For example, in (a) one has to define many classes and to deal with virtual methods. In (b) not all fields in an instance are always used. We prefer using alternative (b), because it is more efficient than (a) since there are no virtual methods and it is more suitable for the pattern matching we will introduce in [Section 4.3]. However, (a) is the *cleaner* solution.

For alternative (b) the class definition in *C++* for `Nat` looks like the definition in [Fig. 5] where we first define the enumeration type *Nat_constructor*, two constructor functions (the first for `zero` and the second for `succ`), and finally the two fields c and n as described above. The parameter *Nat_constructor* in the constructor signature is used, because there might be type constructors with the same argument signatures.

```
class Nat {
  typedef enum { zero, succ } Nat_constructor;
  Nat(Nat_constructor _c)          : c(_c)       { }
  Nat(Nat_constructor _c, Nat &_n) : c(_c), n(_n) { }
  ...
  Nat_constructor c;
  Nat             n;
}
```

Figure 5: Class definition for `Nat`

Remark. The template concept in *C++* can be used to implement polymorphic types of ASM-SL like the above `BTree` type.

4.2 Garbage Collection

In functional languages and also in ASM-SL heap cells are allocated on the fly while evaluating expressions; heap cells are used to store functional structures. A garbage collector analyzes all heap cells and frees those memory cells which are not used. In *C++* memory has usually to be allocated and deallocated by hand. To automate memory allocation and deallocation there is a well-known technique of *reference counting* [Wilson 1992] and *smart-pointers*. Reference counting is used to keep track of the number of elements referring to an instance. The smart-pointer technique automatically increments and decrements this number when copying and assigning object instances.

The definitions in [Fig. 6] shows a smart-pointer class where we first define a copy constructor and then an assignment operator. A pointer to the element is stored in the variable *elem* and this pointer value is usually shared by several smart-pointer instances.

Note that this class definition is not complete. The copy constructor is used in *C++* when creating a new instance as a copy of an existing one. A pointer to the element is stored in the variable *elem* and this pointer is usually shared by several smart-pointer instances.

Since we are using smart-pointers instead of elements directly we use the name *NatImp* for the class definition of `Nat` in the last section and define *Nat* as a subclass of the type *SmartPointer* for *NatImp*. We define it as a subclass instead of simply a type alias for $SmartPointer{<}NatImp{>}$, because we want to use the notation $Nat(zero)$ and $Nat(succ, n)$ for creating elements as in the previous section. Hence, in the class definition for *Nat* we include the definitions for the corresponding constructors. Additionally, we define *NatImp* as a subclass of *Reference* which supports reference counting. The classes *Reference*, *NatImp*, and *Nat* are also shown in [Fig. 6].

```
template <class T>
class SmartPointer {
  SmartPointer(const SmartPointer &x) {
    elem = x.elem;
    reference();
  }
  SmartPointer &operator=(const SmartPointer &x) {
    x.reference(); dereference();
    elem = x.elem;
    return *this;
  }
  void reference()   { if (elem) elem->reference();   }
  void dereference() { if (elem) elem->dereference(); }
  ...
  T *elem;
}
class Reference {
  void reference()   { counter++; }
  void dereference() {
    counter--;
    if (counter==0) delete this;
  }
  int counter = 0;
}
class NatImp : public Reference {
  typedef enum { zero, succ } Nat_constructor;
  NatImp(Nat_constructor _c)          : c(_c)        { }
  NatImp(Nat_constructor _c, Nat &_n) : c(_c), n(_n) { }
  ...
  Nat_constructor c;
  Nat             n;
}
class Nat : public SmartPointer<NatImp> {
  Nat(Nat_constructor _c) : Nat(new NatImp(_c)) { }
  Nat(Nat_constructor _c, Nat &_n) ...
  Nat(NatImp *n) ...
  ...
}
```

Figure 6: Type definition for `Nat` with smart-pointer

4.3 Pattern Matching

Pattern matching [Jones 1987] is one of the famous features of functional programming. A pattern is either a variable or a constructor with patterns as arguments. Pattern variables must not appear multiply in a pattern (the pattern must be linear). Consider the following example in ASM-SL which is similar to *set-comprehensions* in functional programming:

```
var succ(x) in xs
  f(x) := ...
  ...
endvar
```

We assume that `xs` is a set of elements of type `Nat` and `f` is a unary dynamic function. The above *for-all* rule takes all elements in `xs` which match the pattern `succ(x)` where `x` is a pattern variable matching anything. The rule inside `var` ... `endvar` is executed in parallel for each y in `xs` where `x` is bound such that $y =$ `succ(x)`.

Patterns can be replaced by predicates and selector functions. For the `Nat` type we could define *isZero*, *isSucc*, and *getSucc* and we could translate the above rule to something like the following *C*++ pseudo code:

```
set<Nat> xs;
Nat      y;
for (y in xs) {
  if isSucc(y) {
    Nat x = getSucc(y);
    f(x) = ...;
    ...
  }
}
```

We can imagine that this would work, but it makes compilation difficult and the compiled code would be hard to understand (especially for more complicated patterns). Hence, we try to compile patterns more intuitively.

Remember our smart-pointer instance which contains a pointer to its element. We use the null pointer to denote that this instance is not bound and can be matched to anything. We define a match operator as a modified assignment which returns *true* if the righthand-side (the term) can be assigned to the lefthand-side (the pattern) and where all variables in the lefthand-side are bound according to the righthand-side. If the matching is not possible, then the match operator returns *false*.

The matching algorithm works as follows. We have two expressions *lhs* and *rhs* of the same type. Both are smart-pointer instances (for the same element type). If a smart-pointer instance contains a null pointer, then we say it the instance is unbound. Otherwise it points to an element. Since the lefthand-side *lhs* corresponds to a pattern in ASM-SL it may contain an unbound instance. On the other hand the righthand-side *rhs* is a term and can not contain an unbound instance. If *lhs* is unbound, then the matching succeeds and we set the pointer in *lhs* to the pointer in *rhs*. Otherwise *lhs* and *rhs* point to elements. If the constructors in both elements are different, then the matching fails. Otherwise they have the same constructor and we match among the arguments. The matching succeeds if the matching of all arguments succeeds.

We now come back to our example and analyze what happens. We use the function $=_m$ for the matching operator:

```
set<Nat> xs;
Nat      y;
for (y in xs) {
  Nat x;
  if (Nat(succ,x) =_m y) {
    f(x) = ...;
    ...
  }
}
```

This has a nice appearance; in particular the syntactical structure of the pattern construct is preserved. Unfortunately, it does not work and the question is why? The problem is located in the constructor invocation $NatImp(succ, x)$ which we use for the constructor definition of $Nat(succ, x)$. The constructor is defined as follows (see the class definition in [Fig. 6]).

```
NatImp(Nat_constructor _c,  Nat &_n) : c(_c), n(_n) { }
```

The variable x and the formal parameter $\_n$ are of type *Nat* which is a smart-pointer class. When invoking the constructor, *succ* is copied to $\_c$ which is copied to c and the notation $n(\_n)$ implies that the field n (declared in class *NatImp*) is initialized using the copy constructor in [Fig. 6] to create a copy of $\_n$ which is an alias for x, because the parameter $\_n$ is passed by reference. Hence n becomes an unbound instance since $\_n$ (alias for x) is one. In the matching algorithm the pointer in n is modified, but not the pointer in x and therefore x is still unbound after the matching.

By analyzing this problem we can see how to fix it. We have to memorize that n is a copy of x and when n should be matched we delegate the matching to x. Therefore we use an additional field in class *Nat* which denotes the origin where the instance got its content. If the instance is not a copy of another instance, then we set this pointer to *null*. Additionally we adapt the matching algorithm.

If we match *rhs* against *lhs* and the origin pointer in *lhs* is different from *null*, then we match *rhs* against the instance denoted by the origin pointer in *lhs*. Now our implementation works fine.

Remark. The patterns for *let* and *case* expressions can be treated similarly. Additionally, our solution would also work for non-linear patterns.

4.4 Let Expressions

Up to now, we defined the compilation scheme for functional types and pattern matching. Functions and rules in ASM-SL can be compiled as methods in *C++*. However, there is a problem with local variables. In ASM-SL local variables can be introduced in expressions. This is not possible in *C++*, because a variable declaration is a statement and not allowed inside expressions. For example, consider the following function call for a function f with arity 2:

```
f(let x = 1 in x + 2 endlet, y)
```

This is a valid term in ASM-SL (assuming that `y` and `f` are defined properly), but can not be compiled as is into *C++*. Moreover, unfolding of *let*-expressions does not work since the expression on the lefthand-side may be a pattern:

```
f(let (x,y) = g(7) in x+y endlet,5)
```

On the other hand consider the following term where the *let*-expression is lifted outside the function call.

```
let (x,y) = g(7) in f(x+y,5) endlet
```

This term can be translated into *C++*, because we can first declare the variables x and y, use pattern matching to match $g(7)$ against (x, y), make the function call, and return the value:

```
X x;
Y y;
Tuple2(x,y) =_m g(7);
return f(x+y,5);
```

In our compilation scheme we use this technique of lifting *let*-expressions. Instead of defining the *lift*-algorithm for complete ASM-SL we define it for a small *lambda*-language [Barendregt 1981]. The extension to ASM-SL is straightforward. We use the following lambda language (only with first-order terms, because ASM-SL does not support higher-order functions):

$$\begin{aligned} \mathit{term} ::= &\ \texttt{let}\ \mathit{pattern} = \mathit{term}\ \texttt{in}\ \mathit{term} \\ &\mid \mathit{funid}(\mathit{term}) \\ &\mid \mathit{variable} \end{aligned}$$

The lifting for a term t is the expression $lift(t, id)$ where id is the term $\lambda x \rightarrow x$ denoting the identity function and *lift* is defined as follows (we assume that there are no name clashes while lifting variables):

$$\begin{aligned}
&lift(variable, f) &&= f(variable)\\
&lift(\texttt{let}\ pattern = t_1\ \texttt{in}\ t_2, f) &&= lift(t_1, \lambda t \rightarrow\ \texttt{let}\ pattern = t\ \texttt{in}\ lift(t_2, f))\\
&lift(funid(t), f) &&= f(lift(t, \lambda t' \rightarrow funid(t')))
\end{aligned}$$

This algorithm preserves the semantics of an expression and transforms it to an expression of the form $\texttt{let}\ p_1 = t_1\ \texttt{in}\ \texttt{let}\ p_2 = t_2\ \texttt{in}\ \ldots\ \texttt{let}\ p_n = t_n\ \texttt{in}\ t$ such that $t_1, \ldots, t_n, t$ contain no *let*-expressions.

Before we translate an ASM-SL expression to *C++* we transform it using the *lift*-algorithm to obtain the above special form. The *let*-expressions are then compiled as variable declaration statements together with pattern matching and the resulting term t is compiled to the statement `return t;`.

5 Applications

In this section we briefly describe the FALKO application and the production cell case study. The production cell is an academic case study which we used to test the compiler and FALKO is the reason why we developed this compiler.

5.1 FALKO

FALKO [Börger et al. 2000] is a software system for railway simulation. The software consists of three components namely the train supervision, interlocking system, and the process simulator. The first two components are manually encoded in *C++* and the process simulator is designed using ASM-SL.

The formal specification in ASM-SL is part of an HTML documentation and we implemented a tool to extract these formal parts from the documentation. Additionally, the tool can modify the HTML files in order to pretty print the formals parts (keywords in bold-face, generating index files, automatically inserting suitable hyperlinks, ...). Since this is an industrial project the HTML documentation (including the specification) is not public.

The specification is detailed enough such that it can be executed using the ASM-Workbench with an additional oracle for the external functions in the specification.

The ASM-Workbench was used to debug the specification for small test scenarios. We compiled this specification into *C++* using the introduced compilation scheme and the generated code is used successfully since January 1999 in the final product release. Until now (March 2001) no bugs in the compilation scheme have been discovered and only two *specification bugs* occurred. When the specification bugs were discovered the team which implemented the other components fixed the bugs directly in the generated code, because they were not familiar with the ASM specification and the provided tool environment. They also introduced a new feature in the compiled code.

This illustrates that the generated code is readable enough so that people not familiar with the compilation scheme can fix and extend the produced code. Meanwhile, the bugs have been fixed in the specification, the new feature was introduced, and the specification was recompiled into *C++* to prevent inconsistencies between the specification and the code.

For more information about the FALKO specification and the generated code like size and effort we refer the reader to [Börger et al. 2000].

5.2 Production Cell

In early stages, we tried our compiler on the well-known production cell case study [FZI 1998]. We took the ASM-specification in [Börger and Mearelli 1997], extracted the function and rule definitions, and translated them into ASM-SL. An HTML version of this specification (the input for our compiler) is available [Schmid 1999b]. However, this version does not contain the describing text from the ASM specification in [Börger and Mearelli 1997].

The FZI in Karlsruhe provides a graphical visualization for the case study. We compiled the specification—using our compiler—into *C++* and implemented the interface between the compiled code and the graphic visualization. The resulting code [Schmid 1999b] successfully controls the simulator.

6 Conclusion and Related Work

We presented a transformation scheme for the compilation of Abstract State Machines written in ASM-SL into efficient *C++* code which was applied in an industrial middle sized project. We showed how to compile functional language features as well as ASM features. Except for *let*-expressions our compilation scheme preserves the structure of the specification in the compiled code. The most important part is the translation of simultaneous updates into sequential statements. The introduced technique ensures that the sequential execution of rules is semantically equivalent to their parallel execution. However, correctness of the compiled code is a quite complicated issue. For instance, there must be a formal specification of *C++*. It must be proven that our compiler is correct implying that the Haskell compiler must compile correctly our compiler, etc. We refer the reader to [Goerigk and Langmaack 2001] for a discussion about such topics. See also [Stärk et al. 2001] for a proven to be correct compilation scheme of Java programs into Java Virtual Machine bytecode, which we have implemented in AsmGofer [Schmid 1999a].

The XASM tool [Anlauff 2000] compiles ASMs in the XASM syntax into C-code. More precisely, the compiler generates an abstract machine and that code is executed by an interpreter. The XASM language is untyped and static functions have to be defined in C.

In [Mearelli 1997] Mearelli manually translated the ASM specification of the production cell case study [Börger and Mearelli 1997] to *C++* code. His translation is not related to our compilation and works only for special examples where parallel execution of rules is not necessary.

Extensive research has been done on garbage collection in *C++* [Wilson 1992, Boehm 1993, Smith and Morrisett 1997]. Reference counting is the simplest solution. For our FALKO project we compared the compiled specification with reference counting with an implementation for automatic garbage collection [Boehm 2001]. The performance was nearly the same. However, the generated code using automatic garbage collection needs a lot more memory than the code using reference counting.

There are more efficient pattern matching algorithms than our implementation ([Augustsson 1985, Jones 1987, Maranget 1994], e.g.). However, we preferred readability of the compiled code; performance for pattern matching was not an important issue for our examples.

The main advantage when using ASMs and compiling them into *C++* instead of using directly *C++* is the possibility to specify on a very high level of abstraction enabling the customer to understand the specification. Furthermore, it is possible the generate more efficient code by improving the compilation scheme without changing the specification; support for other target languages (Java, e.g.) is straightforward.

Acknowledgments. We thank Egon Börger and Peter Päppinghaus for many comments on this paper.

References

[Anlauff 2000] Anlauff, M. (2000), "XASM – An extensible, component-based Abstract State Machines language", In [Gurevich et al. 2000], pages 69–90.

[Augustsson 1985] Augustsson, L. (1985), "Compiling pattern matching", In *Conference on Functional Programming Languages and Computer Architecture*, pages 368–381.

[Barendregt 1981] Barendregt, H. (1981), *The Lambda Calculus. Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*, North-Holland.

[Barnard 1994] Barnard, D. T. (1994), "A simple functional language compiler", Technical Report 94-371, Department of Computing and Information Science.

[Boehm 1993] Boehm, H. J. (1993), "Space efficient conservative garbage collection", *ASM SIGPLAN Notices*, 28:197–206.

[Boehm 2001] Boehm, H. J. (2001), "A garbage collector for *C* and *C++*", Web pages at `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[Börger and Mearelli 1997] Börger, E. and Mearelli, L. (1997), "Integrating ASMs into the software development life cycle", *Journal of Universal Computer Science*, 3(5):603–665.

[Börger et al. 2000] Börger, E., Päppinghaus, P., and Schmid, J. (2000), "Report on a practical application of ASMs in software design", In [Gurevich et al. 2000], pages 361–366, Online available at `http://www.tydo.de/files/papers/`.

[Del Castillo 1998] Del Castillo, G. (1998), "The ASM Workbench. an open and extensible tool environment for Abstract State Machines", In *Proceedings of the* 28^{th} *Annual Conference of the German Society of Computer Science*, Technical Report, Magdeburg University.

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

[FZI 1998] FZI (1998), "Case study Production Cell", Web pages at `http://www.fzi.de/divisions/prost/projects/production_cell/ProductionCell.html`.

[GHC 2001] GHC (2001), "Glasgow Haskell Compiler", Web pages at `http://www.haskell.org/ghc/`.

[Goerigk and Langmaack 2001] Goerigk, W. and Langmaack, H. (2001), "Will informatics be able to justify the construction of large computer based systems?", Technical Report 2015, Christian-Albrechts-Universität Kiel.

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

[Gurevich et al. 2000] Gurevich, Y., Odersky, M., and Thiele, L., editors (2000), *Abstract State Machines, ASM 2000*, number 1912 in Lecture Notes in Computer Science, Springer-Verlag.

[Jones 1987] Jones, S. P. (1987), *The Implementation of Functional Programming Languages*, Prentice Hall.

[Maranget 1994] Maranget, L. (1994), "Two techniques for compiling lazy pattern matching", Technical Report 2385, INRIA.

[Mearelli 1997] Mearelli, L. (1997), "Refining an ASM specification of the Production Cell to *C++* code", *Journal of Universal Computer Science*, 3(5):666–688.

[Papaspyrou 1996] Papaspyrou, N. S. (1996), "A framework for programming denotational semantics in *C++*", *ACM SIGPLAN Notices*.

[Paulson 1996] Paulson, L. C. (1996), *ML for the Working Programmer*, Cambridge University Press.

[Robson 2000] Robson, R. (2000), *Using the STL. The C++ Standard Template Library*, Springer-Verlag.

[Schmid 1999a] Schmid, J. (1999), "Executing ASM specifications with AsmGofer", Web pages at `http://www.tydo.de/AsmGofer/`.

[Schmid 1999b] Schmid, J. (1999), "Production cell", Web pages at `http://www.tydo.de/ProductionCell/`.

[Smith and Morrisett 1997] Smith, F. and Morrisett, G. (1997), "Mostly-copying collection: A viable alternative to conservative mark-sweep", Technical Report TR97-1644, Computer Science Department Cornell University.

[Stärk et al. 2001] Stärk, R. F., Schmid, J., and Börger, E. (2001), *Java and the Java Virtual Machine. Definition, Verification, Validation*, Springer-Verlag, See web pages at `http://www.inf.ethz.ch/~jbook/`.

[Thompson 1999] Thompson, S. (1999), *Haskell. The Craft of Functional Programming*, Addison-Wesley, second edition.

[Wilson 1992] Wilson, P. R. (1992), "Uniprocessor garbage collection techniques", In Bekkers, Y. and Cohen, J., editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Springer-Verlag.