An Executable Specification of C♯

Horatiu V. Jula, Nicu G. Fruja

Computer Science Department, ETH Zürich, CH–8092 Zürich, Switzerland
{julah,fruja}@inf.ethz.ch

Abstract The Abstract State Machines (ASMs) theory has been applied to rigorously formalize the semantics of the C♯ programming language. The developed ASM model is validated here by providing an executable model which supports the experimentation with the ASM model. The executable model is a refinement of the ground model and is implemented in AsmL – the Abstract State Machine Language, which is embedded into Microsoft Visual Studio .NET. We report here on the development of AsmL Executable Specifications of the C♯ semantics by depicting our techniques on building and using AsmL as a Specification Language.

1 Introduction

In this paper we present our implementation of the Abstract State Machine (ASM) model, that has been defined in [4] to formalize the C♯ static and dynamic semantics. We show all the techniques we have used to get an efficient execution.

This work is part of a larger project which aims to establish some outstanding properties of C♯ and CLR virtual machine by mathematical proofs, such as the C♯ type safety and the correctness of the CLR bytecode verifier. As part of this effort, an ASM model in form of an abstract interpreter has been developed in [4] for the *full* C♯ , including a thread and memory model [11]. The implementation of the ASM model for C♯ comes as a method to check the faithfulness with respect to C♯ of the modeling decisions we had to take where the C♯ Language Specification exhibits gaps (see [12] for details). By providing an executable version of the C♯ model, we can test the internal correctness of the model and its conformance to the experiments we run in C♯.

We did our implementation in AsmL [1,2]. AsmL is a specification language based on Gurevich's ASMs [14]. An interested reader can find the ASM semantics of the core of AsmL in [3].

Specifications written in AsmL can be run as programs and are also called executable specifications. AsmL supports mathematical set operations – such as comprehension and quantification – that proved themselves to be very useful for our implementation. Unlike most popular programming languages, AsmL is non-deterministic and this turned out to be in our favor. The expressiveness is another important aspect of AsmL: the implementation of the abstract model rules are very close to their abstract form.

In order to interpret the C♯ programs, our application needs to store the source code programs in some abstract structures. The annotated abstract syntax trees (ASTs) are a feasible solution for this purpose. To build these trees, we use a parser generator called *Coco* [6]. The application checks then if the nodes of the AST are type consistent. Before the interpreter can run the input program, it has to be initialized. For this, certain actions are performed such as syntactical transformations described in [4], computation of sets of class members, finding the entrypoint of the program etc. Of particular interest is the C♯ threads execution, that we implemented following the lines of the ASM model in [11]. AsmL proved itself to be a good solution for implementing the two execution policies: serial and parallel mode.

The paper is organized as follows. In Section 2 we give a brief overview of the ASM model for C♯ and we introduce the AsmL programming language. The Section 3 explains the implementation details. The Section 4 presents the related work. Section 5 draws some conclusions, as well as future work directions.

2 Method

2.1 The ASM model for the C♯ semantics

In [4], the ASM method has been used to build up a model that formalizes the semantics of the C♯ language. The ASM model reflects the decisions underlying the C♯ implementation as defined by the C♯ standard [7]. It comes as an abstract interpreter that gives a way to reflect the run-time features encountered upon executing a C♯ program. C♯ is layered into modules of orthogonal language features which yields a sequence of sublanguages $C\#_{\mathcal{I}} \subset C\#_{\mathcal{C}} \subset C\#_{\mathcal{O}} \subset C\#_{\mathcal{E}} \subset C\#_{\mathcal{D}} \subset C\#_{\mathcal{T}} \subset C\#_{\mathcal{U}}$ (see [4] for details). For each sublanguage, there is defined a submachine which is a conservative extension of its predecessor. The whole ASM model is then a (parallel) composition of all the submachines.

One of the main benefits of having the interpreter in a form of an ASM is that one can specify the static and dynamic parts of the semantics separately. The model captures the dynamics by ASM rules describing their run-time effect on the abstract program state, guided by a walk through the corresponding AST.

2.2 AsmL

The AsmL language has a very high-level of abstraction. It is a very powerful language, which can be used to model reliable systems; the AsmL specifications are, of course, executable, therefore the model can be tested immediately by simply executing it.

We used in our approach some powerful AsmL features like: disjunctive types, structure cases, high-level constructions for defining sequences, sets or maps, quantifying expressions such as **forall-holds**, **exists** and statements like **match**, **forall**, **choose**. We had a lot of benefits by using these features: the manipulation of the data structures (e.g. maps, sets, sequences, disjunctive types) became very easy, because these features are integrated in the language syntax, not just packed into a set of libraries. Therefore, the code which uses these powerful AsmL constructions is a lot more readable and shorter than its equivalent from all of the common programming languages like C++ or Java. The AsmL code is also more robust, due to the constraint that the statements within a step must be consistent (without conflicts) because they run in parallel.

3 Approach

This section illustrates the details of the implementation. We depict in Section 3.1 our method for building an AST for a C♯ program. Section 3.2 explains the algorithm we used to perform type inference and checking for the above derived AST. We discuss in detail in Section 3.3 all the actions that are performed for initializing the interpreter. Section 3.4 gives a few examples of execution rules implementing their formalization in [4]. We provide details in Section 3.5 on how we made possible the integration of our interpreter with Visual Studio .NET [15]. In Section 3.6 we describe what are the capabilities of the interpreter. In Section 3.7 we describe some enhancements and refinements of the ground model [4] and Section 3.8 presents a few minor problems we have encountered in the AsmL reference manual [4].

The AsmL implementation of the model [4] simulates the behavior of the C♯ programs by interpreting their code as follows: first, one parses the input program and the predefined classes (e.g. `object`, `Type`, `Exception`, `Console`) from the library file `System.cs` and one generates the annotated AST, then a type checking for the program's expressions is performed and finally, if the type constraints are satisfied, the abstract state machine is initialized and the program is executed *step by step* by a recursive evaluation of its AST.

3.1 AST generation for the input program

The AST is built using a parser generated by the *Coco Parser Generator* [6]. A specification file (*CSharp.atg*), which specifies the C♯ tokens, the C♯ grammar and the actions for each grammar production, must be provided for the parser generator. Then, a parser for C♯ programs (*Parser.cs*) is generated from the specification file.

The AST nodes have the type *Pos*. The class *Pos* is used to encapsulate all the information needed by an AST node, namely: the *label*, which describes the kind of a node, the *type* of the node which is relevant only if the node represents an expression node and the pointers to the *parent*, to the *first* son and to the *next* brother.

The AsmL definitions for an AST node and its label are shown in Fig. 1: Two important methods – *setFirst* and *setNext* – are listed in Fig. 2. The *setFirst* and the *setNext* sets the first child and the next brother of a node, respectively.

We define the notions of *pattern* and *context* of a node. The pattern is actually the description of a node. It is a n-tuple which contains the label of the node

Fig. 1. The class *Pos*

```
class Pos
  var label as Label
  var posType as RetType = Undefined()
  var up as Pos = null
  var first as Pos = null
  var next as Pos = null

type Label = String or
    (String, String) or
    (String, String, String) or
    (String, String, String, String)
```

Fig. 2. The methods *setFirst* and *setNext*

```
class Pos
  [EntryPoint]
  setFirst(p as Pos?)
    step
      first := p
      if p ne null then
        p.up := me
    step skip
```

```
class Pos
  [EntryPoint]
  setNext(p as Pos?)
    step
      next := p
      if p ne null then
        p.up := me.up
    step skip
```

and the references to the children. The pattern encapsulates all the information *provided* by an AST node, without any concern if the node is evaluated or not. The context is quite similar to the pattern, but if a child node is evaluated, instead of the child reference we have the value of the child, i.e. the value returned after the evaluation of the child. This value has the type *Result*.

3.2 Type checking

Type inference is performed in a single step for all the AST nodes, through a recursive manner. To a type-consistent statement node we assign the type `void`. An AST node is *type-consistent* if the type inference algorithm does not return the *Undefined* type for the node, i.e. it can infer a valid type for that node.

The definitions underlying the type checking algorithm are the following:
- an expression is type-consistent if its type can be found through the type inference algorithm;
- a statement is type-consistent if all its children (expressions or statements) are type-consistent;
- a method is type-consistent if its body is type-consistent and its return type and arguments' types are not *Undefined*;
- a class/struct field is type-consistent if its type is not *Undefined*;
- a program unit (i.e. a class or struct) is type-consistent if all its members (fields and methods) are type-consistent;
- a program is type-consistent if all its units are type-consistent;

Fig. 3. The method *getType*

```
class Pos
  getType(m as Method) as RetType
    match pattern()
      // exp1 bop exp2
      (("bop", bop as String), exp1 as Pos, exp2 as Pos) :
        match (exp1.getType(m), exp2.getType(m))
          (t1 as ValidType, t2 as ValidType) :
            ...
            elseif bop in {"*", "/"} then
              if t1 is NumericType and t2 is NumericType then
                let t = sup(t1,t2)
                me.posType := t
                return t
              else return Undefined()
          otherwise return Undefined()
      // exp1 = exp2
      (("aop", "="), exp1 as Pos, exp2 as Pos) :
        match (exp1.getType(m), exp2.getType(m))
          (t1 as ValidType, t2 as ValidType) :
            me.posType := t1
            return t1
          otherwise return Undefined()
      ...
      // a statement
      otherwise
        // checks in parallel if all the children are type-consistent
        let ok = forall c in getChildren() holds
                    c.getType(m) ne Undefined()
        if ok then
          me.posType := void()
          return void()
        else return Undefined()
```

Some self-explanatory examples of type inference are illustrated in Fig. 3 in AsmL syntax. One can see in Fig. 3 how the method *getType* infers the type of an AST node representing the binary operations / and $*$, an assignment operation and a statement, respectively. For instance, in the case of division and multiplication, the result type is computed recursively as the supremum of the operands' types, which must be numerical types (see [4] for details). If one of them is not a numerical type, then the result type is set to *Undefined*.

Fig. 4 shows how we check in parallel the type consistency for all the method bodies from all the units of a program.

Fig. 4. The type consistency check

```
step
  let typesOk =
    forall u in prog.units holds
      forall m in u.members where m is Method holds
        (m as Method).body.getType(m as Method) ne Undefined()
step
```

3.3 Machine initialization

If the program is type-consistent, the machine is initialized, i.e. the global variables are initialized. When the machine starts, the application performs the following actions:

- Finds all the inherited members for each class C and adds them to class C's definition. If a member from a base class already exists in class C, then it is not incorporated anymore into class C's definition. So, every class includes in its definition a set of distinct inherited members. As the process of gathering inherited members starts with the *first superclass*, the programs have always a polymorphic behavior, based on the calling of the *most specialized method*. More explicitly, every method from class C has the newest version.
- Adds the `this` parameter to all non-static methods. It is inserted at the first position in the method parameters' list.
- Adds the base constructor call at the beginning of every non-static constructor's body – in accordance with [7].
- Performs some syntactic transformations (see [4,7] for details) like the following:

`++x`	⇝ `x = x + 1`	*pre-increment expression*
`typeof(t)`	⇝ `new Type(t)`	*typeof expression*
`base`	⇝ `(`*BaseClass*`)this`	*base-access expression*
delegateObj`(`*args*`)`	⇝ *delegateObj*`.Invoke(`*args*`)`	*delegate invocation*
`o->f`	⇝ `(*o).f`	*pointer-field access*
`o->m(`*args*`)`	⇝ `(*o).m(`*args*`)`	*pointer-method access*

- Determines the set of local variables for each method.
- Type checking.
- If we meet some expressions like $pexp[i]$, where $pexp$ is an expression of a pointer type (e.g. `int`*, `void`*), we replace $pexp[i]$ with $*(pexp + i)$ as suggested in [4].
- Checks if all the unsafe expressions/statements are enclosed by blocks declared as unsafe – as constrained by [4] (i.e. `unsafe` $\{\ldots$ *unsafeStm* $\ldots\}$).
- If the type constraints and the unsafe code constraints are satisfied, the application finds the program's entry-point (the `Main` method).
- If the entry-point is found, one creates a delegate with the entry-point added to its invocation list. Then, the main-thread of the program is created and is associated with the entry-point's delegate. Shortly, the program's main-thread will execute the entry-point.
- Finally, the program's main-thread is started.

3.4 Program execution

The execution begins by executing the body of the `Main` method. We use the variable *pos* to reference the current position in the AST. Each AST node is evaluated recursively through the following algorithm:

– First, the context of the node is analyzed.
– If the context has unevaluated elements, one recursively evaluates these elements by moving *pos* to each of them, in a predefined order, according to the definition of the corresponding execution rules from the ASM model [4].
– If the context has no unevaluated elements, one computes the value of the node and keeps it in the *values* global variable, which maps the AST node references to the value returned after their evaluation.

A program can have multiple threads and therefore it is maintained a thread queue. Each thread has its own context, which contains: the *stack of call frames*, the *current method*, the reference to the *current position* in the current method's AST, the *values* of the current method's AST nodes (the AST of a method is the AST of its body), the *stack of the still-to-be-handled exceptions* and the *trace* of each exception object from the exception stack. A frame is associated with a method call and consists of: the *method signature*, the reference to the *current position* in the method's AST, the *addresses* where the local variables and the formal parameters are allocated in the memory and the *values* of the AST nodes. An exception object's trace is the chain of methods this object is propagated through, after it is thrown for the first time until it is caught. We have two thread execution policies: a serial execution and a parallel execution. In the case of the serial execution, at each step we choose an active thread and we evaluate the current position *pos* in the current method's AST. In the case of the parallel execution, at each step we evaluate in parallel the current positions in the current methods' ASTs from all the active threads. We will not insist anymore on the execution mechanism, because it is described in the execution rules of the ASM model [4].

Fig. 5 shows both the abstract rules of the ASM model [4] for the execution of an `if` statement and their implementation in AsmL. Similarly, Fig. 6 shows the corresponding evaluation rules for the postfix operator `++` and their implementation.

3.5 Integration with Visual Studio .NET

The integration with Visual Studio .Net is very simple. First, the AsmL plugin must be installed. Then, the AsmL specification can be written in an AsmL source file. Before the project compilation, the AsmL code is translated into C♯ code by the AsmL plugin. AsmL classes and their methods and also the global functions (translated as methods of a class named `_GLOBAL_`) can be used (imported) in the C♯ code. We have defined interface methods in the AsmL specification, i.e. methods which are called from the C♯ code. Usually, we used

Fig. 5. The `if` statement

$$
\begin{array}{ll}
\texttt{if (}exp\texttt{)}\ stm_1\ \texttt{else}\ stm_2 & \rightarrow pos := exp \\
\texttt{if (}^{\blacktriangleright} val\texttt{)}\ stm_1\ \texttt{else}\ stm_2 & \rightarrow \textbf{if}\ val\ \textbf{then}\ pos := stm_1\ \textbf{else}\ pos := stm_2 \\
\texttt{if (}True\texttt{)}\ ^{\blacktriangleright} Norm\ \texttt{else}\ stm & \rightarrow \textsc{YieldUp}(Norm) \\
\texttt{if (}False\texttt{)}\ stm\ \texttt{else}\ ^{\blacktriangleright} Norm & \rightarrow \textsc{YieldUp}(Norm)
\end{array}
$$

```
("if_else", exp as Pos, Pos, Pos) : pos := exp
("if_else", val as Boolean, stm1 as Pos, stm2 as Pos) :
    if val then pos := stm1 else pos := stm2
("if_else", true, Norm, Pos) : yieldUp(Norm())
("if_else", false, Pos, Norm) : yieldUp(Norm())
```

Fig. 6. The postfix `++` operator

$$
\begin{array}{l}
vexp\ \texttt{++} \rightarrow pos := vexp \\
^{\blacktriangleright} adr\ \texttt{++} \rightarrow \textbf{let}\ old = memValue(adr, type(pos))\ \textbf{in} \\
\qquad\qquad \textbf{if}\ \neg UopException(\texttt{++}, old)\ \textbf{then} \\
\qquad\qquad\quad \textsc{WriteMem}(adr, type(up(pos)), Apply(\texttt{++}, old)) \\
\qquad\qquad\quad \textsc{YieldUp}(old)
\end{array}
$$

```
("++", exp as Pos) : pos := exp
("++", adr as Adr) :
      let val = memValue(adr, pos.posType)
      let (result, _) = apply("++", val)
      writeMem(adr, pos.up.posType, result)
      yieldUp(val)
```

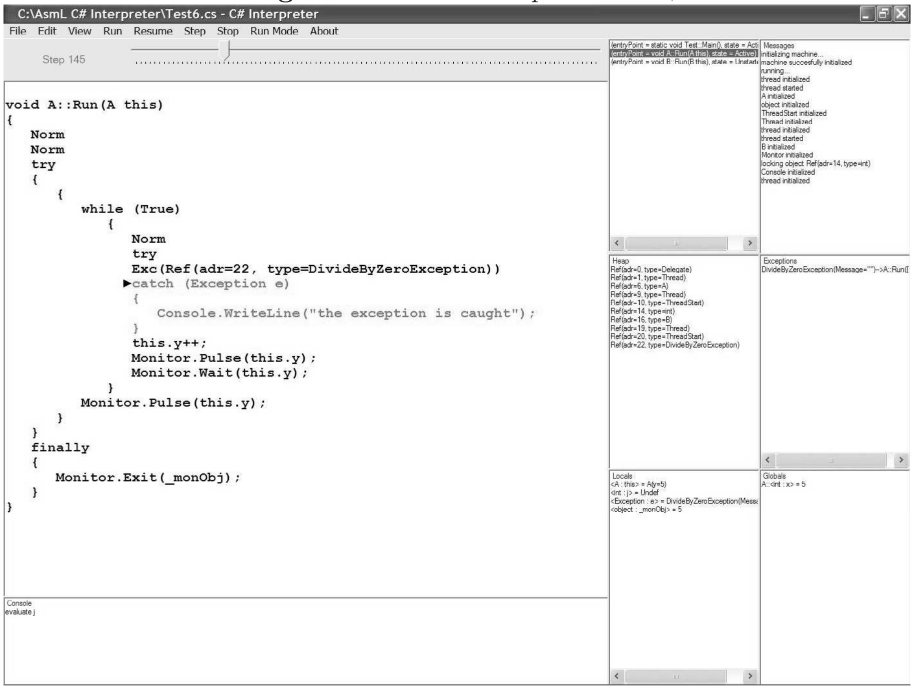
these methods for printing in the graphical interface some parameters of the program state (e.g. the thread queue, the frame stack, the global variables) or outputs like console or error messages.

3.6 The application's capabilities

Fig. 7 shows a screenshot of the application. At each step the application prints the following outputs:

- *the thread queue*, with information about each thread, namely the name, the state and the associated method
- *the current configuration of the current method's AST*; the currently executing expression/statement node is colored in pink and the evaluated nodes are colored in blue
- *the frame stack*, with information about each frame, namely the method signature and the arguments' values
- *the static fields*, with information about each field, namely the type, the name and the field's value

Fig. 7. The AsmL interpreter for C♯



– *the local variables*, with information about each variable, namely the type, the name and the variable's value
– *the heap references*
– *the exception stack*
– *the exception stack trace*
– *the error/debug messages*
– *the console messages*

We can pause/resume/stop the execution or change the thread execution policy (switch to the serial or the parallel mode) at any point during the program execution. In the serial mode we cannot choose which thread we want to visualize; at each step, we will see only the current thread. But in the parallel mode we can choose the thread we want to visualize.

The thread independent outputs are: the thread queue, the static fields, the heap references, the error/debug messages and the console messages. The others are thread dependent, because they belong to a specific thread. Hence, when the current visible thread (the thread we visualize during the current step) is changed, all the thread dependent outputs are updated accordingly.

3.7 Challenges of the AsmL implementation

Memory allocation [4] provides a specification of an *abstract memory management*. To allocate a new memory address (or a new object reference), the ASM model [4] uses the ASM construct **import** to provide a completely fresh element from the universe of addresses (or universe of object references). As the **import** is not part of the AsmL, we had to develop our own algorithms for the allocations. Although the creation of new objects has exactly the meaning of **import**, we choose our own implementation, because we intend to obtain a fresh integer value to represent the new address. We describe here only the algorithm which allocates a memory address. First, it tries to find a free memory location (which is not occupied either by a reference or a value). If it does not succeed, the memory capacity is increased with a new location (assumption of infinite memory). Thus, one returns the found *free* address, if there is one, otherwise we return the location situated immediately after the last allocated memory address. The content of the returned address is initialized with *Undefined* and the address is marked as allocated ($isAdr(adr) := true$). The algorithm is implemented in AsmL and is illustrated in Fig. 8:

Fig. 8. The allocation of a new memory address

```
newAdr() as Adr
  choose adr in Indices(mem) where not isRef(adr)
    and not isAdr(adr)
      mem(adr) := Undefined()
      isAdr(adr) := true
      isRef(adr) := false
      return adr
  ifnone
    let adr = Adr(mSize)
      mSize += 1
      mem(adr) := Undefined()
      isAdr(adr) := true
      isRef(adr) := false
      return adr
```

Model enhancements and refinements We enhanced the ASM model [4] with new features, which are also implemented in the AsmL specification, like the transition rules for the `do`, `for`, `switch`, `goto` *case*, `goto default` statements.

Because in the ASM model the type of the AST nodes' values is not incorporated in the value itself, we need to keep the track of the value's type, hence, in many places where we need the type of a value, we actually carry a tuple formed by the value and its corresponding type.

Syntactical transformations The syntactical translations represented a real challenge, because they required complex modifications in the structure of the program's AST. They are implemented in parallel in a single scan. First, one recursively calls the syntactic sugar routine for all the children in parallel, then we actually do the syntactical transformations for the current node, making use of the **match** statement. We use compiler reserved names for the new variables that may appear during this process, i.e. we add "_" in front of their names.

Fig. 9. Parallel execution of threads

```
shared execParallelCSharp()
  step
    step foreach i in Indices(threads) where alive(threads(i))
      // executes a step from the thread i
      threadContext(threads(i)).execStep(threads(i))
  step skip
```

Parallel execution of threads We experienced problems when we tried to implement the parallel execution of threads by using the **forall** statement, because of the parallel memory accesses or parallel object lockings. Therefore, we used the **step foreach** statement (see Fig. 9). Actually, the threads are executed serially in single step.

3.8 Remarks on the AsmL reference manual

Upon developing the AsmL implementation, we have encountered a few minor gaps in the AsmL reference manual [1], which we describe here briefly.

Option types The AsmL standard defines in [1, §5.1.2] an option type in the form `t?` as a type that includes all the values of type `t` plus the special value `null`. There is no restriction specified for the type `t` concerning its kind: reference type vs. non-reference type. There is provided even an example where a variable can be declared of the type `Boolean ?`. Example 22 of the same standard defines a variable of the type `Integer ?`. So, there should apparently be no worry on involving non-reference types in option types. However, when we run such programs we get the following compile-time error:

```
? not applicable to non-reference type (use 'T or Null' instead)
```

Therefore the AsmL reference manual [1] should explicitly state that `?` is applicable only to reference types `t` – a constraint that makes sense considering that the type `t` is extended with the `null` value. Notice that, unlike other languages, the AsmL reference types do not include the `null` value.

Subtype relation In [1, §5.4], it is defined the subtype relationship. One of the properties this relation has to satisfy is the following:

> *Subtype relationships extend through instantiations of type families of structure types but not through instantiations of type families of class types. For example, if T is a subtype of type S then the instantiated type* `Set of T` *is a subtype of type* `Set of S`*, since* `Set of T` *is a structure.*

However, AsmL tests indicated us that although, for example, the type `Byte` is a subtype of `Integer`, i.e. every value of type `Byte` can be used where an integer is expected, `Set of Byte` is not considered a subtype of `Set of Integer`. On the other hand, examples signaled that `Set of Byte` is considered a subtype of `Set of Object`.

This leads us to the idea that in the current AsmL implementation, the subtype relation satisfies a stronger condition where either S and T are both reference types or S is the type `Object`. The same remark applies also to the other type families, i.e sequences and maps.

Parameter modifiers In AsmL one can add modifiers to the method parameters. `out` and `inout` are two of the modifiers that can be applied to a parameter. While the former is applied to define an output parameter, there is no description in [1, §10.1] for the `inout` modifier.

4 Related work

A similar work has been done for Java in AsmGofer [5]. AsmGofer is an ASM interpreter embedded in the functional programming language Gofer (a subset of Haskell). In our implementation, the static part of the ASM interpreter for C♯ is bigger than the dynamic part (this does not apply for Java), because most universes are handled as lists. Currently, a project on implementing executable specifications in AsmL for the .NET CLR virtual machine has been started guided by the second author. Another formal definition of the Java semantics has been defined in an operational style using the Typol formalism [16]. One uses *natural semantics* to specify the object oriented and inheritance features of Java and *structural operational semantics* to describe concurrency. This formalism enables us to get a modular Java specification. Using the *Centaur* system, this semantics permits to generate a development environment called *Jitan* for parallel object-oriented programming and visualization; the specified semantics communicates with the environment thanks to notifications.

5 Conclusion and Future work

This paper describes the executable specifications which validate the ASM model for C♯ [4]: it showed that a refinement of the ground model is executable in AsmL. The validation serves also to provide experimental tests of the C♯ ASM model. A purely theoretical specification often contains subtle errors, which are sometimes very hard to discover. An implementation of the theoretical specification can reveal such errors, by showing a misbehavior during its execution. Furthermore, by using AsmL one can easily develop a robust prototype of a real application, in our case a C♯ interpreter, and can explore and simulate design approaches before the implementation begins. Therefore, the prototype can serve as a solid base for the development of a reliable application.

The AsmL specifications are very precise and have a very high level of abstraction. Therefore, they are more readable and comprehensive. They can be easily integrated in the Visual Studio .Net Environment. They can also communicate with the rest of the system by simply defining some interface methods (entry-points) in the AsmL code, which can be used by the rest of the system to retrieve some data from the abstract state machine.

The future development of the project includes the implementation of the C$\#_{\mathcal{U}}$ *memory model* mentioned in Section 2.1 (see [4] for details); the memory model is changed into an array of bytes; it is a very low-level memory model, which makes use of the functions *pack* for encoding the data which is written into the memory and *unpack* for decoding the data which is read from the memory. We will also *divide the memory* in *stack* and *heap*. There will be different allocation policies for these two memory models. Currently, they are just logically separated. We also want to implement a scheduler for the parallel execution of threads, which will handle the simultaneous memory accesses and the simultaneous locking of objects. At the end, we plan to extend the implementation with *generic types*, as introduced in [8].

References

1. Abstract State Machine Language (AsmL), Foundations of Software Engineering Group, Microsoft Research, Web pages at `http://research.microsoft.com/foundations/AsmL/`.
2. Y. Gurevich, B. Rossman, W. Schulte, Semantic Essence of AsmL, Microsoft Research Technical Report MSR-TR-2004-27, 2005.
3. U. Glässer, Y. Gurevich and M. Veanes, Abstract communication model for distributed systems, IEEE Transactions on Software Engineering, vol. 30, no. 7, 2004.
4. E. Börger, N. G. Fruja, V. Gervasi, R. F. Stärk. A High–Level Modular Definition of the Semantics of C♯. Journal Theoretical Computer Science, 2004.
5. J. Schmid. Executing ASM Specifications with AsmGofer, Web pages at `http://www.tydo.de/AsmGofer/`
6. H. Mössenböck, Albrecht Wöss, Markus Löberbauer. The Compiler Generator Coco/R, Web pages at `http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/`
7. S. Wiltamuth and A. Hejlsberg. C♯ Language Specification. MSDN, 2003.
8. C♯ Language Specification 2.0. MSDN, 2004.
9. Microsoft .NET Framework 1.1 Software Development Kit. Download from `http://msdn.microsoft.com/netframework/howtoget/`.
10. R. F. Stärk and J. Schmid and E. Börger. Java and the Java Virtual Machine–Definition, Verification, Validation. Springer–Verlag, 2001.
11. E. Börger and R. F. Stärk. Abstract State Machines. A Method for High–Level System Design and Analysis. Springer–Verlag, 2003.
12. N. G. Fruja. Specification and Implementation Problems for C♯, Proceedings of the 11th International Workshop ASM 2004, Halle, Germany.
13. R. F. Stärk and E. Börger. An ASM specification of C# Threads and the .NET memory model. 11th International Workshop on Abstract State Machines, ASM 2004, Wittenberg, Germany.
14. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1993.
15. Visual Studio .NET 2003 web site. `http://msdn.microsoft.com/vstudio/`
16. Isabelle Attali, Denis Caromel, Marjorie Russo. A Formal Executable Semantics for Java. Proceedings of Formal Underpinnings of Java - An OOPSLA'98 Workshop, Vancouver, October 1998.