An Infrastructure for Implementing Compilers for Concurrent Abstract State Machine Languages

Kristian Magnani, Mariza A. S. Bigonha, Roberto S. Bigonha, Fabíola F. Oliveira
Universidade Federal de Minas Gerais, Departamento de Ciência da Computação,
Belo Horizonte, Brazil, 31.270-901
{kristian, mariza, bigonha, fabiola}@dcc.ufmg.br

and

Vladimir O. Di Iorio
Universidade Federal de Viçosa, Departamento de Informática,
Viçosa, Brazil, 36570-000
vladimir@dpi.ufv.br

Abstract

This paper proposes a new approach for concurrency in abstract state machines (ASM) and presents the MIR Architecture, an infrastructure designed to serve as basis for abstract state machine compilers. This new concurrency model improves ASM with truly concurrent capabilities. Besides the implementation of this new approach, the proposed infrastructure also allows the implementation of optimizations specially designed in order to address specific ASM optimization opportunities.

Keywords: MultiAgent Systems, ASM, Distributed System, Intermediate Representation Language.

1 Introduction

Abstract State Machines (ASM) is a formal semantic method introduced by Yuri Gurevich in order to provide operational semantic for algorithms [11, 12]. This paper describes a new model for specifying concurrent systems based on ASM and proposes an infrastructure, called MIR, to implement ASM-like languages. This infrastructure provides two important features:

- It implements the concurrency model proposed in this paper. This model is based on Lamport's concept of distributed systems [15], and it is intended to improve the original ASM model.
- It is also designed to allow some optimizations to be performed over ASM specifications [20, 24, 25]. These optimizations can be those specific for the ASM model, and do not overlap with the customary code optimizations.

According to Gurevich, ASM are abstract machines whose states are algebraic structures, which can be viewed as abstract memories. The arguments of the functions in the algebra are locations of the memory, whereas the values of the functions are their contents [4]. Some basic concepts related to the ASM model are introduced following.

Vocabulary and states: a *vocabulary* or *signature* is a finite collection of function names, each with a fixed arity. A *state* is a nonempty set called the *superuniverse*, together with interpretations of the function names.

Transition Rules: a transition rule resembles a program written in an imperative language.In a transition rule there is no iteration command,because it is not necessary due to the intrinsic cyclic execution of an ASM. Given an *initial state*, the transition rule is applied to it, leading to a new state, over which the transition rule is applied again. This process repeats over and over, until no modifications are observed in the state. Although the vocabulary stays unchanged along the execution, its interpretation is modified by the transition rule, from state to state. The execution of ASM transition rules produces update sets, which is used to obtain the new state. Transition rules are *update rule*, *block constructor* and *conditional constructor*. An *update rule* is an expression $f(\bar{t})$:= t_0, where f is the name of a function, $\bar{t}$ is a tuple of terms whose length equals the arity of f and t_0 is another term. Terms have no free variables and are recursively built using names of distinct elements of the superuniverse and the application of function names to other terms. The effect of the execution of such a rule is that, in the *next* state, the value of the function f at the point $\bar{t}$ is the value denotated by t_0, evaluated at the *current* state. A *conditional constructor* is an expression with the following format: **if** g_0 **then** R_0 **elseif** g_1 **then** R_1 ... **elseif** g_k **then** R_k **endif** The semantics is: the rule R_i, $0 \leq i \leq k$, will be executed if the boolean terms $g_0, ..., g_{i-1}$ evaluate to *false* and g_i evaluates to *true*, in a given state. A *block constructor* is a set of rules $R_0, R_1, \ldots, R_k$. The effect of the execution of a block is the execution of all its components at the same time.

Programs and Runs: a program is a transition rule. A *run* is a sequence of states. Each state is generated by firing an update set in the previous state.

The original Gurevich's ASM does not fit to the most general concept of distributed systems. Particularly, the coherence condition requires only one agent be active at each time, while the others remain blocked. In other words, a transition made by an agent (see Section 3.2) is *atomic* with respect to others agents. This active agent acts over a world that, in the very moment of its action, can be changed exclusively by it. There is no real concurrency in the strict sense of this word. Moreover, there is no real parallelism between agents, just the interleaved execution of a transition rule, which also occurs in the single-agent case. It is as if the execution of a transition rule is always perfomed inside a critical section. This approach leads to a simpler, more treatable model, but it does not reflect the reality of distributed systems.

According to Lamport [15], a distributed system may be defined as a collection of processes that are somehow separated and communicate with each other by exchanging messages. If inside a system the message transmission delay is not negligible comparing to the time between events in a single process, then such a system can be considered distributed.

One of the remarkable features of a distributed system is that it is not always possible to say if an event will occur before or after another one. So, it is possible to get some non-predictable behaviour from a distributed system, regarding that the order of events is not predictable. It is responsibility of the designer to use the available synchronism mechanisms in order to avoid such undesirable, anomalous behaviour.

2 Related Work

Del Castillo presents in [7] a definition of an *evolving algebra abstract machine* (EAM) as a platform for developing ASM tools and [6] introduces an implementation called ASM-Workbench. The ASM-Workbench [5, 6] describes a system that is able to transform an ASM specification to a C++ program. The specification language is called ASM-SL, and it is a typed ASM specification language based in functional programming language ML. The ASM-Workbench is an important implementation. It is also given a formal definition of the EAM ground model in terms of a universal ASM. [8] (apud [7]) performs a description of a functional interpreter for ASM, with applications for functional programming languages. Some extensions to the language of ASM are proposed, as well. [13] (apud [7]) presents a Prolog interpreter for ASM specifications that are made in a particular language.

The AsmGofer is an ASM programming environment presented by Schmidt [22, 21], which extends the Gofer functional language. It provides an interpreter, and therefore it is not so fast as a compiled specification could be. On the other hand, it is usefull in order to build prototypes.

Anlauff presents in [1] the XASM, an ASM language, together with a compiler for it. A formal definition of the language is given by Kutter in [14].

The current AsmL version of Gurevich is AsmL 2, also known as AsmL for the Microsoft .NET, and it can be found at [2]. It does not provide multi-agent mechanisms. On the other hand, it benefits from the vast library of the Microsoft .NET platform.

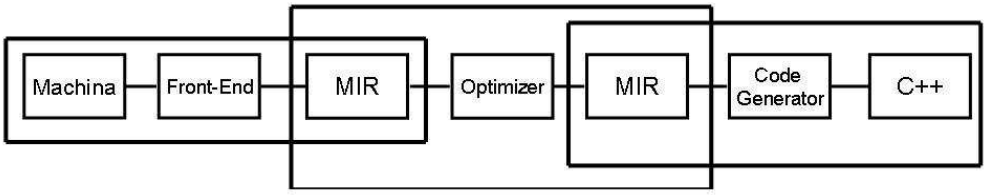

Figure 1: The Machĭna Project.

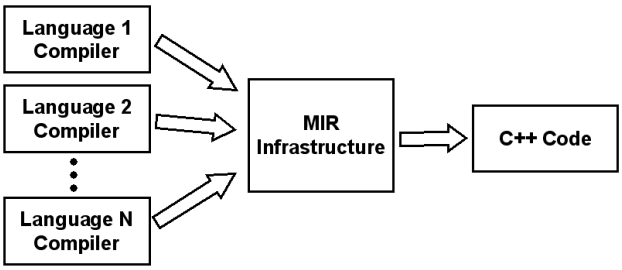

Figure 2: The Context of MIR.

Finally, Visser have developed the EvADE compiler [27], which implements an optimization: the common sub-expression elimination. However, this one does not belong exclusively to the ASM model.

All these systems are concerned with only a few optimizations, and none of them addresses the concurrency issue nor mention an intermediate representation with features for disttributed ASM. The infrastructure presented by this paper aims to properly address the concurrency issue, using an intermediate representation language, and it also provides an optimization environment where specific ASM optimizations can be plugged in order to produce efficient code. The infrastructure proposed and how these results are achieved are explained in the sequel.

3 The MIR Architecture

The idea of developing a general infrastructure for *concurrent* ASM compilers arised from the purpose of bringing this powerful semantic modeling methodology to the world of distributed systems. It would be helpful to have a general infrastructure available, which could be the basis of compilers aiming at different ASM oriented languages. As it provides the concurrent execution capability, such an infrastructure would be usefull to implement concurrent algorithms, at the same time providing a formal semantic model over which one could prove or deduce some properties. The MIR architecture was designed in order to answer these needs. MIR stands for Machĭna Intermediate Representation. It was originally used as an intermediate representation for the Machĭna language under the Machĭna project. The MIR project has grown up and nowadays it serves as the basis of compilers for concurrent ASM oriented languages.

The Machĭna project, illustrated in Figure 1, is composed of three distinct parts. The first one comprises the Machina front-end, which translates Machina program to MIR, the intermediate representation language. The second part receives a MIR file, optimizes it, considering the optimizations specially designed for the ASM model and produces as output a MIR file optimized. The third part translates from MIR to C++.This paper addresses the design and implementation of MIR. For details of the Machĭna project refers to [3, 16, 17, 19].

The main feature of the MIR Infrastructure is to produce C++ code from a MIR specification and thus allowing rapid development for ASM compilers. Figure 2 shows MIR usage context.

Another remark worth mentioning about MIR is that it was designed to be easily *optimized*, as shortly introduced in Section 3.5. These optimizations do not overlap with those who are normally performed by the C++ compiler. Rather, they belong exclusively to the ASM model. For details, see [18].

MIR Infrastructure means all the classes and software components that compose the infrastructure presented in this paper and its implementation. The expression *MIR Architecture* refers to the general structure of an ASM specification using the MIR Infrastructure .

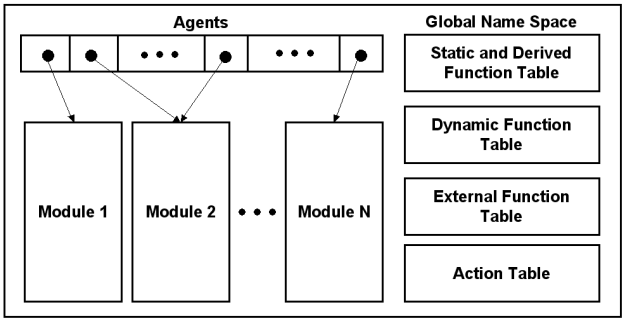

Figure 3: The MIR Architecture.

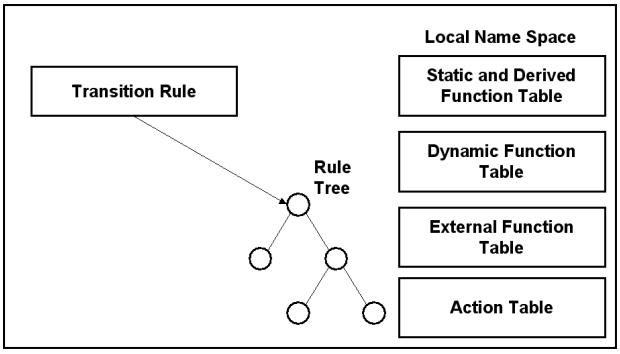

Figure 4: A Module of MIR.

3.1 Agents, Modules and Other Elements

A MIR specification is basically composed by a set of *agents*, each of them of a given type, and a *common* Global Name Space, which is accessible by each agent belonging to that MIR specification. This picture is depicted in Figure 3. The Global Name Space is defined as tables for *static*, *derived*, *external* and *dynamic functions*, and for the *actions*, as well. Static functions are those which can not have values in points of their domain changed by update rules. These functions are defined by parameterized expressions, and remain unchanged during the whole execution of the MIR architecture. It is not allowed to call a dynamic function inside its definition, as well. A derived function is similar to a static function, except that it may call dynamic functions. Dynamic functions can have values in points of their domain changed or even defined by update rules. External functions are those defined outside the MIR architecture definition, and only their signatures and return types are known. They are usefull to model interaction with the environment. Finally, actions are the abstraction of agents. They allow the implementation of the notion of *submachine*, as pointed out by Tirelo [24]. Static and derived functions are grouped together, because their definitions are closely related.

The type of an agent is a *module*. Figure 4 presents the structure of a module in MIR. A module contains a *transition rule* augmented with some support structures: the *update lists* and the Local Name Space. Like the global one, the Local Name Space consists of the tables for static, derived, external and dynamic functions, and for the actions, as well.

The transition rule is a tree of rules. The nodes of such a tree are given by the simple and compounded rules of the MIR Architecture. Simple rules appears always in the leaves of a rule tree, while compounded rules are used in order to build rules from other ones. Simple rules are:

- *update* rule, which changes the value of a dynamic function at a specific point;
- *create* rule, which creates a new agent and starts its execution;
- *destroy* rule, which deletes an agent;
- *stop* rule, which indicates that the execution of the rule must be halted;

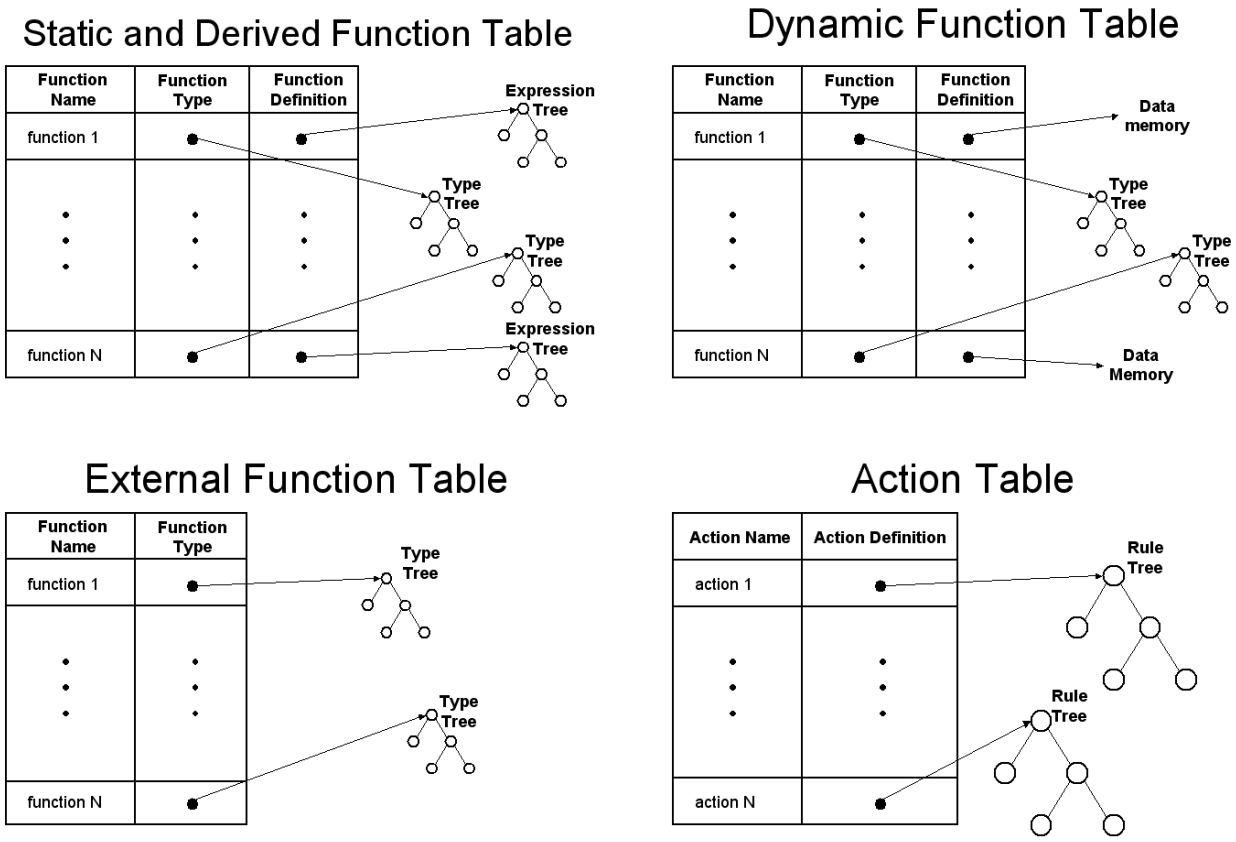

Figure 5: The Structures of the Tables.

- *return* rule, used inside actions to indicate that the execution must return to the calling rule;
- *action call* rule, which starts an action as a submachine;

The compounded rules of the MIR Architecture are:

- *conditional* rule, the traditional *if-then-else* rule;
- *forall* rule, which applies a rule to *all* the elements of a set;
- *choose* rule, which applies a rule to *a randomly chosen* element of a set;
- *let* rule, which allows *ad-hoc* expressions definitions;
- *case* rule, which executes a rule according to the *value* of an expression;
- *with* rule, which executes a rule according to the *type* of an expression;
- and the *block* rule, used in order to define a rule that is the union of many rules.

The Static and Derived Function Table is a list whose entries have three components: Function Name, Function Type and Function Definition, as depicted in Figure 5. The function type is a tree of types, whose root is possibly a functional type node, except when the function is nullary. A Type Tree is a tree whose nodes are either basic types or type constructors. High order functions are not allowed. The Function Definition is a tree, as well, and it can be recursively constructed from basic expressions and expressions constructors. The definition of each of the type and expression nodes are beyond the scope of this paper, and the reader is referred to [18] in order to get more details about them.

The entries of the Dynamic Function Table have also three components, but its third component, the Function Definition, now leads to a dynamic mapping between points in the function domain and their values. The Dynamic Function Table is illustrated in Figure 5.

As external functions are defined outside the MIR architecture definition, the entries of the external functions table have just two components: the Function Name and the Function Type. External functions are written in C, and the communication protocol and the parameter mapping policy are defined in Section 3.3. The External Function Table is depicted in Figure 5.

The Action Table represents the table of agent abstractions, and its entries are pairs whose first component is the Action Name and the second component is the Rule Tree. This table is presented in Figure 5.

3.2 The MIR Approach for Concurrency

The original definition for concurrent ASM programs, as presented in [11], is too limited to model properties usually found in concurrent systems. In the Gurevich's model, two or more agents are not allowed to execute their transition rules simultaneously, which make it inapproprieted to model real multiagent systems.

According to Zambonelli [29], an *agent* is a software entity that exhibits *autonomy*, *situatedness* and *proactivity*. The concept of agent used here pursuits these features in the following sense:

Autonomy: there is a transition rule associated with each agent, giving it an autonomous and independent behaviour from other agents.

Situatedness: every agent is immersed in a common global context where they can interact among each other, which are referenced by their names or by the special word *self*[1].

Proactivity: an agent has the freedom on calling actions over other agents.

Agents also interact with each other through cooperation, coordination or negotiation. This kind of interaction is called *sociality*.

This section presents a new approach for concurrency in ASM in order to cope with real concurrent systems. The main features of this new approach are the possibility of simultaneous moves by different agents and the concept of "delayed knowledge" [15], meaning that agents may store copies of the global state which may not always be kept up to date.

Our intention is to model systems with agents of different speed of execution. When the update set of an active agent is processed, it means that the agent has had enough time to execute an iteration of the associated transition rule. This time can differ from agent to agent, leading to an unpredictable delay between the moment some changes are performed by one agent and the moment when these changes are perceived by another agent. In a distributed system, the delay may be also associated with the time for information transmission.

The proposed model does not provide any primitive for communication between agents. The model is general enough to allow inconsistent simultaneous updates produced by different agents. All communication must be explicitly programmed by the designer of an ASM concurrent specification. For example, suppose a set of agents in a *distributed* system, meaning that all communication is handled via message exchange. The designer of the system may represent communication channels by ASM functions which are updated by a *sender* agent and read by a *receiver* agent, avoiding inconsistent updates. Many ASM-based languages [26, 2] offer mechanisms for hiding information and visibility control, and also for the definition of abstractions for ASM rules. These concepts may be used by the designer to ensure that all communication will be executed using only the defined channels, allowing an agent to update only the ASM functions related to its output channel.

In the MIR Architecture, the scope of a function or action may be declared either *local* or *global*. Local declaration specifies the elements that belong to a specific module, and therefore these elements are accessible just inside that module. Conversely, a globally declared function or action can be accessed by each agent at execution in the context of a MIR specification. Global elements are declared at the top-most level, namely, the MIR specification outside the modules. Name conflicts between elements of both scopes are not allowed.

Every agent in a MIR specification is executed concurrently, and each of them has local copies of the global dynamic functions it makes use of. The local copies are used by the transition rule of the agent, and occasionally a synchronization window opens and the local copies and their global correspondents are

[1]The interpretation of *self* is the agent itself in whose transition rule this word is found.

updated. As argued in [15], in a concurrent system, it is sometimes impossible to say in advance which one of two events will occur first, and therefore the relation "happened before" is only a partial ordering of the events of the system. This is particulary true for the event of synchronization in the MIR approach for concurrency, since the synchronization is made by each agent at the end of the execution of its transition rule, and it is not possible to assure that every agent will be going to finish it at the same time.

The execution of an agent transition rule produces three types of update lists, namely: the *local* updates, the *import* updates and the *export* updates. These update lists are maintained inside the MIR module, and are employed in the proper situation.

The local updates are related to the update rules that act over locally defined dynamic functions. They are fired at the end of every execution of the transition rule. The entries of the local update list are cleaned up after they are fired, and then the new execution of the transition rule will fill in it again. This gives the local update list a transient, dynamic nature. The import and export update lists, however, are associated with globally defined dynamic functions.

The import updates are fired every time the local copy of a global dynamic function needs to be refreshed. This happens at two occasions:

1. the import is done when the execution of an agent starts;
2. it is also done at the moment the values of dynamic functions of the agent are synchronized with the global name space.

The entries of the import update list of a module are all those globally defined functions used as *right* hand side values by the transition rule of the module. This list can be constructed from the analysis of the transition rule, and it remains unchanged during all the execution. The export updates take effect when the local copy of a global defined dynamic function is to be uploaded into its global counterpart, hence its local value becoming available to other agents which imports that dynamic function. The entries of the export update list of a module are all those updates of global dynamic functions used as *left* hand side values by the transition rule of the module.

According to Lamport [15], a distributed system may be defined as a collection of processes which are somehow separated, and they communicate with each other by exchanging messages. If inside a system the message transmission delay is not negligible comparing to the time between events in a single process, then such a system can be considered distributed. The concurrent MIR approach follows the above definition. More specifically, each agent performs a data exchange at the end of each iteration of its transition rule, and it is at this very moment that the agent updates its knowledge about the external world. Between two consecutive data synchronizations, the agent gets its transition rule executed, which may take an unbounded period of time. This time differs from agent to agent since it depends upon several factors, like the size of the rule, the parts of this rule that are executed in fact, processor speed and other hardware resources, and so on. Meanwhile, the external environment can be modified by other agents, but these changes are perceived by an arbitrary agent only at its next synchronization, and this time is not negligible. As it occurs with real life distributed systems, the perception of reality may differ depending on the moment of observation.

3.3 MIR Native Interface

MIR Architecture allows the existence of *external functions*, also known as *oracle functions*, to be written as C++ *functions* The external functions are present in the ASM model since the beginning and they provide a way to specify interaction with the environment. As argumented by Gurevich in [11], an oracle function does not need to be consistent between different execution steps of a transition rule. But the oracle should be consistent at different uses in the same execution step of the transition rule. This effect can be achieved by caching the accesses of the external functions at the same iteration, and that is done in MIR Infrastructure. In order to get the external functions written in C++ to be called from the MIR specification properly, it is required that they obey some conventions. These conventions are called *MIR Native Interface*, or MNI, for short, and they determine a common protocol upon which both the MIR specification and the C++ external function can rely.

In the external functions written in C++ it is not allowed the void return type, as they are indeed not procedures, but functions, and so some value is expected from them. On the other side, the function can be a nullary one. The mapping from the types of MIR and the correspondent C++ types are presented in Table 1. They are the only types allowed both in the signature and as return type of the C++ functions

MIR Type	Correspondent C++ Type
boolean	`bool`
character	`char`
integer	`int`
real	`double`
string	`std::string`
$(T_1,\ldots,T_n)$	`struct`
list of T	`std::vector<`T`>`

Table 1: Type mapping in the MIR Native Interface. The `std::` prefix means that the element belongs to the *Standard Template Library* of C++ [23].

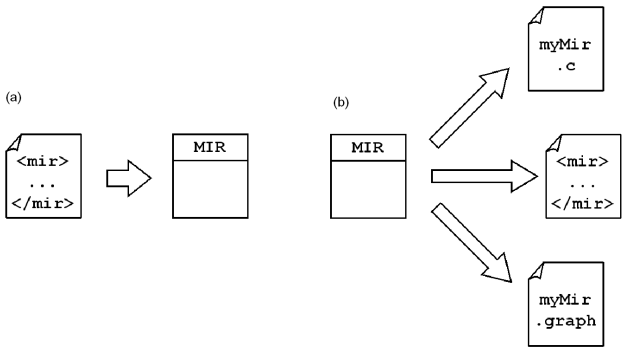

Figure 6: The features of the MIR implementation.

used as external ones. Due to implementation restrictions, the type of a parameter must be a basic type, a tuple, or a list of the allowed types.

3.4 Highlights of MIR Implementation

The implementation of MIR is heavily based on design patterns, particularly on the *visitor* design pattern. According to Gamma et al. [9], a visitor is a design pattern that represents an operation to be performed on the elements of an object structure without changing the classes of the elements upon which it operates. This concept is directly applied in the implementation of the following operations.

Serialization: the MIR representation of an ASM specification can be saved in permanent store. This saving process is called *serialization*. MIR implementation allows its serialization through XML files, according to a specific format determined by a XSD (*XML Schema Definition*) [28]. This serialization may happen in both ways: it is possible to get a MIR object from a serialization file, as well as a MIR object can be serialized in such a file. The main advantages of this represention are: XML files are just plain text files, so they can be edited using many alternatives according to the needs of the programmer; XML files are easily readable not just by humans, but also by machines, as it is relatively simple to build parsers for them, and even some libraries are available in order to automate this work; and finally, an XML representation is quite easy to produce from the hierarchical structure of the MIR architecture.

Compilation to C++ Code: the MIR implementation provides a visitor in order to obtain C++ code that reflects the architecture under definition. The generated code matches the C++ standards, so it is possible to compile it into different machines and different operating systems with a few changes. Additionally, the adoption of C++ as target language allows the generation of efficient, fast code, which is essential in many scenarios. The existence of several C++ compilers, some of them available without charge, frees the user of the MIR Infrastructure of being dependent upon specific compilers and vendors.

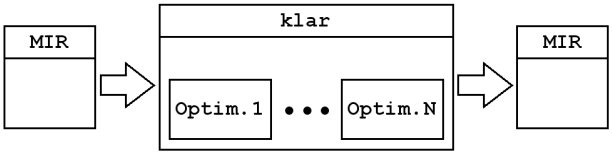

Figure 7: The optimization process of a MIR specification by using the kℓar framework.

Direct Execution: the MIR implementation allows its direct execution. In other words, the MIR objects can be executed in an interpreted fashion, without the need of being converted to C++ code and then compiled. This is usefull, for instance, in building step-by-step symbolic debbugers. The visitor that provides this feature can be viewed as a virtual machine for executing MIR specifications, a concept used nowadays and that offers some advantages, as providing an abstraction layers over different machines and operating systems.

Visualization: it is also possible to obtain a visual representation of a MIR specification through the generation of its description in the DOT language of the GraphViz software. The DOT language is a language designed for description of graph-like structures, and it is used in GraphViz software. This program and the definition of the DOT language can be found at [10].

3.5 MIR and Optimization

It is not enough to a modern compiler to get executable code; this code should be also *optimized*. For the target language of the MIR Infrastructure, namely, C++ code, there are several compilers that address this request with efficiency. However, there are some optimization possibilities that belong exclusively to the ASM model, and therefore they are not performed by the existing C++ compilers, as argued in [20] and in [25]. These possibilities are the ones we are interested in.

In order to address this special situation, it is under development a framework [17] that provides the proper environment to optimize MIR specifications, as depicted in Figure 7. The optimizations are easily added or removed as plugins, thus facilitating the development of new optimizations.

As examples of such ASM-related optimizations, we can point out the Update Scale Optimization and the Deviation Optimization, taken from [19, 20, 25]. Some brief comments about them are made in the sequel.

Update Scheduling Optimization: let $B = U_1, U_2$ be a block where U_1 and U_2 are updates, so that U_1 is `y:=z` and U_2 is `x:=y`. According to the semantics of ASM, these updates could not be direct converted to a sequence of attributions in C++ code, because `x` could receive a wrong value, according to the model. Instead, the updates are compiled into code that inserts some entries in the list of updates to be processed at the end of the iteration of the rule. Unfortunately, this can be time consuming, as the operations involved may have high computational costs. It is desirable to avoid paying this cost whenever possible. Some intructions, like U_2 above, could be performed immediately without the loss of the model features. Moreover, the order in which the elements of a block are compiled can result in a greater or smaller number of direct updates. For instance, if U_1 and U_2 are commuted, no insertion in the update list would be necessary, and both updates could be carried out directly. The optimization proposed schedules the rules inside a block in order to make the number of direct updates maximal.

Deviation Optimization: a transition rule R must be compiled into an infinite loop of the form `L:` R`; goto L;` . In the case of conditional rules, like `if` g `then` R_1 `else` R_2, it is compiled into `L: if (`g`)` R_1`; else` R_2`; goto L;`. Now suppose that R_2 does not update any dynamic function used in g, and any external function call can be found in g, as well. In this case, once R_2 executed, it is not necessary to get back to `L`, as g was not changed. An optimized code would be like `L: if (`g`)` $\{R_1\}$`; else {L2:` R_2`; goto L2; } goto L;`. This optimization detects possibilities like the one above, producing code that results in more efficient deviations.

4 Conclusions

After a brief review of the ASM model, this paper has presented the MIR Architecture which consists of an infrastructure for developing concurrent ASM oriented languages. The most important contributions of the proposed infrastructure are: (1) It can be used in order to implement a whole family of languages targeting the ASM model. (2) The concurrent approach and the ASM modelling distributed agents are new, and it can be used to precisely describe distributed algorithms in ASM. (3) Optimizations can be plugged in, allowing the enhancement of the generated code. Moreover, these pluggins can also be used to other types of program manipulation, e.g. refactoring, given that the necessary information is provided.

Future work includes the formal definition of the concurrent ASM model used in the infrastructure and the demonstration that important properties of disttributed systems hold for it. Some examples of these properties includes the absence of starvation and the possibility of mutual exclusion.

References

[1] M. Anlauff. XASM – An Extensible, Component-Based Abstract State Machines Language. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 69–90. Springer-Verlag, 2000.

[2] asml. The AsmL 2 for Microsoft .NET Homepage: http://research.microsoft.com/fse/asml/. Consulted in 20th April 2005.

[3] Roberto S. Bigonha, Fábio Tirelo, Marcelo A. Maia, Marcelo Silva, Marco Túlio O. Valente, Mariza A. S. Bigonha, and Vladimir O. Di Iorio. Projeto Machina. Technical Report LLP007/99, Universidade Federal de Minas Gerais, Julho 1999.

[4] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[5] G. Del Castillo. The ASM Workbench: an Open and Extensible Tool Environment for Abstract State Machine. In *28th Annual Conference of the German Society of Computer Science*, 1998.

[6] G. Del Castillo. *The ASM Workbench: A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models*. PhD thesis, Universität Paderborn, 2000.

[7] G. Del Castillo, I. Durdanović, and U. Glässer. An Evolving Algebra Abstract Machine. In H. Kleine Büning, editor, Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95), volume 1092 of *LNCS*, pages 191–214. Springer, 1996.

[8] D. Diesen. *Specifying Algorithms Using Evolving Algebra. Implementation of Functional Programming Languages*. Dr. scient. degree thesis, Dept. of Informatics, University of Oslo, Norway, March 1995.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] GraphViz. The GraphViz Homepage: www.graphviz.org, 2005. Consulted in February 2005.

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

[12] Yuri Gurevich. Evolving algebras: An attempt to discover semantics, 1991.

[13] A. M. Kappel. Executable Specifications Based on Dynamic Algebras. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 698 of *Lecture Notes in Artificial Intelligence*, pages 229–240. Springer, 1993.

[14] P. Kutter. The Formal Definition of Anlauff's eXtensible Abstract State Machines. TIK-Report 136, Swiss Federal Institute of Technology (ETH) Zurich, June 2002.

[15] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[16] Mário Celso Candian Lobato. Proposta de Dissertação: Um Arcabouço para Compilação de Linguagens de Especificação ASM, 2005.

[17] Kristian Magnani. Proposta de Dissertação: kℓar - Um Arcabouço para Otimizações em Máquinas de Estado Abstratas, 2005.

[18] F. Oliveira, K. Magnani, M. Bigonha, and R. Bigonha. MIR: Machĭna Intermediate Representation. Technical Report RT001/04, Laboratório de Linguagens de Programação - Departamento de Ciência da Computação - Universidade Federal de Minas Gerais, 2004.

[19] Fabíola F. Oliveira. Proposta de Tese: Otimizações em Máquinas de Estado Abstratas, 2004.

[20] Fabíola F. Oliveira, Roberto S. Bigonha, and Mariza A. S. Bigonha. Otimização de Código em Ambiente de Semântica Formal Exeutável Baseado em ASM. *Proceedings of 8th Brazilian Symposium on Programming Languages*, pages 172–185, May 2004.

[21] J. Schmidt. Executing ASm Specifications with AsmGofer, 1999.

[22] J. Schmidt. Introduction to AsmGofer, 2001.

[23] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 2000.

[24] Fábio Tirelo. Uma ferramenta para execução de um sistema dinâmico discreto baseado em Álgebras evolutivas. Master's thesis, Universidade Federal de Minas Gerais, 2000.

[25] Fábio Tirelo and Roberto S. Bigonha. Técnicas de Otimização de Programas Baseados em Máquinas de Estado Abstratas. *Proceedings of 4th Brazilian Symposium on Programming Languages*, pages 144–157, 2000.

[26] Fábio Tirelo, Marcelo A. Maia, Vladimir O. Di Iorio, and Roberto S. Bigonha. Projeto Machina: A linguagem de especificação ASM. Technical Report LLP008/99, Universidade Federal de Minas Gerais, Julho 1999.

[27] J. Visser. Evolving algebras. Master's thesis, Faculty of Technical Mathematics and Informatics, Delft University of Technology, Zuidplantsoen 4, 2628 BZ Delft, The Netherlands, 1996.

[28] Priscilla Walmsley. *Definitive XML Schema*. Prentice Hall PTR, 2001.

[29] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.