The MIR Architecture: An Infrastructure for Implementing Compilers for Concurrent Abstract State Machine Languages

Kristian Magnani [1] **, Mariza A. S. Bigonha** [1] **, Roberto S. Bigonha** [1] **, Vladimir O. di Iorio** [2]

[1]Laboratório de Linguagens de Programação - Departamento de Ciência da Computação
Universidade Federal de Minas Gerais

[2]Departamento de Informática
Universidade Federal de Viçosa

`kristian@ufmg.br, vladimir@dpi.ufv.br, {mariza, bigonha}@dcc.ufmg.br`

Abstract. *The MIR Architecture is is a infrastructure designed to serve as basis for abstract state machine (ASM) compilers. It provides concurrent execution capabilities, which are usefull to implement concurrent algorithms. This paper presents an overview of the MIR architecture, its approach for concurrency and the main features of its implementation.*

1. Introduction

This paper describes a new model for specifying concurrent systems based on ASM and proposes an infrastructure, called MIR, to implement concurrent ASM compilers. We begin by reviewing the basic concepts relative to the ASM model, and then we present the adopted approach for concurrency.

1.1. The Abstract State Machine Model

Abstract State Machines (ASM) is a formal semantic method introduced by Yuri Gurevich in order to provide operational semantic for algorithms ([Gurevich, 1995], [Gurevich, 1991]). ASMs are abstract machines whose states are algebraic structures. These algebraic structures can be viewed as abstract memories. The arguments of the functions in the algebra are the locations of the memories, whereas the values of the functions are their contents [Börger and Stärk, 2003].

Vocabulary and states A *vocabulary* or *signature* Σ is a finite collection of function names, each with a fixed arity. A *state* $\mathcal{U}$ for Σ is a nonempty set called the *superuniverse*, denoted by $[\mathcal{U}]$, together with interpretations of the function names of Σ. A *basic function* with arity r is a $[\mathcal{U}]^r \rightarrow [\mathcal{U}]$ function. When $r = 0$, a function is called *distinct element*. The superuniverse always contains the distinct elements *true*, *false* and *undef*, defined as *logical constants*. The element *undef* is used for representing partial functions, for example, $f(\overline{a}) = undef$ means that function f is undefined for tuple $\overline{a}$.

Locations and Update Sets A *location* of $\mathcal{U}$ is a pair $(f, (a_1, \ldots, a_n))$ where f is a function name and $a_1, \ldots, a_n$ are elements of $[\mathcal{U}]$. An *update* for $\mathcal{U}$ is a pair (l, v), where l is a location and v is an element of $[\mathcal{U}]$. An *update set* is a set of updates. An *update set* U is *consistent* if, for any location l and elements a, b, it is true that if $(l, a) \in U$ and $(l, b) \in U$ then $a = b$.

The result of *firing* a consistent update set U in a state $\mathcal{U}$ is a new state $\mathcal{U} + U$ where, for every location l:

$$(\mathcal{U} + U)(l) = \begin{cases} a & \text{if } (l, a) \in U; \\ \mathcal{U}(l) & \text{if there is no } a \text{ such that } (l, a) \in U. \end{cases}$$

Transition Rules A transition rule is very similar to a program written in an imperative language, although some important differences can be observed. For instance, there is no iteration command. This absence is justified by 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. The update set produced by the execution of a transition rule R in a state $\mathcal{U}$ is denoted by $[\![R]\!]^{\mathcal{U}}$. Basic rules are *update rule*, *block constructor* and *conditional constructor*.

An *update rule* is an expression $f(\overline{t}) := t_0$, where f is the name of a function, $\overline{t}$ 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 $[\mathcal{U}]$ and the application of function names to other terms. The result of the execution of an update rule is

$$[\![f(\overline{t}) := t_0]\!]^{\mathcal{U}} = \{((f, \overline{t}^{\mathcal{U}}), t_0^{\mathcal{U}})\}$$

i.e, an update set with a single update, built with the evaluation of the terms involved in the state $\mathcal{U}$.

A *conditional constructor* is an expression with the following format:

$$\textbf{if } g_0 \textbf{ then } R_0 \quad \textbf{elseif } g_1 \textbf{ then } R_1 \ldots \textbf{ elseif } g_k \textbf{ then } R_k \quad \textbf{endif}$$

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 state $\mathcal{U}$.

A *block constructor* is a set of rules $R_0, R_1, \ldots, R_k$. The result of the execution of a block rule is:

$$[\![R_0, R_1, \ldots, R_k]\!]^{\mathcal{U}} = \bigcup_{i=0}^{k} [\![R_i]\!]^{\mathcal{U}}$$

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. If S_0 is the *initial state*, then

$$S_0 \overset{U_0}{\Rightarrow} S_1 \overset{U_1}{\Rightarrow} S_2, \overset{U_2}{\Rightarrow} \cdots, \qquad \text{where } S_{i+1} = S_i + U_i, \; i = 0, 1, \ldots$$

1.2. Concurrent Abstract State Machines

In a *sequential* ASM, update sets are generated by the execution of a single transition rule. In a *distributed* ASM, several agents may be running different programs. According to Zambonelli ([Zambonelli et al., 2003]), an *agent* is a software entity that exhibits *autonomy*, *situatedness* and *proactivity*. Agents interact between themselves through cooperation, coordination or negotiation. This kind of interaction is called *sociality*.

The program associated with an agent A is denoted by $P(A)$. An agent is said to be active if it is executing its transition rule.

In a *distributed run* $S_0, S_1, \ldots$, a **subset** of the active agents in each step i is chosen to contribute to the update set U_i. We will call *choice*(i) this subset of active agents, in a similar way to the one adopted in [Schönfeld, 1998]. If *choice*$(i) = \{A_1, A_2, \ldots, A_k\}$ then the update set U_i is defined as:

$$U_i = [\![P(A_1)]\!]^{\mathcal{U}_1} \cup [\![P(A_2)]\!]^{\mathcal{U}_2} \cup \ldots \cup [\![P(A_k)]\!]^{\mathcal{U}_k}$$

At the step i of a run, U_i is the union of the update sets produced by the execution of the programs associated with agents $A_1, A_2, \ldots, A_k$, in states $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_k$, respectively. For $j = 1, 2, \ldots, k$, each state $\mathcal{U}_j$ is equivalent to a state S_q of the run, with the following restrictions:

1. $q \leq i$, i.e., $\mathcal{U}_j$ is one of the previous states of the run.
2. $j \notin choice(r)$ for all $r \in \{q+1, \ldots, i-1\}$, i.e., if an agent is chosen in a step of the run, it must "update" its "knowledge" about the global state before it is chosen again in a future step of the same run.

Important points of these definitions are:

- Only a **subset** of the active agents is used to produce a new global state, implying non determinism.
- The contributions of different agents for the update set may be based on different states, which means that agents have different "knowledge" about the global state.

Our intention is to model systems with agents with 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 the associated transition rule. In a distributed system, the delay may be also associated with the time for information transmission.

1.3. Related Work

Del Castillo introduces in [Del Castillo et al., 1996] the concept of an *evolving algebra abstract machine* (EAM) as a platform for developing ASM tools and [Del Castillo, 2000] presents an inplementation called ASM-Workbench. The ASM-Workbench ([Del Castillo, 1998], [Del Castillo, 2000]) introduces a system that is able to transform an ASM specification to a C++ program. The source 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, but optimization was not one of its features. It is also given a formal definition of the EAM ground model in terms of a universal ASM. [Diesen, 1995] (apud [Del Castillo et al., 1996]) 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. [Kappel, 1993] (apud [Del Castillo et al., 1996]) presents a Prolog interpreter for ASM specifications that are made in a particular language.

The AsmGofer is an ASM programming environment presented by Schmidt ([Schmidt, 2001], [Schmidt, 1999]), which extends the Gofer functional language. It provides an interpreter, and therefore it is not so fast as a compiled specification would be. On the other hand, it is usefull in order to build prototypes.

Anlauff presents in [Anlauff, 2000] the XASM, an ASM language, together with a compiler for it. This compiler makes use of an optimization, namely, the efficient representation of dynamic functions using hash tables. A formal definition of the laguage is given by Kutter in [Kutter, 2002]. Although this is a very relevant issue in the ASM model, no futher optimizations are provided, nor even an environment where other optimizations could be easily developed.

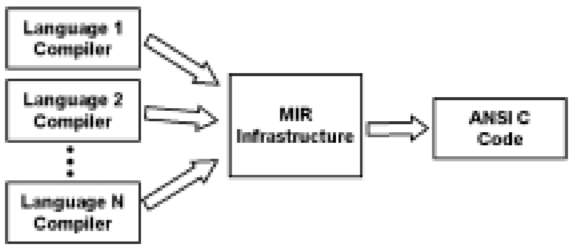

Figure 1: The Context of MIR.

Finally, Visser have developed the EvADE compiler ([Visser, 1996]), which implements an optimization: the common sub-expression elimination. However, this is the only optimization, and it does not belong exclusively to the ASM model.

Most of these systems are concerned with only a few optimizations, and none of them addresses the concurrency issue or even 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 is also binded with an optimization environment, in order to produce efficient code.The infrastructure proposed and how these results are achieved are explained in the remaining of this text, which is organized as follows: Section 2 presents the MIR Architecture. Section 3 addresses some important features of the MIR implementation. The context of optimization of MIR is introduced in Section 4. Finally, Section 5 concludes the paper.

2. The MIR Architecture

The idea of developing a general infrastructure for ASM compilers arised from the purpose of doing some experiences with languages oriented by the ASM model. It would be helpful to have a general infrastructure available, which could be the basis of compilers aiming at different ASM oriented languages. The MIR Architecture was designed in order to answer this need. Moreover, it provides the concurrent execution capability, which is usefull to implement concurrent algorithms. MIR stands for Machĭna Intermediate Representation, because originally it was used as an intermediate representation for the Machĭna language. For details of its contructions see [Oliveira et al., 2004a]. The Machĭna language and a compiler for it is presented by Tirelo in [Tirelo, 2000]. MIR project has grown up and nowadays it consists of a entire separated project, which aims its own purpose: to serve as the basis of compilers for ASM oriented laguages. A new version of the Machĭna front-end compiler is under development by Lobato ([Lobato, 2005]), and it will make use of the MIR Infrastructure.

Figure 1 shows MIR usage context. The main feature of the MIR Infrastructure is to produce ANSI C code from a MIR specification and thus allowing rapid development for ASM compilers.

Another remark worthy mentioning about MIR is that it is designed to be *optimized*, as shortly introduced in Section 4. These optimizations do not overlap with those who are normally performed by the C compiler. Rather, they belong exclusively to the ASM model.

Before we proceed with the presentation of MIR Architecture itself, it is necessary to make some preliminary remarks. *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 . It can be viewed as the “language” of the

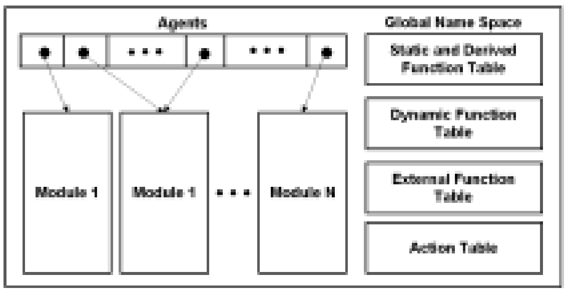

Figure 2: The MIR Architecture.

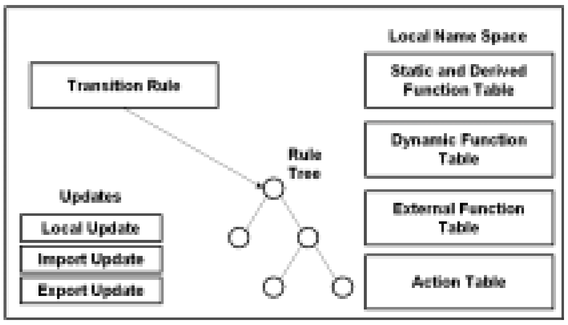

Figure 3: A Module of MIR.

infrastructure. Finally, if the MIR Architecture is a language, a *MIR Specification* is a "program" written in that language.

2.1. 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 2. 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 by that this last restriction is banished. By contrast, dynamic functions can have values in points of their domain changed or even defined by update rules. External functions are functions defined outside the MIR architecture definition, and only its signature and return type are known inside it. 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* ([Tirelo, 2000]), as pointed out by Tirelo. The static and derived functions are grouped together, as their definitions are closely related.

The type of an agent is a *module*. Figure 3 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 is the tables for static, derived, external and dynamic function, and for the actions, as well.

The transition rule is a tree of rules. The nodes of such a Rule Tree are given by the basic rules and by the rules constructors of the MIR Architecture. A *Rule Tree* is recursively defined through the grammar presented in Figure 4. Basic rules are *update* rule, *create* rule, *destroy* rule, *stop* rule, *action call* rule and λ rule. Basic rules appears

$$
\begin{array}{lll}
R & ::= & \text{I := E} \\
 & | & \textbf{if } E\ R_1\ R_2 \\
 & | & \textbf{forall } (I_1 : E_1, \ldots, I_n : E_n)\ R \\
 & | & \textbf{choose } (I_1 : E_1, \ldots, I_n : E_n)\ R \\
 & | & \textbf{create } I : \text{esclarecer...} \\
 & | & \textbf{destroy } I \\
 & | & \textbf{stop} \\
 & | & \textbf{let } (I_1 : E_1, \ldots, I_n : E_n)\ R \\
 & | & \textbf{case } (k_1 : R_1, \ldots, k_n : R_n, R_{otherwise}) \\
 & | & \textbf{with } E\ (I_1 : T_1 : R_1, \ldots, I_n : T_n : R_n, R_{otherwise}) \\
 & | & \textbf{actioncall } I(E_1, \ldots, E_n) \\
 & | & \lambda \\
 & | & R_1, R_2
\end{array}
$$

Figure 4: The recursive definition of a rule.

$$
\begin{array}{lll}
T & ::= & \textbf{bool} \mid \textbf{char} \mid \textbf{int} \mid \textbf{real} \mid \textbf{string} \\
 & | & \textbf{enum} \\
 & | & (T_1, \ldots, T_n) \\
 & | & \textbf{set } T \\
 & | & \textbf{list } T \\
 & | & T_1 \cup \ldots \cup T_n
\end{array}
$$

Figure 5: The recursive definition of a type.

always in the leaves of a rule tree. On the other side, rule constructors are used in order to build rules from other ones. Rule constructors of MIR Architecture are *conditional* rule, *forall* rule, *choose* rule, *let* rule, *case* rule, *with* rule and *block* rule.

The static and derived functions table is a list whose entries have three components: *Function Name*, *Function Type* and *Function Definition*, as depicted in Figure 6. 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. The definition of possible type trees are given by the grammar of Figure 5. High order functions are not allowed. The Function Definition is a tree, as well, and it can be recursively constructed following the productions of the grammar presented in Figure 10.

The entries of the dynamic functions table have also three components. The difference between the static and derived functions table is that the third component, the Function Definition, leads to a dynamic mapping between points in the function domain and their values. The dynamic functions table is illustrated at Figure 7.

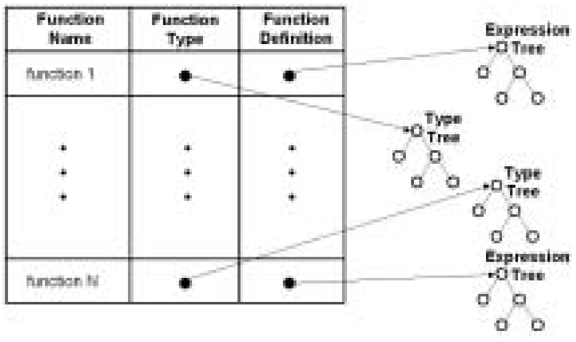

Figure 6: The static and derived functions table.

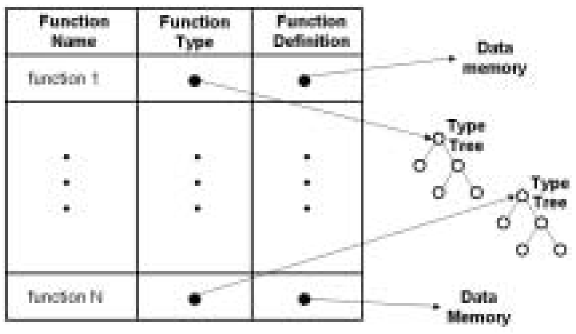

Figure 7: The dynamic functions table.

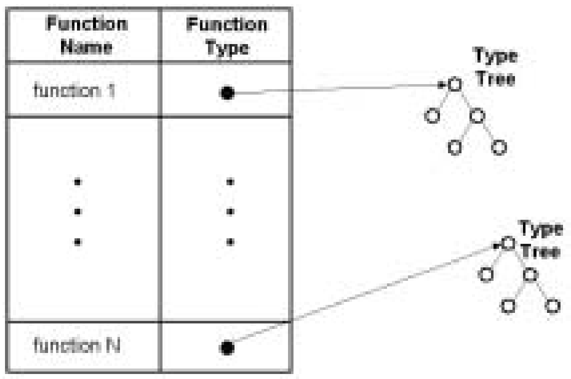

Figure 8: The external functions table.

As the 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 2.3. The external functions table is depicted in Figure 8.

The action table represents the table of agent abstractions, and its entries are pairs whose first component is the Action Name and whose second component is the Rule Tree. This table is presented in Figure 9.

2.2. The MIR Approach for Concurrency

In the MIR Architecture, the scope of a function or action may be declared either *locally* or *globally*. Local declaration means that such an element belongs to a specific module, and therefore this element is accessible just inside that module. Conversely, a global declared function or action can be accessed by each agent at execution in the context of a

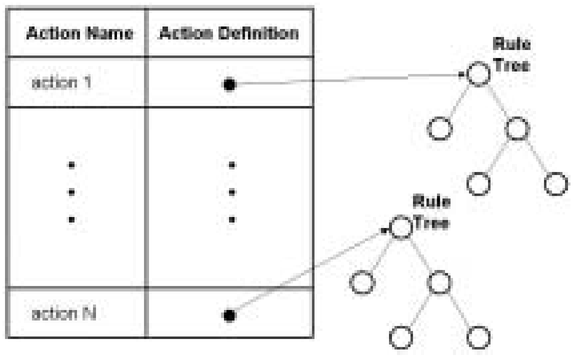

Figure 9: The action table.

$$\begin{array}{lcl} E & ::= & \textit{literal} \\ & | & \textit{unop}\ E \\ & | & E_1\ \textit{binop}\ E_2 \\ & | & \textbf{funcall}\ I(E_1, \ldots, E_n) \\ & | & \textbf{agregate}\ \text{esclarecer} \\ & | & \textbf{if}\ E\ E_1\ E_2 \\ & | & \textbf{let}\ (I_1 : E_1, \ldots, I_n : E_n)\ E \\ & | & \textbf{case}\ (k_1 : E_1, \ldots, k_n : E_n, E_{otherwise}) \\ & | & \textbf{with}\ E\ (I_1 : T_1 : E_1, \ldots, I_n : T_n : E_n, E_{otherwise}) \\ & | & \textbf{self} \end{array}$$

Figure 10: The recursive definition of an expression.

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 updated. As argued in [Lamport, 1978], in a concurrent system, it is sometimes impossible to say which one of two events ocurred 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 gives rise to three types of updates, namely: the *local* updates, the *import* updates and the *export* updates. These updates are maintened in three corresponding groups inside the MIR module, and each group is employed in the proper situation.

The local updates are related to the update rules that act over local 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 feature. The import and export updates, 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 in the every moment a synchronization of the agent with the global name space has to happen.

The entries of the import update list of a module are all those updates of local functions with the values of global dynamic functions used as *right* hand side values by the transition rule of the module. This list is permanent, as it can be construted from the analysis of the transition rule. 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.

MIR Type	**Correspondent C Type**
bool	`short`
char	`char`
int	`int`
real	`double`
string	`char*`
$(T_1,\ldots,T_n)$	`struct`
list of T	T`[]`

Table 1: Type mapping in the MIR Native Interface.

According to Lamport ([Lamport, 1978]), 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 proccess, then such a system can be considered distributed. The concurrent MIR approach follows the above definition. More specifically, each agent performs a synchronization 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 synchronizations, the agent gets its transition rule executed, which takes an unbounded period of time. This time differs from each 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.

2.3. MIR Native Interface

MIR Architecture allows the existence of *external functions*, also known as *oracle functions*, to be written in ANSI C code. 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 [Gurevich, 1995], 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 used as external ones. Due to implementation restrictions, the type of a parameter must be a basic type, a tuple, or a list of basic types.

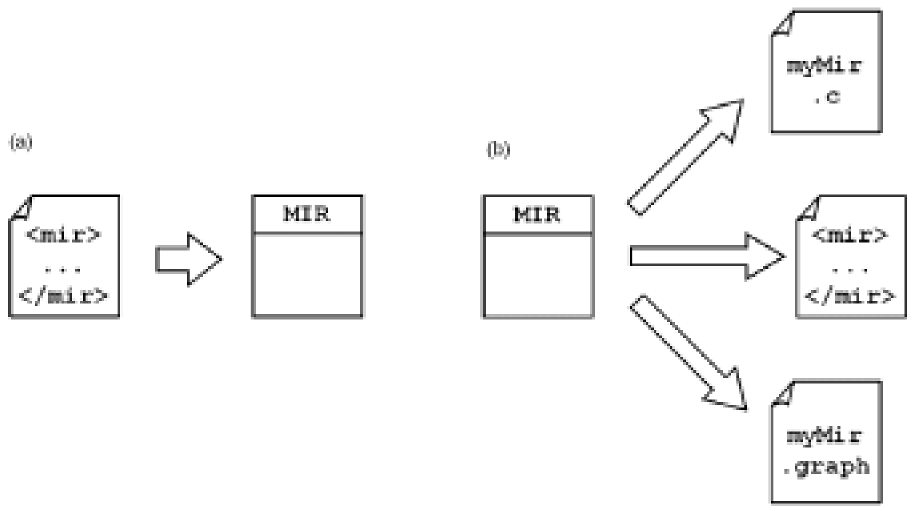

Figure 11: The features of the MIR implementation.

3. Highlights of MIR Implementation

The architecture proposed in this paper is implemented as C++ classes and they are made available as a dynamic linkage library. This implementation attempts to address some needs, as detailed in the following paragraphs and in Figure 11. *Visitors* are provided in the current implementation in order to improve it with the desired capabilities. According to Gamma et al. ([Gamma et al., 1995]), 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.

Serialization MIR implementation allows its serialization through XML files, according to a specific format determined by a DTD. 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 ANSI C Code MIR implementation provides a visitor in order to obtain C code that reflects the architecture under definition. The generated code matches the ANSI C standard, so it is possible to compile it into different machines and different operating systems. 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.

Direct Execution MIR implementation allows its direct execution. In other words, the MIR objects can be executed in a 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.

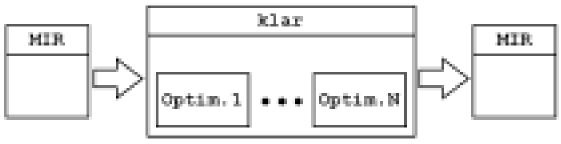

Figure 12: The optimization proccess of a MIR specification by using the kℓar framework.

Visualization It is also possible to obtain a visual representation of a MIR specification through the generation of its description in the DOT language. 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 [GraphViz, 2005].

4. MIR and Optmization

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 caught by the existing C compilers, as argumented in [Oliveira et al., 2004b] and in [Tirelo and Bigonha, 2000]. These possibilities are the ones we are interested in.

In order to address this special situation, it has been under development the kℓar framework ([Magnani, 2005]). This framework provides the proper environment to optimize MIR specifications, as depicted in Figure 12. The optimizations are easily added or removed as plugins, which facilitates the development of new optimizations. Results of this tool are expected soon.

5. Conclusions

After a brief review of the ASM model and the explanation of the concurrent model adopted, this paper has presented the MIR Architecture, which consists in a infrastructure for developing concurrent ASM oriented languages. The main motivation of the MIR is the desire of doing some experiences with the proposal of languages oriented by ASM model. We believe that this desire can be shared with other researchers all over the world, and therefore MIR Architecture can contribute providing the basis for concurrent abstract state machine compilers. Its current implementation was presented, and its main features were pointed out. The kℓar environment, under development, will complement the environment with the optimization capability.

References

Anlauff, M. (2000). 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.

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

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

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

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

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

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GraphViz (2005). The GraphViz Homepage: www.graphviz.org. Consulted in February 2005.

Gurevich, Y. (1991). Evolving algebras: An attempt to discover semantics.

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

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

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

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

Lobato, M. C. C. (2005). Proposta de Dissertação: Um Arcabouço para Compilação de Linguagens de Especificação ASM.

Magnani, K. (2005). Proposta de Dissertação: kℓar - Um Arcabouço para Otimizações em Máquinas de Estado Abstratas.

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

Oliveira, F. F., Bigonha, R. S., and Bigonha, M. A. S. (2004b). Otimização de Código em Ambiente de Semântica Formal Exeutável Baseado em ASM. *Proceedings of 8th Brazilian Symposium on Programming Languages*, pages 172–185.

Schmidt, J. (1999). Executing ASm Specifications with AsmGofer.

Schmidt, J. (2001). Introduction to AsmGofer.

Schönfeld, W. (1998). Interacting Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University.

Tirelo, F. (2000). Uma ferramenta para execução de um sistema dinâmico discreto baseado em Álgebras evolutivas. Master's thesis, Universidade Federal de Minas Gerais.

Tirelo, F. and Bigonha, R. S. (2000). Técnicas de Otimização de Programas Baseados em Máquinas de Estado Abstratas. *Proceedings of 4th Brazilian Symposium on Programming Languages*, pages 144–157.

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

Zambonelli, F., Jennings, N. R., and Wooldridge, M. (2003). Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370.