A Framework for Optimizations in Abstract State Machines

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

Abstract

The Abstract State Machines methodology offers a powerful, easy-to-use mechanism to formally specify the semantics of algorithms. The kℓar framework adds to it optimization capability, allowing the transformation of ASM specifications into efficient programs, which is important in order to use the specifications as realistic programs. Moreover, the optimizations are modules to be plugged-in "on the fly", so that independent developers can develop their own optimizations without concerning about the internal details of the kℓar framework. Finally, the wide set of constructions of the language understood by the framework allows its use as a target for compilers aiming the ASM methodology.

1 Introduction

Abstract State Machines (ASM) is a formal specification method introduced by Yuri Gurevich in order to provide operational semantic for algorithms [12, 13]. Basically, 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 [5]. There is a transition rule fired over each state S_n, which changes it, producing a new state S_{n+1}. An ASM specification is made up from three elements: a vocabulary, which remains unchanged along the whole execution; an initial state; a transition rule, which acts over the current state, producing new states or algebras.

One of the main advantages of this methodology is that it is considerably easy to get executable code from an ASM specification. In the process of translating it into executable, real-world code, there are some optimizations opportunities that could be addressed in order to get efficient code. Getting executable code from a formal specification of an algorithm is useful in order to experiment with that algorithm, but it is necessary to turn this code into *efficient* code if it is aimed to be used in a professional environment. In addition to traditional code optimizations, there are some that belong exclusively to the ASM methodology, being not addressed by ordinary compiler technologies. Examples of such optimizations are presented in Section 3

This paper presents the kℓar framework, which has been developed to offer the proper environment where independent optimizations can be easily plugged in. kℓar is specially designed so that each optimization is an independent module. This approach brings some advantages, namely:

- Independent developers are able to develop their own optimizations without worring about interference with the optimizations that coexist in ASM compiler. Since the language of optimized specifications at the output of plugged optimizers is the same as the input language of every optimizer, the object of an optimizer is always the same: a program in that language.
- By the same reason, the optimizations can be applied in any order. A particular, empirical better order can be chosen aiming the better final code.
- The optimizations are designed as dynamic libraries, so they can be plugged "on the fly". There is no need of compiling the optimizations together with the cradle, even if new versions of the cradle are released. All that is required is to configure a special XML file inside the kℓar folder. Details about this dynamic configuration and how to design such optimizations are given in Section 4

The language provided by the kℓar framework is general enough to be used as a target for compilers aiming the ASM methodology. Its constructions allow even the specification of multi-agent systems. This language is known as the MIR language. MIR stands for Machĭna Intermediate Representation. It is used as an intermediate representation for the Machĭna language under the Machĭna project [3, 4, 24, 16]. The MIR project has grown up and nowadays it serves as the basis of compilers for concurrent ASM-oriented languages. A general overview of the MIR language is presented in [18], while [17] provides all the details about the language, as well as about the infrastructure developed in order to assist the language usage.

2 Related Work

Del Castillo *at alii.* present a definition of an *evolving algebra abstract machine* (EAM) as a platform for developing ASM tools [7, 8] and introduce an implementation called ASM-Workbench [9]. The ASM-Workbench describes a system that is able to transform an ASM specification into a C++ program. The specification language is called ASM-SL, and it is a typed ASM specification language. 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. Diesen performs a description of a functional interpreter for ASM, with applications for functional programming languages [10] (apud [9]). Some extensions to the language of ASM are proposed, as well. Kappel presents a Prolog interpreter for ASM specifications that are made in a particular language [14] (apud [9]).

The AsmGofer is an ASM programming environment presented by Schimdt [21, 22], 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 useful in order to build prototypes.

Anlauff presents the XASM, an ASM language, together with a compiler for it [1]. A formal definition of the language is given by Kutter [15].

The current AsmL version of Gurevich is AsmL 2, which can be found at [2]. It is also known as AsmL for the Microsoft .NET. As an advantage, it benefits from the vast library of the Microsoft .NET platform.

Finally, Visser has developed the EvADE compiler [26], which implements an optimization: the common sub-expression elimination. However, this one does not belong exclusively to the ASM model.

All these systems are not concerned with optimizations that are particular to the ASM methodology, and none of them provides an infrastructure where optimizations could be easily added. The framework presented in this paper aims to properly provide an optimization environment to which specific ASM optimizations can be plugged in order to produce efficient code.

3 Optimizations in ASM

The kℓar framework optimizes ASM specifications. Given a specification written in the language understood by the environment, it applies the optimizations according a configuration file, yielding to a new specification in the same language. This specification can be then translated into C++ code making use of the infrastructure provided for that matter. This code, when compiled, can be optimized following the traditional imperative code optimizations. The point is that the optimizations of interest inside the kℓar framework are those which belong exclusively to the ASM methodology, not overlapping with the traditional, imperative code optimizations. This section presents a review of two of that optimizations, which were originally proposed in [20, 25]. Section 5 presents in Table 1 a small benchmark that measures the impact of the implementation of the update scheduling optimization.

3.1 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 can not be directly converted to a sequence of assignments into C++ code. Instead, the updates must be compiled into code that inserts entries in the list of updates to be processed only at the end of the iteration of the rule. This can be time consuming, considering that the operations involved may have high computational costs. However, some assignments, like U_2, could be performed immediately without the loss of the model features, provided the location at the left hand-side of the assignment is not used further as a right-value.

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 in place. The optimization algorithm, originally proposed by Tirelo and Bigonha [25] and improved by Oliveira *et alii* [20], schedules the rules inside a block in order to minimize the number of insertion in the update list. The scheduling of updates is performed based upon the construction of a so called *graph of conflicts*. Starting from this graph, the objective is to successively remove vertices in the best possible order, generating the code associated with each one. Considering that this is a NP-Complete problem [6, 20], Oliveira *et alii* make use of a heuristic to perform this task achieving good results [20].

3.2 Jump Optimization

According to ASM semantics, the main transition rule R of an agent must be compiled into an infinite loop of the form: `L:` R`; goto L;`. Conditional rules, like `if` g `then` R_1 `else` R_2, are 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 no external function call can be found in g, as well. In this case, once R_2 was executed, it is not necessary to get back to `L` because the reevaluation of g will produce the same value. An optimized code for it could be

`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 execution, with unnecessary reevaluations not being performed at all.

4 The kℓar Environment

One of the most important contributions of kℓar is to provide a common optimization environment where optimizations can be easily added and removed. In order to achieve this feature, it was designed and implemented an environment which provides a well-defined manner of plugging and unplugging optimizations. This section shortly presents its project and implementation decisions. Details can be found in [17].

According to the adopted approach, optimizations are supposed to be developed inside the kℓar framework as *dynamic linkage libraries*. This strategy makes possible to attach optimizations to the pre-compiled core of the kℓar. The core itself was developed in standard C++ [23], summing up almost 30.000 lines of code and defining over than 150 classes.

kℓar belongs to a bigger research project as showed in Figure 4. Lobato developed ACOA, a framework for languages targeting kℓar [16], while Oliveira is concerned with the development of optimizations tailored to the ASM methodology [19].

The general overview of the kℓar framework showing the main class diagram is presented in Figure 1. Some design patterns can be recognized by the inspection of this diagram. These patterns help clarifying the understanding of the kℓar as a whole, therefore they are pointed and explained in the following sections. It is convenient to highlight the KlarLibrary class, since the kℓar framework should be accessed throughout instances of this class. The class Module, present in some methods signatures, is another class of special interest. It represents ASM specifications in the MIR language.

4.1 Design Patterns in kℓar

Design patterns, originally proposed by Gamma *et alii* [11], can be defined as "the description of communicating objects and classes that are customized to solve a general design problem in a particular context". They serve many useful purposes, among them, the understanding of a software design, because they are well-known solutions for recurring problems. Once the presence of a design pattern is identified, the experienced programmer recognizes the main implications of its presence in the behaviour of the program.

4.1.1 Design Pattern State and the class KlarLibrary

The design pattern *State* is manifest in kℓar through the following classes:

- KlarLibrary - It is the pattern context.

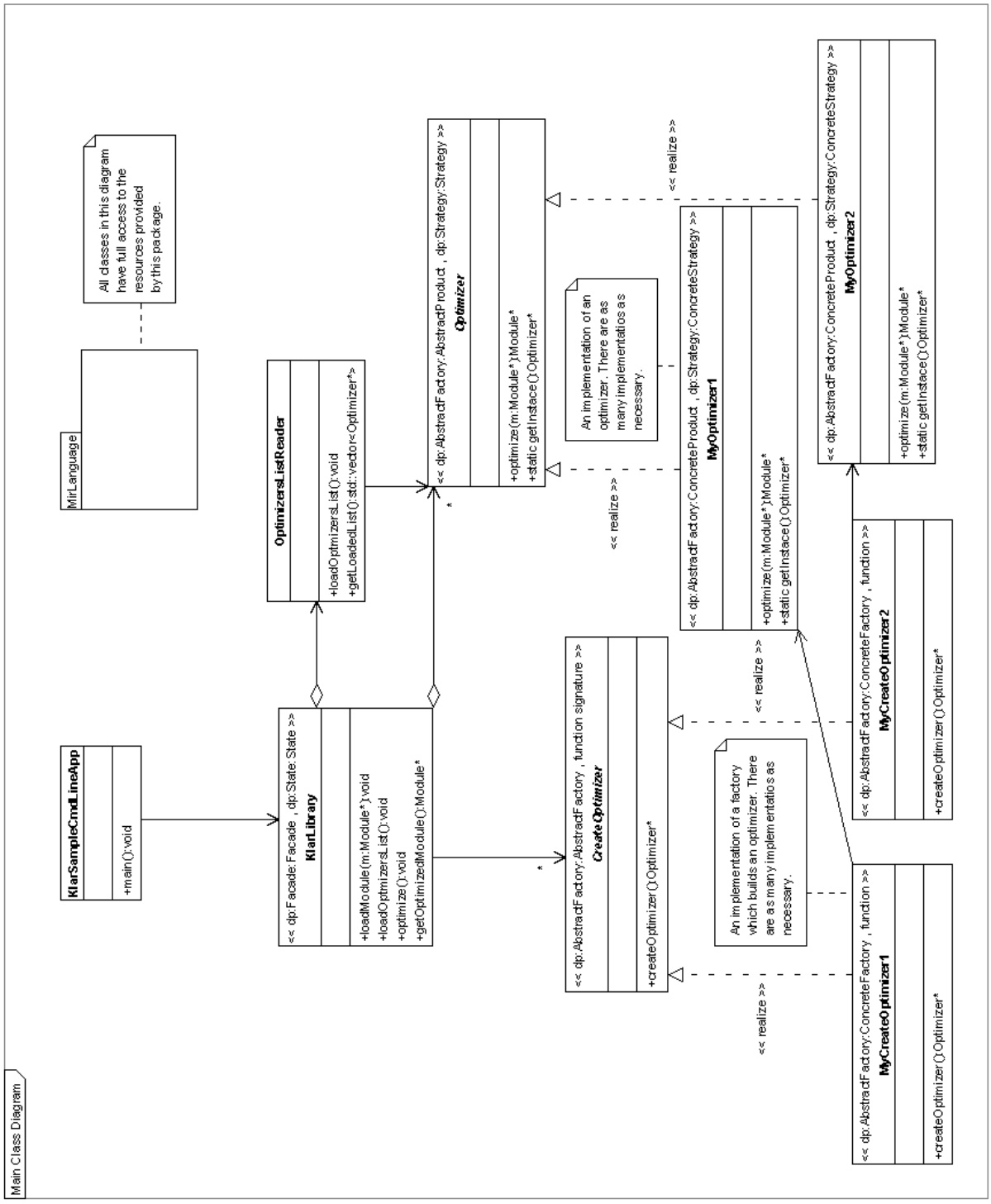

Figure 1: Class diagram: general overview of the kℓar architecture.

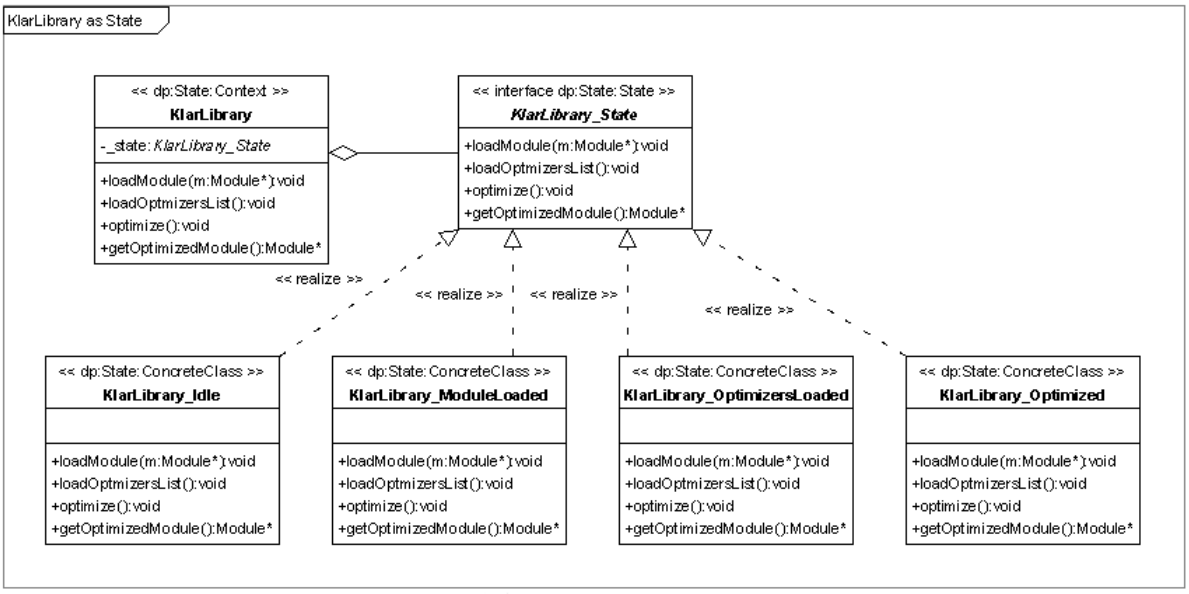

Figure 2: Class diagram of the pattern *State* manifest in kℓar.

- KlarLibrary_State - It is the pattern's state, and it is implemented as an abstract class in C++, in such a way that all of the methods belonging this class are virtual.
- KlarLibrary_Idle - It represents the state before any usage of KlarLibrary. A call to the method loadModule loads a given module written in the MIR language and makes an instance of the class KlarLibrary_ModuleLoaded the current state. Others methods, if invoked, raise exceptions.
- KlarLibrary_ModuleLoaded - It represents the state just after a module is loaded. In this state, the optimizations list can be loaded by means of the method loadOptimizersList, leading to the state KlarLibrary_OptimizersLoaded. If the method loadModule is invoked, the current state remains KlarLibrary_ModuleLoaded, but a new module has been loaded. Other methods, if invoked, raise exceptions.
- KlarLibrary_OptimizersLoaded - It represents the state just after the optimizatons list is loaded. In this state, a loaded module can be optimized by invoking the method optimize, leading to the state KlarLibrary_Optimized. If loadOptimizersList is invoked, the current state remains KlarLibrary_OptimizersLoaded, and the current list of optimizations is the last loaded. If the method loadModule is invoked, the current state is KlarLibrary_ModuleLoaded, this time with a new loaded module. A call to getOptimizedModule raises an exception.
- KlarLibrary_Optimized - It represents the final state of the global optimization of a module. At this point, the natural procedure is to obtain the optimized module through the method getOptimizedModule. Calling all the other methods leads to the correspondent state.

The relationship among the classes is shown in the class diagram of Figure 2.

4.1.2 Design Pattern Abstract Factory and the instantiation of objects from the classes derived from Optimizer

The goal of the design pattern *Abstract Factory* in kℓar is to provide the desired modularity in the optimizations development, yielding to optimizations that are interchangeable and can be applied successively, in any order. The optimizations are dynamically loaded, so when kℓar is built, there is no information about which optimizations will be available at runtime. At that point the pattern Abstract Factory plays its role.

It provides not only the abstract interface for the optimizers, but it additionally provides the abstract interface of the class whose responsibility is to create the appropriated optimizer. This approach includes some compiler-dependent details, specially those involved in the development of dynamic linkage libraries. Figure 1 shows the classes involved in the design pattern Abstract Factory, as well as the relation between them.

- KlarLibrary - It is the client of the factories.
- CreateOptimizer - Its role is to be the abstract factory of the optimizers. In kℓar, it is implemented as a function signature, because it is supposed to be exposed externally to the dynamic linkage library corresponding to the optimizer.
- MyCreateOptimizer1, MyCreateOptimizer2, etc. - They represent functions with the same signature of CreateOptimizer, each one instantiating the appropriated optimizer.
- Optimizer - Abstract class which determines the minimal common interface that every optimizer must supply.
- MyOptimizer1, MyOptimizer2, etc. - These classes represent the several optimizations to be developed for usage inside the kℓar framework.

4.1.3 Design Pattern Facade and the class KlarLibrary

The kℓar framework was conceived to be accessed only by means of the class KlarLibrary, whose objects represent a facade for the other classes, namely those that implement the functionalities of the system. The class diagram presented in Figure 1 illustrates the role of the pattern *Facade* in kℓar. The class KlarSampleCmdLineApp is a sample command line program that makes use of kℓar by means of its facade, the class KlarLibrary.

Although facade classes are in general unique, which could be achieved through the pattern *Singleton*, this is not the case here. Instead, the design decision was made in order to allow the instatiation of many objects from this class. Consequently, an integrated development environment which allows several projects open at the same time could assign one object of the class KlarLibrary for each project, or even to each module belonging to a project. It is important to say that the classes which support the MIR language are not accessed through a facade, being available to direct access.

4.1.4 Design Pattern Singleton and the concrete classes inheriting from Optimizer

kℓar assumes that, between calls to the method optimize, no information is stored in the objects of the concrete classes derived from Optimizer. In other words, these classes do not provide the notion of a state to be preserved or altered between successive calls to the method optimize. These classes exhibit only an functional behaviour. As the optimization list and the order of their application remain the same, a given module passed as argument to the method which performs the optimization must always yield the same result, no matter when it is executed. This justifies the adoption of the design pattern Singleton by the concrete classes derived from Optimizer. The class diagram presented in Figure 1 illustrates the presence of this design pattern in kℓar. It is worth mentioning that every class which inherits from Optimizer must provide a static method instance, allowing the access to the unique instance of the class. Additionally, all of its constructors are supposed to be private.

4.1.5 Design Pattern Strategy and the concrete classes inheriting from Optimizer

The design pattern Strategy is used to define interchangeable algorithms. This design pattern is manifest in kℓar through the classes illustrated in Figure 1, with the class KlarLibrary playing the role of the Context, the class Optimizer being the Strategy itself and the concrete classes that derive from Optimizer being the implementation of the algorithms, more specifically, the optimizations algorithms. Although Strategy is commonly employed to *commute* mutually exclusive algorithms, in the kℓar framework it is used to *successively apply* different algorithms, in order to allow that the final behaviour is the cascading of all algorithms involved.

4.1.6 Other Design Patterns

The design patterns *Composite* and *Visitor* play an important role in the kℓar framework. They are extensively used in the set of classes developed in order to provide support to the usage of the MIR language.

4.2 Kinds of Optimizations

According to their nature, the optimizations are classified in the context of the kℓar framework into two categories: *embedded* optimizations and *plugged* optimizations.

4.2.1 Embedded Optimizations

The expression *embedded optimizations* refers to those ones which modify the way specifications written in the MIR language are translated into equivalent C++ code. These optimizations do not act upon specifications transforming them into other, more efficient specifications whose semantics remains the same. Instead, they provide a new mapping from ASM constructions into C++ code elements.

Dynamic functions inside ASM specifications are translated into corresponding hashtables in the C++ code. In the available implemetantion of the kℓar framework, the class std::map, belonging to the Standard Template Library of C++, is employed with this purpose, since it is an efficient implementation of a hashtable. Now suppose that another data structure is found more efficient to represent dynamic functions, or even a new implementation of a hashtable offering advantages over the standard implementation is made available. In this case, it would be necessary to change the way a dynamic function is translated into C++ code. Further away, accesses to points in the dynamic funtion domain could be done differently in C++, both as right-values and left-values. This change has the purpose to obtain a more efficient final code, but this is transparent to the ASM specification itself. This is the reason why they are called embedded optimizations.

The classes that represent modules of the MIR language were structured according to the pattern *Composite*, and this design, together with the code generation by a *Visitor*, makes it easy to cope with the task of changing the way a module is translated into C++. The implementation of these optimizations is achieved inheriting from the class CodeGenerator, which is the visitor responsible to generate C++ code. Next, the methods those are responsible for the translations that should be changed must be overwritten. In the dynamic function example, these methods should be:

- visitDynamicFunction(DynamicFunction *df), which performs the translation of a dynamic function into its actual C++ entity;
- visitUpdate(Update *u) and visitImmediateUpdate(ImmediateUpdate *u), which make use of dynamic functions as left-values;
- and finally visitDynamicFunctionCall(DynamicFunctionCall *fc), which makes use of dynamic functions as right-values;

All the other methods remain unchanged. Cascading changes are achieved by means of successively derivations of the changing classes.

4.2.2 Plugged Optimizations

Plugged optimizations act *transforming* a given ASM specification. The result of such optimizations is another ASM specification that preserves the original semantics, but at the same time it is more efficient according to some aspect. Examples of optimizations that fall into this category were presented in Section 3

In order to develop and configure a plugged optimization it is necessary to obey a specific protocol, to be more exactly, the steps that should be followed are:

1. Implements the class that encapsulates the optimization. This class can have any name, provided that it inherits from klar::Optimizer. This will force the implementation of the method optimize(Module*), which is pure virtual in the base class. This is the method called by the framework in the moment the optimizations are requested, and it receives as argument the module to be optimized, returning the optimized one.

2. Now it is necessary to provide the right factory for the optimizer developed, as reported in Section 4.1.2. In this case, the factory is not implemented as a class; instead, it is a function with name and signature Optimizer *CreateOptimizer(), exposed externally by means of a external "C" modifier.

3. Compile the optimizer with the appropriated options. This compilation should yield a dynamic linkage library, exposing the function Optimizer *CreateOptimizer(), used as the factory of instances of the optimizer.

4. Add an entry to the configuration file `optimizers.cfg`, which groups together the list of optimizations to be applied and useful information about them, like the order in which they are supposed to be applied and where they should be found. This file follows a XML-like syntax, given in Figure 3.

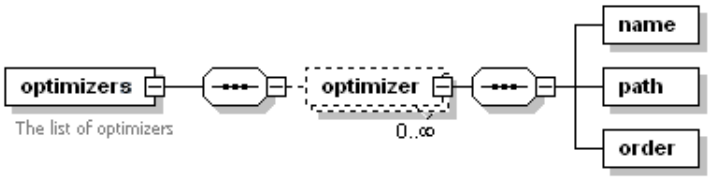

Figure 3: Sintax of the configuration file `optimizers.cfg`.

When the optimization proccess starts, the optimizations listed by the configuration file are loaded and then successively applied to the module to be optimized. The input of optimization n is the output of the optimization $n-1$. The output of the last optimization is the global result of the optimization proccess.

5 Validation and Results

In order to test the kℓar framework, it was developed some sample programs that exercise every construction of MIR, covering all the language. These examples are successfuly built and they behaved as expected.

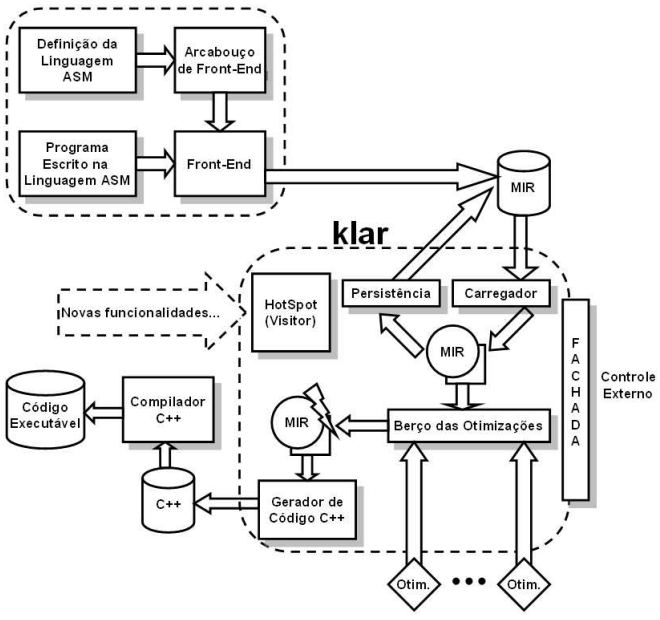

Figure 4: The kℓar used as a target for ASM compilers.

Additionally, it is used as a back-end for an ASM compiler, as depicted in Figure 4.

To validate the ease of optimizations development in kℓar, an optimization was implemented and then plugged into the framework. The chosen optimization was the update scheduling [20, 25], presented in Section 3.1. Figure 5 depicts the class diagram of this implementation. As argued before, the development of optimizations in kℓar is eased by the infrastructure provided by the framework. In the example showed in Figure 5, the optimization was implemented following these steps:

1. The class ImmediateUpdateOptimizer was declared as a subclass of Optimizer. The obvious implementation of the method instance was provided.
2. The method optimize was implemented. This method makes use of the DefUseVisitor, which is provided by the framework. This visitor supplies the information about dynamic functions usage along a rule, and this information is used in order to perform the immediate update optimization. Additionally, the optimization algorithm is graph based, so a Graph class was developed.
3. Finally, an entry was added to the configuration file.

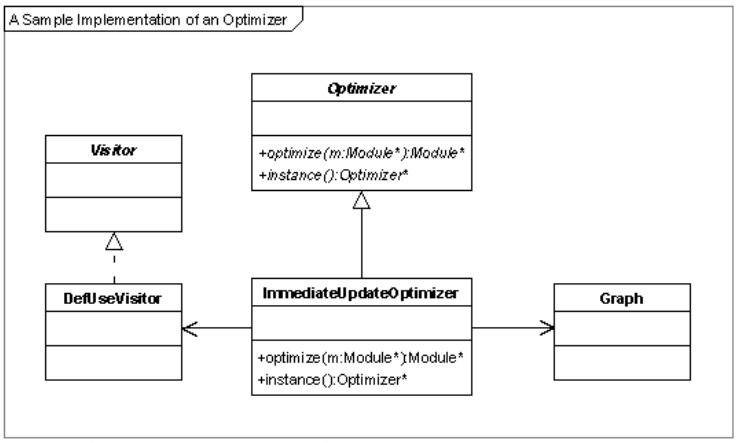

Figure 5: Class diagram of an optimizer implemented for usage in kℓar.

The implementation work is eased by means of the following resources offered by the framework:

1. The analysis of use and definition of values of the dynamic function points is made by the DefUseVisitor, which is part of the framework. There is no need to concern about the details involved.
2. A Graph class is given by the framework, as many optimizations are based on graph modeling. This class is used by the implemented optimizer.
3. The configuration of the optimization is defined by a simple file entry.
4. The classes for representation of programs in MIR are available, so it is easy to understand and to change programs to be optimized.
5. In order to add the optimization to the framework, there is no need to understand the internal details of the framework itself, just the simple interface of the optimizers.

This optimization was successfully developed and configured, leading to a performance improvement from 8% up to 15% [17], as depicted by the benchmark of Table 1. Basically, this rate depends on several factors, among them how many updates instructions are executed comparing to non-updates ones, as this optimization just acts over update instructions.

The implementation of this optimization has allowed us to conclude about the overall quality of the framework regarding to the commitment with the easiness of the development of optimizations. First of

SelSort

Problem Size	1000	2000	4000	8000	16000
Non-Optimized	0.975 s	3.891 s	15.248 s	1m03.717 s	4m22.785 s
Optimized	0.872 s	3.363 s	13.406 s	56.873 s	3m53.961 s
Percentual Improvement	10.56	13.56	12.07	10.74	10.96

Fibonacci

Problem Size	10e3	20e3	40e3	80e3	16e3
Non-Optimized	0.04 s	0.076 s	0.154 s	0.308 s	0.616 s
Optimized	0.038 s	0.071 s	0.143 s	0.292 s	0.585 s
Percentual Improvement	5	6.57	7.14	5.19	5.03

Counting

Problem Size	2e6	4e6	6e6	8e6	10e6
Non-Optimized	4.107	8.214	12.313	16.427	20.506
Optimized	3.457	6.912	10.353	13.798	17.257
Percentual Improvement	15.82	15.85	15.91	16.00	15.84

Table 1: A small *benchmark* that measures the impact of the implemented optimization.

all, the use of the framework is facilitated by its availability through a facade that allows the control of the optimization proccess using just four methods. The interface requirements to be fulfilled by plugged optimizers are quite simple, as explained in Section 4.1.2 This simplicity does not limit the generality and flexibility of the framework. Every optimization that acts transforming ASM programs can be implemented as plugged optimization, and optimizations that change the way ASM programs are compiled can be implemented as embedded optimizations. Other techniques that work by code transformation, like refactoring, can be plugged into the framework, provided that the plug-in mechanism is general enough. The inheritance can be also used to implement optimizations based upon those already developed, which makes the tunning of optimizations more flexible.

6 Conclusions and Further Work

kℓar facilitates the development of optimizations tailored to the ASM methodology. As presented in this paper, this framework provides an infrastructure to support of the development of such optimizations. Further away, kℓar defines a standard that these optimizations are supposed to follow in order to be properly used inside its context.

The classes that support the usage of the ASM-oriented MIR language as well as the code generation capability make the framework eligible as a target for compilers aiming the ASM methodology. A further work concerning this question is the development of *visual* tools to ease the task of programming in that language.

A sample optimization was implemented in order to illustrate how easy the development and configuration of such optimizations inside the kℓar framework is, as depicted in Section 5. It provided a performance gain to specifications optimized through it when compared with the non-optimized versions. As the results present, this optimization does not overlap with the customary C++ code optimizations. Although this implementation, the matter of discovering and implementing specific ASM-oriented optimizations is far away from being tottaly fullfilled; on the contrary, there is a vast, unknown field to be explored. We believe that kℓar is valuable tool that can be used to perform such research. As the result, we expect that the power of the ASM methodology for giving formal, operational semantics for algorithms will be bound to the automatic generation of executable code that fulfills the efficiency requirements of production code.

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., 2005.

[3] R. S. Bigonha, F. Tirelo, V. O. Di Iorio, and M. A. S. Bigonha. A linguagem de especificação formal machina 2.0. Technical Report LLP001/2005, UFMG, 2005.

[4] R. S. Bigonha, F. Tirelo, M. A. Maia, M. Silva, M. T. O. Valente, M. A. S. Bigonha, and V. O. Di Iorio. Projeto Machina. Technical Report LLP007/99, UFMG, 1999.

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

[6] T. H. Cormem, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw Hill, 1990.

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

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

[9] G. Del Castillo, I. Durdanović, and U. Glässer. An Evolving Algebra Abstract Machine. In H. Kleine Büning, editor, Proceedings of CSL'95, volume 1092 of *LNCS*, pages 191–214. Springer, 1996.

[10] D. Diesen. *Specifying Algorithms Using Evolving Algebra. Implementation of Functional Programming Languages*. PhD thesis, Dept. of Informatics, Univ. of Oslo, 1995.

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

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

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

[14] A. M. Kappel. Executable Specifications Based on Dynamic Algebras. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 698. Springer, 1993.

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

[16] M. C. C. Lobato. Proposta de Dissertação: Um Arcabouço para Compilação de Linguagens de Especificação ASM, 2005.

[17] K. Magnani. klar: Um Arcabouço para Otimizações em Máquinas de Estado Abstratas. Master's thesis, DCC, UFMG, 2006.

[18] K. Magnani, M. A. S. Bigonha, R. S. Bigonha, V. O. Di Iorio, and F. F. Oliveira. An Infrastructure for Implementing Compilers for Abstract State Machines. In *31th Latin American Conference on Informatics*, 2005.

[19] F. F. Oliveira. Proposta de Tese: Otimização de Código em Ambiente de Semântica Formal Executável Baseado em ASM, 2004.

[20] F. F. Oliveira, R. S. Bigonha, and M. A. S. Bigonha. Otimização de Código em Ambiente de Semântica Formal Executável Baseado em ASM. *Proc. of 8th SBLP*, May 2004.

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

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

[23] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 2000.

[24] F. Tirelo. Uma ferramenta para execução de um sistema dinâmico discreto baseado em Álgebras evolutivas. Master's thesis, UFMG, 2000.

[25] F. Tirelo and R. S. Bigonha. Técnicas de Otimização de Programas Baseados em Máquinas de Estado Abstratas. *Proceedings of 4th SBLP*, 2000.

[26] J. Visser. Evolving algebras. Master's thesis, Faculty of Technical Mathematics and Informatics, Delft University of Technology, 1996.