UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

EMILENA SPECHT

An Approach for Embedded Software Generation Based in Declarative Alloy Models

Thesis presented in partial fulfillment of the requirements for the degree of Master in Computer Science.

Prof. Dr. Luigi Carro
Advisor

Porto Alegre, October 2008.

CIP – CATALOGING-IN-PUBLICATION

Specht, Emilena

An Approach for Embedded Software Generation Based in Declarative Alloy Models / Emilena Specht – Porto Alegre: PPGC UFRGS, 2008.

66 f.:il.

Thesis (MSc.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2008. Advisor: Luigi Carro.

1. Embedded systems. 2. Embedded software. 3. System level modeling and synthesis. 4. Declarative languages. 5. Design space exploration. 6. Code generation. I. Carro, Luigi. II. Título.

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, for all the help and patience dedicated to me since I was an undergraduate student. I can say that he gave me the opportunity to learn more than I would ever realize during my academic journey.

Also I would like to thank my boyfriend and my family, for all the motivation and support.

A special thanks to the undergraduate students Ronaldo, who implemented the compiler, and Crístofer, who implemented applications in Esterel. They were the great contributors of this research.

I am deeply grateful to Julius, Lisane, Ricardo Redin and Caco for their contribution, and other friends in LSE for the happy hours spent together.

Thanks to the professors Flávio Wagner, Erika Cota and Luís Lamb, for participating in the research and giving valuable advises, and to the professors that have accepted to be members of my dissertation board.

Thanks for all of those that helped me somehow, and unfortunately I forgot to mention here.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

AAL	Alloy Annotation Language
CASE	Computer Aided Software Engineering
CoS	Coin Storage
DrS	Drink Storage
DSE	Design Space Exploration
FSM	Finite State Machine
GUI	Graphical User Interface
IMDCT	Inverse Modified Discrete Cosine Transform
LoC	Lines of Code
MDA	Model Driven Architecture
MIC	Model-Integrated Computing
MIT	Massachusetts Institute of Technology
MoC	Models of Computation
OCL	Object Constraint Language
OO	Object Orientation
Sig	Alloy signature
UML	Unified Modeling Language
V&V	Validation and Verification
VM	Vending Machine

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This work proposes a new approach for embedded software development, by combining the abstraction and model verification properties of the Alloy declarative language with the broad acceptance in industry of Java. The approach comes into play since software automation in the embedded domain has become a major need, as currently most of the development time is spent designing software for such hard-constrained resources products. Design automation tools for embedded systems must meet the demand for productivity and maintainability, but constraints such as memory, power and performance must still be considered.

Design automation tools deal with productivity and maintainability by allowing high-level specifications, which is hard to accomplish on the embedded domain due to the mixed behavior nature of many embedded applications. Approaches that provide means for formal verification are also attractive, but their usage is usually not straightforward, and for this reason they are not that helpful in dealing with time-to-market constraints. By using Alloy, based in first-order logic, it is possible to obtain high-level specifications and formal model verification with a single language.

This work shows the powerful abstraction provided by the Alloy language for embedded applications, as well as rules for obtaining automatically Java code from Alloy models. The Java source code generation from Alloy models, combined with an estimation tool, provides design space exploration to match tight embedded software design constraints, what is usually not taken into account by standard software engineering techniques.

Keywords: Embedded systems, embedded Software, system level modeling and synthesis, declarative languages, design space exploration, code generation.

Uma Abordagem para Geração de Software Embarcado Baseada em Modelos Declarativos Alloy

RESUMO

Este trabalho propõe uma nova abordagem para o desenvolvimento de sistemas embarcados, através da combinação da abstração e propriedades de verificação de modelos da linguagem declarativa Alloy com a ampla aceitação de Java na indústria. A abordagem surge no contexto de que a automação de software no domínio embarcado tornou-se extremamente necessária, uma vez que atualmente a maior parte do tempo de desenvolvimento é gasta no projeto de software de produtos tão restritos em termos de recursos. As ferramentas de automação de software embarcado devem atender a demanda por produtividade e manutenibilidade, mas respeitar restrições naturais deste tipo de sistema, tais como espaço de memória, potência e desempenho.

As ferramentas de automação de projeto lidam com produtividade e manutenibilidade ao permitir especificações de alto nível, tarefa difícil de atender no domínio embarcado devido ao comportamento misto de muitas aplicações embarcadas. Abordagens que promovem meios para verificação formal também são atrativas, embora geralmente sejam difíceis de usar, e por este motivo não são de grande auxílio na tarefa de reduzir o tempo de chegada ao mercado do produto. Através do uso de Alloy, baseada em lógica de primeira-ordem, é possível obter especificações em alto-nível e verificação formal de modelos com uma única linguagem.

Este trabalho apresenta a poderosa abstração proporcionada pela linguagem Alloy em aplicações embarcadas, assim como regras para obter automaticamente código Java a partir de modelos Alloy. A geração de código Java a partir de modelos Alloy, combinada a uma ferramenta de estimativa, provê exploração de espaço de projeto, atendendo assim as fortes restrições do projeto de software embarcado, o que normalmente não é contemplado pela engenharia de software tradicional.

Palavras-Chave: Sistemas embarcados, software embarcado, modelagem e síntese em nível de sistema, linguagens declarativas, exploração de espaço de projeto, geração de código.

1 INTRODUCTION

Embedded systems are present in our daily life more than most people realize. According to (NOKIA 2006), by 2010 there will be about 4 billion cell phones operating around the world. In fact, cell phones, which are one of the most important embedded systems nowadays, offer many features besides telephone services, such as digital cameras, organizers, games, internet access, MP3 players, radio, etc. These several features increase design complexity, while the current reduced time-to-market windows stress the embedded system design process.

The embedded systems domain is driven by reliability, cost, and time-to-market factors (GRAAF; LORMANS; TOETENEL, 2003). In current applications, software is responsible for the major bottleneck, and in many cases software solutions are preferred to hardware ones, because software is more flexible, easier to update and can be reused (GRAAF; LORMANS; TOETENEL, 2003). Industry statistics from (MURRAY, 2005) reveal that lines of embedded software code, which are considered by (KAN, 2002) as an indicative of software complexity, are growing at about 26 percent annually. Due to this growing complexity and short time-to-market windows, statistics from (LAPEDUS, 2006) show that more than one half of current embedded design projects are running behind schedule.

To cope with the increased complexity, developers have been looking for higher levels of abstraction during system specification. Current research on embedded systems design emphasizes that the use of techniques starting from higher abstraction levels is crucial to the design success. Some authors like (SELIC, 2003) and (GOMAA, 2000) argue that this approach is the only viable way for coping with the complexity that is found in the new generations of embedded systems. To deal with tight time-to-market, automatic synthesis tools are required to generate code from high-level models.

In software engineering, CASE (*Computer Aided Software Engineering*) tools generate source code from high-level models, and for this reason they are widely used to automate the software development process. However, such tools are often limited for a specific application domain, therefore, they cannot be used for embedded software development, which usually includes multiple domains. A mobile phone specification, for example, requires not only signal processing for telecommunication domain, applicable to the discrete-time model of computation (MoC) (LEE, 2002). It also requires sequential logic to describe other applications embedded in the mobiles that were previously mentioned (such as organizers and games). In fact, embedded systems are naturally heterogeneous, and consequently, automation tools should support different MoCs. Nevertheless, most of the existing software automation tools do not have this feature. Even tools like ForSyDe (SANDER; JANTSCH, 2004), specific for embedded software synthesis, usually do not provide the desired abstraction.

Besides all the challenges introduced from the multiple domain nature of the embedded applications, the embedded software development requires support for some specific domain requirements, such as real-time operation, energy consumption and limited memory footprint. Software designers are still in the quest for the best trade-off between area and performance, and embedded software is often still written in assembly.

Tools and conventional software engineering practices cannot be directly applied to the embedded software domain without considering the specific requirements of the embedded product. Code reuse, for example, is indispensable for reducing development time in conventional software design. On this point of view, reuse would be very useful also in the embedded software development. However, a code developed for reuse is usually generic, and for this reason, more memory and execution time consuming. Consequently, reusable code is usually not suited for the embedded domain.

Conventional and embedded software are different in many aspects, as shown in Table 1.1. Due to the tight constraints, the embedded systems domain requires customized design processes and development tools, which are usually not considered by standard methodologies. The use of high-level tools that provide design space exploration in embedded software development is somewhat mandatory, to combine high abstraction with a certain platform control, in order to ensure that the embedded equipment will make good use of its resources. Consolidated methodologies, like Object Orientation (OO), introduce a delay in the embedded software development. Larger codes are produced when OO is used, requiring more data and program memories and also increasing application execution time (CHATZIGEORGIOU; STEPHANIDES, 2002; MATTOS et al., 2005).

Table 1.1: Comparative between conventional and embedded software

	Conventional Software	**Embedded Software**
Time-to-market	Long	Short
Memory	No constraints	Constrained
Power	No constraints	Constrained
Application Domain	Single	Multiple
Code Reuse	Nice	Often leads to Overheads
Code Optimization	Dispensable	Necessary
Automation	Considerable	Poor

Embedded systems are also often used in life-critical situations (KOOPMAN, 2007), where reliability and safety are more important criteria than pure performance. This fact points to the need of a careful support for validation and verification (V&V), from the requirement analysis to the developed software test. Software quality is a prior requirement in the embedded software domain (WOODWARD; MOSTERMAN, 2007). As V&V techniques are expensive and may represent 80% of the total cost of the software development in safety-critical systems (KOOPMAN, 2007), it is necessary to find a trade-off among quality, cost and development time (SOMMERVILLE, 2006).

Formal Techniques are suited for safety-critical systems because they provide a better failure coverage, and as well as for conventional systems, they provide means for

finding errors early in a design process. Tools that offer formal verification for designers and do not imply in an additional increase in development time are valuable contributions. However, tools for these checking activities are often not used in current embedded software development, usually because of the cost and extra complexity induced by their use. In another aspect, declarative languages are known for their well-defined semantics, for providing natural tools for program correctness and allowing implementations in a more natural fashion (HUDAK, 1989).

1.1 Main Goal

The main goal of this research, as shown in Figure 1.1, is the proposal of a new approach able to:

- Allow specification of embedded applications using a declarative language that provides a high abstraction level;
- From the declarative code, automatically generate its correspondent imperative code, suited for the available processor resources. This implies in the development of a high-level design space exploration (DSE) tool together with a code generator;
- Use tools and available implementations of the FemtoJava processor (ITO; CARRO; JACOBI, 2001), developed in the same research group. This implies in using Java, which could be easily adapted to run in FemtoJava processors, as the target imperative language of the code generator.

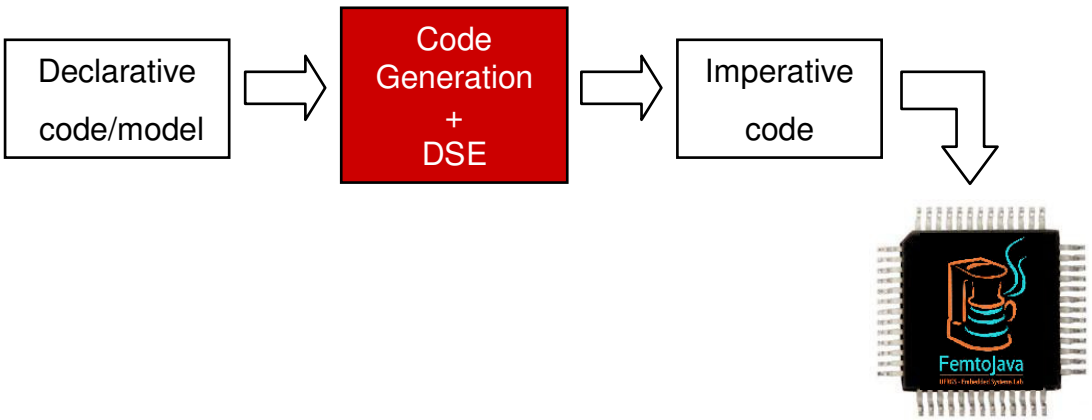

Figure 1.1: Main goal: Generate dedicate Java code from a declarative specification

1.2 Methodology

In order to achieve the research goal, this work presents a study regarding the abstraction power of some general-purpose declarative languages, together with the impact in performance and memory induced by their use. Such task aims to check if the regular declarative languages provide more abstraction than imperative languages concerning applications that use different MoCs. Also, the possibility of improving

memory usage and performance by generating imperative code from declarative code is evaluated.

This work also proposes a new approach for embedded software generation, which provides high abstraction and formal verification together with automation. Alloy (MIT SOFTWARE DESIGN GROUP, 2008), a declarative modeling language empowered by lightweight formal verification tools, is used to accomplish this task. Some translation rules that were used to build an Alloy-to-Java compiler in the research group are also shown. The compiler developed in the group analyzes the verified high-level specification and automatically generates Java source code from Alloy models. Experimental results show the gains obtained in code size and overall quality, benefiting the design process as a whole.

1.3 Text Organization

The work is organized as follows. Section 2 discusses the related work on declarative languages usage, software automation and design space exploration for embedded systems. Section 3 presents a study with some declarative languages and their use in embedded system applications. Section 4 presents the Alloy language and explains the main motivations for its adoption. Section 5 introduces the proposed approach for embedded software development automation, and shows case studies that are further used to demonstrate the feasibility of the approach. Section 6 presents the translation rules and algorithms for software synthesis from Alloy models. Section 7 shows experimental results for software synthesis and design space exploration. Section 8 presents a brief analysis on the proposed approach, and Section 9 draws main conclusions and discusses future works.

2 RELATED WORK

This section presents the work related to this research. It includes comparison between imperative and declarative paradigms, usage of declarative languages in embedded systems development, automatic software generation for conventional and embedded systems, and finally, other approaches based in Alloy models.

2.1 Declarative Versus Imperative for Embedded Systems

Comparisons between declarative and imperative paradigms are restricted to certain algorithms or models of computation. (OKASAKI, 1999) and (GIEGERICH; KURTZ, 1995) compare functional and imperative implementations through red-black trees and suffix tree constructions, respectively. (STANSIFER, 1989) uses some benchmarks (sorting and puzzles solutions) to compare functional and imperative paradigms, but the comparison considers only performance. (WADLER, 1999) gives some hints comparing both paradigms, but gives no numbers.

There are various reports on applications for embedded systems described in functional languages, a type of declarative languages, such as (ARMSTRONG et al., 1993). These works present case studies, extensions of known functional languages, or even propose new languages. However, these proposals are suited for only one model of computation in the embedded systems domain: (WALLACE; RUNCIMAN, 1995) proposes extensions to handle reactive systems, the Lustre language (CASPI et al., 1987) is suitable for synchronous-dataflow applications, and (ARMSTRONG et al., 1995) targets at very specific real-time applications, mainly used in telecom systems.

In (CARRO et al., 2006) the authors implemented a concrete case study to demonstrate the feasibility of using a high-level, general-purpose logic language in the design and implementation of an application targeting wearable computers. The case study has tight embedded system constraints and was implemented in a high-level language without efficiency or architectural concerns. The compile-time optimizations and native code transformations made it possible to execute the code in the embedded device without high-level code modifications.

2.2 Automatic Software Generation

There are many academic and commercial solutions for software automation. Most of them focus on the management of huge domain specific systems, such as databases (XDOCLET TEAM, 2004) and web-based ones (JGENERATOR TEAM, 2008), because conventional software is usually suited for a particular domain. Tools for conventional software automation aim at minimizing complexity issues arising from the

large size of the software project, instead of optimizing software for hardware constrained device.

Many UML-based software automation tools are available, such as Poseidon for UML (GENTLEWARE SOFTWARE, 2008), Rational Rose (RATIONAL SOFTWARE, 2008), UniMod (EXECUTABLE UML UNIMOD, 2008), Rhapsody (TELELOGIC, 2008), which extract code from UML diagrams and follow the Model Driven Architecture (MDA) approach (OBJECT MANAGEMENT GROUP, 2000). MDA encourages the use of models in software development, and allows the functionality and behavior of the system to be separated from implementation details.

Some of the UML-based tools may be considered partial generators, as they are capable of generating only the structural code for applications from structural diagrams (e.g. class and deployment diagrams). Other tools, such as CodeGenie (DOMAIN SOLUTIONS, 2008), generate the structural code plus architectural data structures (hash tables, lists, queues, etc), and a restricted amount of research tools can generate behavioral code from a combination of UML diagrams, like Rhapsody (TELELOGIC, 2008).

The UML-based automation tools here discussed can provide great abstraction to the designer, but none of them embodies formal verification of the modeled solution. Besides, most of them have been developed before the definition of a precise semantics for UML (OBJECT MANAGEMENT GROUP INC, 2003), and they use their own *ad hoc* semantics. A formal semantics for the source language is essential in software automation to avoid ambiguous interpretations.

2.3 Embedded Software Modeling and Synthesis

Embedded software synthesis can be compared to the target code generation (in Java, C or assembly) from synchronous languages such as Esterel (BERRY; GONTHIER, 1992), and Lustre (CASPI et al., 1987). However, these programming languages are not designed to be used as specification languages, but indeed represent an improvement on standard imperative languages for implementing reactive systems. To provide graphical specification, StateCharts (HAREL, 1987) was proposed as an extension of state machines for the specification of discrete event systems. Although these approaches have a formal model, and provide some mechanisms for formal verification, these languages do not provide high abstraction in comparison to traditional programming languages.

Frameworks and environments for embedded system modeling have also been proposed, such as the commercial Simulink (THE MATHWORKS, 2008) and Scade (ESTEREL TECHNOLOGIES, 2008), and the academic POLIS (BALARIN et al., 1999), Metropolis (SANGIOVANNI-VINCENTELLI, 2007), ForSyDe (SANDER; JANTSCH, 2004) and Ptolemy II (BROOKS et al., 2007).

Simulink (THE MATHWORKS, 2008-c) is a graphical environment for dynamic and embedded system design and simulation. It provides a customizable set of block libraries that allows modeling and simulating a variety of time-varying systems, and also supports algorithms developed using Matlab (THE MATHWORKS, 2008-a). Add-on products that extend Simulink software include tools for supporting multiple domains, code generation, testing, verification and validation tasks. One example is the

Real-Time Workshop tool (THE MATHWORKS, 2008-b), that generates C code from Simulink models and MATLAB code.

Scade (ESTEREL TECHNOLOGIES, 2008) is also a graphical environment, for modeling, simulating and synthesizing embedded software. It is suited for safety-critical applications, and its code generator is qualifiable for DO-178B, which provides guidelines for the production of software for airborne systems. Scade provides block libraries like Simulink, and supports discrete dataflow and FSM modeling, however, the code generation from FSM and the block libraries is not qualifiable for DO-178B. Scade may be integrated with tools like Reqtify (GEENSYS, 2008), which provide requirement tracing and coverage analysis.

POLIS (BALARIN et al., 1999) provides code generation based on the model of Concurrent Finite State Machines (CFSM) created by the authors. The formal specification embedded within POLIS enables to interface directly with existing formal verification algorithms based on finite state machines (FSM). However, the set of applications generated and the scheduler (specifically generated for the set of tasks) compounds a static set; hence no dynamic behavior is possible. Moreover, regarding to specification power, it does not allow the specification of data structures.

Metropolis (SANGIOVANNI-VINCENTELLI, 2007) was proposed as an extension of Polis. The Metropolis MetaModel specification language is based on process networks and is targeted for system-level specification. Two verification techniques are implemented in Metropolis: the formal verification using the model checker SPIN (HOLZMANN, 1997) and the simulation trace checking (CHEN et al., 2006). When SPIN is used, a Promela (HOLZMANN, 1997) specification is generated from a system level specification. SPIN is an automata-based model checker, which verifies the Promela model for correctness by performing random or iterative simulations or even generates a C program that performs a fast exhaustive verification of the system state space. Unfortunately, these formal verification techniques usually represent a boost in quality at the price of extra development time, something that cannot be afforded in current stringent time-to-market needs.

ForSyDe (SANDER; JANTSCH, 2004) is a framework for modeling and synthesizing embedded systems. Specifications in this framework are modeled in Haskell, to obtain high-level abstraction and formal semantics to the model. To provide abstraction for many domains, ForSyDe requires knowledge and classification of several different models of computation, what may increase development time. Moreover, the abstraction provided by Haskell is similar to OCaml, being close to standard programming languages.

The Ptolemy II (BROOKS et al., 2007) is a Java-based component assembly framework with a graphical user interface, which provides simulation and prototyping for heterogeneous systems. Ptolemy II includes a growing suite of domains, each of which realizes a model of computation. It also includes a component library, in which most components are domain polymorphic, in that they can operate in several of the domains. Despite of the appealing heterogeneity support, the Ptolemy II framework does not provide tools for formal verification.

Model-Integrated Computing (MIC), an approach that is acquiring many adepts in the embedded systems community, fully adopts the model-based development paradigm (SZTIPANOVITS; KARSAI, 1997). Tools that follow MIC present rich environments for modeling, analyzing and synthesizing embedded systems, considering the

integration of their hardware, software and environments. However, the approach relies on Domain-specific modeling languages (KARSAI et al., 2003), which are still not appropriate for applications that mix models of computation.

Most of the current design exploration tools in embedded system development explore parallelism and provide flexibility at the same time, by taking advantage of the availability of reconfigurable processors and MPSoCs (BENINI et al., 2005). Some other approaches like the Real Time Workshop (THE MATHWORKS, 2008-b) and DESEJOS tool (MATTOS; CARRO, 2007) allow design space exploration based on the reuse of commonly used algorithms previously implemented. The issue in these approaches is that libraries must be restricted to some algorithms, and only some kinds of applications are supported in a fixed language. Besides, the effort to maintain a library up to date usually leads to discontinuous works.

2.4 Usage of the Alloy Declarative Language

Some academic works have proposed the use of Alloy for formal verification of programs or system specifications. (MASSONI; GHEYI; BORBA, 2004) proposes an approach that transforms UML class diagrams with OCL constraints to Alloy code in order to verify class diagrams. As behavioral diagrams are not analyzed, this approach only verifies the system structure.

Recently, a method has been proposed for translating sequential Java programs to DynAlloy in order to formally verify Java programs and its dynamic properties (GALEOTTI; FRIAS, 2007). DynAlloy checks pre and post conditions of a behavior by extending the Alloy language and (GALEOTTI; FRIAS, 2007) checks if these conditions are reflected on the Java code by translating it into DynAlloy. Another approach uses an annotation language based on Alloy for annotating Java code, allowing a fully automatic compile-time analysis (KHURSHID; MARINOV; JACKSON, 2002).

Although these works also propose the use of Alloy for formal verification, they start with Java programs, and hence miss the higher abstraction mechanism that is required to produce more code with less describing effort for complex embedded software specification. Differently from these approaches, we propose to automatically generate Java codes from verified Alloy models.

3 ANALYSIS OF THE USE OF DECLARATIVE LANGUAGES FOR EMBEDDED SOFTWARE DEVELOPMENT

Declarative languages, as they provide more abstraction than traditional languages like Java, could be used to code embedded system applications in order to deal with the increased demand for programmability in this area. Comparative results of the use of declarative languages to describe embedded applications are presented. The MP3* was designed, an embedded application containing the IMDCT algorithm together with an Address Book, and the games Sokoban and Tic-tac-toe. All the applications were coded in Ocaml and Prolog to analyze the resulting abstraction and performance, and then compared to the Java equivalent codes. For some applications, a comparison with Esterel v. 5.21 (BERRY, 2008) was also provided. The main objective of this study is the analysis of the abstraction level achieved with the shift from the imperative programming paradigm to the declarative paradigm, considering its impact in terms of performance and memory in the embedded systems domain.

3.1 Comparison Methodology: The MP3* Application

The methodology of this study includes the implementation of the MP3* application. The MP3* is composed by four small applications: an IMDCT algorithm (an essential part of an MP3 player), an Address Book and Sokoban and Tic-tac-toe games. The MP3* was coded in Java, Ocaml, and Prolog. A part of the MP3* (the Address Book and Tic-tac-toe) was also coded in Esterel, to compare the abstraction achieved in general-purpose languages with the abstraction achieved by using an embedded system language.

Java was the imperative language chosen for the applications implementation because it provides abstraction through its APIs and object-oriented style and its use is spread in the embedded systems community (STROM et al., 2003). The concepts here presented could also be applied to other languages like C. However, the comparison with declarative languages would not be fair in terms of abstraction. Ocaml, the main implementation of the Caml language, was the functional language chosen for its powerful module system and widespread use. Prolog is the main example of logic languages.

Some components that are part of the MP3* (or similar components) can be often found in embedded devices, since they involve different behaviors and, consequently,

different models of computation. The Tic-tac-toe, with the system playing with the user, and Sokoban games include scalar data processing, where the system must compute the user inputs and choose the best move. Also, the control behavior is present in the sequencing of plays and user inputs. The Address Book behavior is mostly control-flow with some dataflow, since the manipulation of the data structure after some user input predominates. Besides, the user options suggest a reactive control behavior. The IMDCT algorithm, a typical stream data processing, can be considered mostly dataflow.

The implementations in Java, Ocaml, and Prolog maintain the same basic algorithms, but use different control and data structures. This was done to take advantage of the structures provided by the paradigms that each language supports. Lists, for example, are the basic data structures of functional languages and have optimized access functions. On the other hand, clauses are the basis of logic languages, while sequential structures like arrays predominate in imperative languages.

The Tic-tac-toe and Sokoban games were implemented using lists as data structures in Ocaml. In Java, matrixes were used as maps in Sokoban and to save the plays in the Tic-tac-toe game. The Sokoban Java code was adapted from a version available in the web (NUGROHO, 1999), and its graphical interface was maintained. In Prolog, the Tic-tac-toe game uses the predicate construct, while lists were used in Sokoban. In Esterel, the Tic-tac-toe game was implemented using state machines and single Strings as the map positions.

The Java Address Book uses the Vector class and a class Person, with name and phone as attributes. The Ocaml Address Book uses lists and tuples to store the contacts, and the Prolog version again uses dynamic predicates. Although Ocaml is a language with imperative features, in the MP3* application we tried to use only its functional constructions, since we wanted to observe the amount of possible abstraction achieved. In Esterel, the Address Book uses a Java Vector data structure, and its manipulation methods are implemented in Java, in a separated file. These methods are later invoked by the Esterel Address Book source code. The control state machine in Esterel implements the user menu and method calls.

The cosine table of IMDCT in Java is a final array, in Prolog it reproduces a table composed by predicates, and in Ocaml it is a list. The output IMDCT array is an array in Java and a list in Ocaml and Prolog.

3.2 Results

The results were obtained by actual tests executed under the same conditions for all languages and datasets. Tests and experiments were executed on an AMD Sempron 2500+ running at 1750 MHz with 512MB of DDR3200 RAM memory. The operating system used was a Debian GNU/Linux 3.1 with GNU/Prolog, Ocaml, GCC/GCJ 4.0, and SUN JDK1.5.

The abstraction achieved in number of lines of code and the impact in terms of performance and memory usage of each implementation were evaluated. In many cases, the number of lines of code provides a good measure of abstraction. A 32-bit multiplier with 6689 gates may be described using only 31 lines of code in VHDL, for example. Although the complexity of different commands in different languages is not always reflected by the number of lines of code, a measure of this complexity is not easy to

obtain. Other related metrics like learning time and development time were not used because they are strongly dependent on the programmers.

3.2.1 Achieved Abstraction

Figure 3.1 shows the percentage of Lines of Code (LoC) of two Ocaml and Prolog applications, compared with the Java ones. The numbers above the columns are the absolute number of LoC of each application. As said before, Sokoban requires a Graphical User Interface. The LoC numbers in Sokoban means Total LoC minus GUI LoC.

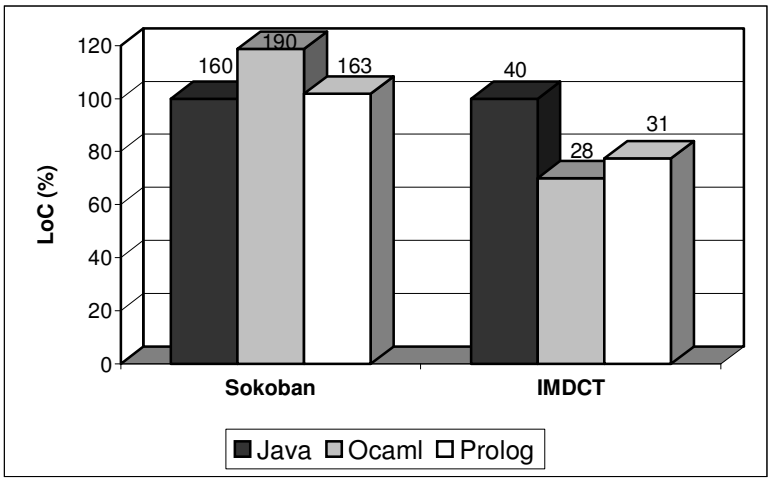

Figure 3.1: Sokoban and IMDCT size in LoC

Figure 3.2 also shows the percentage of LoC of Ocaml and Prolog applications, but now compared with Java and Esterel ones. In the Address Book, the Esterel line count means the lines of code in Esterel plus the lines of code that were implemented in Java.

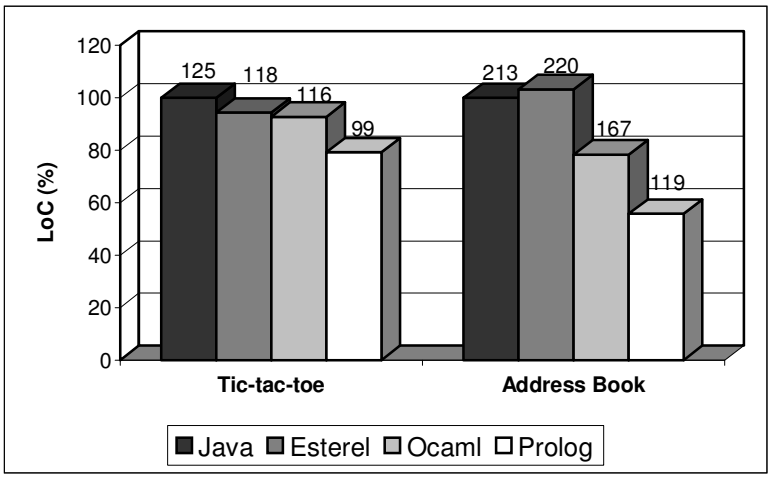

Figure 3.2: Tic-tac-toe and Address Book size in LoC

The charts in 3.1 and 3.2 show that, in terms of Lines of Code, declarative languages can achieve a lower line count than Java for some applications. Using Prolog instead of

Java or Esterel, one can reduce nearly by half the effort in terms of lines to code an Address Book. However, as the Sokoban result shows, this is not always the case, since the lack of adequate data structures available for certain applications increases the number of Lines of Code. In this case, the Sokoban code in Ocaml is about 20% larger than its correspondent in Java.

The Tic-tac-toe Ocaml and Prolog implementations, although with about 10% and 20% less Lines of Code than the Java one, provided somewhat less abstraction considering data structures. A matrix to represent the game board is more intuitive and easier to deal with. The smaller number of Lines of Code was achieved in Ocaml and Prolog applications because of the cleaner control structures. Despite the fact that Java provides more abstraction considering data structures, the Esterel Tic-tac-toe achieves less LoC, also because of the abstract control structures and succinct commands and syntax of Esterel.

The use of lists is not adequate when there is a need for a data structure that changes elements besides adding elements and querying. In order to maintain the referential transparency, a list cannot be modified, that is, a new modified list is returned each play. Despite its safety, there is an unnecessary memory overhead, which is not adequate for embedded systems.

The case of Sokoban is similar to the Tic-tac-toe one, considering data structures in Ocaml. A board with more positions makes the situation even worse for the Ocaml Sokoban implementation, where dealing with strings is necessary to avoid the extra overhead, when considering each position as an element of a list. Besides, the Ocaml String Library does not offer functions like the *java.String.split(String expression)*. The Java AWT classes also help on manipulating the Sokoban board map, which makes the implementation in Java even easier. The Prolog implementation of Sokoban stores positions of the objects in the game as dynamical facts loaded from a file specified in the initialization. Prolog, as Ocaml and most declarative languages, has difficulties to handle input and output (very limited in these languages).

The Address Book implementation is interesting in Ocaml, due to the simplicity of dealing with lists and tuples. The Prolog implementation of Address Book uses dynamically loaded facts to store contact information. One can store the contact archive as a set of facts and load them when needed. Since the Address Book application is limited to loads, saves, and queries about these facts, this is the application where Prolog has the best results in terms of abstraction and performance. The Esterel language again seems not to be a good choice for implementing an Address Book application. As data structures must be still implemented in Java, an extra time is necessary to switch between languages and maintain the coherency among files.

A comparison in terms of control structures, data structures manipulation, and code conciseness among Prolog, Ocaml, and Java may be observed in the search operation of an Address Book written in Figure 3.3 to 3.5.

The search clause in Prolog has an instance of the Address Book (database) and a variable holding the name of the contact to be found as parameters. If a name that belongs to a contact in the database matches the name obtained from parameters, a value for the Phone variable will be returned. Otherwise, the commands after the semicolon will be executed. In this case, a “not found” message will be delivered to the standard output (screen).

```
search(X,Name) :-
      contact(Name,Phone),
      write(Name),
      write(' has the number: '),
      write(Phone);
      write('\nContact not found in database!\n').
```

Figure 3.3: Search operation in Prolog

```
let rec search name list =
  match list with
     [ ] ->  Printf.printf "\nContact not
           found in database!\n";
   | head::tail -> let (n,p) = head in
         match n = name with
           true -> Printf.printf
               "\n%s has the number: %s\n" n p;
          | false -> search name tail;;
```

Figure 3.4: Search Operation in Ocaml

In Ocaml, the *pattern matching* construct allows simple access to the components of complex data structures. *Pattern matching* is used to recognize the form of a value and lets the computation be guided accordingly, associating each pattern to an expression to compute. In the search function in Figure 3.4, *pattern matching* is applied to a list. A list can be either empty (the list is of form [ ]) or composed of a first element (its head) and a sublist (its tail). The list is then of form *head::tail*.

In the Address book in Ocaml, each element of the list is a tuple representing a contact, of form (*name,phone*). The *pattern matching* on the list tries to match the name obtained from parameters with the first component of the tuple in the current head of the list. If the matching does not happen, the interpreter will keep iterating recursively, considering the rest of the list (tail). As soon as a matching occurs, the function automatically finishes its execution. If there are no matches for the name and the considered list becomes null, a “not found” message will appear on the screen.

The Address Book in Java has its contacts stored in a *Vector* data structure. A search on a *Vector* must be controlled by an auxiliary variable (*i*), to ensure that the search is not going to exceed the boundaries of the data structure and throw an exception. Also, a flag must be used to advise that the loop must be interrupted, in case an object Person with the same name as that searched was found.

In Prolog, the search engine is totally transparent to the programmer, providing abstraction by concentrating on what has to be done, and not how to accomplish this task. In Ocaml, the *pattern matching* and recursion mechanisms simplify what in Java is done using flags and for-loop commands. A search method in Java also depends strongly on which data structure is used. In Esterel, the same Java search method stated above is invoked by the Esterel Address Book source code.

```
public void search(String name){
  int i=0;
  boolean found=false;
  while((i<list.size())&&(found==false)){
    if(((Person)list.get(i)).getName().equals(name)){
      String phone=((Person)list.get(i)).getPhone();
      System.out.println(name+"has the number: "+phone);
      found=true;
     }
    i = i+1;
   }
   if (found==false)
   System.out.println("Contact not found in database!");
}
```

Figure 3.5: Search operation in Java

The IMDCT algorithm in Ocaml and Prolog, considering data structures, presented the same abstraction as the Java version. In this case there is almost no difference between using lists, predicates, or an array. Control is usually more abstract in Ocaml and Prolog due to the use of recursion instead of loops and pattern matching instead of for-loop and if-then-else commands. This feature also makes the code cleaner. This is the reason for the short code of IMDCT in Ocaml and Prolog, when compared to the Java IMDCT.

3.2.2 Performance and Memory Usage

Table 3.1 shows a comparison in terms of static memory consumption by interpreted code, compiled code, and virtual machine size. Java interpreted code uses less static memory for all applications; however, there is a penalty in the JVM memory usage. The amount of memory used by the Prolog virtual machine is variable, depending on the predicates required by the application. The Address Book in Prolog required the use of dynamic predicates to add, edit and delete contacts from the database. For this reason, the virtual machine size for the Address Book in Prolog is 15% higher than the JVM size. Considering the sum of the interpreted code size with the virtual machine size, it is possible to see that the result is usually better for applications coded in Ocaml.

The amount of static memory used by the applications when they execute natively varies according to their type and size. For three applications it is smaller in Prolog implementations; the Address Book in Prolog requires only 3 KB of memory for the executable file, more than thirty times less than the amount required by Java and Ocaml ones.

Table 3.1: Static memory usage

Application	**Tic-tac-toe**			**Sokoban**		
Language	Java	Ocaml	Prolog	Java	Ocaml	Prolog
Interpreted (KB)	3.5	47	16	22	49	63
Native (KB)	26	61	125	102	79	40
VM (KB)	262	116	162	262	116	279
VM + Interpreted (KB)	265.5	163	178	284	165	342
Application	**IMDCT**			**Address Book**		
Language	Java	Ocaml	Prolog	Java	Ocaml	Prolog
Interpreted (KB)	1.8	22	15	6.4	49	14
Native (KB)	16	18	4	35	63	3
VM (KB)	262	116	120	262	116	299
VM + Interpreted (KB)	263.8	138	135	268.4	165	313

Figure 3.6 shows a performance evaluation of the IMDCT algorithm. The plot shows the time required to process the amount of data varying from 10 to 1,000,000 samples. As one can see, Ocaml and Prolog have a performance degradation two orders of magnitude larger than Java. This result can be explained by the use of recursion to implement iteration, needed in the algorithm. When interpreters use recursion, it is necessary to allocate another frame to store data of that pass. The allocation process incurs in time and memory penalties. The same result pattern was observed in the interpreted code and in the native compiled code. This means that, albeit optimizations used in compiler of Prolog and Ocaml, native code follows the same idea of interpreted code to obtain the answer. Another implementation of the IMDCT algorithm that benefits from recursion can possibly match up Java performance.

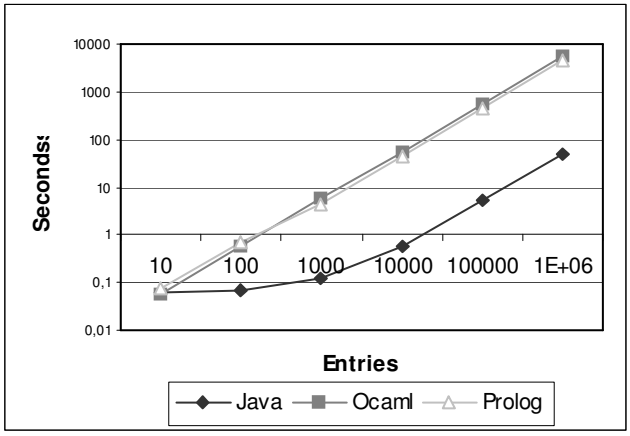

Figure 3.6: Execution time comparison in Java, Ocaml and Prolog

Backtracking and instantiation processes in Prolog and recursion in Ocaml lead to a high memory allocation and usage. For this reason, declarative languages use a lot of memory bandwidth in execution time and each memory access slows down the program execution. Mechanisms such as lazy evaluation in Ocaml, that improve the performance of functional languages avoiding unnecessary computations, also increase memory usage.

Prolog and Ocaml execution environments are based on a stack machine, which leads to the same memory problem described above, but can also draw some advantages as a good garbage collection scheme, and in some cases a predictable runtime memory usage.

3.3 Conclusions of the Declarative Languages Analysis

Results showed that we can describe different computation models in Prolog and Ocaml but, unfortunately, these results also showed that the performance and memory penalties cannot be neglected. The abstraction level reached with the use of Prolog (measured in Lines of Code) is near 50% in some cases, although in the worst case it was a little higher than Java. For applications like Sokoban, the Java code is more abstract and still uses less memory. Nevertheless, for applications that mix models of computation like the Address Book and the Tic-tac-toe game, declarative languages presented more abstraction mechanisms than Esterel, a language that is specific for reactive embedded systems.

The abstraction results for declarative languages were not satisfactory for all kinds of embedded applications studied. This means that by using the single metric of LoC, even by assuming a perfect interpreter for a declarative language, one would not achieve orders of magnitude gains in the code abstraction. Nevertheless, some features of declarative languages are still highly desirable, like those concerning the error ratio in programs. Prolog and the other logic programming languages are based on logic, and so its programs can be logically organized and written, which leads to less errors and maintenance costs. Functional languages, with the use of recursion and *pattern matching* mechanism, also allow a cleaner code.

4 MOTIVATION FOR USING ALLOY

This section aims to explain why the Alloy language was chosen to be part of the proposed approach. It presents Alloy's main features and some basic knowledge on Alloy, required for understanding the further abstraction analysis. The study evaluating the Alloy abstraction power is presented, comparing applications modeled using Alloy with their correspondent code in Java and Esterel.

4.1 Alloy's Main Features

Alloy (JACKSON, 2006) is a modeling language based on first-order logic, used for expressing complex structural constraints and behavior. With the underlying idea that code is a poor medium for exploring abstractions (JACKSON, 2006), the Alloy language provides declarative high-level descriptions (or models) allied with lightweight formalism. Alloy, which is freely available on the Web, has been used to model and analyze all kinds of system, but mostly systems that involve complex structured states. It has proved useful in applications as general security (CHEN et al., 2006), interoperating networks (ZAVE, 2005) and dynamic reconfigurations (WARREN, 2006).

The Alloy Analyzer is a constraint solver that provides fully automatic simulation and checking of Alloy models. It translates constraints to be solved from Alloy into Boolean constraints, which are then fed to an off-the-shelf SAT solver. All problems are solved within a user-specified scope that bounds the size of the domains, and thus renders the problem finite (and reducible to a Boolean formula).

What makes Alloy distinct from other languages used for specification is that models in Alloy are both declarative, analyzable, support complex structural states (instead of complexity due to event sequencing) and are usually much smaller than their respective implementations. None of these features is new in themselves. Formal specifications in Z are declarative and structural, and model-checking languages such as SMV are analyzable. What is new is the combination of features, especially of declarative modeling and analyzability (JACKSON, 2006).

4.2 Principles of the Alloy Language

Alloy models can express both structure (in a software sense, a connection among objects) and behavior. A signature is the Alloy structuring mechanism (JACKSON, 2006). Each signature represents a set of atoms, and may also introduce some fields,

each representing a relation. A signature, on a top view, is like a set data structure. The language provides operators for sets, relations and signature inheritance as well.

Regarding the Alloy modeling purposes, facts, predicates, functions, and assertions are model constraints. Facts are constraints that are assumed always to hold, different than predicates and functions, which are constraints valid under some conditions or used for simulation. Assertions are implications to be checked. From the implementation point of view, a portion of the application behavior is modeled using predicates and functions, which perform operations within the data structures.

Some useful libraries are available in the current version of Alloy (Analyzer 4.1.8). An ordering library is particularly applicable to provide state machine modeling without timing properties. Alloy has some specific primitives for analysis: the *run one* command is used to check a property or to call a predicate to be checked; and the *check* primitive indicates an assertion to be checked. Finally, the search space is bounded to allow an exhaustive search for specification flaws within the scope.

4.3 Analysis of the Abstraction Achieved by Using Alloy

It was shown before that one can describe different computation models in declarative general-purpose languages such as Prolog and OCaml, but the abstraction level reached is not much higher than the one provided by the Java imperative language. In addition, none of these declarative languages provide built-in support for model verification. Complementing the previous analysis, a new study includes Alloy as a representative of declarative languages set. In order to analyze the Alloy abstraction capabilities, some of the same applications presented before (a Tic-tac-toe game and an Address Book) plus a Crane high-level control system were modeled using Alloy.

The Address Book and Tic-tac-toe applications are the same as those described in Section 3.1. The Crane system (MOSER; NEBEL, 1999) is composed of both data and event driven components, however, the crane modeled already specifies a solution based in functional blocks for the problem under consideration, and does not express the system requirements. Figure 4.1 shows the comparison of the number of lines of code for each application, which does not include lines of code used for input and output interfaces.

The implementation of the Tic-tac-toe game in Alloy is almost 50% shorter than the Java one, and about 30% shorter than the Esterel version. The results for Address Book are even more encouraging: the implementation in Alloy is also 50% shorter than the Java one, and around 60% shorter than the Esterel implementation. These results show that Alloy can provide abstraction in both data structures and control. Considering the crane control system, a high-level interface with module communication was implemented. In this case, results also show the advantage of the Alloy implementation, which has around 40% and 70% less lines of code than the Java and Esterel implementations, respectively. Both crane control system and address book applications take advantage of data structures abstraction in Alloy. As in other declarative languages like Prolog, the search engine in a data structure is totally transparent to the programmer by means of relational operations, providing abstraction by concentrating in what has to be done, and not in how to accomplish this task.

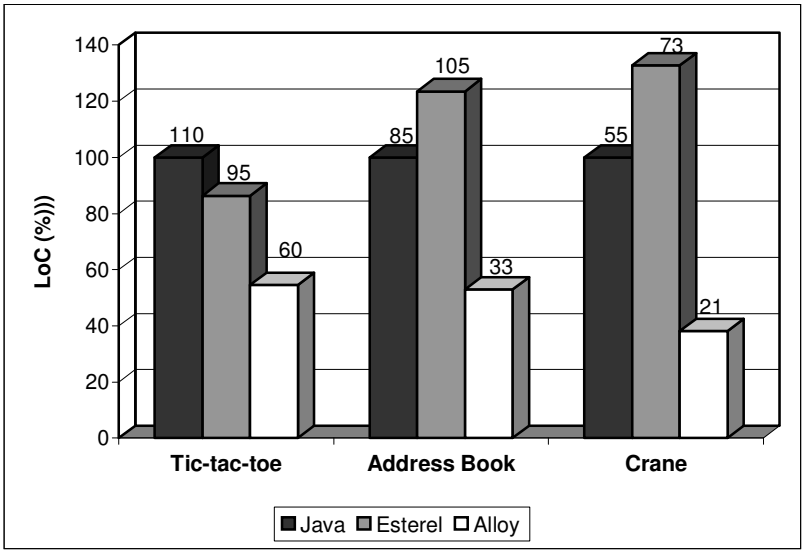

Figure 4.1: Comparison in LoC of applications coded using different paradigms

The numbers in Figure 4.1 confirm that the state machine approach of Esterel is not sufficient to express abstract applications that mix control and data structure usage, which are very common in embedded systems. Besides, Java and Esterel are languages focused on implementation details (for example, a variable signal must have a type). In Alloy, these details are usually not important, and the same Alloy model may be reused and mapped into a customized implementation, according to the requirements of a particular system. Based on these results, the main motivations for the use of Alloy are the gains in expressiveness and abstraction together with its simulation, checking capabilities and its object-oriented syntax, favoring its adoption by the embedded system community.

5 PROPOSED APPROACH

This work proposes the usage of Alloy as the high-level specification language, followed by automatic generation of Java code from the Alloy specification. Java is used as the target language due to its portability and it broad adoption in embedded devices (LAWTON, 2002). Nevertheless, the concepts presented here can also be ported to other target languages like C/C++ or Ada.

5.1 Design Flow

Figure 5.1 shows the proposed design flow for embedded software automation. Following the requirements description, all functional ones are described with the Alloy language (this includes structural and behavioral specifications). After, the Alloy Analyzer analyzes the Alloy model, and the designer can check if the modeled application complies with its requirements.

The verified and requirements-compliant Alloy model enters the Model Translator, generating thus Java sources that implements the modeled application. Different Java implementations are generated for the same Alloy model, varying the data structures used in the code. For all these implementations, physical properties such as required memory size, execution time, power and energy consumption are estimated, considering that the code will run in a certain platform.

The platform adopted in this work is composed by different cores with the same instruction set. It is based in different implementations of a Java processor, FemtoJava (ITO; CARRO; JACOBI, 2001; BECK; CARRO, 2003), previously developed in the research group. The FemtoJava processor implements a Java execution machine directly in hardware, by using a stack mechanism compatible with the Java Virtual Machine (JVM) specification. It is emphasized that the choice of FemtoJava as the target platform was due to the possibility of analyzing and instrumenting generated code. However, the proposed design flow could be adapted to use J2ME or other platform.

In order to use more elaborated behaviors than those currently supported by the approach and also support legacy code reuse, the approach foresees the use in the Alloy model of Java methods and classes imported from a repository. The integration between the Java code repository and the Alloy model will be accomplished by using annotations from Alloy Annotation Language (AAL) (KHURSHID; MARINOV; JACKSON, 2002), process that is currently under development in the research group.

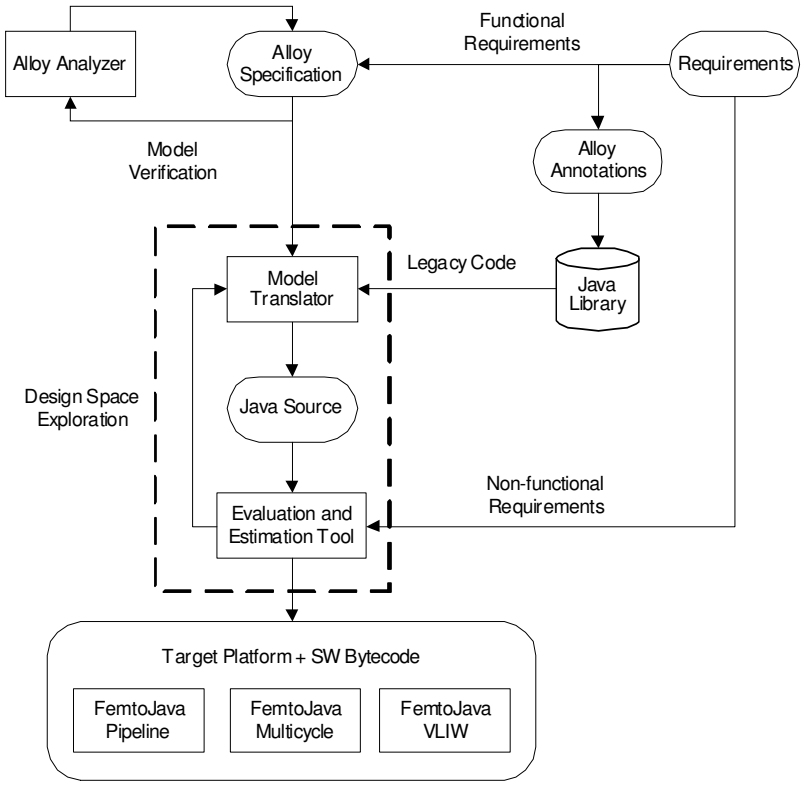

Figure 5.1: Proposed approach design flow

5.2 Case Studies

This section presents three small case studies used to validate and exemplify the approach. Alloy models were built for an Address Book, an elevator control system and for a Vending Machine.

5.2.1 Address Book

The Address Book behavior is mostly control-flow with some embedded data processing, since the manipulation of the data structure after some user input predominates. Besides, the Address Book user options suggest a reactive control behavior. The Address Book has only two fields (name and phone), and allows addition and deletion of an entry, and searching for a phone.

Some ideas of the Address Book Alloy model presented here were taken from the examples of (JACKSON, 2006). *Name* and *Phone* are empty signatures, and Book is a signature that holds names and their respective phones. Figure 5.2 shows the Alloy model for the Address Book.

```
1.  open util/ordering[State] as ord
2.  sig Name, Phone {}
3.  one sig Menu{}
4.  sig Book {
5.   contact: Name -> one Phone
6.  }
7.  sig State {
8.    book : one Book,
9.    addition,  deletion, found, done, exit: set Menu
10. }
11. fact exclusive {no m1, m2: Menu | m1 = Menu && m2 = Menu && disj[m1,m2]}
12. fact initialState {
13.   let s0 = ord/first | no s0.done &&
14.  (s0.addition = Menu || s0.deletion = Menu || s0.found = Menu || s0.exit = Menu)
15. }
16. fact stateTransition {
17.   all s: State, s': ord/next[s] {
18.     one n: Name | one a : Phone {
19.     (Menu in s.addition) =>
20.         add [s.book,s'.book,n,a]  && s'.done = Menu {
21.     (Menu in s.deletion) =>
22.         del [n,s.book,s'.book]    && s'.done = Menu {
23.     (Menu in s.found) =>
24.         some lookup [s.book,n]    && s'.done = Menu
25.  }}}}
26. }
27. pred add [b, b': Book, n: Name, a:Phone] {
28.    ((no lookup [b,n]) && (#b.contact[Name] < 7)) =>
29.       { b'.contact = b.contact + n->a }
30. }
31. pred del [n: Name, b, b': Book] {
32.       (#b.contact[Name] > 0) => {
33.                 (some lookup[b,n]) =>
34.                    { b'.contact = b.contact - n->Phone }
35.  }
36. }
37. fun lookup [b: Book, n: Name]: set Phone {
38.     n.(b.contact)
39. }
40. pred solve {
41.    ord/last.done = Menu
42. }
43. run solve for 3 but 3 State
```

Figure 5.2: An Alloy Address Book

In order to supply the Address Book control information, a design pattern called *Menu* is used, which is expressed by the fact *stateTransition* in the Alloy model. In the model, each state holds its own Book and atoms that specify an operation to be performed in the book. The fact that describes state transitions calls the operation predicate, according to these atoms. The final state is reached when the *done* atom of the current state receives the token *Menu*, which means that an operation has just finished its execution.

There are many ways to model an Address Book in Alloy. The way described in this work was chosen because then it is possible to obtain Java code from the model without

disturbing the Alloy Analyzer simulation output. Choosing a minimum of states for the simulation, one operation is performed at a time and, by asking the Alloy Analyzer for other solutions, one could get examples of all operations performed.

The assertion *delUndoesAdd* in Figure 5.3 says that an addition from book *b* resulting in book *b'*, followed by a deletion using the same name *n* results in a book *b''* whose address mapping is the same as the one of the original book *b*. Supposing that the model is not complete and the *add* and *del* operation do not check for existing contacts in the Address book (in Figure 5.2, lines 28, 32 and 33 are suppressed), as suggested by (JACKSON, 2006). The checking of the assertion returns a counterexemple, a scenario in which the assertion is violated, in Figure 5.4.

```
assert delUndoesAdd {
        all b,b',b'':Book,n:Name,a:Phone |
        add[b,b',n,a] && del[n,b',b''] => b.contact = b''.contact
}

check delUndoesAdd for 3
```

Figure 5.3: Checking if an addition undoes a deletion in an Address book

In the diagram produced by the Alloy Analyzer shown in Figure 5.4, *b* and *b'*, the values of the book in the first and second states, are both *Book1*. This happens because the name/phone link to be added is already present, so the execution of *add* has no effect. The execution of del, on the other hand, removes the link, resulting in the empty book *Book0*, that corresponds to *b''* in the model. If the lines 28, 32 and 33 in Figure 5.2 are present in the model, the Analyzer says that no counterexemple has been found for the assertion in Figure 5.3.

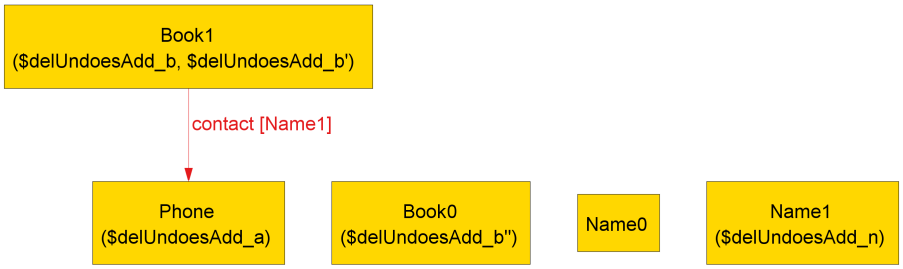

Figure 5.4: Couterexemple of the assertion delUndoesAdd

5.2.2 Elevator Control System

In this case study, a simple elevator control system based in Figure 5.5 was modeled in Alloy. A *Request Solver*, that resolves various requests into a single requested floor, and a *Unit Control*, that moves the elevator to this requested floor, compose the whole system. An existing *Request Resolved* software is assumed, and only the *Unit Control* module was modeled.

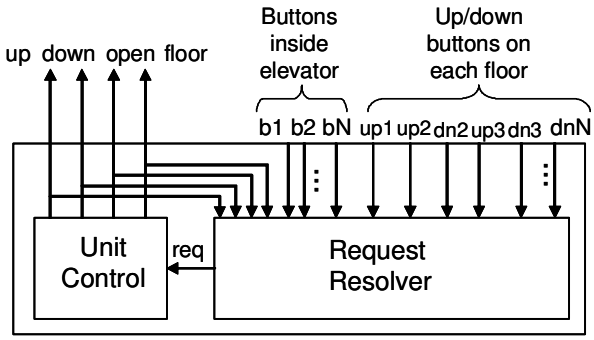

Figure 5.5: The Elevator Control System

The Elevator model needs information about the service request, and the current floor to determine what to do next. These concepts are expressed by *reqFloor* and *currentFloor* that are signed as *Floor* in the elevator model illustrated in Figure 5.6. The current floor and the requested floor are input in the control system.

The output of the elevator control system is the direction of the elevator. The direction is modeled by the *Direction* signature in the Alloy model, and can be *stop*, *up* and *down*. For each state change, the control algorithm determines the direction considering the requisition and the current floor. For example, when the requested floor is higher than the current one, the *currentFloor* of the next state receives the *currentFloor* of the current state plus one and the direction is set to *up*. This example also uses the design pattern *Menu* for the state machine control.

Through the Alloy Analyzer, it is possible to simulate the signatures and operations, and also, check assertions about the model. Simulations are always welcome to check if the model is in accordance with the specification. A simulation of the Elevator Control System model is shown in Figure 5.7. There, properties described in the Elevator model, such as reaching the required floor, can be verified.

Checking assertions on simple applications like the Elevator Control System or the Address Book seems unnecessary, but it prevents from undesirable bugs that may appear when the application is already in the market. On the other hand, for security applications, checking assertions about the model, more than a desirable feature, are indispensable.

```
open util/ordering[State] as ord
open util/ordering[Floor] as ford
sig Floor {}
one sig Direction {}
one sig Menu{}
sig State {
  currentFloor:  one Floor,
  reqFloor : one Floor,
  stop, up, down: set Direction,
  nDone, done : set Menu
}
fact initialState {
  let s0 = ord/first |
  one cf: Floor | s0.currentFloor = cf &&
   s0.stop= Direction &&
   one rf: Floor | s0.reqFloor = rf  &&
   s0.nDone = Menu
}
fact {
  no s: State | s.nDone=Menu && s.done=Menu
}
fact f1 {
  no disj d1, d2: Direction | (Direction in d1 && Direction in d2)
}
fact stateTransition {
 all s: State, s': ord/next[s] {
(gt[s.reqFloor, s.currentFloor] && no s'.done) =>
       { s'.reqFloor = s.reqFloor &&
         s'.currentFloor = next[s.currentFloor] &&
         s'.up = Direction &&  s'.nDone = Menu } else {

(lt[s.reqFloor, s.currentFloor] && no s'.done) =>
       { s'.reqFloor = s.reqFloor &&
         s'.currentFloor = prev[s.currentFloor] &&
         s'.down = Direction  && s'.nDone = Menu } else {

(s.currentFloor == s.reqFloor) =>
       {s'.reqFloor = s.reqFloor &&
         s'.currentFloor = s.currentFloor &&
        s'.stop = Direction && s'.done = Menu }
}}}
}
pred solve {
 ord/last.done = Menu
}
run solve for 4 but 4 State
```

Figure 5.6: The Elevator Control System in Alloy

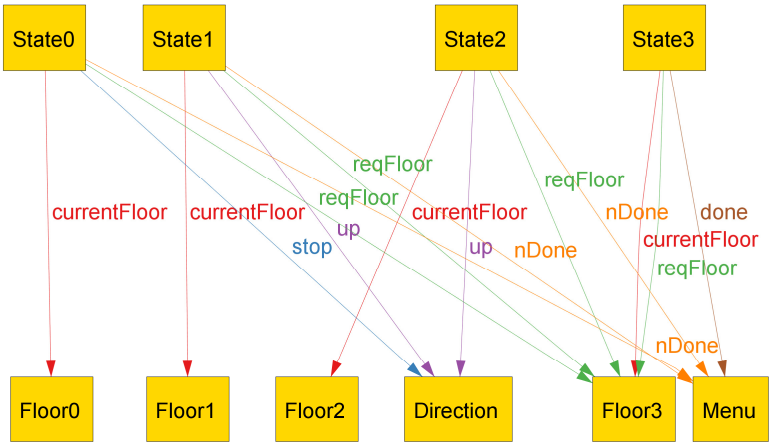

Figure 5.7: The Elevator Control System simulation

5.2.3 Vending Machine

The Drink Vending Machine is a reactive embedded system, where the user leads its behavior by signaling options to the machine. The modeled machine has the following requirements:

i) there are three kinds of drinks: water, tea and soft drink;

ii) every drink has a price

iii) the machine works by inserting coins in it;

iv) a sell is only completed when the needed amount of coins has been inserted;

v) the machine does not sell drinks if whose storage is empty.

The vending machine Alloy model is shown in Figure 5.8.

Instead of using a pattern like *Menu*, the Vending Machine application assumes that the model represents a state machine from untimed model of computation (JANTSCH, 2004). Therefore, a designer shall use *run* commands to inform which are the possible states of the machine. The two commands *run* in lines 43 and 44 of Figure 5.8 indicate that the possible states of the application are either waiting for someone to add coins, or expecting to sell a drink.

The simulation obtained from the *run* command in Figure 5.8 - line 41 is shown in Figure 5.9[1]. It shows a case where all the drinks in the machine are consumed, and there are no coins left in the temporary machine coin storage. Coins are added into the coin storage until there are coins enough to buy a drink. Considering that in the initial state (Figure 5.8 - line 12) there is at least one element of each drink (Figure 5.8 - line 15), and every state the machine is either receiving coins or selling drinks (Figure 5.8 - line 17), the first possible solution has 10 states. The scope of the simulation is bounded together with the *run* command (Figure 5.8 - line 42). The number after the *for*

[1] *VendingMachine*, *coinStorage* and *drinkStorage* were replaced by *VM*, *CoS* and *DrS* respectively, in order to improve visualization.

statement indicates the bound: to at most 10 objects in each signature. If the scope has smaller bounds than 10, the Alloy Analyzer indicates that no simulation instance was found.

```
1.  open util/ordering[State] as ord
2.  abstract sig Drink { price: Int }
3.   sig Water extends Drink {}
4.   sig Softdrink extends Drink {}
5.   sig Tea extends Drink {}
6.   sig Coin {}
7.   sig VendingMachine{
8.     coinStorage:   set Coin,
9.     drinkStorage:  set Drink
10.  }
11.  sig State { machine: VendingMachine }
12.  fact InitialState {
13.     let s0 = ord/first.machine | no s0.coinStorage
14.     let s0 = ord/first.machine | all d:Drink | d in s0.drinkStorage
15.     some Tea and some Water and some Softdrink
16.  }
17.  fact DrinkBuying {
18.     all s: State, s': ord/next[s] | some d: Drink | some c: Coin |
19.     d.price <= #s.machine.coinStorage =>
20.                     buyDrink[s.machine,s'.machine,d]
21.                     else addCoin[s.machine,s'.machine,c]
22.  }
23.  fact Prices {
24.       all w: Water      | w.price = 1
25.       all s: Softdrink  | s.price = 2
26.       all t: Tea        | t.price = 3
27.  }
28.  fact MaximunCoinAmount { #Coin = (sum d : Drink | d.price) }
29.  fact AlwaysDoSomething{ all s:State,s': s.ord/next |
30.                     s.machine.coinStorage != s'.machine.coinStorage }
31.  fact AllMachineInSomeState{ all m:VendingMachine |
32.                                 some s: State | m in s.machine }
33.   pred buyDrink[m, m': VendingMachine, d: Drink] {
34.       m'.drinkStorage =  m.drinkStorage - d
35.      #m'.coinStorage  = #m.coinStorage  - d.price
36.   }
37.   pred addCoin[m, m': VendingMachine, c: Coin] {
38.       m'.coinStorage  = m.coinStorage + c
39.       m'.drinkStorage = m.drinkStorage
40.   }
41.   run { no ord/last.machine.drinkStorage and
42.         no ord/last.machine.coinStorage } for 10
43.   run addCoin
44.   run buyDrink
```

Figure 5.8: Vending machine Alloy model

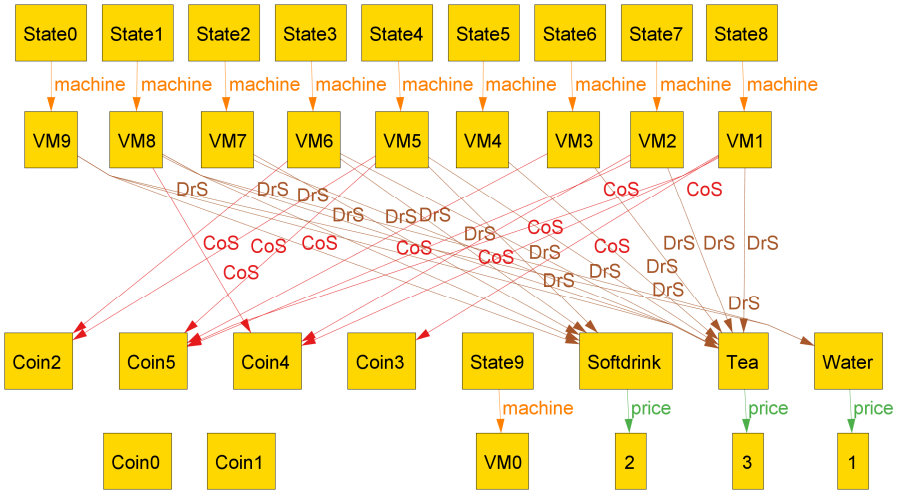

Figure 5.9: Simulation of the Vending Machine in the Alloy Analyzer.

6 CODE GENERATION

This section shows the translation rules for generating Java code from Alloy models. Such rules were further used to build an Alloy-to-Java compiler in the research group. The translation process is demonstrated by using case studies previously presented. The compiler went through some updates regarding data structures, to allow execution of generated code on the FemtoJava platform (ITO; CARRO; JACOBI, 2001) and estimation using DESEJOS tool (MATTOS; CARRO, 2007). There are three compiler versions: the first one uses the rules developed by the author of this work; the second one aggregates rules related to the state machine synthesis that were developed in the group, and the third version does not include new rules, but replace Java standard data structures by data structures supported in the estimation process.

The parser and the lexical analyzers of the Alloy-to-Java compiler were automatically generated using CUP (HUDSON; FLANNERY; ANANIAN, 1999) and JFlex (KLEIN 2004) tools, respectively. Currently, the third version of this compiler is composed of 3 automatically generated plus 35 hand-coded Java classes that encapsulate data needed by the code transformation process.

6.1 Generation of Classes and Attributes

Classes and attributes are extracted from declared signatures and their relations in the model, respectively. Signatures with fields are translated into classes that encapsulate the signature's fields as its attributes; the empty ones are translated into the Java String type. The algorithm for generation of Java classes and attributes is show in Figure 6.1 (where *Sig* corresponds to an Alloy signature).

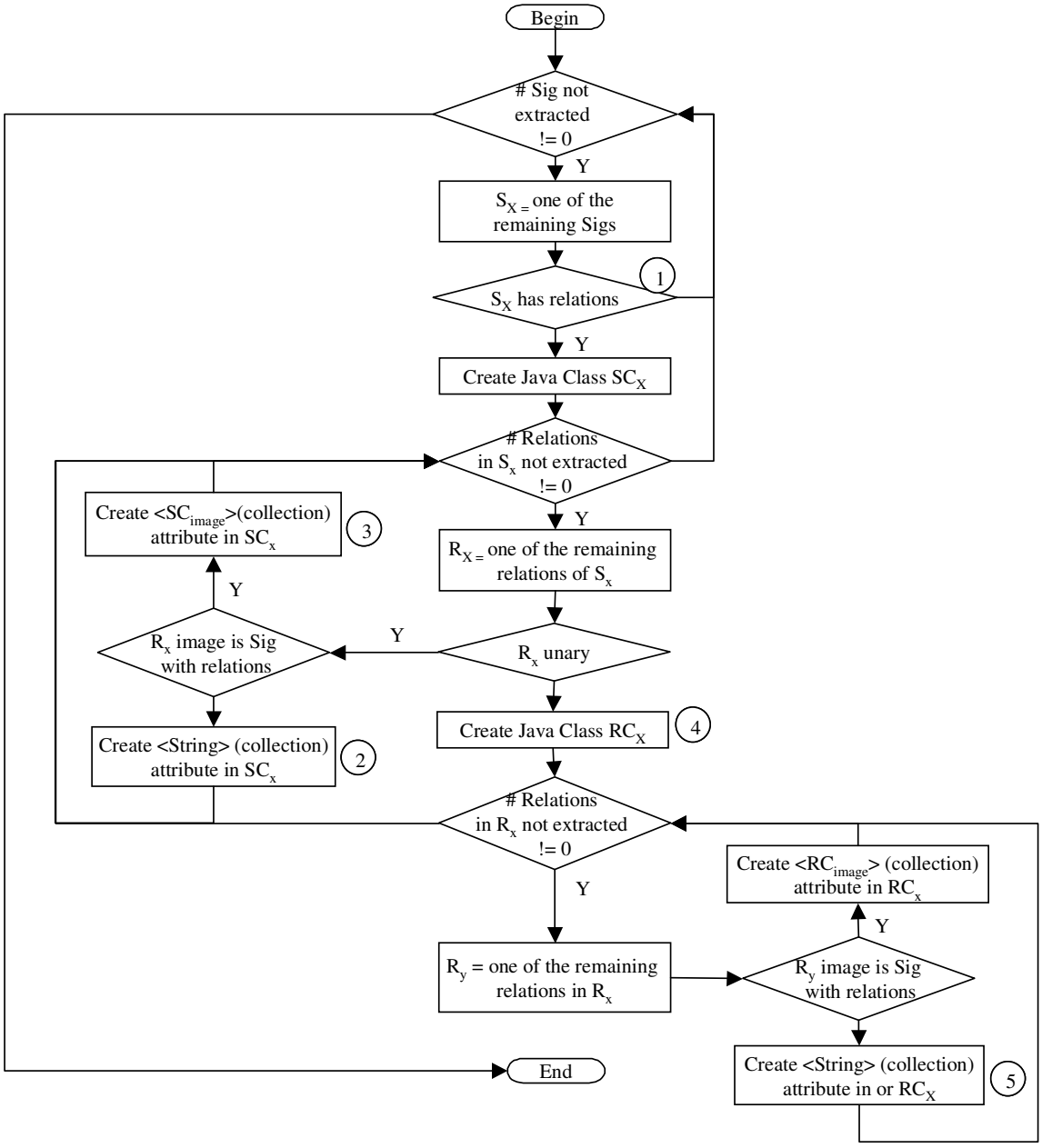

Figure 6.1: Procedure for generating classes and attributes

Figure 6.2 shows the structural portion of the *VendingMachine* class generated from the signature of the same name, according to the box 1 in Figure 6.1. The *VendingMachine* signature has two fields (relations): *coinStorage* and *drinkStorage*. The *coinStorage* relation is unary, and its image is a set of elements of the signature *Coin*. As *Coin* has no fields, *coinStorage* is mapped to an attribute of type String inside the *VendingMachine* class, according to the box 2 in Figure 6.1. The *drinkStorage* relation is unary and its image is of type *Drink*, which is not an empty signature. For this reason, *drinkStorage* is created as an attribute of type *Drink*, according to the box 3 of the algorithm.

Boxes 2 and 3 in Figure 6.1 show that it is possible to create single or composite attributes (collections). This depends on the multiplicity of the fields in a relation. Due

to the declared fields in the model with multiplicity *set*, the *VendingMachine* class contains two collections, where the *FastList* data structure was chosen by the designer (Figure 6.2, lines 2 and 3).

Alloy structural declaration	Java generated class structure

open util/ordering[State] as ord
abstract sig Drink { price: Int }
sig Water extends Drink {}
sig Softdrink extends Drink {}
sig Tea extends Drink {}
sig Coin {}
sig VendingMachine{
  coinStorage: set Coin,
  drinkStorage: set Drink
}
sig State { machine: VendingMachine }
	1. public class VendingMachine {
	2. protected FastList<Drink> drinkStorage;
	3. protected FastList<String> coinStorage;
	4. public VendingMachine() {
	5. this.drinkStorage = new FastList<Drink>();
	6. this.coinStorage = new FastList<String>();
	7. }
	8. ...
	9. public abstract class Drink { … }
	10. public class Water extends Drink { … }
	…

Figure 6.2: Structural portion of the generated Java class VendingMachine

The Vending Machine application cannot be an example of the translation performed by steps 4 and 5 in Figure 6.1. An example of a signature that contains a non-unary relation is shown in the Address Book structural declaration, in Figure 6.3. The relation *contact* is not unary, once it relates *Book*, *Name* and *Phone* at the same time. In this case, a Java class *Contact* is created (step 4) and *Phone* and *Name* are created as its attributes (Figure 6.3, lines 8 and 9), according to the step 5.

Alloy structural declaration	Java generated class structure

open util/ordering[State] as ord
sig Name, Phone {}
one sig Token {}
sig Book {
contact: Name -> one Phone
}
sig State {
  book: one Book,
  addition, deletion, found, done: set Token
}
	1. public class Book {
	2. protected FastList<Contact> contact;
	3. public Book () {
	4. this.contact = new FastList<Contact>();
	5. }
	6. ...
	7. public class Contact {
	8. name: String;
	9. phone: String;
	10. }

Figure 6.3: Structural portion of the generated Java class Book

The collections are not implemented using Java standard libraries (J2SE and J2ME APIs), which provide structures like *LinkedList*, *Hastable* and *Vector*. This is due to the fact that part of these APIs was implemented in native code, and the target platform FemtoJava executes only Java bytecodes. Therefore, the Javolution library (DAUTELLE, 2007) was chosen because it is totally developed in Java.

6.2 Generation of Methods

Methods are extracted from declared predicates and functions in the model. Methods extracted from predicates are typed as void; the ones extracted from functions are typed with the function's type declared in the model. The procedure for generating Java methods is shown in Figure 6.4.

The class method membership is extracted from its formal parameters list. In the pattern used for state machine modeling, a state is a signature that holds fields, which

usually change across states. Predicates/functions perform such changes, and they have at least a pair of parameters with the same type of the signature that models the state machine - one holding the field value in the current state and another holding its value in the next state. This happens because Alloy is a declarative language, and keeps the property of referential transparency: a variable can be changed by its value, then, it cannot be assigned more than once. In this way, the translation tool sets a method as a member of the class generated from the paired-signature declared in the formal parameters list. If there is no such pattern, the method is set as static into the Main class.

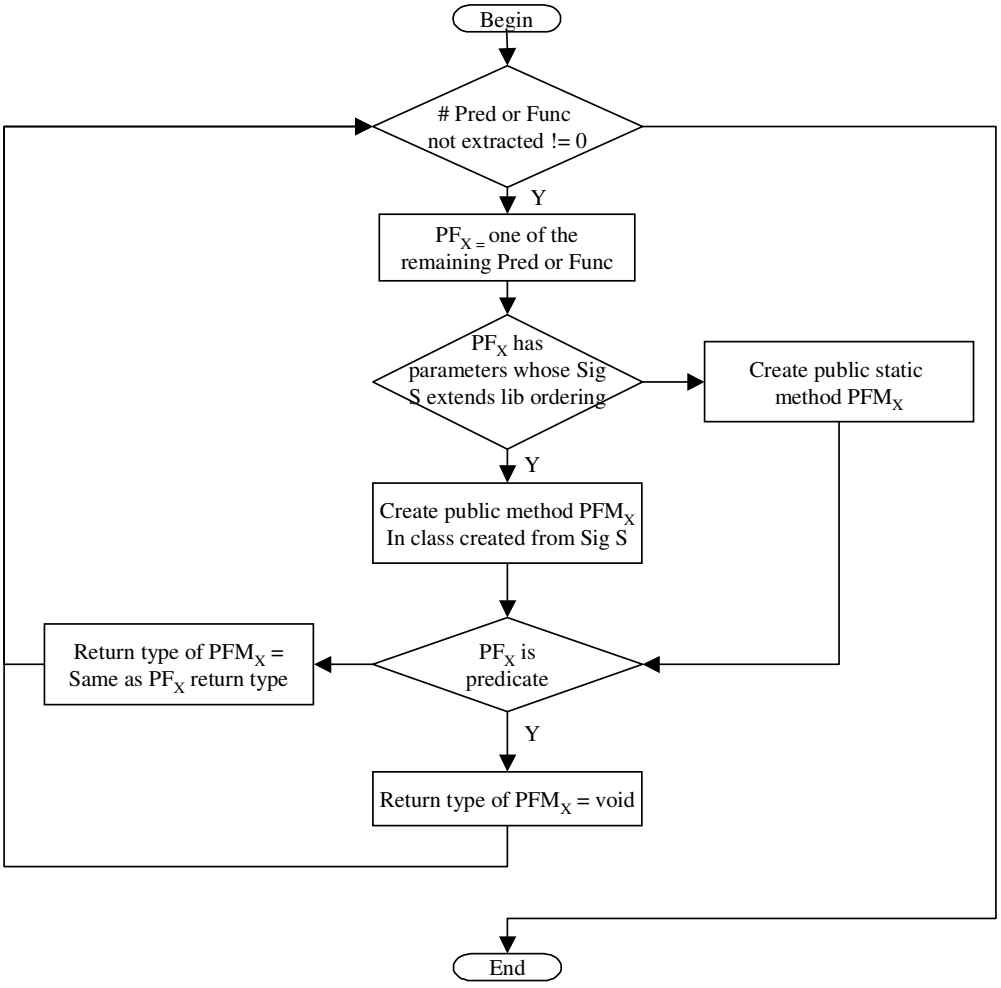

Figure 6.4: Procedure for Java methods generation

Figure 6.5 shows the generated methods from the predicates declared in the Vending Machine model. The translation process handles method polymorphism by generating all possible combinations of the polymorphic attributes. This may be awkward considering good practices of software engineering, but it is necessary in order to disambiguate between all possible state machine branches. This generation does not interfere in the reuse capabilities and design process quality, because the approach is applied to the high-level Alloy models, and not to the generated object oriented code.

Alloy behavioral declaration	Java generated method structure

pred buyDrink[m, m': VendingMachine, d: Drink]{
m'.drinkStorage = m.drinkStorage - d
#m'.coinStorage = #m.coinStorage - d.price
}
pred addCoin[m, m': VendingMachine, c: Coin]{
m'.coinStorage = m.coinStorage + c
m'.drinkStorage = m.drinkStorage
}
	public class VendingMachine {
	…
	public void addCoin(String c) {
	AlloyOperations.alloyUnion(
	this.coinStorage,c);
	}
	public void buyDrink_Water(Water d) {
	AlloyOperations.alloySubtraction(
	this.drinkStorage, new Water(d));
	}
	public void buyDrink_Softdrink(Softdrink d){
	AlloyOperations.alloySubtraction(
	this.drinkStorage, new Softdrink(d));
	}
	. public void buyDrink_Tea(Tea d) {
	AlloyOperations.alloySubtraction(
	this.drinkStorage, new Tea(d));
	}

Figure 6.5: Behavioral portion of the generated model from Alloy predicates

6.3 Generation of Alloy Operations

If within the model any Alloy built-in set and/or relational operation is called, it is necessary to generate Java code that contains an implementation for each operation. Hence, every application has an automatically generated library that implements these operations, in order to respect the behavior modeled within the Alloy model.

In the compiler versions 1 and 2, all available data structures either inherited the *AbstractCollection* abstract class or implemented the *Map* interface, both from the Java standard library (package *java.util*). In this way, all coded Alloy operations operated over *AbstractCollection* or *Map*. In the current version of the compiler (version 3), the code is generated for the four structures that are available in Javolution library (DAUTELLE, 2007): *FastList*, *FastMap*, *FastSet* and *FastTable*. Now, the operations are coded for these structures. Figure 6.6 shows the Union Alloy operation, coded for the *FastList* structure.

```
1. public static boolean alloyUnion(FastList c, Object a) {
2.   if(a != null && !c.contains(a)){
3.       c.addLast(a);
4. .     return true;
5. . }
6.   else
7.       return false;
8. }
```

Figure 6.6: Java method generated from an Alloy Union operation

6.4 Synthesis of the State Machine

A model may use states to allow a sequence of operations and order events on the automatic simulation performed by the Alloy Analyzer. In order to use this feature in a model, facts shall be declared with initial state constraints, state transition constraints and what is expected in the last state. In this way, it is possible to express the untimed model of computation (untimed MoC), as classified by (JANTSCH, 2004).

In the untimed MoC, the application waits for a command to be executed by an internal or external actor and proceeds with the computation based on the actor's action. In addition, the untimed MoC does not consider timing properties, and just the events order is important (JANTSCH, 2004).

As a declarative language, Alloy was not conceived to provide imperative features such as explicit loop statements in models and input/output constructs. Hence, it is not straightforward to model an application that requires a menu for the user interaction, for example, a common requirement in untimed MoC. At the first moment, it was considered to use information from the Alloy Analyzer output (an application state machine simulation, for example) besides of the Alloy model to generate Java code. Nevertheless, due to the random and non-deterministic feature of the Alloy Analyzer simulation, the decision was to rely only in the Alloy model for code generation.

In order to obtain a menu, for example, it would be necessary to implement a function for generating automatically all the possible simulation outputs of the Alloy Analyzer for a specific model, and also, a function for extracting a menu from all the outputs. This suggests an approach based on the old synthesis from traces (BIERMANN et al., 1975). Besides, the developer would be forced to model the application in a certain way that causes the Alloy Analyzer to simulate only the outputs necessary to build a menu. In this way, the code generation would be bound to the simulation, and the developer would probably not visualize other simulation outputs besides those related to the code generation. Another idea that occurred lately was to take the equations that the Alloy Analyzer uses to build the simulation instead of the simulation outputs themselves in order to generate Java code, and this implementation is among the future works.

As discussed before, the current code generation approach is based only in the Alloy model, and it suggests two ways for generating state machine for an application. The first one, adopted by the compiler version 1, is based in the use of a specific design pattern. The design pattern concept was adopted for modeling behaviors that cannot be inferred directly from an Alloy model, such as the ones that include interactive loops. In this case, by using different design patterns, other models of computation and behaviors could be modeled. The second way is more straightforward for developing state machines once it assumes that the application is modeled exclusively under untimed MoC. In this way, the designer just writes the operations that will be added into the state machine by the Alloy's *run* commands instead of describing detailed state transition constraints.

The design pattern used for state machine modeling and adopted by the first version of the compiler is called *Menu*. Its usage is marked by the singleton empty signature *Menu* and the fact *stateTransition* in the model. Address Book and the Elevator Control System were developed using this pattern, as shown in Section 5. Figure 6.7 presents a diagram describing the devised design pattern.

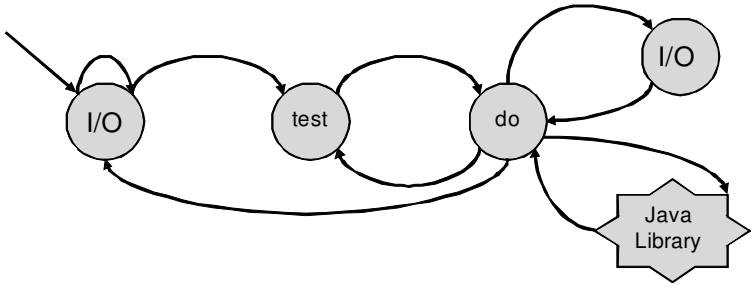

Figure 6.7: Menu design pattern state machine

It can be seen from Figure 6.7 that the *Menu* pattern is composed of four states. The center-left *I/O* state catches the first user selection from the *Menu* options. Proceeding, the state machine reaches the *test* state, where a specific behavior is selected to the chosen *Menu* option. After, the machine goes to *do* state, where the behavior is executed. Each time an Alloy operation is called within the *do* state, a runtime Java library is requested, in order to obtain the Java implementation of the called Alloy operation. The right-upper *I/O* state allows to ask for data input from the *do* state. When the *do* state finishes its execution, the machine can branch to the *test* state or to the left-centered *I/O* state, depending on the control flow specified within the *do* state.

A piece of code of the Address Book generated from an Alloy model that uses the pattern *Menu* is shown in Figure 6.8.

Alloy state machine specification	Java Menu of the State Machine

one sig Menu{ }
sig State {
book : one Book,
addition, deletion, found, done, exit: set Menu
}
fact stateTransition {
all s: State, s': ord/next[s] {
one n: Name ǀ one a : Phone {
(Menu in s.addition) =>
add [s.book,s'.book,n,a] && s'.done = Menu {
(Menu in s.deletion) =>
del [n,s.book,s'.book] && s'.done = Menu {
(Menu in s.found) =>
some lookup [s.book,n] && s'.done = Menu
}}}}
pred solve {
ord/last.done = Menu
}
	public enum Menu {
	addition, deletion, found, done, exit
	}
	while(true) {
	if(menu_enum.ordinal()== Menu. addition()) {
	// read name and phone
	// add name and phone
	menu_enum = Menu.Done; }
	if(menu_enum.ordinal()== Menu. deletion ()){
	// read name
	// del contact
	if(menu_enum.ordinal()== Menu. found ()){
	// read name
	// return phone
	menu_enum = Menu.Done; }
	if(menu_enum.ordinal()==Menu. done. ()){
	// read new Menu option }
	if(menu_enum.ordinal()== Menu. exit.()){
	break; }
	}

Figure 6.8: Portion of the synthesized reactive state machine by Compiler v.1

As mentioned before, the second version of the compiler uses Alloy predicates and functions called by a *run* command to infer the state machine. Not all run commands are considered in the state machine generation process; only the ones calling predicates or functions transformed into Java methods of classes instantiated within the State

signature are used by the generation process. The algorithm used by the compiler v. 2 (and compiler v.3) is show in Figure 6.9. This procedure does not include synthesis of the input actions.

A portion of the generated reactive state machine for the Vending Machine model is illustrated in Figure 6.10. The I/O generation follows the same principle as the first compiler approach: constructs are inserted before the method call (lines 15 to 17 – Figure 6.10). The *menu_enum* Java enumeration is created to save the current state of the state machine. After any machine operation, it goes to *done* state (line 21 – Figure 6.10), where a new action can be taken.

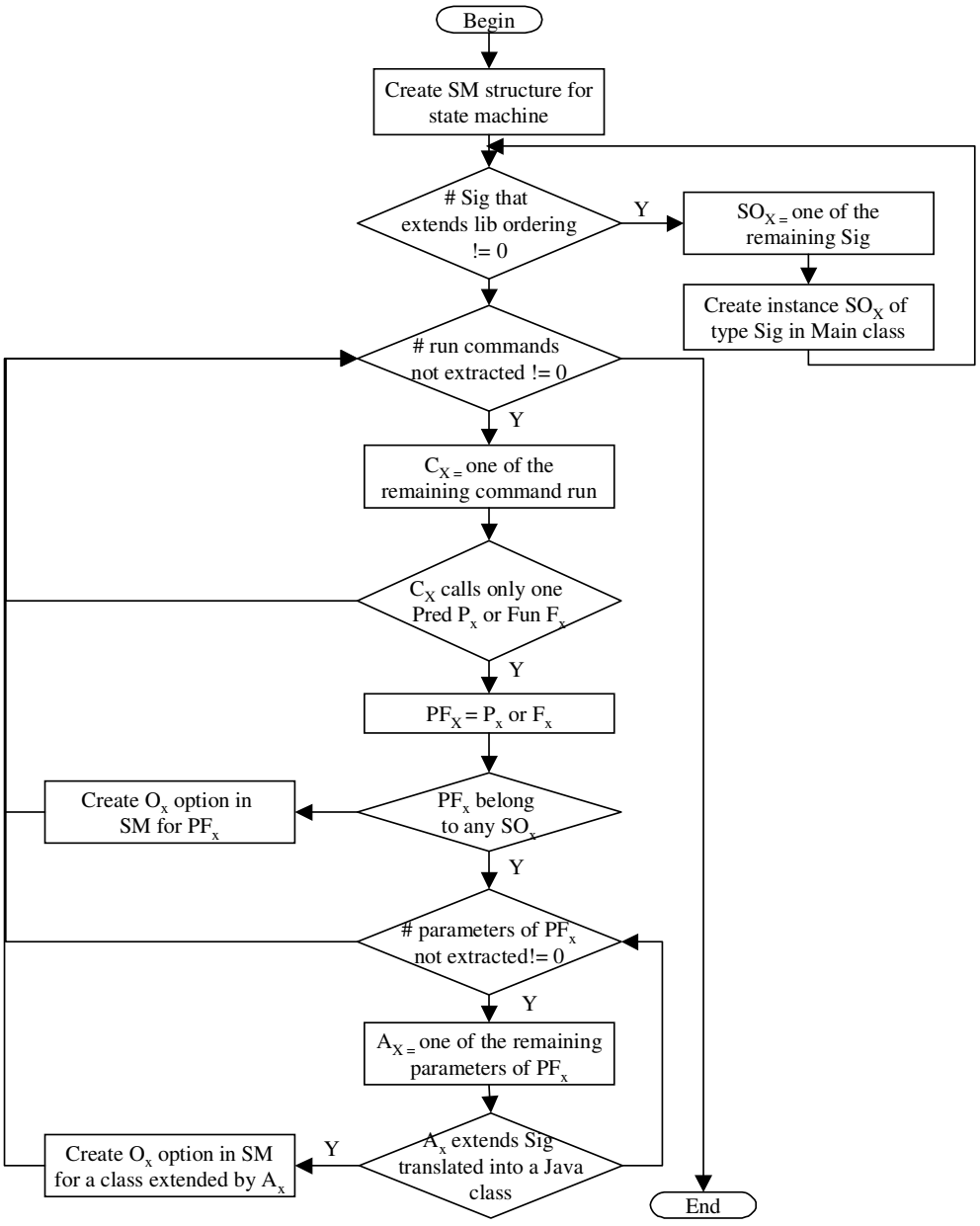

Figure 6.9: Procedure for synthesizing the Java reactive state machine

Alloy state machine specification	Java synthesized state machine

fact DrinkBuying {
all s: State, s': ord/next[s] |
some d: Drink | some c: Coin |
d.price <= #s.machine.coinStorage =>
buyDrink[s.machine,s'.machine,d]
else addCoin[s.machine,s'.machine,c]
}
run addCoin
run buyDrink
	1. public class Main {
	2. public static void main(String[] args) {
	3. VendingMachine machine =
	4. new VendingMachine();
	5. Menu menu_enum = Menu.done;
	6. InputStreamReader stdin =
	7. new InputStreamReader(System.in);
	8. BufferedReader console =
	9. new BufferedReader(stdin);
	10. try {
	11. while(true){
	12. if(menu_enum.ordinal()==
	13. Menu.addcoin.ordinal()) {
	14. try {
	15. System.out.println("Enter Coin:");
	16. String c = console.readLine();
	17. machine.addCoin(c);
	18. }
	19. catch (IOException e)
	20. { e.printStackTrace(); }
	21. finally { menu_enum = Menu.done; }
	22. }
	...

Figure 6.10: Portion of the synthesized reactive state machine by Compiler v.2

7 EXPERIMENTAL RESULTS

This section shows the experimental results regarding code generation based on the rules previously explained and design space exploration capabilities of the approach.

7.1 Code Generation

In order to analyze the generated Java code, the Address Book, the Drink Vending Machine and the Elevator Control System Alloy applications were submitted to the code translator. The Address Book and the Vending Machine mix control and data-flow portions; the Elevator Controller is totally control-flow.

Figure 7.1 shows the quantitative analysis between the Alloy and Java constructions existing in the model and in the Java code. The results show that mixed applications benefit more of the proposed approach in terms of the amount of generated lines of code. This is due to the absence of explicit data structures in the Alloy models, which on the other hand must be generated in the Java code; this is evident when considering the less intensive use of diverse Alloy constructions by the address book, but yet the generated code is the largest among the presented applications.

Regarding the Elevator Control System, the transformation was not advantageous considering the gain in abstraction. However, the generated code was based on a verified model, being not necessary to perform exhaustive tests on the Java code. This reduction on the amount of code generated is due to the lack of data structures in the Elevator Alloy model and the need of fewer Alloy operations to model this system. The transformation process created only one class, which contains the Java main method with the main state machine, and one enumeration containing the menu options.

The approach does not cover all Alloy operators and constructions, such as facts. In the applications modeled, facts are basically used for State machine manipulation and simulation purposes. Therefore, applications that have facts in different situations are still not supported.

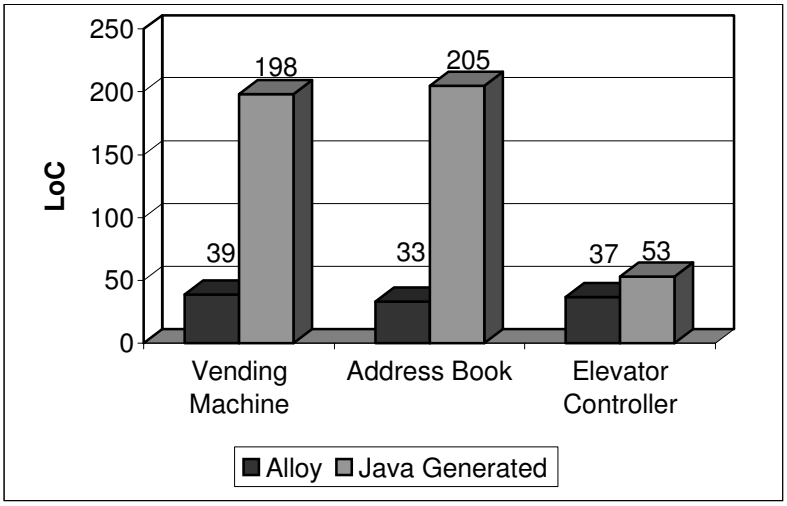

Figure 7.1: Number of lines written in Alloy and generated in Java

7.2 Design Space Exploration

This section presents the results in terms of memory use, performance, consumed energy and power for the Java generated code from Alloy models.

The FemtoJava processors used in this study were the multicycle version (ITO; CARRO; JACOBI, 2001) and a pipeline version (BECK; CARRO, 2003). The multicycle version is suited for embedded applications that require small area and low performance, while the pipeline version is improved in terms of performance, but increasing area and power consumed. The instructions implemented in the multicycle version are executed in 3, 4, 7 or 14 cycles due to the memory accesses. The pipeline version deals with data dependency by using *forwarding* and it has five stages: instruction fetch, decode, operand fetch, execution and write.

The results presented were estimated by the DESEJOS tool (MATTOS; CARRO, 2007). The tool calibration for performance was accomplished by a cycle-accurate processor simulator (BECK; CARRO, 2003), and the calibration for energy consumption was performed by the Synopsis Power Compiler tool (SYNOPSYS, 2008), from the synthesis of the processor descriptions in VHDL.

Estimates of physical properties have been obtained for the Address Book and the Drink Vending Machine Java code automatically generated from their Alloy model. The execution instances are the following: 100 insertions and 100 searches in the Address Book, and insertion and consumption of 60 coins and 60 drinks in the Vending Machine.

Table 7.1 presents physical properties estimates for three solutions of the Address Book, while Table 7.2 shows the result of the same experiment for five different solutions of the Vending Machine, executing both on FemtoJava multicycle and pipeline. Each solution has the same FSM; the only variation between them is the assignment of distinct data structures to the existing collections.

Table 7.1: Results of Power, Energy and performance for the Address Book

Solution	Multicycle			Pipeline		
	Performance (cycles)	Energy consumption (Joules)	Average power (mWatts)	Performance (cycles)	Energy consumption (Joules)	Average power (mWatts)
# 1	359,253,773	1.3076	18.1988	168,216,156	1.1820	40.6535
# 2	1,731,066	0.0061	17.7807	970,068	0.0045	32.8869
# 3	1,731,066	0.0061	17.7807	970,068	0.0045	32.8869

Table 7.2: Results of Power, Energy and performance for the Vending Machine

Solution	Multicycle			Pipeline		
	Performance (cycles)	Energy consumption (Joules)	Average power (mWatts)	Performance (cycles)	Energy consumption (Joules)	Average power (mWatts)
# 1	33,466,145	0.1344	20.0855	18,259,424	0.1096	30.9115
# 2	34,575,240	0.1394	20.1674	18,902,957	0.1145	31.1649
# 3	34,512,434	0.1391	20.1578	18,870,747	0.1134	30.0570
# 4	8,314,412	0.0388	23.3420	5,025,923	0.0303	31.2090
# 5	8,251,699	0.0384	23.3265	4,993,165	0.0292	30.4029

In the Address Book, three solutions were generated for the *Contact* data structure: #1 uses *FastList*, #2 uses *FastMap* and #3 uses *FastSet*. The five generated solutions for the Vending Machine contain the following pairs of data structures representing the coin and drink storage, respectively: #1:(*FastList*, *FastList*), #2:(*FastList*, *FastMap*), #3:(*FastList*, *FastSet*), #4:(*FastSet*, *FastMap*) and #5:(*FastSet*, *FastSet*).

Results show that only by varying used data structures, there are significant design tradeoffs between solutions, due to the variation in total number of cycles. In the Address Book application, solutions based in *FastSet* and *FastMap* were better regarding performance, because these structures are implemented with hashing in Javolution. For the Vending Machine application, solutions #4 and #5 take advantage of not using linked lists as well, reducing algorithmic complexity from O(n) to O(1) when accessing data. Results for both applications are coherent. All solutions in Vending Machine have intensive elements insertion and deletion operations over object collection, and insertion in hashing data structures is usually fast. The Address Book takes even more advantage in using hashing structures, because both insertion and search are fast in this case (LAFORE 2002).

Concerning physical properties only, there are significant tradeoffs between the multicycle and pipeline architectures. The pipeline versions consume a less energy than the multicycle ones; besides, there are significant performance improvements in the pipeline versions. If the extra power consumption is acceptable, it is very advantageous to adopt the pipeline architecture. Clearly, results presented in Table 7.1 and 7.2 show the importance and the need of design space exploration in order to properly support software automation.

Figure 7.2 and 7.3 present the results in terms of code size for both applications. These codes size do not include memory consumed by the Javolution library.

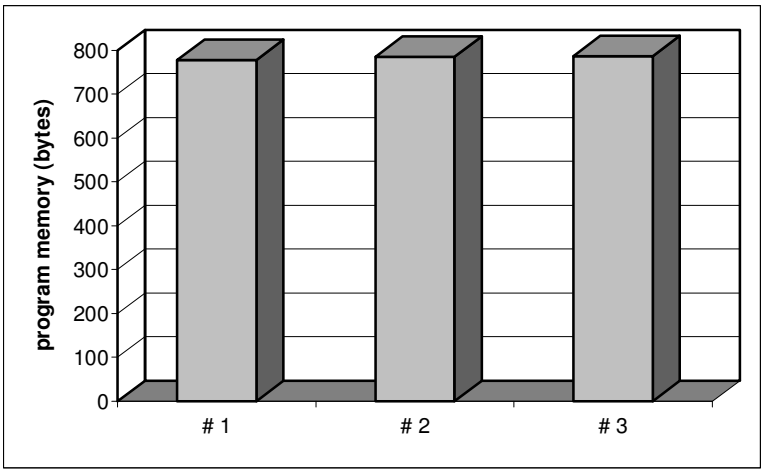

Figure 7.2: Memory results for the Address Book solutions

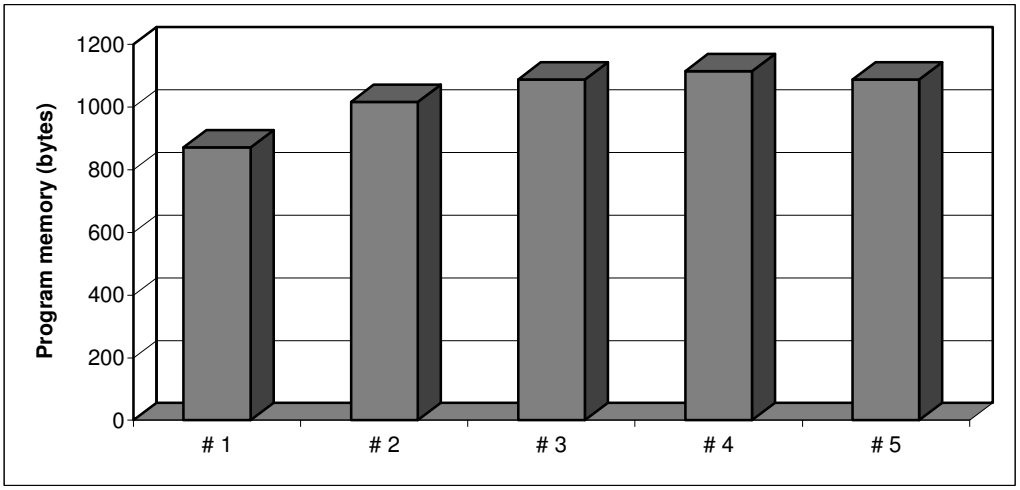

Figure 7.3: Memory results for the Vending Machine solutions

The FemtoJava pipeline and multicycle instruction set are the same, therefore, the program memory occupation is the same in both multicycle and pipeline embedded hardware platforms. As expected, the generated codes are very similar regarding their size, however, the algorithms behave very different during execution. It can be seen in Figure 7.3 the program memory overhead of hashing data structures (*Map* and *Set*) against linked lists, due to the information needed to maintain a hash function. This cannot be noticed in the memory results for Address Book. Memory experiments are usually more significative for applications larger than the ones presented, or in this case, applications that manipulate huge data structures.

8 ANALYSIS OF THE PROPOSED APPROACH

The main advantage of this approach for automating embedded software development is that the input language provides data structure abstraction, and this also allows the later design space exploration. Other FSM-based approaches for embedded software synthesis like Esterel do not follow the increasing ubiquity of embedded systems, which nowadays demand complex and higher-level data structures to represent their applications. Their design space exploration methodologies are restricted to those allowed by FSM optimizations.

Another advantage is that the Alloy Analyzer already performs formal checking and model simulation. Using the best SAT solvers available today, the analyzer can examine spaces that are several hundred bits wide (JACKSON, 2006). Like model checking, it considers all possible scenarios within the introduced bounds. Unlike model checking, the Alloy Analyzer does not examine all the possible scenarios; instead, it searches for a bad scenario (one that results in failure) from a pool of randomly generated ones. This makes the proposed approach even more attractive, because the developer can model a system, check the integrity of the model based on lightweight and object-based formal method, and automatically obtain the source code optimized for the available resources.

Alloy is an attractive language to model complex problems, but its simplicity may cause a power loss. As a First-order logic language, Alloy is not suited to model problems that involve higher-order logic or temporal logic. A support for temporal logic is desirable for specification, validation and verification of reactive and real-time embedded systems. The approach to code generation presented could offer means for including timing and synchronization primitives in the generated Java code, but a high-level verification of such constraints would not be possible.

Authors like (HUDAK, 1989) claim that it is possible to express concurrency in a more natural fashion by using declarative languages. This characteristic is highly desirable with today's multi-core CPU design. However, the approach does not support multithread yet, neither provides means for parallelism extraction. Future works plan to provide other DSE capabilities besides data structure selection, including those related to concurrency exploration.

According to (JACKSON, 2006), the Alloy language was developed for specifications modeling and analysis. Therefore, there is a huge gap between an Alloy specification and the code implementation itself. The approach here presented showed that it is possible to overcome this gap for applications that implement basically the untimed MoC, and use some data structure. The initial purpose of the research was to

use a declarative language for modeling a variety of MoCs. This idea still remains, and future works plan to expand the approach to support more MoCs, or even more heterogeneous specifications. The fact is that, despite of offering libraries for Integer and numeric operations like addition, subtraction, multiplication and division, Alloy does not provide good ways for modeling intensive calculations. For this reason, more effort will be required for the approach to support dataflow models and still provide the desired abstraction.

Regarding the source language, many Alloy constructions are still not covered by the proposed approach, and therefore cannot be mapped to Java code. The language coverage is done gradually, starting from modeling examples and then, if required, creating new rules for generating their code. One considerable issue is that the approach does not cover Alloy facts, except for those related to the state machine modeling. In section 4.3, most of the abstraction achieved when comparing alloy models to Java and Esterel respective codes was due to the use of facts. They have a similar role as Aspects in the UML approach. As facts are intrinsic in the code, they are harder to be translated than other constructions. Research on facts translation already started in the group, and once the approach is able to translate all possible facts to Java code, the set of Alloy constructions will be almost totally covered.

The approach still presents limitations in I/O mapping. Currently, outputs and inputs are performed automatically in the generated code, and only standard I/O (that reads characters from the keyboard and sends characters to the screen) is supported. It is somewhat mandatory to provide means for modeling outputs production whenever it is necessary. Also, interfaces to support other I/O mechanisms, such as port access in FemtoJava, should be provided soon.

Embedded software designers can benefit from the approach, once it provides design space exploration besides the abstraction in development. However, there is always an overhead introduced by the automation process. By comparing the hand-written Address Book in Section 3.2.1 with an Address Book that implements the same requirements generated by the Alloy-to-Java compiler (Section 7.1), it is possible to see that the hand-written code is about 60% of the size of the automatically generated code. One of the next steps of the research is to instrument hand-written codes and compare their physical attributes with those obtained from automatically generated codes, in order to evaluate the quality of the automation process and measure its impact in physical constraints.

9 CONCLUSIONS

This work proposed a software development approach based in the use of a declarative language and targeted to the embedded domain. In the proposed approach, the high-level specification is described using the Alloy declarative language, in order to provide high abstraction and formal verification at the same time. From the Alloy specification, it is possible to obtain Java source code suited for the target platform available resources. Rules were devised to map Alloy models into Java source code, for applications that implement the untimed model of computation. A code translator developed in the group produces different Java code by varying its data structures. Thanks to the integration between the approach and DESEJOS estimation tool, a designer may explore different solutions, verify the performance and memory footprint for each one of them, and choose the one that fits better to the target resources.

Current problems with available high-level tools in the embedded systems context have also been presented. Among the main issues of these tools is the fact that only a few provide support for heterogeneous applications, which are massively present in the new generation of embedded systems. Also, many tools for embedded development do not provide great abstraction even for a single MoC. On the other hand, tools that provide high abstraction level, like the UML based ones, are often apart from optimization strategies or are not enough reliable to support development of certain types of embedded systems.

The proposed approach has proved to be highly abstract for at least five applications. For three of them, modeled under the untimed MoC, it was managed to obtain both abstraction and automation. The choice of the Alloy language for specification was not in vain: other declarative languages analyzed did not provide enough abstraction to motivate the development of an approach based on them. Besides, tools that provide lightweight formal verification such as the Alloy Analyzer are a valuable contribution for the entire embedded software development process.

In the future, the approach shall provide interfaces to deal with input and output actions, as well as the support of intensive dataflow applications, in addition to Alloy's fact construction. The translation scheme of AAL annotations into an Alloy model is currently being coded in the group, in order to enable the use and verification of legacy code as well as software libraries, maximizing reuse. Moreover, it is necessary do deal with complex examples, which might mix various models of computation and concurrency, in order to observe with more details the abstraction gain one can achieve with the proposed methodology.

REFERENCES

ANDROMDA TEAM. **AndroMDA v. 3.3**. Available at: <http://www.andromda.org>. Visited on: 2008.

ARMSTRONG, J. et al. **Concurrent programming in Erlang**. New York: Prentice-Hall, 1993.

BALARIN, F. et al. Synthesis of Software Programs for Embedded Control Applications. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 18, n. 6, p. 834-849, June 1999.

BECK, A.; CARRO, L. Low Power Java Processor for Embedded Applications. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEM-ON-CHIP, VLSI-SOC, 2003, Darmstadt. **Proceedings…** Darmstadt: Technische Universitädt, 2003. p. 239-244.

BERRY, G. **Esterel Compiler v. 5.21**. Available at: <http://www-sop.inria.fr/meije/esterel/esterel-eng.html>. Visited on: 2008.

BERRY, G. and GONTHIER, G. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. **Science of Computer Programming**, Amsterdam, v. 19, n. 2, p. 87-152, 1992.

BENINI, L. et al. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. **The Journal of VLSI Signal Processing**; Netherlands, v. 41, n.3, p. 169-182, Sept. 2005.

BIERMANN, A. W.; BAUM, R. I.; PETRY, F. E. Speeding up the Synthesis of Programs from Traces. **IEEE Transactions on Computers**, New York, v. c-24, n. 2, p. 122-136, Feb.1975.

BRISOLARA, L. et al. A Comparison between UML and Function Blocks for Heterogeneous SoC Design and ASP Generation. In: MARTIN, G.; MULLER, W. (Ed.). **UML for SOC Design**. Netherlands: Springer, 2005. p. 199-222.

BROOKS, C. et al. **Heterogeneous Concurrent Modeling and Design in Java**. Berkeley: EECS Department, University of California, Berkeley, 2007. v.1.

CARRO, M. et al. **High-level languages for small devices: a case study**. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURE AND SYNTHESIS FOR EMBEDDED SYSTEMS, CASES, 2006, Seoul, Korea. *Proceedings*...New York: ACM, 2006. p. 271-281.

CASPI, P. et al. LUSTRE: A declarative language for programming synchronous systems. In: ACM SIGALT-SIGPLAN SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, POPL, 14., 1987. **Proceedings…** New York: ACM, 1987. p. 178-188.

CHATZIGEORGIOU, A. and STEPHANIDES, G. Evaluating Performance and Power of Object-Oriented vs. Procedural Programming in Embedded Processors. In: ADA-EUROPE INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE TECHNOLOGIES, ADA-EUROPE, 7., 2002, Vienna, Austria. **Proceedings…** Berlin: Springer, 2002. p. 65-75. (Lecture Notes in Computer Science, v. 2361).

CHEN, C. et al. Design and Validation of a General Security Model with the Alloy Analyzer. Alloy Workshop, Portland, Oregon, November 6, 2006.

DAUTELLE, J. Fully Deterministic Java. In: AIAA SPACE CONFERENCE AND EXPOSITION, 2007, Long Beach, CA. **Proceedings…**[S.l.]: AIAA, 2007. p.18-20.

DAVID, A.; MOLLER, M. O.; YI, W. Formal Verification of UML Statecharts with Real-Time Extensions. In: INTERNATIONAL CONFERENCE ON FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING, FASE, 5., 2002, Grenoble, France. **Proceedings…** Berlin: Springer, 2002. p. 208-241. (Lecture Notes in Computer Science, v. 2306).

DOMAIN SOLUTIONS. **CodeGenie v. 4.3**. Available at: <http://www.ooagenerator.com/codegenie.htm>. Visited on: 2008.

ESTEREL TECHNOLOGIES. **Scade v. 6.0**. Available at: <http://www.esterel-technologies.com/products/scade-suite/>. Visited on: 2008.

EXECUTABLE UML UNIMOD. **Unimod v. 1.3**. Available at: <http://unimod.sourceforge.net>. Visited on: 2008.

GALEOTTI, J. P.; FRIAS, M. DynAlloy as a Formal Method for the Analysis of Java Programs. **Software Engineering Techniques: Design for Quality**, v. 227, p. 249-260, 2007.

GEENSYS. **Reqtify v. 3.1.1.** Available at: <http://www.geensys.com/?Outils/Reqtify>. Visited on: 2008.

GENTLEWARE SOFTWARE. **Poseidon for UML**. Available at: <www.gentleware.com/products.html>. Visited on: 2008.

GIEGERICH, R.; KURTZ, S. A Comparison of Imperative and Purely Functional Suffix Tree Constructions. **Science of Computer Programming**, Amsterdam, v. 25, p. 187-218, 1995.

GOMAA, H. **Designing Concurrent Distributed, and Real-Time Applications with UML**. Reading, MA: Addison-Wesley, 2000.

GRAAF, B.; LORMANS, M.; TOETENEL, H. Embedded Software Engineering: the state of the practice. **IEEE Software**, Los Alamitos, CA, v. 20, n. 6, p. 61-69, Nov./Dec. 2003.

HAREL, D. StateCharts: a Visual Formalism for Complex Systems. **Science of Computer Programming**. New York, v. 8, p. 231-274, 1987.

HOLZMANN, G. J. The Model Checker SPIN. **Proceedings of IEEE**, Piscataway, NJ, v. 23, n. 5, p. 279-295, 1997.

HUDAK, P. Conception, Evolution, and Application of Functional Programming Languages. **ACM Computing Surveys**, New York, v. 21, n. 3, p. 359-411, Sept. 1989.

HUDSON, S.E., FLANNERY F.; ANANIAN, C.S. **CUP: Parser Generator for Java v. 11**, 1999. Available at: <http://www2.cs.tum.edu/projects/cup/>. Visited on: 2008.

ITO, S. A., CARRO, L.; JACOBI, R. P. Making Java work for microcontroller applications. **IEEE Design & Test of Computers**, California, v. 18, n. 5, p. 100-110, Sept./Oct. 2001.

JACKSON, D. **Software Abstractions – Logic, Language and Analysis**. Cambridge: The MIT Press, 2006.

JANTSCH, A. **Modeling Embedded Systens and SoC's – Concurrency and Time in Models of Computation**. San Francisco, CA, USA: Morgan Kaufmann, 2004.

JZOX. **JGenerator v. 2.2 MX**. Available at: <http://www.jzox.com>. Visited on: 2008.

KAN, S. H. **Metrics and Models in Software Quality Engineering**. Boston: Addison-Wesley, 2002.

KARSAI, G. et al. Model-Integrated Development of Embedded Software**. Proceedings of the IEEE**, Piscataway, NJ, v. 91. n. 1, p. 145-164, Jan. 2003.

KLEIN, G. **JFlex: The Fast Scanner Generator for Java v. 1.4.1**, 2004. Available at: <http://jflex.de>. Visited on: 2008.

KOOPMAN, P. **Reliability, Safety, and Security in Everyday Embedded Systems**. In: LATIN-AMERICAN SYMPOSIUM ON DEPENDABLE COMPUTING, LADC, 3., 2007, Morella, Mexico. **Proceedings...** Berlin: Springer, 2007. p. 1-2. (Lecture Notes in Computer Science, v. 4746).

KHURSHID, S., MARINOV D.; JACKSON, D. An Analyzable Annotation Language. **ACM SIGPLAN Notices**, New York, v. 37, n. 11, p. 231-245, 2002.

LAFORE, R. **Data Structures & Algorithms in Java**. Indianapolis: Sams, 2002.

LAPEDUS, M. Half of design projects are late. **EE Times**, March 2006. Available at: <http://www.eetimes.com>. Visited on: 2008.

LAWTON, G. Moving Java into Mobile Phones. **Computer**, Los Alamitos, CA, v. 35, n. 6, p. 17-20, June 2002.

LEE, E. A. Embedded Software. **Advances in Computers**, New York, v. 56, p. 56-97, 2002.

LEE, H. B. and ZORN, B. G. BIT: A Tool for Instrumenting Java Bytecodes**.** In: USENIX SYMPOSIUM ON INTERNET TECHNOLOGIES AND SYSTEMS, 1997, Monterey, CA. **Proceedings…** Berkeley, CA: USENIX Association, 1997. p. 73-82.

MASSONI, T., GHEYI, R. and BORBA, P. A UML class diagram analyzer. In: WORKSHOP ON CRITICAL SYSTEMS DEVELOPMENT WITH UML, 2004. **Proceedings…** [S.l.: s.n.], 2004. p. 100-114.

MATTOS, J. et al. Making Object Oriented Efficient for Embedded System Applications. In: BRAZILIAN SYMPOSIUM INTEGRATED CIRCUIT DESIGN, SBCCI, 18., 2005. **Proceedings…** New York: ACM Press, 2005. p.104-109.

MATTOS, J.; CARRO, L. Object and Method Exploration for Embedded Systems Applications. In: BRAZILIAN SYMPOSIUM INTEGRATED CIRCUIT DESIGN, SBCCI, 20., 2007, Rio de Janeiro. **Proceedings…** New York: ACM Press, 2007. p.318-323.

MIT SOFTWARE DESIGN GROUP. **Alloy Language and Alloy Analyzer v. 4.1.8**. Available at: <http://alloy.mit.edu/>. Visited on: 2008.

MOSER, E.; NEBEL, W. Case study: System model of Crane and embedded control. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 1999. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, 1999. p. 721.

MURRAY, C. OS Rides To Rescue. **EE Times**, October 2005. Available at: <http://www.eetimes.com>. Visited on: 2008.

NOKIA. **Internet key to next phase of growth in mobile phone industry**. Press Releases. November 29th, 2006. Available at: <http://press.nokia.dk/release.php?id=2775015>. Visited on: 2008.

NUGROHO, R. P. **Java Sokoban**. Jakarta, Indonesia, 1999. Available at: <http://javaboutique.internet.com/Sokoban>. Visited on: 2008.

OBJECT MANAGEMENT GROUP. Model Driven Architecture. [S.l.], 2000. White Paper v. 3.2, November, 2000. Available at: <http://www.omg.org>. Visited on: 2008.

OBJECT MANAGEMENT GROUP. **Precise Semantic Action for UML**. In *OMG Unified Modeling Language Specification v. 1.5*, March, 2003. Available at: <http://www.omg.org>. Visited on: 2008.

OKASAKI, C. Functional Perl: Red-black Trees in a Functional Setting. **Journal Functional Programming**, Cambridge, v. 9, n. 4, p. 471-477, July 1999.

RATIONAL SOFTWARE. **Rational Rose**. Available at: <http://www-01.ibm.com/software/awdtools/developer/rose/index.html>. Visited on: 2008.

SANDER, I.; JANTSCH, A. System Modeling and Transformational Design Refinement in ForSyDe. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, New York, v. 23, n. 1, p. 17-32, Jan. 2004.

SANGIOVANNI-VINCENTELLI, A. Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design. **Proceedings of the IEEE**, Piscataway, NJ, v. 95, n. 3, p. 467-506, 2007.

SCHWARTZ, J.; FREUDENBERGER, S.; SHARIR, M. Experience with the SETL Optimizer. In **ACM Transactions on Programming Languages and Systems, TOPLAS**, New York, v. 5, n.1, p. 26-45, 1983.

SEBESTA, R. W. **Concepts of Programming Languages**. 7th ed. Boston: Addison Wesley, 2005.

SELIC, B. Models, Software Models and UML. In: LAVAGNO, L.; MARTIN, G.; SELIC, B. (Ed.). **UML for Real**: Design of Embedded Real-Time Systems. Boston: Kluwer Academic, 2003. p. 1-16.

SOMMERVILLE, I. **Software Engineering**. 8th ed., Reading: Addison-Wesley, 2006.

STANSIFER, R. Imperative versus Functional. **SIGPLAN Notices**, New York, v. 25, n. 4, p. 69-72, Nov. 1989.

STROM, O. et al. On the Utilization of Java Technology in Embedded Systems. **Design Automation for Embedded Systems**, New York, v. 8, n. 1, p. 87-106, Mar. 2003.

SYNOPSYS. **Synopsys's Power Compiler**. Available at: <http://www.synopsys.com/products/power/power_ds.html>. Visited on: 2008.

SZTIPANOVITS, J.; KARSAI, G. Model-Integrated Computing. **Computer**, Los Alamitos, CA, p. 110-112, Apr. 1997.

TAN, Y. et al. **A Novel JAVA Processor for Embedded Devices.** In:INTERNATIONAL WORKSHOP ON SYSTEMS, ARCHITECTURES, MODELING AND SIMULATION, SAMOS, 2005, Samos, Greece. **Proceedings…** Springer: Berlin, 2005. p. 112-121. (Lecture Notes in Computer Science, v. 3553).

TELELOGIC. **Rhapsody v. 7.3**. Available at: <http://modeling.telelogic.com/modeling/products/rhapsody/index.cfm>. Visited on: 2008.

THE MATHWORKS. **Matlab v. 7**. Available at: <http://www.mathworks.com/products/matlab/>. Visited on: 2008.

THE MATHWORKS. **Real-Time Workshop v. 7.1**. Available at: <http://www.mathworks.com/products/rtw/>. Visited on: 2008.

THE MATHWORKS. **Simulink v. 7.1**. Available at: <http://www.mathworks.com/products/simulink/>. Visited on: 2008.

WADLER, P. Why no one uses functional languages. **ACM SIGPLAN Notices**, New York, v. 33, n. 8, p. 23-27, 1998.

WARREN, I. et al. An Automated Formal Approach to Managing Dynamic Reconfiguration. In: IEEE/ACM INTERNATIONAL CONFERENCE ON

AUTOMATED SOFTWARE ENGINEERING, ASE, 21., 2006, Tokyo, Japan. **Proceedings…** [S.l.: s.n.], 2006. p. 18-22.

WALLACE, M.; RUNCIMAN, C. Extending a Functional Programming System for Embedded Applications. **Software- Practice and Experience**, [S.l.], v. 25, n. 1, p. 73-96, Jan. 1995.

WOODWARD, M. V.; MOSTERMAN, P. J. Challenges for embedded software development. In: IEEE INTERNATIONAL MIDWEST SYMPOSIUM ON CIRCUITS AND SYSTEMS, MWSCAS, 50., 2007, Montreal, Canada. **Proceedings…** [S.l.: s.n.], 2007. p. 630-633.

XDOCLET TEAM. **Xdoclet v. 1.2, 2004**. Available at: <http://xdoclet.sourceforge.net/xdoclet/index.html>. Visited on: 2008.

ZAVE, P. A Formal Model of Addressing for Interoperating Networks. In: INTERNATIONAL SYMPOSIUM OF FORMAL METHODS EUROPE, 13., 2005, Newcastle, UK. **Proceedings…** Berlin: Springer, 2005. p. 318-333. (Lecture Notes in Computer Science, v. 3582).

APPENDIX A UMA ABORDAGEM PARA GERAÇÃO DE SOFTWARE EMBARCADO BASEADA EM MODELOS DECLARATIVOS ALLOY

Este anexo apresenta um resumo expandido da dissertação.

Introdução

Os sistemas embarcados estão cada vez mais presentes no dia a dia das pessoas. De acordo com uma projeção da empresa Nokia (NOKIA 2006), haverá aproximadamente 4 bilhões de celulares em operação no mundo. Estes aparelhos, que estão entre os sistemas embarcados mais importantes da atualidade, oferecem muitas vantagens além de serviços telefônicos, como câmeras digitais, agendas, jogos, acesso à internet, tocadores MP3, rádio, etc. Estes recursos diversificados aumentam a complexidade de desenvolvimento, enquanto o período de tempo exigido para que os novos equipamentos cheguem ao mercado está cada vez menor.

O domínio de sistemas embarcados é dirigido por fatores de confiabilidade, custo e tempo de chegada ao mercado (GRAAF; LORMANS; TOETENEL, 2003). Nas aplicações atuais, o software é responsável pelo maior gargalo e, em muitos casos, soluções em software são preferíveis a soluções em hardware, pois o software é mais flexível, fácil de atualizar e pode ser reusado (GRAAF; LORMANS; TOETENEL, 2003).

Para lidar com a crescente complexidade, desenvolvedores têm procurado especificar o sistema utilizando altos níveis de abstração. A pesquisa atual em projeto de sistemas embarcados enfatiza que o uso de técnicas que se iniciem nas primeiras etapas do ciclo de desenvolvimento é crucial para o sucesso do projeto. Alguns autores como (SELIC, 2003) e (GOMAA, 2000) argumentam que esta abordagem é a única maneira de lidar com a complexidade encontrada nas novas gerações de sistemas embarcados. Para lidar com o curto tempo de chegada ao mercado, são necessárias ferramentas para síntese automática capazes de gerar código a partir de modelos abstratos.

Na engenharia de software, há ferramentas CASE que geram código a partir modelos abstratos, e por esta razão elas são amplamente utilizadas para automatizar o processo de geração de software. No entanto, estas ferramentas são geralmente restritas a um domínio de aplicação específico, e por este motivo não são recomendadas para projeto de sistemas embarcados, que geralmente inclui múltiplos domínios. A especificação de um celular, por exemplo, inclui não apenas processamento de sinal

para o domínio de telecomunicações, aplicável ao modelo de computação tempo-discreto (LEE, 2002), como também lógica seqüencial para descrever jogos e agendas. Mesmo ferramentas como ForSyDe (SANDER; JANTSCH, 2004), específicas para síntese de software embarcado, geralmente não promovem a abstração desejada.

Além dos desafios introduzidos pelo fato dos sistemas embarcados compreenderem múltiplos domínios, o desenvolvimento de software embarcado requer suporte a alguns requisitos específicos do domínio, como operação em tempo real, limitação no tamanho máximo de memória de programa ocupada e consumo de energia. Dispor de ferramentas que ofereçam exploração de espaço de projeto é uma contribuição valiosa na busca por melhor relação entre tamanho de memória e tempo de execução.

Além das restrições físicas inerentes ao domínio de embarcados, curto tempo de chegada ao mercado e a presença de aplicações que envolvem múltiplos domínios, o software convencional e embarcado diferem em muitos outros aspectos, conforme mostra a tabela A.1. Assim sendo, as práticas convencionais de engenharia de software não podem ser aplicadas diretamente ao domínio de sistemas embarcados sem considerar os requisitos específicos do produto embarcado.

Tabela A.1: Comparativo entre software convencional e embarcado

	Software Convencional	**Software Embarcado**
Tempo de chegada ao mercado	Longo	Curto
Memória	Sem restrições	Restrita
Potência	Sem restrições	Restrita
Domínio de aplicação	Único	Múltiplo
Reuso de código	Benéfico	Geralmente prejudicial
Otimização de código	Dispensável	Necessária
Automação	Considerável	Pobre

Os sistemas embarcados também são comumente empregados em situações críticas (KOOPMAN, 2007), onde a segurança é um critério tão ou mais importante do que performance, e por este motivo é recomendável dispor de um cuidadoso suporte à validação e verificação (V&V). No entanto, ferramentas que oferecem atividades de verificação desde as primeiras etapas geralmente não são utilizadas no domínio embarcado por introduzirem aumento de custo e complexidade. Por outro lado, linguagens declarativas são conhecidas pela semântica bem definida, por promoverem ferramentas naturais para correção dos programas e possibilitar implementações de forma mais intuitiva (HUDAK, 1989).

Este trabalho propõe a geração automática de código para aplicações embarcadas a partir de modelos Alloy (MIT SOFTWARE DESIGN GROUP, 2008) e apresenta as regras de transformação dos modelos para a linguagem Java. Essa abordagem alia automação no desenvolvimento de *software* à confiabilidade da verificação promovida pelo uso de uma linguagem declarativa. Além disto, várias instâncias de código são geradas, de forma a melhor aproveitar os recursos disponíveis em diferentes versões do processador FemtoJava (ITO; CARRO; JACOBI, 2001), desenvolvido no grupo.

Trabalhos Relacionados

Comparações entre os paradigmas imperativos e declarativos estão em sua maioria restritas a certos algoritmos ou modelos de computação, como em como

(ARMSTRONG et al., 1993), onde a proposta de utilização de linguagem funcional é aplicável a apenas um modelo de computação no domínio embarcado. (WADLER, 1999) por sua vez promove comparações entre ambos paradigmas, mas sem utilizar números. Em (CARRO et al., 2006) um estudo de caso concreto é implementado a fim de demonstrar a viabilidade do uso de uma linguagem lógica de alto nível no projeto de aplicações embarcadas, apesar não demonstrar nenhuma preocupação com arquitetura ou eficiência do código.

Muitas das soluções existentes para a automação de software focam no gerenciamento de grandes sistemas de domínio específico, como banco de dados (XDOCLET TEAM, 2004), já que o software convencional é geralmente aplicável a um domínio em particular. São várias as ferramentas de automação baseadas em UML, como Poseidon for UML (GENTLEWARE SOFTWARE, 2008), que costumam proporcionar ao desenvolvedor uma capacidade de abstração considerável. Infelizmente nenhuma destas inclui verificação formal da solução modelada, e muitas foram desenvolvidas antes da definição de uma semântica precisa para UML (OBJECT MANAGEMENT GROUP INC, 2003), essencial para evitar interpretações ambíguas na automação do software.

A síntese de software para o domínio embarcado pode ser comparada a geração de código fonte (Java, C ou assembly) a partir de linguagens síncronas como Esterel (BERRY; GONTHIER, 1992) ou linguagens gráficas para especificação de sistemas de eventos discretos como StateCharts (HAREL, 1987). Apesar destas abordagens constituírem modelos formais e promoverem mecanismos para verificação formal, as linguagens não apresentam ganho significativo em abstração quando comparadas a linguagens de programação tradicionais. Por outro lado, abordagens baseadas em Model-Integrated Computing (MIC) estão adquirindo muitos adeptos na comunidade de embarcados por adotarem o paradigma de desenvolvimento baseado em modelos abstratos (SZTIPANOVITS; KARSAI, 1997). No entanto, por utilizarem linguagens de modelagem de domínio específico (KARSAI et al., 2003), ainda não são apropriadas para aplicações que apresentam diversos modelos de computação.

Ambientes para modelagem de sistemas embarcados também foram propostos tanto no meio acadêmico quanto comercial. Simulink (THE MATHWORKS, 2008-c) e Scade (ESTEREL TECHNOLOGIES, 2008) são ambientes comerciais gráficos para projeto e simulação de sistemas embarcados. Simulink provê um conjunto de bibliotecas que permite modelagem e simulação de uma gama de sistemas variantes no tempo, enquanto o Scade é mais adequado para descrição de aplicações críticas de segurança. O ambiente POLIS (BALARIN et al., 1999) desenvolvido em meio acadêmico promove geração de código baseada no modelo de Máquina de Estados Finita Concorrente (CFSM) criado pelos autores, não permitindo especificação de estruturas de dados. Metropolis (SANGIOVANNI-VINCENTELLI, 2007) foi proposto como extensão para Polis, e as técnicas de verificação formal por ele utilizadas representam aumento no tempo de desenvolvimento, o que não é bem aceito em função da exigência de curto tempo de chegada ao mercado.

Outros ambientes para modelagem e síntese de sistemas embarcados desenvolvidos em meio acadêmico procuram considerar a heterogeneidade das aplicações embarcadas, como ForSyDe (SANDER; JANTSCH, 2004). Para promover abstração para vários domínios, ForSyDe requer conhecimento e classificação de vários modelos de computação diferentes, o que pode vir a aumentar o tempo de desenvolvimento, além da abstração promovida ser próxima a de linguagens de programação comuns. Ptolemy II

(BROOKS et al., 2007) é um ambiente baseado em componentes Java com interface gráfica, que proporciona simulação e prototipação de sistemas heterogêneos. Apesar do considerável suporte a aplicações heterogêneas, Ptolemy II não provê ferramentas para verificação formal.

A maior parte das ferramentas para exploração do espaço de projeto no desenvolvimento de sistemas embarcados exploram paralelismo e proporcionam flexibilidade, beneficiando-se da disponibilidade de processadores reconfiguráveis e MPSoCs (BENINI et al., 2005). Outras abordagens como o Real Time Workshop (THE MATHWORKS, 2008-b) provêm exploração baseada no reuso de algoritmos comumente utilizados. O problema nestas abordagens está na restrição das bibliotecas a certos algoritmos, e apenas determinados tipos de aplicação são suportados em uma linguagem fixa.

Alguns trabalhos acadêmicos foram propostos com o intuito de utilizar Alloy principalmente para verificação formal de programas. (MASSONI; GHEYI; BORBA, 2004) propõe uma abordagem que transforma diagramas de classe UML com construções OCL para código Alloy, para fins de verificação. Outra abordagem utiliza uma linguagem de anotação baseada em Alloy (AAL) para inserir anotações em um código Java, possibilitando uma análise total automática em tempo de compilação (KHURSHID; MARINOV; JACKSON, 2002). Apesar destes trabalhos utilizarem Alloy, as especificações partem da linguagem Java, e por este motivo perdem nos mecanismos de abstração introduzidos pelo uso de Alloy para especificação de sistemas.

Análise do Uso de Linguagens Declarativas no Desenvolvimento de Sistemas Embarcados

Linguagens declarativas poderiam ser utilizadas no desenvolvimento de sistemas embarcados, uma vez que permitem descrições mais intuitivas do que as de linguagens tradicionais como Java. Foi implementado o MP3*, aplicação embarcada composta por um algoritmo IMDCT (utilizado em tocadores MP3), uma Agenda de telefones e os jogos da Velha e Sokoban. Todas aplicações que compõem o MP3* foram codificadas nas linguagens declarativas Ocaml (funcional) e Prolog (lógica), e adicionalmente, partes do MP3* como Jogo da Velha e Agenda foram codificadas em Esterel versão 5.21 (BERRY, 2008). O objetivo deste estudo é a análise do nível de abstração alcançado para aplicações heterogêneas através da mudança de paradigma imperativo para declarativo, considerando impacto em termos de desempenho e memória no domínio de embarcados.

Testes foram executados nas mesmas condições para todas linguagens e conjuntos de dados. A abstração foi avaliada em número de linhas de código. Resultados mostraram que é possível descrever diferentes modelos de computação em Prolog e Ocaml, mas por outro lado, o custo em desempenho e memória não deve ser negligenciado. A percentagem de linhas de código obtida com o uso de Prolog é de aproximadamente 50% com relação a Java em casos como o da Agenda, e no pior caso apresentou quantidade de linhas de código um pouco superior a do equivalente Java. Para aplicações como o Sokoban, o código Java é mais abstrato e utilizou menor quantidade de memória do que os equivalentes em Prolog e Ocaml. Para aplicações como Agenda e Jogo da Velha, que incluem controle e manipulação de dados e conseqüentemente envolvem diferentes modelos de computação, as linguagens declarativas apresentam maior quantidade de mecanismos de abstração do que Esterel, linguagem específica para sistemas embarcados reativos.

Os resultados em abstração para as linguagens declarativas não foram satisfatórios para todos os tipos de aplicações estudados: com base na métrica de linhas de código, mesmo assumindo interpretadores perfeitos para as linguagens declarativas, não é possível obter ganho de ordens de magnitude em abstração. Ainda assim, algumas características das linguagens declarativas são desejáveis, como a reduzida quantidade de erros apresentados em um código declarativo. Prolog, entre outras linguagens, são baseadas em lógica, logo, programas escritos nestas linguagens podem ser logicamente organizados, o que induz a menor taxa de erros e menor esforço de manutenção do código. Linguagens funcionais, com o uso de recursão e mecanismos como o de *pattern matching*, também permitem descrever um código com maior clareza.

Motivação para o Uso de Alloy

Alloy (JACKSON, 2006) é uma linguagem de modelagem baseada em lógica de primeira ordem, utilizada para expressar características estruturais complexas e comportamento. Partindo do princípio de que códigos não são meios ideais para expressar abstrações, a linguagem Alloy promove descrições em alto nível aliadas a verificação formal leve. Alloy Analyzer é um solucionador de restrições que provê simulação totalmente automática e verificação de modelos Alloy cujo escopo (finito) é especificado pelo desenvolvedor. O que diferencia Alloy de outras linguagens é o fato dos modelos serem ao mesmo tempo declarativos, analisáveis, suportarem estados de estrutura complexa e geralmente muito menores que suas respectivas implementações.

Modelos Alloy podem expressar tanto estrutura quanto comportamento. *Signature* é o mecanismo de estruturação de Alloy (JACKSON, 2006), semelhante a uma estrutura de dados do tipo conjunto. Cada *signature* representa um conjunto de átomos, além de possibilitar a introdução de campos, cada qual representando uma relação. A linguagem provê operadores para conjuntos, relações e herança. As restrições de modelos são representadas por fatos (restrições permanentes), predicados e funções (restrições válidas sob certas condições ou usadas para simulação) e asserções (implicações a serem verificadas). Do ponto de vista de implementação, parte do comportamento da aplicação é modelado utilizando predicados e funções, responsáveis pelas operações sobre estruturas de dados. A linguagem Alloy também disponibiliza bibliotecas como a de ordenamento, particularmente útil na modelagem de máquinas de estados sem expressar propriedades temporais.

Complementando o estudo demonstrado anteriormente, onde aplicações em linguagens declarativas foram comparadas a implementações em linguagens imperativas, um novo estudo inclui Alloy como representante das linguagens declarativas. Com o objetivo de analisar a capacidade de abstração promovida por Alloy, algumas aplicações apresentadas anteriormente (Jogo da Velha e Agenda) e um sistema de controle de guindaste foram modelados em Alloy. O controle de guindaste foi implementado apenas parcialmente, como interface em alto nível para módulos de uma solução proposta em (MOSER; NEBEL, 1999). A comparação novamente utilizou linhas de código como métrica de abstração, e não inclui linhas utilizadas para interface de entrada e saída (não disponível em Alloy).

A implementação do Jogo da Velha em Alloy é quase 50% menor do que a equivalente em Java, e aproximadamente 30% menor do que a versão em Esterel. Os resultados da Agenda foram ainda mais animadores: a implementação em Alloy é da mesma forma 50% menor que a implementação em Java, e aproximadamente 60% menor que a implementação em Esterel.Os resultados mostram que Alloy é capaz de

promover abstração tanto em estruturas de dados quanto controle. Considerando o sistema de controle de guindaste, resultados também mostram a vantagem da implementação Alloy, que apresenta aproximadamente 40% e 70% a menos de linhas de código do que as implementações em Java e Esterel respectivamente.Com base nestes resultados, entre as principais motivações para o uso de Alloy estão os ganhos em expressividade e abstração, juntamente com a capacidade de simulação, verificação e sintaxe orientada a objeto, favorecendo sua adoção na comunidade de embarcados.

A Abordagem Proposta

A Figura A.2 apresenta o fluxo da abordagem proposta para automação de software embarcado. Todos os requisitos funcionais são descritos na linguagem Alloy, e o modelo é então analisado pelo Alloy Analyzer. O modelo Alloy verificado é inserido no Tradutor de Modelos, que gera diferentes implementações em Java a partir de um mesmo modelo. Três estudos de caso foram modelados em Alloy, com o intuito de exemplificar e validar a abordagem: Agenda Telefônica, Sistema de Controle de Elevador e Máquina de Vendas. Para todos códigos fonte gerados, propriedades físicas (quantidade de memória necessária, tempo de execução, potência e consumo de energia) são estimadas, considerando que o código será executado em uma das configurações da plataforma FemtoJava (ITO; CARRO; JACOBI, 2001; BECK; CARRO, 2003).

A escolha pelo FemtoJava como plataforma alvo foi feita pela possibilidade de analisar e instrumentar o código gerado. No entanto, o fluxo poderia ser adaptado para execução em J2ME ou outra plataforma. Do mesmo modo, o tradutor que aplica esta abordagem poderia gerar código fonte em outras linguagens além de Java, como C/C++ ou Ada. De forma a permitir reuso de código legado, a abordagem prevê o uso de classes e métodos Java importados de um repositório em modelos Alloy, o que será possível através da integração com a Linguagem de Anotação Alloy (AAL) (KHURSHID; MARINOV; JACKSON, 2002).

Detalhando os estudos de caso, a Agenda modelada em Alloy possui apenas dois campos (nome e telefone), e permite adição, exclusão de entradas e busca por telefone. *Nome* e *Fone* são *signatures* vazias, e *Agenda* é uma *signature* que engloba relações entre os nomes e seus respectivos telefones. Para modelar informação de controle da Agenda (menu interativo), o *design pattern Menu* é utilizado, expressado pelo fato *transicaoDeEstados* no modelo. Cada estado contém sua própria instância de *Agenda* e átomos que especificam uma determinada operação a ser executada na Agenda. O fato que descreve a transição de estados faz uma chamada ao predicado da operação, de acordo com estes átomos. O estado final é alcançado quando o átomo *fim* do estado atual recebe o token *Menu*.

O Sistema de Controle do Elevador é composto por um *SolucionadorDeRequisições* (existente), que trata várias requisições e indica apenas um andar, enquanto o *ControleDeUnidade* (modelado) move o elevador para o andar indicado. As entradas e saída do sistema de controle de elevador são respectivamente andar requisitado, andar atual e direção do elevador. As entradas são expressas pelos atributos *AndarReq* e *AndarAtual* que estão relacionados com a *signature Andar* no modelo. A direção foi modelada em Alloy através da *signature Direção*, cujo domínio pode ser *parar*, *subir* ou *descer*. Para cada mudança de estado, o algoritmo do controle determina a direção considerando o andar requisitado e o andar atual. Este exemplo também utiliza o *design pattern Menu* para o controle da máquina de estados.

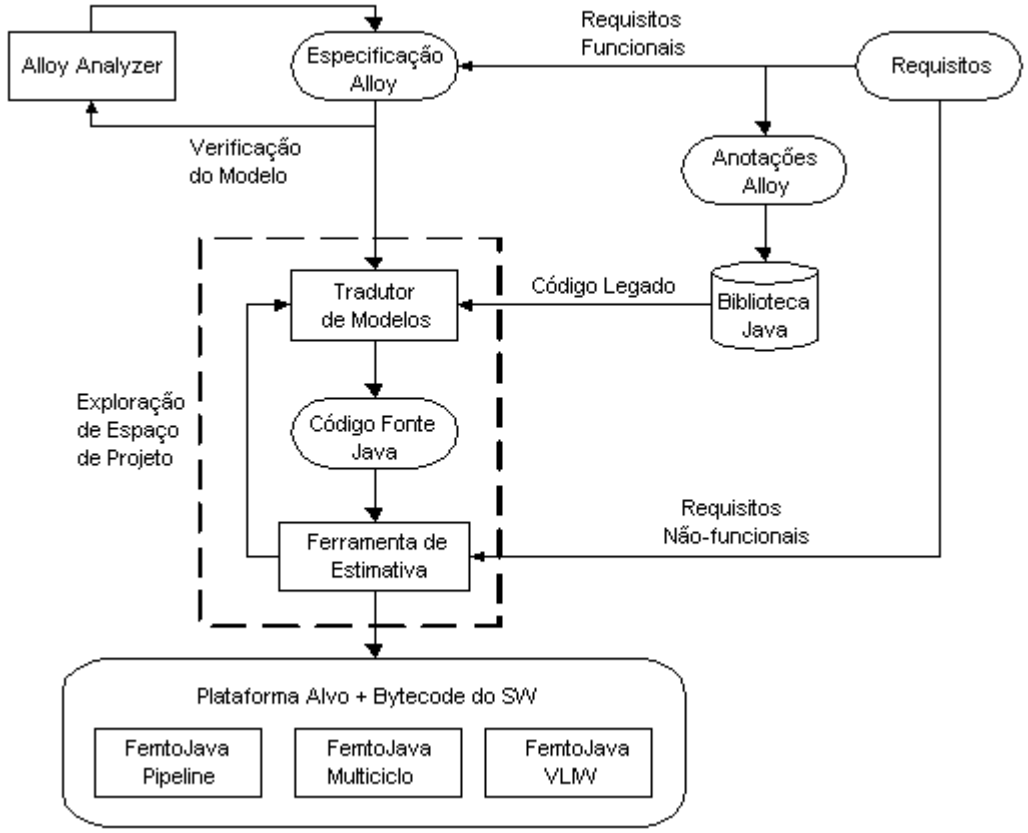

Figura A-1: Fluxo da abordagem proposta

A Máquina de Venda de Bebidas consiste em um exemplo de sistema embarcado reativo, onde o usuário indica o comportamento esperado escolhendo as opções disponíveis na máquina. A máquina apresenta três tipos de bebidas, cada qual com um preço específico, e funciona quando moedas são inseridas. Uma venda é completada apenas quando uma quantidade suficiente de moedas foi inserida, desde que estoque não esteja vazio. Ao invés de utilizar o *design pattern Menu*, no modelo da Máquina de Bebidas é assumido o modelo de computação *untimed* (JANTSCH, 2004). Assim sendo, é possível utilizar comandos *run* para informar quais os estados possíveis da máquina, como executar (via comando *run*) o predicado *AdicionarMoedas* ou *ComprarBebida*.

Geração de Código

As regras aqui apresentadas foram utilizadas no desenvolvimento de um compilador Alloy-Java no grupo de pesquisa. Há três versões para o compilador: a primeira utiliza as regras desenvolvidas pelo autor deste trabalho; a segunda agrega regras relacionadas à síntese de máquina de estados desenvolvidas no grupo, e a terceira versão substitui o suporte a estruturas padrão Java (que estendem a classe *AbstractCollection* ou a interface *Map*) por estruturas suportadas no processo de obter estimativas, como *FastList*, *FastMap*, *FastSet* e *FastTable*, disponíveis na biblioteca Javolution (DAUTELLE, 2007).

Classes e atributos são extraídos de *signatures* e suas relações no modelo, respectivamente. *Signatures* com campos são traduzidas em classes, sendo os campos

encapsulados como atributos da classe; os vazios são traduzidos em campos do tipo String. Criação de atributos simples ou compostos (coleções) é dependente da multiplicidade dos campos em uma relação (multiplicidade *one* ou *set*). As coleções não são implementadas utilizando as bibliotecas padrão Java (APIs J2SE e J2ME) devido ao fato de de parte destas APIs serem implementadas em código nativo, e a plataforma alvo FemtoJava executa apenas bytecodes Java. A biblioteca Javolution (DAUTELLE, 2007) foi escolhida justamente por ser totalmente desenvolvida em Java. Caso o modelo Alloy apresente chamadas a operadores relacionais ou de conjunto (como o operador *union*), é necessário gerar código Java que contém implementação para a operação utilizada, um para cada estrutura de dados.

Métodos são extraídos dos predicados e funções do modelo. Métodos extraídos de predicados são do tipo *void*, enquanto os extraídos de funções retornam tipo idêntico ao declarado como retorno da função no modelo Alloy. A classe a que os métodos pertencem é determinada através da lista de parâmetros formais do predicado/função. Em um *design pattern* utilizado para modelagem de máquina de estados, *estado* é uma *signature* que contém campos, os quais geralmente sofrem modificações de um ciclo para o outro. Predicados/funções proporcionam estas modificações, e estes geralmente possuem pelo menos um par de parâmetros do mesmo tipo da *signature* que modela a máquina de estados - um representa o estado atual e outro representa o próximo estado. Caso este padrão não for identificado no predicado/função, o método é colocado como estático na classe *Main*. O processo de tradução também trata o polimorfismo de métodos, gerando todas as combinações possíveis de atributos polimórficos.

O modelo faz uso de estados, a fim de permitir uma seqüência de operações e eventos ordenados na simulação automática promovida pelo Alloy Analyzer. Para permitir a simulação de um modelo, fatos devem ser declarados com restrições de estado inicial, restrições de transição de estados e o que é esperado no estado final. Desta forma, é possível expressar o modelo de computação *untimed*, classificado por (JANTSCH, 2004). Como linguagem declarativa, Alloy não foi concebido para apresentar características imperativas como laços e construções de entrada e saída. Em um primeiro momento, foi considerado utilizar-se também da saída do Alloy Analyzer para gerar uma máquina de estados. No entanto, devido à característica randômica e não-determinística da simulação do Alloy Analyzer, a decisão foi tomar como base apenas o modelo Alloy para a geração de código, utilizando o *design pattern Menu* (compilador versão 1) e o modelo de computação *untimed* (versão 2).

Resultados Experimentais

Uma análise quantitativa foi efetuada considerando os modelos da Agenda Telefônica, Sistema de Controle do Elevador e Máquina de Bebidas submetidos ao tradutor de código. Resultados mostram que aplicações de comportamentos mistos e que contém estruturas de dados apresentam maior vantagem em relação à quantidade de linhas de código geradas. Para a Máquina de Bebidas, por exemplo, 198 linhas de código Java são geradas a partir das 39 linhas do modelo Alloy. Da mesma forma, a redução do número de linhas de código geradas para o Sistema de Controle do Elevador é devido à falta de estruturas de dados, e conseqüentemente, da necessidade de gerar código para os elementos que operam sobre as estruturas.

Propriedades físicas foram estimadas para o código obtido a partir dos modelos da Agenda Telefônica e da Máquina de Bebidas. As instâncias de execução foram: 100 inserções e 100 buscas na Agenda, e inserção e consumo de 60 moedas e 60 bebidas na

Máquina de Bebidas. Resultados mostram que é possível obter diferenças significativas em soluções que apresentam apenas estruturas de dados diferentes. Na Agenda Telefônica, soluções baseadas em FastSet e FastMap foram melhores considerando desempenho em ciclos, devido à implementação baseada em tabelas hash. Na Máquina de Bebidas, certas soluções também são vantajosas por não utilizarem listas encadeadas, reduzindo a complexidade de O(n) a O(1) no algoritmo de acesso a dados.

As arquiteturas de pipeline e multiciclo por si só apresentam diferenças significativas no que se refere a desempenho, potência e energia. As versões pipeline consomem menos energia e apresentam melhorias significativas em desempenho, logo, são recomendáveis se for aceitável um consumo extra de energia. Mesmo conhecendo estas diferenças, a exploração do espaço de projeto ainda é importante no sentido de quantificar as propriedades físicas. Como o conjunto de instruções é o mesmo, a área da memória de programa das soluções pipeline e multiciclo também é equivalente, embora os algoritmos tenham comportamentos muito diferentes durante a execução. Na Máquina de Bebidas é possível observar o acréscimo em memória de programa com o uso de estruturas hash, devido à informação necessária para manter tais funções. Este comportamento já não pode ser observado na Agenda, uma vez que experimentos de memória são geralmente mais significativos para aplicações que manipulem estruturas de dados maiores.

Análise da Abordagem Proposta

A principal vantagem desta proposta, que visa automatizar o desenvolvimento de software embarcado, é a abstração das estruturas de dados promovida pela linguagem de entrada. Outras abordagens para síntese de software embarcado baseadas em máquina de estados finitos, como Esterel, não seguem a crescente demanda dos sistemas embarcados, que atualmente exigem complexas estruturas de dados para representar suas aplicações. Outra vantagem é a verificação formal e simulação de modelos promovida automaticamente pelo Alloy Analyzer.

Alloy é uma linguagem atrativa para modelar problemas complexos, mas sua simplicidade pode torná-la incapaz de modelar problemas que envolvam lógicas de mais alta ordem ou temporal. Autores como (HUDAK, 1989) alegam que é possível expressar concorrência de uma forma mais natural utilizando linguagens declarativas, mas a abordagem não suporta aplicações multitarefa, nem mesmo promove meios para extração de paralelismo. Além disto, a proposta inicial da pesquisa era usar uma linguagem declarativa para modelar múltiplos modelos de computação. Como Alloy não proporciona boas maneiras de modelar aplicações com cálculos intensivos, é necessário um esforço maior para que a abordagem suporte modelos orientados a dados e ainda promova a abstração desejada.

A abordagem ainda não suporta algumas construções Alloy, como fatos (exceto os relacionados à modelagem da máquina de estados). Como os fatos estão intrínsecos no código, são mais difíceis de serem traduzidos do que outras construções, apesar desta tarefa já ter sido iniciada no grupo. A abordagem também apresenta limitações no mapeamento de entrada e saída: atualmente, entradas e saídas são manipuladas automaticamente no código gerado, e apenas entrada e saída padrão são suportadas. Além disto, interfaces para suportar outros mecanismos de entrada e saída, como acesso a portas do FemtoJava, estarão disponíveis em breve.

Desenvolvedores de software embarcado podem beneficiar-se da abordagem, uma vez que esta promove exploração do espaço de projeto além da alta capacidade de abstração no desenvolvimento. No entanto, sempre há um overhead introduzido pelo processo de automação. Um dos próximos passos da pesquisa é instrumentar códigos manuais e comparar seus atributos físicos com os obtidos por instrumentação de códigos gerados automaticamente, a fim de avaliar a qualidade do processo de automação e mensurar seu impacto nas propriedades físicas do sistema embarcado.

Conclusão

Este trabalho propôs uma nova abordagem para o desenvolvimento de software embarcado, onde uma especificação em alto nível é descrita utilizando a linguagem declarativa Alloy, com o objetivo de proporcionar ao desenvolvedor abstração e verificação formal ao mesmo tempo. Regras que mapeiam modelos Alloy em código Java foram apresentadas, sendo que a ferramenta tradutora desenvolvida no grupo produz diferentes códigos fonte, variando suas estrutura de dados. Graças à integração entre a abordagem proposta e a ferramenta de estimativa DESEJOS, o desenvolvedor pode avaliar diferentes soluções observando tempo de execução e quantidade de memória necessária, e escolher a mais adequada aos recursos disponíveis na plataforma-alvo.

Problemas que persistem com o uso das ferramentas de alto nível existentes para desenvolvimento de sistemas embarcados também foram apresentados. Entre os problemas apontados está o restrito suporte a aplicações heterogêneas, ou mesmo abstração limitada na descrição de apenas um modelo de computação. Por outro lado, ferramentas que promovem boa abstração como as baseadas em UML geralmente produzem código não-otimizado ou com confiabilidade reduzida, o que dificulta sua utilização para certos tipos de sistemas embarcados. A abordagem baseada em Alloy comprovou ser altamente abstrata para pelo menos cinco aplicações, resultados melhores do que os obtidos utilizando outras linguagens declarativas, além de beneficiar o processo de desenvolvimento do software embarcado como um todo pela disponibilidade de ferramentas que promovem verificação formal leve, como o Alloy Analyzer.

Entre os trabalhos futuros está o desenvolvimento de interfaces para lidar com ações de entrada e saída, suporte a aplicações totalmente orientadas a dados, bem como a tradução de construções "fato" da linguagem Alloy. Um esquema de tradução de anotações AAL em modelo Alloy está sendo codificado no grupo, de forma a permitir o uso e verificação de bibliotecas de código legado e assim maximizar o reuso. Por fim, é necessário provar a eficácia da abordagem com exemplos mais complexos, que envolvam mais modelos de computação e concorrência, e detalhar os ganhos de em abstração que podem ser obtidos com a metodologia proposta.

ATTACHMENT A PAPER SUBMITTED TO SBCCI 2007

The paper "**Analysis of the Use of Declarative Languages for Enhanced Embedded System Software Development**", developed by the authors Emilena Specht, Ricardo Redin, Luigi Carro, Luís Lamb, Erika Cotta, Flávio Wagner, all from the Informatics Institute, Federal University of Rio Grande do Sul, Brazil, was presented in the SBCCI 2007, that took place in Rio de Janeiro, Brazil. It was published in the Proceedings of the 20th annual conference on Integrated circuits and systems design (SBCCI).

In this paper, comparative results of the use of declarative languages to describe embedded applications are presented. The main objective of the study was the analysis of the abstraction level achieved with the shift from the imperative programming paradigm to the declarative paradigm, considering its impact in terms of performance and memory in the embedded systems domain.

Analysis of the Use of Declarative Languages for Enhanced Embedded System Software Development

Emilena Specht, Ricardo Redin, Luigi Carro, Luís Lamb, Erika Cota, Flávio Wagner
Informatics Institute
Federal University of Rio Grande do Sul
Porto Alegre – RS – Brazil
{emilenas, rmredin, carro, lamb, erika, flavio}@inf.ufrgs.br

ABSTRACT

With the increased demand for programmability in embedded applications, the pressure for producing high performance software in a timely fashion has grown over the years. For this reason, declarative languages, as they provide more abstraction than traditional languages like Java, could be used to code embedded system applications. In this paper we present comparative results of the use of declarative languages to describe embedded applications. We designed the MP3*, an embedded application containing the IMDCT algorithm (an essential part of an MP3 player) together with an Address Book and Sokoban and Tic-tac-toe games. We coded all applications in Ocaml and Prolog to analyze the resulting abstraction and performance, and then compared them to the Java equivalent codes. For some applications, a comparison with a language that is especially oriented for embedded systems was also provided. The main objective of this study is the analysis of the abstraction level achieved with the shift from the imperative programming paradigm to the declarative paradigm, considering its impact in terms of performance and memory in the embedded systems domain.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and Embedded Systems. D.1 [**Programming Techniques**]: Applicative (Functional) Programming, Logic Programming D.2.11 [**Software Architectures**]: Data Abstraction; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages, Measurement, Performance.

Keywords

System Level Modeling and Synthesis, Modeling Languages, Embedded Software.

SBCCI'07, September 3–6, 2007, Rio de Janeiro, Brazil.

1. INTRODUCTION

Embedded systems are everywhere and usually offer several functionalities. Cell phones are examples of embedded systems. It is predicted by [1] that by 2009 there will be over 3 billion mobile phone users worldwide, which means that embedded systems will be even more dominant in the market.

Nowadays, cell phones are much more than communicating devices: they are also MP3 players, digital cameras, address books, games, and so on. For this reason, the embedded software market continues to enjoy steady growth. The estimated Average Annual Growth Rate (AAGR) between 2004 and 2009 is 16% for embedded software [2].

Besides the variety of applications present in embedded equipments, embedded software is becoming more complex. Industry statistics [3] reveal that lines of embedded software code, which can be considered an indicative of software complexity [4], are growing at about 26 percent annually. Due to this growing complexity and short time to market, more than one half of current embedded design projects are running behind schedule [5].

One can also observe the shift from hardware components development to software components development and the growth of methodologies that favor reuse at several abstraction levels. As stated in [6], from 60% to 90% of a system is very similar to a previously developed system, and this portion may be reused. The same fact was observed by [7], where 95% of the embedded systems components are reused, and 90% of these systems are composed by software.

The only hope for the designer to cope with the ever growing complexity of embedded designs and their tight time-to-market constraints is to increase the abstraction level. Hence, a developer should use languages that provide abstraction in order to accelerate embedded software implementation. There are many languages that have been used to implement certain types of embedded software, such as Esterel [8] and SDL [9].

The SDL language is suited for expressing algorithms that perform uniform arithmetic operations on a continuous stream of data, while Esterel provides high-level control constructs for reacting to many different simultaneous inputs [10]. There is no specific embedded system language that provides abstraction for every model of computation, a feature that is required for most of the current embedded software, where a single product like a cell-phone exhibits several different functions.

On the other side of the software design spectrum, declarative general-purpose languages are known for permitting descriptions

at a very high level of abstraction and providing program correctness [11][12]. Declarative languages can also explore concurrency without requiring programmer's knowledge [13]. They are based on sound mathematical foundations and have well-defined semantics. Logic and functional languages, such as Prolog and Ocaml, are examples of declarative languages. In this work, we analyze the declarative general-purpose languages as an alternative to achieve abstraction in the embedded software development.

We designed the MP3*, an embedded application containing an IMDCT algorithm (an essential part of an MP3 player) together with an Address Book and Sokoban and Tic-tac-toe games, in Ocaml, Java (J2SE), and Prolog. The main objective of this study is the analysis of the abstraction level gain (measured in lines of code) achieved with the shift from the imperative programming paradigm to the declarative paradigm. We also coded a part of the MP3* in Esterel, to compare the abstraction achieved by using a language that is specific for embedded control systems.

The remaining of the paper is organized as follows. Section 2 discusses related work, with examples of the use of declarative languages in embedded software design. Section 3 describes the comparison methodology and the implementation of the MP3* application. Section 4 shows the main experimental results and a discussion on these results. Section 5 draws main conclusions and discusses future work.

1. RELATED WORK

There are various reports on applications for embedded systems described in functional languages, such as [14]. These works present case studies, extensions of known functional languages, or even propose new languages. However, these proposals are suited for only one model of computation in the embedded systems domain: [14] proposes extensions to handle reactive systems, the Lustre language [15] is suitable for synchronous-dataflow applications, and [16] targets at specific applications.

The Ptolemy II framework [17] provides simulation and prototyping for heterogeneous systems. It does not assume a particular model of computation, such as dataflow, functional, finite-state machines, statecharts, communicating sequential processes, or Petri nets. Rather, it accommodates all of these models. Although Ptolemy provides abstraction for many domains, it requires knowledge and classification of several different models of computation, and this may increase development time.

Other comparisons between declarative and imperative paradigms do exist, but they are restricted to certain algorithms or models of computation. [18] and [19] compare functional and imperative implementations through red-black trees and suffix tree constructions, respectively. [20] uses some benchmarks (sorting and puzzles solutions) to compare functional and imperative paradigms, but the comparison considers only performance. [21] gives some hints comparing both paradigms, but gives no numbers.

In [22] the authors implemented a concrete case study to demonstrate the feasibility of using a high-level, general-purpose logic language in the design and implementation of an application targeting wearable computers. The case study has tight embedded system constraints and was implemented in a high-level language without efficiency or architectural concerns. The compile-time optimizations and native code transformations made it possible to execute the code in the embedded device without high-level code modifications.

The authors of [22] were concerned with the feasibility of using logic languages (a branch of declarative languages) in embedded systems. In this paper we are concerned with the actual abstraction reached by the use of these languages in terms of code-time savings and with the actual trade-offs implied by the use of declarative languages in terms of performance and memory usage for embedded systems.

2. COMPARISON METHODOLOGY

2.1 Embedded Constraints

Embedded systems are often used in life-critical situations, where reliability and safety are more important criteria than performance [23]. However, embedded systems are usually real-time, where the time at which a computation takes place can be more important than the computation itself, which means that timing and predictability must be taken into account.

Memory size is important for embedded systems design. Memory is expensive, huge, and power hungry, if compared with the rest of the system. Embedded systems usually rely on batteries, and the more memory to access, the more energy is consumed. A simple controller that uses the Intel 8051 processor has at most 128 Kbytes of external memory available, half being for the executable code. On the other side, a Texas OMAP-DM270, a platform suited for cell phones, offers this number as internal memory and allows 128 MB of external SDRAM memory and 16 MB of external flash memory. Depending on the application, a considerable amount of memory must be available.

One could think that, if there is a lot of memory available, one would not have to save memory when developing an application. Embedded systems development has not reached this level of memory independence, because saving memory is worthwhile, as it frees memory to other applications and reduces the overall power dissipation.

2.2 The MP3* application

The methodology of this study includes the implementation of the MP3* application, containing an IMDCT algorithm (part of an MP3 player) together with an Address Book and Sokoban and Tic-tac-toe games. The MP3* was coded in Java, Ocaml, and Prolog. We also coded a part of the MP3* (the Address Book and Tic-tac-toe) in Esterel, to compare the abstraction achieved in general-purpose languages with the abstraction achieved by using an embedded system language.

We used Java because it is an imperative language that provides abstraction through its APIs and object-oriented style and its use is spread in the embedded systems community [24]. The concepts here presented could also be applied to other languages like C. However, the comparison with declarative languages would not be fair in terms of abstraction. Ocaml, the main implementation of the Caml language, was the functional language chosen for its powerful module system and widespread use. Prolog is the main example of logic languages.

Some components that are part of the MP3* (or similar components) can be often found in embedded devices, since they involve different behaviors and, consequently, different models of computation. The Tic-tac-toe, with the system playing with the user, and Sokoban games include scalar data processing, where the system must compute the user inputs and choose the best move. Also, the control behavior is present in the sequencing of plays and user inputs. The Address Book behavior is mostly control-flow with some dataflow, since the manipulation of the data structure after some user input predominates. Besides, the user options suggest a reactive control behavior. The IMDCT algorithm, a typical stream data processing, can be considered mostly dataflow.

The implementations in Java, Ocaml, and Prolog maintain the same basic algorithms, but use different control and data structures. We have done this to take advantage of the structures provided by the paradigms that each language supports. Lists, for example, are the basic data structures of functional languages and have optimized access functions. On the other hand, clauses are the basis of logic languages, while sequential structures like arrays predominate in imperative languages.

The Tic-tac-toe and Sokoban games were implemented using lists as data structures in Ocaml. In Java, matrixes were used as maps in Sokoban and to save the plays in the Tic-tac-toe game. The Sokoban Java code was adapted from a version available in the web [25], and its graphical interface was maintained. In Prolog, the Tic-tac-toe game uses the predicate construct, while lists were used in Sokoban. In Esterel, the Tic-tac-toe game was implemented using state machines and single Strings as the map positions.

The Java Address Book uses the *Vector* class and a class *Person,* with *name* and *phone* as attributes. The Ocaml Address Book uses lists and tuples to store the contacts, and the Prolog version again uses dynamic predicates. Although Ocaml is a language with imperative features, in the MP3* application we tried to use only its functional constructions, since we wanted to observe the amount of possible abstraction achieved. In Esterel, the Address Book uses a Java *Vector* data structure, and its manipulation methods are implemented in Java, in a separated file. These methods are later invoked by the Esterel Address Book source code. The control state machine in Esterel implements the user menu and method calls.

The cosine table of IMDCT in Java is a final array, in Prolog it reproduces a table composed by predicates, and in Ocaml it is a list. The output IMDCT array is an array in Java and a list in Ocaml and Prolog.

2.3 Expected abstractions

Imperative programming languages are directly based on the von Neumann architecture, were programmers have to handle variable management and value assignment. In declarative languages, in turn, the instructions are declarations, not assignments or control-flow instructions. The evaluation order of mathematical expressions is controlled by recursion and conditional expressions.

Another abstraction expected by the use of declarative language is the expression of concurrency. The graph representation of programs in functional languages exposes many opportunities to concurrent execution, without the programmer concern. Programs in logic languages are also naturally parallel [13]. The concurrency aspect is not covered by this work and will be explored as a future work.

In terms of data structures, the use of lists as the main data structure is a characteristic of declarative languages. Indeed, lists are almost the unique data structure used in the declarative paradigm. When the programmer has to work with lists, he/she does not need to specify how to build a new list, he/she has only to declare how the new list looks like. Trivial operations in lists are implemented by the interpreter system. Matching elements of two lists is a common operation that is performed automatically by the underlying interpreter. On the other hand, the Esterel language presents specific features to abstract control structures, and data structures have to be defined outside the Esterel code, somehow increasing the same complexity one is trying to reduce.

3. RESULTS

Our results were supported by actual tests executed under the same conditions for all languages and datasets. Tests and experiments were done on an AMD Sempron 2500+ running at 1750 MHz with 512MB of DDR3200 RAM memory. The operating system used was a Debian GNU/Linux 3.1 with GNU/Prolog, Ocaml, GCC/GCJ 4.0, and SUN JDK1.5.

We compared the abstraction achieved in number of lines of code and the impact in terms of performance and memory usage of each implementation. In many cases, the number of lines of code provides a good measure of abstraction. A 32-bit multiplier with 6689 gates may be described using only 31 lines of code in VHDL, for example. Although we are aware that the complexity of different commands in different languages is not always reflected by the number of lines of code, a measure of this complexity is not easy to obtain. We avoided using other related metrics like learning time and development time, because they are strongly dependent on the programmers.

3.1 Achieved abstraction

Figure 1 shows the percentage of Lines of Code (LoC) of two Ocaml and Prolog applications, compared with the Java ones. A comparison with memory numbers is shown in Table 1. As said before, Sokoban needs a Graphical User Interface. The LoC numbers in Sokoban means Total LoC minus GUI LoC.

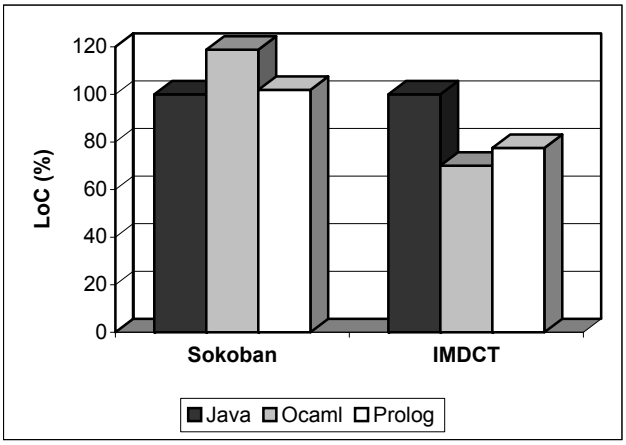

Figure 1. LoC per application

Figure 2 also shows the percentage of LoC of Ocaml and Prolog applications, but now compared with Java and Esterel ones. In the Address Book, the Esterel line count means the lines of code in Esterel plus the lines of code that were implemented in Java.

The charts in Figure 1 and Figure 2 show that, in terms of Lines of Code, declarative languages can achieve a lower line count than Java for some applications. Using Prolog instead of Java or Esterel, one can reduce near half of the effort in lines to code an Address Book. However, as the Sokoban result shows, this is not always the case, since the lack of adequate data structures available for certain applications increases the number of Lines of Code. In this case, the Sokoban code in Ocaml is about 20% larger than its correspondent in Java.

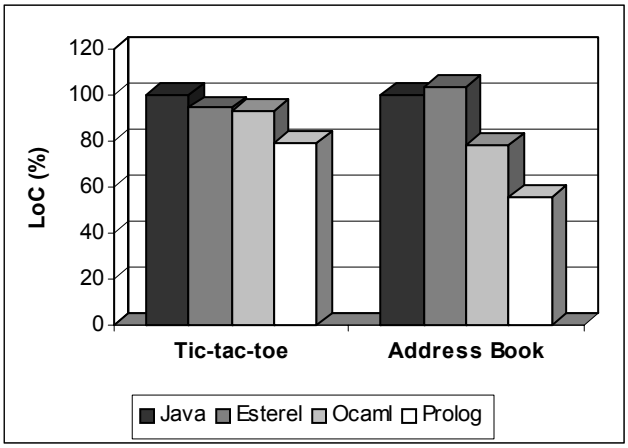

Figure 2. LoC per application (including Esterel)

The Tic-tac-toe Ocaml and Prolog implementations, although with about 10% and 20% less Lines of Code than the Java one, provided somewhat less abstraction considering data structures. A matrix to represent the game board is more intuitive and easier to deal with. The smaller number of Lines of Code was achieved in Ocaml and Prolog applications because of the cleaner control structures. Despite the fact that Java provides more abstraction considering data structures, the Esterel Tic-tac-toe achieves less LoC, also because of the abstract control structures and succinct commands and syntax of Esterel.

The use of lists is not adequate when there is a need for a data structure that changes elements besides adding elements and querying. In order to maintain the referential transparency, a list cannot be modified, that is, a new modified list is returned each play. Despite its safety, there is an unnecessary memory overhead, which is not adequate for embedded systems.

The case of Sokoban is similar to the Tic-tac-toe one, considering data structures in Ocaml. A board with more positions makes the situation even worse for the Ocaml Sokoban implementation, where dealing with strings is necessary to avoid the extra overhead, when considering each position as an element of a list. Besides, the Ocaml String Library does not offer functions like the *java.String.split(String expression)*. The Java AWT classes also help on manipulating the Sokoban board map, which makes the implementation in Java even easier. The Prolog implementation of Sokoban stores positions of the objects in the game as dynamical facts loaded from a file specified in the initialization. Prolog, as Ocaml and most declarative languages, has difficulties to handle input and output (very limited in these languages).

The Address Book implementation is interesting in Ocaml, due to the simplicity of dealing with lists and tuples. The Prolog implementation of Address Book uses dynamically loaded facts to store contact information. One can store the contact archive as a set of facts and load them when needed. Since the Address Book application is limited to loads, saves, and queries about these facts, this is the application where Prolog has the best results in terms of abstraction and performance. The Esterel language again seems not to be a good choice for implementing an Address Book application. As data structures must be still implemented in Java, an extra time is necessary to switch between languages and maintain the coherency among files.

A comparison in terms of control structures, data structures manipulation, and code conciseness among Prolog, Ocaml, and Java may be observed in the search operation of an Address Book written in Figures 3a to 3c.

```
search(X,Name) :-
       contact(Name,Phone),
       write(Name),
       write(' has the number: '),
       write(Phone);
       write('\nContact not found
              in database!\n').
```

Figure 3a. Search operation in Prolog

The search clause in Prolog has an instance of the Address Book (database) and a variable holding the name of the contact to be found as parameters. If a name that belongs to a contact in the database matches the name obtained from parameters, a value for the *Phone* variable will be returned. Otherwise, the commands after the semicolon will be executed. In this case, a "not found" message will be delivered to the standard output (screen).

```
let rec search name list =
  match list with
       [ ] ->  Printf.printf "\nContact not
               found in database!\n";
     | head::tail -> let (n,p) = head in
             match n = name with
                true -> Printf.printf
                     "\n%s has the number:
                      %s\n" n p;
              | false -> search name tail;;
```

Figure 3b. Search operation in Ocaml

In Ocaml, the *pattern matching* construct allows simple access to the components of complex data structures. *Pattern matching* is used to recognize the form of a value and lets the computation be guided accordingly, associating each pattern to an expression to compute. In the search function in Figure 3b, *pattern matching* is applied to a list. A list can be either empty (the list is of form [ ]) or composed of a first element (its head) and a sublist (its tail). The list is then of form head::tail.

In the Address book in Ocaml, each element of the list is a tuple representing a contact, of form (name,phone). The *pattern matching* on the list tries to match the name obtained from parameters with the first component of the tuple in the current head of the list. If the matching does not happen, the interpreter will keep iterating recursively, considering the rest of the list

(tail). As soon as a matching occurs, the function automatically finishes its execution. If there are no matches for the name and the considered list becomes null, a "not found" message will appear on the screen.

The Address Book in Java has its contacts stored in a *Vector* data structure. A search on a *Vector* must be controlled by an auxiliary variable (*i*), to ensure that the search is not going to exceed the boundaries of the data structure and throw an exception. Also, a flag must be used to advise that the loop must be interrupted, in case an object *Person* with the same name as that searched was found.

```
public void search(String name){
  int i=0;
  boolean found=false;
  while((i<list.size())&&(found==false)){
     if(((Person)list.get(i)).
                  getName().equals(name)){
        String phone=((Person)list.get(i)).
                  getPhone();
        System.out.println(name+"has
                  the number: "+ phone);
        found=true;
      }
      i = i+1;
   }
   if (found==false)
   System.out.println("Contact not found in
                   database!");
}
```

Figure 3c. Search operation in Java

In Prolog, the search engine is totally transparent to the programmer, providing abstraction by concentrating on what has to be done, and not how to accomplish this task. In Ocaml, the *pattern matching* and recursion mechanisms simplify what in Java is done using flags and for-loop commands. A search method in Java also depends strongly on which data structure is used. In Esterel, the same Java search method stated above is invoked by the Esterel Address Book source code.

The IMDCT algorithm in Ocaml and Prolog, considering data structures, presented the same abstraction as the Java version. In this case there is almost no difference between using lists, predicates, or an array. Control is usually more abstract in Ocaml and Prolog due to the use of recursion instead of loops and *pattern matching* instead of for-loop and if-then-else commands. This feature also makes the code cleaner. This is the reason for the short code of IMDCT in Ocaml and Prolog, when compared to the Java IMDCT.

3.2 Performance and memory usage

Table 1 shows a comparison in terms of static memory consumption by interpreted code, compiled code, and virtual machine size. Java interpreted code uses less static memory for all applications; however, there is a penalty in the JVM memory usage. The amount of memory used by the Prolog virtual machine is variable, depending on the predicates required by the application. The Address Book in Prolog required the use of dynamic predicates to add, edit and delete contacts from the database. For this reason, the virtual machine size for the Address Book in Prolog is 15% higher than the JVM size.

The amount of static memory used by the applications when they execute natively varies according to their type and size. For three applications it is smaller in Prolog implementations; the Address Book in Prolog requires only 3 KB of memory for the executable file, more than thirty times less than the amount required by Java and Ocaml ones.

Table 1. Static memory usage

Application	**Tic-tac-toe**			**Sokoban**		
Language	Java	Ocaml	Prolog	Java	Ocaml	Prolog
Interp. (KB)	3.5	47	16	22	49	63
Native (KB)	26	61	125	102	79	40
VM (KB)	262	116	162	262	116	279
Application	**IMDCT**			**Address Book**		
Language	Java	Ocaml	Prolog	Java	Ocaml	Prolog
Interp. (KB)	1.8	22	15	6.4	49	14
Native (KB)	16	18	4	35	63	3
VM (KB)	262	116	120	262	116	299

Figure 4 shows a performance evaluation of the IMDCT algorithm. The plot shows the time needed to process the amount of data varying from 10 to 1,000,000 samples. As one can see, Ocaml and Prolog have a performance degradation two orders of magnitude larger than Java. This result can be explained by the use of recursion to implement iteration, needed in the algorithm. When interpreters use recursion, it is necessary to allocate another frame to store data of that pass. The allocation process incurs in time and memory penalties. The same result pattern was observed in the interpreted code and in the native compiled code. This means that, albeit optimizations used in compiler of Prolog and Ocaml, native code follows the same idea of interpreted code to obtain the answer. Another implementation of the IMDCT algorithm that benefits from recursion can possibly match up Java performance.

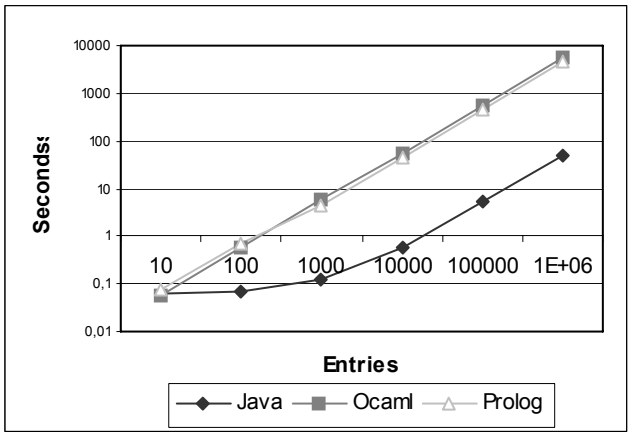

Figure 4. Exec. time comparison in Java, Ocaml, and Prolog

Backtracking and instantiation processes in Prolog and recursion in Ocaml lead to a high memory allocation and usage. For this reason, declarative languages use a lot of memory bandwidth in execution time and each memory access slows down the program execution. Mechanisms such as lazy evaluation in Ocaml, that improve the performance of functional languages avoiding unnecessary computations, also increase memory usage.

Prolog and Ocaml execution environments are based on a stack machine, which leads to the same memory problem described

above, but can also draw some advantages as a good garbage collection scheme, and in some cases a predictable runtime memory usage.

4. CONCLUSIONS AND FUTURE WORK

Results showed that we can describe different computation models in Prolog and Ocaml but, unfortunately, these results also showed that the performance and memory penalties cannot be neglected. The abstraction level reached with the use of Prolog (measured in Lines of Code) is near 50% in some cases, although in the worst case it was a little higher than Java. For applications like Sokoban, the Java code is more abstract and still uses less memory. Nevertheless, for applications that mix models of computation like the Address Book and the Tic-tac-toe game, declarative languages presented more abstraction mechanisms than Esterel, a language that is specific for reactive embedded systems.

The abstraction results for declarative languages were not satisfactory for all kinds of embedded applications studied. This means that by using the single metric of LOC, even by assuming a perfect interpreter for a declarative language, one would not achieve orders of magnitude gains in the code abstraction. Nevertheless, some features of declarative languages are still highly desirable, like those concerning the error ratio in programs. Prolog and the other logic programming languages are based on logic, and so its programs can be logically organized and written, which leads to less errors and maintenance costs. Functional languages, with the use of recursion and *pattern matching* mechanism, also allow a cleaner code.

Yet, declarative languages can explore concurrency without the programmer's knowledge, a feature highly desirable with today's multicore use expanding to every device. This could be a motivation for the exploration of declarative languages in future SOCs and should be further studied. The analysis of other declarative languages, such as the Alloy modeling language, is also among the future works.

REFERENCES

[1] Nokia. The Road to Three Billion Subscribers: Nokia outlines strategy to reduce total cost of mobile phone ownership for consumers in new growth markets. Press Releases. June 02, 2005.

[2] Eurostat. GDP and main components – Current Prices. Retrieved August 19, 2005.

[3] VDC – Venture Development Corporation. The 2005 Embedded Software Strategic Market Intelligence Program. VIII: Embedded Systems market Statistics. Natick, MA, USA, 2006.

[4] Kan, S. H. Metrics and Models in Software Quality Engineering. Addison Wesley, 2002.

[5] CMP Media. 2006 State of Embedded Market Survey. April 2006.

[6] Shandle, J.; Martin, G. Making Embedded Software reusable for SoCs. EEDesing, [S.l.], March 2002. Available at http://www.eedesign.com.

[7] Douglass, B. Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Boston. Addison Wesley, 2003.

[8] Berry, G.; Gonthier, G. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Science of computer programming, vol. 19, n. 2, 1992. Pp. 87-152.

[9] Rockstrom, A.; Saracco, R. SDL - CCITT Specification and Description Language. IEEE Transactions on Communications, vol. 30, n. 6, June 1982.

[10] Edwards, S. A. Languages for Digital Embedded Systems. Kluver Academic Publishers, 2000.

[11] Hudak, P. Conception, evolution, and application of functional programming languages. ACM Computing Surveys, vol. 21, n.3, September 1989. Pp. 359-411.

[12] Fuchs, N. E. Specifications Are (Preferably) Executable. Software Engineering Journal, September 1992.

[13] Sebesta, R. W. Concepts of Programming Languages. Addison Wesley, Boston, 2005.

[14] Wallace, M.; Runciman, C. Extending a Functional Programming System for Embedded Applications. Software-Practice and Experience, vol. 25, n. 1, January 1995. Pp. 73–96.

[15] Caspi, P. et al. LUSTRE: A Declarative Language for Programming Synchronous Systems. Symposium on Principles of Programming Languages, 1987. Pp. 178-188.

[16] Armstrong, J. et al. Concurrent programming in Erlang. Prentice-Hall, 1993.

[17] Brooks, C. et al. Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II). EECS Department, University of California, Berkeley, January 11, 2007.

[18] Okasaki, C. Functional Perl: Red-black trees in a functional setting. J. Functional Programming, vol. 9, n. 4, July 1999. Pp. 471-477.

[19] Giegerich, R, Kurtz, S. A comparison of imperative and purely functional suffix tree constructions. Science of Computer Programming, vol. 25, 1995. Pp. 187-218.

[20] Stansifer, R. Imperative versus Functional. SIGPLAN Notices, vol. 25, n. 4, November 1989. Pp. 69-72.

[21] Wadler, P. Why no one uses functional languages. ACM Sigplan, 1999.

[22] Carro M. et al. High-level languages for small devices: a case study. Proceedings of the CASES 2006, Seoul, Korea, 2006. Pp. 271-281.

[23] Edwards, S. et al. Design of Embedded Systems: Formal Models, Validation, and Synthesis. Proceedings of the IEEE, vol. 85, n. 3, March 1997. Pp. 366-390.

[24] Strom, O. et al. On the Utilization of Java Technology in Embedded Systems, Design Automation for Embedded Systems, vol. 8, n. 1, March 2003. Pp. 87-106.

[25] Nugroho, R. P. Java Sokoban. Jakarta, Indonesia, 1999. Available at http://javaboutique.internet.com/Sokoban(2006)