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

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

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 systems, 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 remainder 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, matrices 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](image-url)
Esterel
Ocaml
Prolog

initialization. Prolog, as Ocaml and most declarative languages, game as dynamical facts loaded from a file specified in the implementation of Sokoban stores positions of the objects in the implementation in Java even easier. The Prolog also help on manipulating the Sokoban board map, which makes the overhead, when considering each position as an element of a list. where dealing with strings is necessary to avoid the extra situation even worse for the Ocaml Sokoban implementation, data structures in Ocaml. A board with more positions makes the The case of Sokoban is similar to the Tic-tac-toe one, considering the overhead, which is not adequate for embedded systems.

Despite its safety, there is an unnecessary memory consumption, 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.

A comparison in terms of control structures, data structures manipulation, and code conciseness among Prolog, Ocaml, and Java 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.

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,
(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.

```java
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;
        } else {
            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.

![Figure 4. Exec. time comparison in Java, Ocaml, and Prolog](image)

Table 1. Static memory usage

<table>
<thead>
<tr>
<th>Application</th>
<th>Tic-tac-toe</th>
<th>Sokoban</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Java</td>
<td>Ocaml</td>
</tr>
<tr>
<td>Interp. (KB)</td>
<td>3.5</td>
<td>47</td>
</tr>
<tr>
<td>Native (KB)</td>
<td>26</td>
<td>61</td>
</tr>
<tr>
<td>VM (KB)</td>
<td>262</td>
<td>116</td>
</tr>
</tbody>
</table>

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.
also among the future works.

which leads to less errors and maintenance costs. Functional
logic, and so its programs can be logically organized and written,
Prolog and the other logic programming languages are based on
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
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


Embedded Software Strategic Market Intelligence Program.
VIII: Embedded Systems market Statistics. Natick, MA,
USA, 2006.


2006.

for SoCs. EEDesing, [S.I.], March 2002. Available at

[7] Douglass, B. Real-Time Design Patterns: Robust Scalable

[8] Berry, G.; Gonthier, G. The Esterel Synchronous
Programming Language: Design, Semantics,
Implementation. Science of computer programming, vol. 19,

[9] Rockstrom, A.; Saracco, R. SDL - CCITT Specification and
Description Language. IEEE Transactions on


functional programming languages. ACM Computing


Addison Wesley, Boston, 2005.

Programming System for Embedded Applications. Software-
96.

Programming Synchronous Systems. Symposium on


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

[18] Okasaki, C. Functional Perl: Red-black trees in a functional

[19] Gigerich, R. Kurtz, S. A comparison of imperative and
purely functional suffix tree constructions. Science of

[20] Stansifer, R. Imperative versus Functional. SIGPLAN

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

case study. Proceedings of the CASES 2006, Seoul, Korea,

Models, Validation, and Synthesis. Proceedings of the IEEE,

[24] Strom, O. et al. On the Utilization of Java Technology in
Embedded Systems, Design Automation for Embedded