Predictable Programs in Barcodes

Alwyn Goodloe, Michael McDougall, Carl A. Gunter, and Rajeev Alur
Department of Computer and Information Science
University of Pennsylvania
200 South 33rd Street
Philadelphia, PA 19104-6389
agoodloe@saul.cis.upenn.edu, mmcdouga@saul.cis.upenn.edu,
gunter@cis.upenn.edu, alur@cis.upenn.edu

ABSTRACT

We explore the challenges for making the programming interfaces for embedded devices open and safe, and present a prototype architecture for delivering verified programs using barcodes. In particular, we consider programs for microwave ovens, which provide a basic open API for controlling cooking times. In our architecture, recipes are written in Java, and their safety properties are formally verified using the model checker Spin. We use off-the-shelf utilities for compressing the byte code, and use two-dimensional barcodes for program delivery. We report on experiments that demonstrate the feasibility of the proposed architecture for predictability and delivery.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*real-time and embedded systems*; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods; model checking*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.11 [**Software Engineering**]: Software Architectures—*domain-specific architectures; languages (e.g., description, interconnection, definition)*

General Terms

Languages, Reliability, Verification

Keywords

Programmability of embedded devices, Code delivery, Active barcodes, Formal verification

1. INTRODUCTION

The aim of this paper is to look at some issues related to the programming of embedded systems through *open Application Programming Interface (API) platforms.* Our specific case study is programming microwave ovens using two-dimensional (2D) barcodes, but the general topic is how embedded systems can be made to have open programming interfaces that enable users and third party vendors to customize their functionality. The open platform idea is really a spectrum that ranges from open source code to modest customization hooks, but it is an important driver in smart devices. For example, Personal Digital Assistants (PDAs) are generally based on open platforms whereas other devices, like cell phones, are sometimes open (the Java phone) but usually not. Other devices, like embedded systems in automobiles, are mainly programmable only by their manufacturer. Microwave ovens occupy an interesting place in this spectrum since they are safety-critical devices that must provide at least a rudimentary open API. Consumers probably would not buy a microwave from Samsung if it could cook food from Samsung, but not from Stouffer's or Pillsbury. The microwave hardware vendors and the third-party software vendors (*viz.* frozen food manufacturers in this case) have a common interest in improving the interface of the device to accept better programs.

Our objective is to explore two issues of particular interest for open APIs on embedded devices: *deliverability* and *predictability*. Deliverability concerns the means used to get a program onto an embedded device. Such devices are often mobile and may have very limited interfaces so practical program delivery is often quite device-specific, as the microwave example shows. Predictability concerns the means of knowing how an open API will be used, or *mis*-used, by a third party vendor. For example, an operator entering a microwave recipe is likely to see an error in a recipe that recommends heating for 30 hours rather than 30 minutes, but a program in a barcode may not receive a similar sanity check by the operator.

This paper describes a prototype implementation of our architecture, together with some experiments and alternatives for various steps. We created a simple microwave oven interface for Java and coded several programs using this interface. Our programs were based on typical recipes we found, but we augmented them with various enhancements that would not be feasible if the operator had to key in the recipe. We studied the problem of predicting resource usage of these programs by expressing desired properties in linear temporal logic and checking them with the model checking tool Spin. We used off-the-shelf compression and barcode encoding techniques to represent the programs, and set up a system to read the code and execute it on a laptop simulating a microwave oven.

Using barcodes printed on food packages to deliver programs seems natural for microwaves. In fact, there have been many similar proposals in recent years to enhance programmability of microwaves. One proposal envisions using a 10 digit number as a program, and a recently-marketed product uses a triple of numbers for programs. However, such programs must be very simple and cannot easily be adapted to the evolution of the devices they program, such as the addition of new sensors and actuators not envisioned when the original code was written. Our architecture uses barcodes to carry programs written in general purpose languages—Java byte-code in our experiments, and thus, provides flexibility. Two-dimensional barcodes have a capacity of 1-2 KB, and our experiments indicate that reasonably interesting recipes can be compressed using off-the-shelf utilities like `gzip` to meet this resource constraint.

Concerning predictability, our experiments show feasibility of applying existing formal verification technology to formulate and verify safety properties such as establishing the the total cooking time for a given recipe does not exceed a specified limit. Unlike typical applications of model checking, our programs are sequential and non-determinism arises almost exclusively from operator actions, not from concurrent interleaving. Applying a model checker like Spin requires significant effort in manually translating the Java recipes into the modeling language of Spin, as well as in formulating the assumptions about the environment. However, blow-up in computational requirements seems to be less critical in our context, and there is potential for developing domain-specific verification tools that are applicable directly to the source code.

The remaining paper is organized as follows. The second section discusses some of the relevant language issues. The third and fourth sections discuss methods related to predictability and delivery respectively. We provide more details about our prototype and experiments in the fifth section. The sixth section summarizes conclusions and discusses future work.

```
1. Make 1 inch slit in plastic
2. 50% power for 5 minutes
3. Remove plastic overwrap
4. Rotate tray 1/2 turn
5. 100% for 1:45
```

Figure 1: A microwave recipe for enchiladas.

```
public static void run(Microwave inMicro) {
    inMicro.display("Make 1 inch slit in plastic");
    inMicro.resetCookTime();
    while (inMicro.getCookTime() < 300) {
        try {
            inMicro.cook(50, 300 - inMicro.getCookTime(), true);
        } catch (PauseException pe) {
            try {
                inMicro.decrementCookTime(1);
            } catch (StartException se) {
                //loop again
            }
        }
    }
    inMicro.display("Remove Overwrap");
    if (!inMicro.canRotate()) {
        inMicro.display("Rotate tray 1/2 turn");
    }
    inMicro.resetCookTime();
    while (inMicro.getCookTime() < 105) {
        try {
            inMicro.cook(100, 105 - inMicro.getCookTime(), true);
        } catch (PauseException pe) {
            try {
                inMicro.decrementCookTime(1);
            } catch (StartException se) {
                //loop again
            }
        }
    }
}
```

Figure 2: The enchilada recipe in Java.

2. PROGRAMMING LANGUAGES

Naturally, our recipe programs need to be written in some programming language. General purpose programming languages have evolved to have high degrees of modularity that help manage software complexity. At the same time, domain-specific languages have been created that adapt modularity to their own particular needs. We therefore have a rich field of language features to choose from in selecting an appropriate language for our programs.

Languages with high degrees of modularity like Java [3](see also `http://java.sun.com/`) have evolved to support resource-rich platforms where the constraints on timing and computational power are fairly generous. This modularity, along with their popularity, makes general purpose languages an attractive choice for taking advantage of open APIs. This explains the excitement that is driving the development of the *Micro Edition*(`http://java.sun.com/j2me/`) of Java, known as J2ME. J2ME attempts to constrain Java in order to allow it to run on devices with limited resources, while leaving the core features of Java intact.

Another strategy is to choose a language that has been designed with embedded systems in mind. Programs for embedded systems typically focus on reacting to the environment instead of transforming data. Consequently, languages that can express interrupt handling and real-time operations elegantly will be a good match. We discuss Esterel, a language for reactive systems, below. The *Functional Reactive Programming* (FRP) family of languages uses a declarative syntax to program reactive systems. Variants of FRP has been used for animation [11] and real-time embedded systems [18].

Our language should have two additional features. It should allow, or even enhance, the predictability and deliverability of its programs. These issues are discussed in general terms in Sections 3 and 4, respectively. It is worth noting mentioning here, however, that the choice of the language must take account of the language's implications for predictability and deliverability.

What would be an appropriate programming language for high level programming of embedded processors? The question is hard to answer in absolute generality, but some range of requirements can be explored in our ongoing case study of programmable microwave ovens. Consider the enchilada recipe from Figure 1 and suppose it was to be delivered as high-level code. One possibility is to create a recipe scripting language. This has many advantages for usability, but domain-specific languages have the disadvantage of being, well, domain specific. We therefore explore a greater level of generality first.

Java is a possible choice. Java is increasingly popular and the J2ME variant of Java is explicitly targeted to run on devices with few resources. Figure 2 shows the same enchilada recipe as a program in Java based on a small (conjectural) microwave object capable of performing operations like cooking and displaying. The program additionally illustrates the potential for enhancements when the program does not need to be keyed in by the user. In particular, it has two features not present in the English recipe: if the microwave has a rotating turntable then step 4 is skipped, and if the user pauses the cooking (by opening the oven door or pressing a 'pause' button) the program will increase the total cooking to account for the food cooling while the oven is paused. This extra functionality could certainly be added to the English recipe, but such a complex recipe would be burdensome for the user.

Another possibility is to choose a language that was designed for programming reactive systems. For example, Esterel [6, 12] (see also `http://www-sop.inria.fr/meije/esterel/esterel-eng.html`) is a language for synchronous programming of reactive systems. The enchilada recipe is given in Esterel in Figure 3. This recipe also skips step 4 if the oven is capable of rotating the food on its own. In order to make the program concise we

```
input Pause, Start, CanRotate;
ouput Power:integer, Rotate;
signal CookTime : integer in
  display("Make 1 inch slit in plastic wrap");
  await Start;
  abort
    loop
      abort
        sustain Rotate || sustain Power(50)
        || every Second do
             emit CookTime(1 + pre(?CookTime)
           end every
      when Pause
      await Start;
    end loop
  when CookTime = 300
  display("Remove plastic overwrap");
  present CanRotate else
    display("Turn tray 1/2 turn");
  end
  emit CookTime(0);
  await Start;
  abort
    loop
      abort
        sustain Rotate || sustain Power(100)
        || every Second do
             emit CookTime(1 + pre(?CookTime)
           end every
      when Pause
      await Start;
    end loop
  when CookTime = 105
```

Figure 3: The enchilada recipe in Esterel.

do not account for food cooling while the oven is paused—adding this functionality is simple.

If we compare the two programs we can see some advantages and disadvantages of each language. In Java, programs are divided into objects which are manipulated using methods. The recipe controls the microwave by invoking the `cook()` method, for example, and therefore passing control to the microwave object. The recipe must then trust the microwave object to rotate the turntable, set the power level, and intercept signals from the user interface. The recipe has no control while the `cook` method is executing, so all the parameters relevant to the cook operation must be grouped as arguments to the method. The Esterel program allows finer grain control that suits the real-time operation of the microwave. The microwave is controlled by setting *signals* such as `Rotate` and `Power`. Signals can be manipulated in parallel. For example, a recipe that called for cooking at 80% power for 20 seconds while only rotating for the first 10 seconds could be written as Figure 4a. The equivalent Java code fragment, shown in Figure 4b, would require two calls to the `cook` method (the third argument of `cook` determines whether the turntable should rotate), requiring the developer to repeat the power setting even though it has not changed. On the other hand, the Java code fragment is more concise (though this is partly because some of the functionality has been moved inside the `cook` method) and more natural for most programmers.

In our current prototype we have chosen to use Java. We feel that its popularity and the availability of tools outweighs the awkwardness of representing real-time programs.

3. PREDICTABILITY

A computer program should do things it is supposed to do, and only those things. In general, it is notoriously difficult to ascertain that a given program meets a specification. Many embedded devices have actuators that can manipulate the physical environment so it is especially frustrating, even dangerous, when a program deviates from its intended behavior. Unlike general purpose computers, many embedded devices offer a very limited interface to their users, making it difficult to diagnose and work around program errors. Programs that control embedded systems are therefore good candidates for analysis techniques that make programs predictable.

In the case of our recipe program, we would like to know that

```
abort
  sustain Power(80)
  || every Second do
       emit CookTime(1
        + pre(?CookTime)
     end every
  || abort
       sustain Rotate
     when Cooktime = 10
when CookTime = 20
```

(a)

```
inMicro.cook(80, 10, true);
inMicro.cook(80, 10, false);
```

(b)

Figure 4: (a) Esterel code fragment. (b) Java code fragment

it will behave as intended for all environments and users. Specifically, we would like to know the following: 1. Will the program terminate? 2. Will the food be cooked for at least 405 seconds? 3. What is the maximum power that will be applied to the food?

Formal methods [9] techniques model programs as mathematical structures which can then be reasoned about mathematically. *Model checking* [8] is a formal methods technique that explores all possible configurations of a finite-state system. With the three questions listed above in mind, we examined Spin [13],an off-the-shelf model checking tool.

Spin is a formal verification tool that analyzes a system by exploring all its possible states. Spin is a mature tool so it is relatively fast and easy to use. Spin's input must be in the form of Promela programs so we had to manually translate our Java program into Promela.

The Java program of Figure 2 is sequential so we do not need to worry about race conditions that arise when two or more threads are interleaved. The nondeterminism comes from the user's actions—when and how often the user pauses and restarts the microwave. Our Promela model includes a simple process that simulates a user.

In a simple sense the answer to our first question is no: a pathological user can always pause the microwave until the food has cooled so much that it needs to be re-cooked for the full 405 seconds. A more precise statement of the question would be "will the recipe terminate if the user eventually stops pausing the microwave?" We augmented our user simulation process so that it would randomly switch into a dormant state where it would stop pausing the microwave. We then constructed a linear temporal logic (LTL) expression, '$\Box(\texttt{user\_stop} \rightarrow \Diamond\texttt{recipe\_finished})$', which encodes our more precise question. Spin verified that the Promela program satisfied the LTL expression. The verification required 35 megabytes of memory and 1.03 million states were visited.

Answering our second question required further changes to the Promela model. We added a new variable that counts the number of seconds of cooking that have taken place. If this counter is added naively then state space becomes infinite—a user can always keep the food cooking for ever, driving the counter arbitrarily high. To overcome this we had to explicitly limit this counter to an arbitrary maximum level. This made the state space finite but it was still too large to search efficiently so we were forced to use an abstraction of our recipe in which each clock tick corresponds to three seconds instead of one. Spin was able to verify that in this abstract model the recipe would always cook the food for at least 405 seconds. The verification required 192 megabytes and visited 5.92 million states. We used a Promela assertion to ensure the cooking time was at least

405 seconds.

Our final question dealt with the maximum power used to cook the food. As was the case for the first question, we need to rephrase this more precisely as "What is the maximum power that will be applied to the food once the user stops pausing the microwave?" In fact, we answered a related question: "will the food be cooked for no more than 405 seconds once the user stops pressing pause?" It would be convenient if Spin could find the maximum amount of time exerted, but we know of no way of finding this maximum short of guessing a maximum and trying it. We modified the counter used above so that it would only increment once the user stopped interfering. We then verified that the counter was no greater than 405 for all states. The verification required 36 megabytes and visited 0.94 million states.

Spin was able to answer all three questions we posed about our program, although some of the analysis required us to use a coarser model of time than we had used initially. The advantages of Spin are its speed and flexibility—constructing and analyzing the model involved some careful thought but the task was mostly straightforward. However, it would be more convenient to use a tool which could take the original program as input; the translation to Promela is error prone, and problems found in the Promela model may not correspond to problems in the original program. An additional problem with Spin was the need to tune the model in order to reduce the state space—it is not always clear whether this tuning changes the fundamental behavior of the model, rendering the analysis irrelevant.

4. DELIVERABILITY

There is a class of embedded devices for which network connectivity is currently either optional, sporadic or impractical. The obvious solution may seem to be traditional media such as flash cards, floppy disks and CDs. Floppy disks, for example, have the advantage of being both familiar and of moderate cost. Yet for some devices, these may not be be feasible. In the case of prepackaged food for programmable microwave ovens, price constraints limit the media cost to a few cents, and the fact that it must be included with the package means that it has to withstand sub-freezing temperature. It must also be convenient enough to use by people uncomfortable with technology such as the elderly. We believe that barcodes provide a viable solution in such situations as the media is extremely cheap, reliable and easy to use. In the rest of this section we shall explore barcodes as means for the delivery of Java bytecode as well as how compression technology can aid in this task.

4.1 Barcodes

Barcodes are interesting because of their low price and convenience. The most common barcode formats are linear codes. The information is represented linearly and vertical redundancy is used to compensate for printing defects and damage while in use. Linear barcodes based on the Universal Product Code (UPC-A) [10] standard are widely used in grocery checkout lines. These use nine to eleven decimal digits and essentially provide an index into a database connected to the reader and cash register. Other linear codes such as Code 39 [2] or Code 128 [1] hold about thirty bytes. The data capacity for linear barcodes is clearly insufficient for delivering programs which has led us to investigate a more recent development in barcode technology—2D barcodes.

As linear barcodes have become almost ubiquitous, there has been a growing desire to store more information in barcode format. This is particularly true in situations where database lookups are impractical and has led to the development of 2D codes. As the name indicates, 2D barcodes store information in both the vertical and horizontal dimensions much the way the letters in the alphabet or pictures use both dimensions to communicate information. Since one dimension can no longer be used for redundancy, error correction coding techniques are usually employed. Though there are many proposed 2D standards, the following are representative of those that have gained industrial acceptance: Aztec code holds 1.9KB [5]; Xerox's DataGlyph (`http://www.dataglyphs.com/`) holds 1KB per sq inch; Data Matrix holds a maximum of approximately 2KB per symbol [4]; Datastrip (patent number 4,782,221) holds a maximum of 1KB per square inch. While the storage capacity for these formats may seem modest, with the aid of compression, a large class of useful programs may be delivered via this medium. There are also several commercial tools on the market that can convert both binary and text files into two dimensional barcode formats. Figure 5 shows the Java class file for the enchilada recipe program given in Aztec and DataGlyph formats. The former was generated by B-Coder from TAL Technologies Inc. and the latter by Xerox's GlyphServer at `www.dataglyphs.com`.

A drawback to the use of 2D barcodes is that they require a somewhat more sophisticated reading device than the one dimensional case; they are usually Charged Coupled Devices (CCD). The prices for industrial-strength hand-held CCD 2D bar-code scanners is currently around $250.00. Less sturdy devices are available at lower prices. In general prices will probably decline somewhat with greater adoption and advances in CCD technology.

Tool	Enchilada (894 Bytes)	Collection (2498 Bytes)
None	100%	100 %
gzip	72%	51 %
jar	180 %	96 %
Pack	60 %	27 %
Sequitur	75 %	54 %

Table 1: Comparison of Compression Programs.

4.2 Compression

In order to decrease the burden imposed by the size constraints associated with using barcodes, we have been investigating compression of small Java programs. The desire to compress Java programs has been around almost as long as the language and was usually driven by limited network bandwidth. Unfortunately, many of the ideas that have emerged for compressing Java programs are not applicable to the embedded environment. Since embedded devices have limited resources, delivery of source code entailing Just-in-Time (JIT) compilation on the device is probably impractical in most causes because of the large memory footprint required. Hence we do not consider compressing source code or proposals such as delivering source code as compact abstract syntax trees [17]. We also ruled out schemes (such as [15], which is targeted for compressing code for embedded systems, or [14]) that alter the KVM or JVM or involve new representations, since these are not likely to gain wide acceptance in the near term. Most such proposals do not have available implementations anyway.

We can report results for the following compression tools: (1) GNU gzip, (2) jar, which uses zip, (3) Pack [16], which is customized for Java bytecode, and (3) SEQUITUR [7], which uses hierarchical grammars. These were applied to the Java program given in Figure 2. We calculated the effect of each program on a collection of small class files. The results are reported in Table 1. Based on this small experiment, Pack seems to be the most effective, probably because it is optimized for Java bytecode.

One aspect of our architecture that we suspect will occur in many

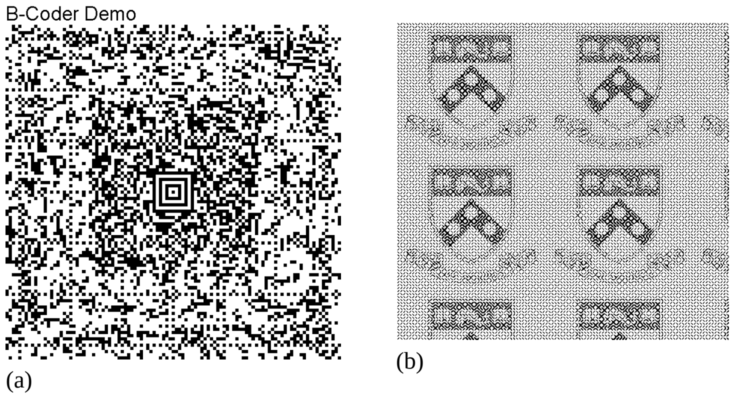

Figure 5: Enchilada Program as (a) Aztec Barcode and (b) Xerox DataGlyph. The DataGlyph format allows images to be embedded in the barcode—we embedded the University of Pennsylvania logo.

other application domains is that programs for a particular embedded device will have similar structure. For example, all the programs for Microwave ovens are recipes. We believe that this fact can be exploited to gain an improvement over most dictionary schemes. The idea is to build a dictionary from a corpus of sample programs. This dictionary is stored at both the compression and decompression locations and used by both algorithms. As part of a simple experiment we created such a dictionary using a basic implementation of the LZ78 algorithm [19]. This algorithm was then modified to use the new dictionary. A 15% improvement was achieved over the original algorithm. We believe that this indicates that the idea has promise.

5. PROTOTYPE ARCHITECTURE

We have implemented a prototype of our proposed architecture. The prototype includes a set of Java classes that form an API to the microwave oven. A recipe developer's program uses these classes to access and respond to the microwave. Different microwave ovens manufacturers will support this API, though manufacturers can customize the implementation details according to the capabilities of the oven.

Once the recipe program is written in Java it is manually translated into Promela, Spin's input language. The recipe developer verifies that the recipe behaves as intended by constructing appropriate linear temporal logic properties and assertions for the Promela recipe, annotating the recipe with extra variables as necessary. Spin's exhaustive search will find any anomalies in the recipe and display an execution sequence that demonstrates the anomaly. If the anomaly is due to a bug in the original Java recipe that recipe and its Promela model must be updated to fix the bug. If the anomaly is due to a discrepancy between the Java recipe and the Promela model the model must be updated to bring the model in line with the original Java recipe.

When Spin shows that a recipe satisfies the necessary properties the Java version of the recipe is compiled to a class file. The class file is compressed using gzip (our system also supports Pack) and converted to an Encapsulated PostScript (EPS) containing Aztec barcode. The EPS file is then printed using a normal laser printer.

We used a Linux workstation with an attached barcode scanner to simulate a microwave oven. The workstation runs a Perl script that takes input from the barcode scanner, decompresses it, and links it to a Java program that displays a mock-up microwave. A user can interfere with the mock-up by opening and closing doors, and pausing and restarting the cooking.

We exercised our prototype system with three microwave recipes, chosen from actual frozen food packages. All three recipes were encoded as barcodes and run on our microwave simulator, though only one recipe was analyzed using Spin.

We used the compiler and virtual machine from Sun's JDK1.3.1 for all the steps where we used Java. The recipe analysis was performed using Spin 3.4.13 and Xspin 3.4.7 on a workstation running RedHat Linux 7.2. The workstation had 512 megabytes of RAM and a 1.5 GHz Pentium 4 processor. A discussion of the analysis can be found in Section 3. The class file of the recipe was compressed using gzip 1.3 and then converted to an Aztec barcode using B-Coder from TAL Technologies version 4.0. The microwave simulator ran on a workstation with 80 megabytes of RAM, a 166MHz Pentium MMX processor and an Imageteam 4410 barcode scanner, running RedHat 6.2.

The recipes, Java classes and Promela models are available at `http://www.cis.upenn.edu/sdrl/mirl`.

6. CONCLUSION

The main contribution of this paper is our experimental prototype which demonstrates feasibility of delivering verified programs in barcodes. Such a set up can be used for open API for controlling myriad of devices from home appliances to medical devices.

We have shown that existing off-the-shelf model checkers like Spin are capable of analyzing the kind of small programs we envision running on top of embedded systems. Unfortunately, Spin requires translating a program into an input language like Promela—an error-prone process that may lead to a model that does not correspond to the original program, and we would like to develop domain-specific tools that can analyze source code.

Finally, while we have not assumed any network connectivity for our prototype, there are interesting architectural possibilities combining barcodes with network access. We plan to explore such alternative architectures in the future.

7. ACKNOWLEDGMENTS

This research was supported in part by NSF award CCR 0208990, NSF award ITR/SY 0121431, and ARO URI award DAAD19-01-1-0473.

8. REFERENCES

[1] I. 15417:2000. Automatic identification and data capture techniques - bar code symbology specification - code 128. Technical report, International Standards Organizatiopn, 2000.

[2] I. 16388:1999. Automatic identification and data capture techniques -bar code symbology specifications – code 39. Technical report, International Standards Organization, 1999.

[3] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, Reading, MA, USA, third edition, 2000.

[4] A. BC11-ISS. Data matrix. Technical report, AIM, 1996.

[5] A. BC13-ISS. Aztec code. Technical report, AIM, 1997.

[6] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: design, semantics, implementation. Technical Report 842, INRIA, 1988.

[7] D. M. C. Nevill-Manning, I.H. Witten. Compression by induction of hierarchial grammars. In J. A. Storer and M. Cohen, editors, *Proceeding Data Compression Conference*, pages 244–253. IEEE Press, 1994.

[8] E. Clarke and R. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[9] E. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[10] U. C. Council. Ansi/ucc1-2000:u.p.c. symbol specification manual. Technical report, American National Standards Institute, 2000.

[11] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[12] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[13] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[14] T. Kistler and M. Franz. A tree-based alternative to java byte-codes. *International Journal of Parallel Programming*, 27(1):21–34, January 1999.

[15] C. C. L. Clausen, U. Oagh-Schultz and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages*, 22(3):1–19, May 2000.

[16] W. Pugh. Compressing java clas files. In *ACM Sigplan Conference on Programming Language Design and Implementation*, pages 247–258. ACM Press, 1999.

[17] C. Stork and V. Haldar. Compressed abstact syntax trees for mobile code. In *Proceeding of Workshop on Intermediate Representation Engineering*, 2001.

[18] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *International Conference on Functional Programming (ICFP '01)*, Florence, Italy, September 2001.

[19] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. IEEE Transactions Information Theory, 24(5):530–536, 1978.