Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement

James Shin Young, Josh MacDonald, Michael Shilman, Abdallah Tabbara, Paul Hilfinger, and A. Richard Newton

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

Abstract

Successive, formal refinement is a new approach for specification of embedded systems using a general-purpose programming language. Systems are formally modeled as Abstractable Synchronous Reactive systems, and Java is used as the design input language. A policy of use is applied to Java, in the form of language usage restrictions and class-library extensions, to ensure consistency with the formal model. A process of incremental, user-guided program transformation is used to refine a Java program until it is consistent with the policy of use. The final product is a system specification possessing the properties of the formal model, including deterministic behavior, bounded memory usage, and bounded execution time. This approach allows systems design to begin with the flexibility of a general-purpose language, followed by gradual refinement into a more restricted form necessary for specification.

1 Introduction

A variety of languages are used in the design and specification of hardware and software for embedded electronic systems[5]. General purpose languages such as C and C++ are usually used for the design of software, while hardware description languages, such as Verilog and VHDL, are used in the design of hardware. Embedded systems design requires integration of hardware and software components, often a difficult task if they are described using dissimilar languages. Instead of using different languages for design of hardware and software, a single language for system specification is desired. We propose a methodology that allows a single-language platform to be used for early design, as well as for guiding subsequent refinement, partitioning, and synthesis steps to obtain a final implementation.

In particular, we describe an approach for using Java as a design and specification language for embedded systems. Java is a pragmatic choice for several reasons. Since it is a member of the C "family" of languages, it is familiar to designers. Unlike C and C++, it provides standard language and library support for concurrency. Its treatment of arrays permits better static and dynamic error checking than is conveniently feasible in C and C++. Finally, while Java's expressive power is comparable to C++, it is a much simpler language, which reduces the difficulty of program analysis, optimization, and transformation. Using Java also has many practical benefits– its widespread adoption by the science and engineering community promises a large base of support, in the form of compilers, debuggers, development environments, and class libraries.

Embedded systems function as sub-components of larger systems, and their design requisites are often quite different from general-purpose computing systems, the domain for which Java is typically used. Many embedded systems are *reactive*[1], operating at speeds dictated by their external environments, and must be reliable and predictable. Therefore, embedded systems should behave deterministically, and operate within bounded resources, including time and memory. However, Java programs in general guarantee neither determinacy nor bounded resource usage.

To address this fundamental incompatibility, our approach anchors programs on formal models of computation. The Abstractable Synchronous Reactive (ASR) model is one we have developed which has properties suitable for embedded systems specification. Restrictions on the usage of Java are applied in order to ensure the programs are consistent with the ASR model. This principle of language restriction is one area where our approach differs most from other proposals to use Java for system specification, such as that of Helaihel and Olukotun[10].

However, restricting the use of Java limits its flexibility and expressiveness. Designers may be discouraged if the resulting language is overly restrictive. We have addressed this concern by developing a methodology for specification development called *successive, formal refinement* (SFR). SFR supports the transformation of a program written in a general-purpose programming language into one that can be embedded within a more restricted model of computation. It consists of a series of static analyses of programs, coupled with an iterative process of incremental, semi-automated program transformation. SFR enables a system to be designed initially in an unrestricted manner using a general-purpose programming language, and then be gradually refined into a restricted form for use as a system specification.

[1] This term was first introduced by Pnueli[9].

In this paper, we present the SFR methodology and describe how it is applied to Java programs to obtain a specification suitable for embedded systems, using the ASR model as a formal basis. We have developed a set of tools, collectively known as *JavaTime*, to aid analysis and transformation of Java source code, and used it to demonstrate application of SFR to a number of examples.

2 Successive, Formal Refinement

Successive, formal refinement rests on two foundations: a *policy of use* on the source language, and a procedure of *incremental transformation*. In general, programs expressed in a general-purpose language cannot be expressed as equivalent systems in a specific model of computation. In SFR, a policy of use is imposed on the source language, S, to make it consistent with the target model, T. A policy of use consists of *restrictions* and *extensions*. The restrictions removes portions of S incompatible with T, while the extensions introduce semantics present in T that have no equivalent in S. The result is a new language S', as shown in Figure 1, whose programs are expressible in T.

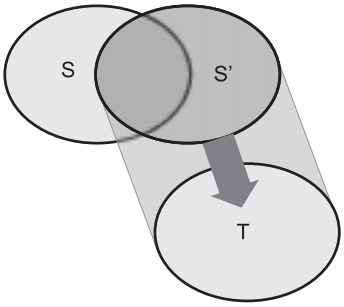


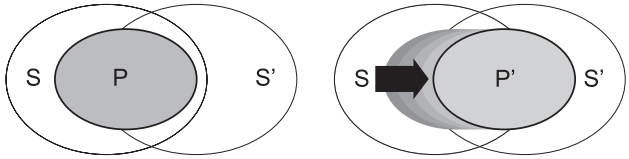
Figure 1. A policy of use applied to language S yields S', compatible with model T.

In this methodology, a user's initial program P is contained within language S (Figure 2a), and cannot be mapped directly onto T. A process of incremental program transformation is used to refine it into a program in language S', to make it valid with respect to the policy of use.

The program is analyzed to verify that the rules in the policy of use are satisfied. If a violation is found, the user is presented with information regarding the nature of the error, and a list of suggested solutions for fixing the problem, including automated program transformations when possible. The user can then modify the program manually or allow the tools to alter it automatically.

This process of analysis and modification is repeated until the program complies with all rules in the policy of use. At this point, the altered version of the program, P', is contained within S' (Figure 2b). Because S' is constructed to be compatible with T, P' corresponds to a system in T.

The SFR process may be applied with respect to a variety of target models, each with its own policy of use. This enables one to use a single general-purpose language for designing systems in an implementation-independent manner, and choose from different models according to the desired implementation style. In addition to the ASR model described in this paper, others might include



(a) (b)

Figure 2. (a) Original program is in language S. (b) Program is embedded in S' through successive, formal refinement.

dataflow, discrete-event, and hierarchical finite state machine models, or models defined by the user. This provides a design language and environment that is highly expressive, and yet can be customized to suit the needs of a particular implementation medium.

Systems for program construction through transformations exist for the development of software[1][13], but ours is the first effort we know of to apply such techniques for the purpose of formal modeling of mixed hardware/software systems. Transformation systems for software typically are used to improve program performance while preserving semantics, while in the SFR methodology transformations are used to restrict and alter a program's semantics.

3 The Abstractable Synchronous Reactive Model

We use the *Abstractable Synchronous Reactive* (ASR) model to specify reactive embedded systems. In the ASR model, time is divided into hierarchically nested *instants*. Systems are represented as collections of *functional blocks*, *channels*, and *delay elements*. Blocks serve to generate output values based on their inputs; channels carry set-valued data between blocks; and delay elements carry data between successive instants in time. ASR systems are *reactive*; that is, they interact with their environments by reacting to the inputs presented to it and producing outputs. The operation of these systems is driven by their environments; if no inputs are provided to a system, it would simply sit idle. Figure 3 shows a graphical representation of a system in the ASR model. The white rectangles are functional blocks, while the shaded rectangle represents a delay element. Channels are drawn as directed edges.

The ASR model is based on the *synchronous* approach to reactive systems, in which systems are said to compute instantaneously and time is divided into instants[2]. Synchronous languages, such as Esterel[3], Lustre[7], and Statecharts[8], also use this approach as the foundation for their formal models of computation. The ASR model is distinguished from other synchronous models in that time may be hierarchically abstracted, as well as function.

Functional blocks calculate output values based on the input values. Inputs and outputs are members of ordered sets, and blocks are restricted to compute only continuous functions between these domains. These idealized blocks produce their outputs synchronously with their inputs, meaning that inputs and their resulting outputs appear in the same instant. Likewise, propagation of values through channels also occurs within a single instant. Channels

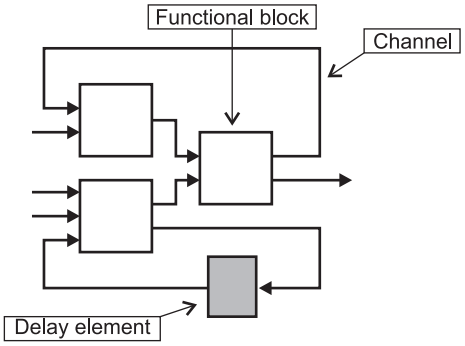


Figure 3. An ASR system.

connect the outputs and inputs of blocks and delay elements together.

Blocks and channels cannot hold values across instants. Instead, values are carried through time by delay elements. At each instant, a delay element's output is equal to the value of its input at the previous instant. From a block's point of view, inputs originating from delay elements appear indistinguishable from external inputs.

It is possible for the input to a block to depend on its output, by introducing cycles in the system graph with no delay elements in the path. To resolve such cyclic dependencies, ASR uses a fixed-point semantics fashioned after the scheme presented by Edwards[4].

Time is viewed as a partially ordered set of instants. The instants may be nested, yielding a hierarchical structure of time, as shown in Figure 4. Thus, although blocks and channels are said to execute instantaneously, their operation can be composed of a set of "sub-instants". Such abstraction of time can be useful in hiding details of implementation[12]. For example, communication of a message between two processors may be viewed as a single instant, rather than as a multitude of instants representing the detailed protocol activities. The amount of physical time associated with an instant is not predetermined. It is associated with the design later, as the solution to a set of temporal constraints imposed by components used to implement the design or via constraints imposed by the system's environment. In that way, the nested refinement described above is symbolic and need not involve "nanoseconds" until later in the design process.

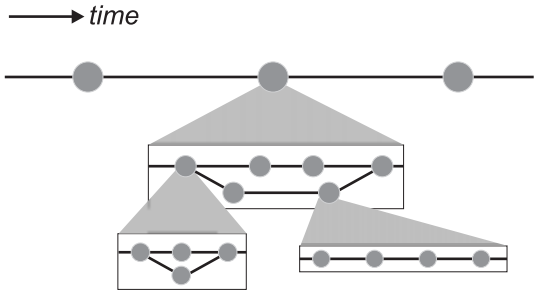


Figure 4. Abstraction of instants in time.

The ASR model has properties that make it useful for specifying embedded systems. ASR systems are deterministic; that is, any particular input can produce only one possible output. Also, they are compositional– an aggregation of blocks is functionally equivalent to a single block, while a collection of blocks and delay elements is equivalent to a system containing one block and one delay element (Figure 5). ASR systems may be represented as a hierarchical composition of ASR subsystems.

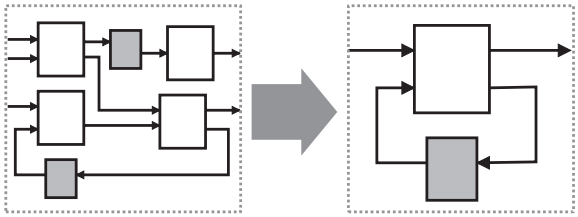


Figure 5. Abstraction of ASR systems in space.

4 Specifying Embedded Systems Using Java

Our approach for design and specification of reactive embedded systems uses Java as the design input language, and SFR is applied to refine programs to be consistent with the ASR model. To begin, we must first clarify the manner in which we interpret Java programs for specification purposes.

Java's ability to dynamically load and link objects prevents a standard Java compiler from globally analyzing and optimizing a program, as any of the classes the program uses may be individually modified and replaced at run time. To enable more detailed analysis of a system's behavior, we assume all the code used to describe it is known at compile time. A class used in describing a particular ASR block cannot be modified without analyzing the other classes to determine the overall effect on the block's behavior.

The Java language specification[6] divides the execution of a Java program into loading, linking, initialization, and finalization. In our approach, the loading, linking, and initialization steps are considered to describe the structure of the system, not the behavior; the program's actions following loading, linking, and initialization are considered to be the specification of the system's behavior. Finalization is disallowed, as it may be considered as representing the termination or destruction of the system.

Java allows concurrency to be specified in the form of user-managed threads that communicate with one another by modifying and reading shared variables. Thus, in general we consider Java programs to describe partial orders of events in a system, as shown in Figure 6.

4.1 Determining the policy of use

In the SFR methodology, a policy of use, consisting of restrictions and extensions, is imposed on the source language to reconcile it with the target model. The user interacts with tools to verify a program's compliance with the policy, and rectify violations through an interactive process of incremental program transformation. In this application, the source is the Java language and the target is the ASR model of computation.

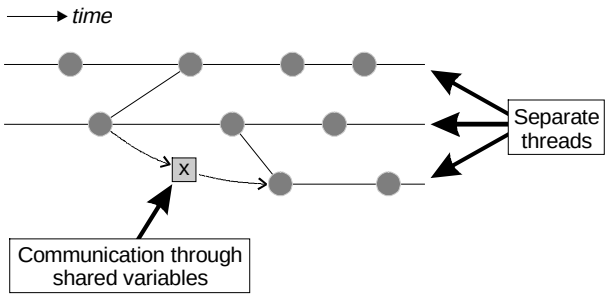


Figure 6. Java programs specify a partial order.

Compliance with restrictions is verified through static analyses of source code. Program transformations may be applied automatically to the abstract syntax tree of the Java program, or manually if the tools are not able to provide a satisfactory transformation. Extensions are made through model-specific class libraries, so that programs undergoing refinement remain valid Java programs throughout, and may remain usable by standard Java development tools. Others have utilized class libraries to model systems[11], but the user-written code is not restricted and verified to ensure the libraries are used correctly.

We may determine the necessary restrictions and extensions by comparing the capabilities and limitations of Java and the ASR model:

- Java programs may be nondeterministic, while ASR systems must be deterministic.
- Java programs may require infinite amounts of memory to execute. In comparison, the memory available to ASR systems is finite.
- The ASR model separates time into hierarchically nested instants, while Java has no notion of time.
- ASR systems operate by reacting to their environments through well-specified inputs and outputs. In Java, no clear distinction between the system and its environment is made.

Therefore, restrictions must be placed on Java programs to guarantee they are determinate and use bounded memory; extensions are needed to distinguish inputs and outputs; and we must define the boundaries of instants.

4.2 Designing an ASR system

A class called ASR is provided to the user for describing ASR systems using Java. It must be used as the base class of a specification, and serves to model the functionality of an ASR system in Java. From the environment's point of view, an object subclassed from it looks like a "black box," operated by providing it with inputs, which causes the system to produce outputs. The ASR class includes output and input ports, used to convey signals to and from the environment. To specify the functionality of the block, the user writes code to fill in the `run` method Figure 7 shows a conceptual sketch of the resulting Java object.

The lifecycle of an object subclassed from the ASR class is considered to be divided into two parts: *initialization* and *behavior*. Initialization includes the invocation of its constructor and execution of static initialization routines. It represents the creation and initialization of the object; if implemented as a hardware/software system, this might correspond to the fabrication of the system and power-on reset behavior.

The object's operational behavior is defined with respect to its `run` method. The object sits idle until its `run` method is invoked

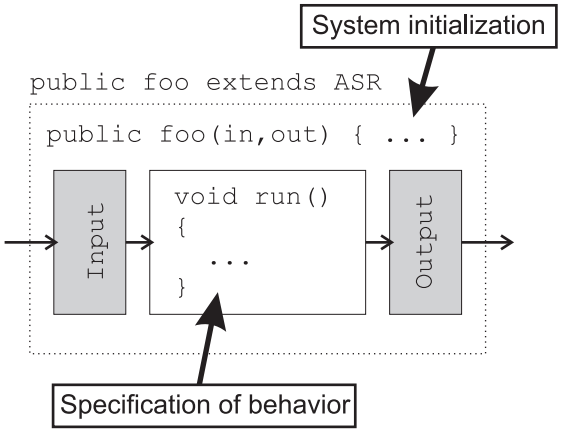


Figure 7. Encapsulation of Java design in ASR class.

by the environment, at which time it produces a set of output values and returns to its idle state. It is the responsibility of the environment to provide the appropriate inputs. With respect to time, an instant is initiated by the invocation of the component's `run` method, and ended when the method completes. The sequence of actions performed in the `run` method can be considered as instants nested within an instant, and are not visible to the object's environment. That is, the execution of the `run` method appears atomic to external observers.

4.3 Restricting the initial design

A specification of an ASR block consists of a single class that extends ASR, and all the classes that this primary class may require, either directly or indirectly. The code invoked by the `run` method of the main class must be analyzed and restricted with respect to the policy of use of the ASR model. Unlike standard Java's dynamic loading model, classes that constitute a specification are considered in our analyses to be bound at compile time, and cannot be restricted and verified independently of one another. This is because verification of a block's behavior requires that all dependencies between classes and methods be statically identified.

The amount of memory in an ASR system is fixed, but Java programs may be written such that they require an unbounded amount of memory, for example if dynamic data structures are used. Therefore, one important restriction is that objects may be instantiated only during initialization. This is accomplished by scanning the code and identifying possible allocation statements. One problem that may be encountered is the use of linked structures, and these types of complex structures should be checked for and eliminated in favor of statically allocated data structures.

We must also take care that an ASR object's internal state may not be externally accessible by requiring the object's variables to

be `private`. This prevents external modification or observation of the object's state, which undermines data encapsulation and abstraction, and may result in unpredictable behavior.

Addition of the notion of instants motivates an additional restriction: Computation of the output must be bounded in time; otherwise, the system's execution would never advance to the next instant. For arbitrary Java programs, verifying bounded execution is impossible, as it is equivalent to solving the halting problem. At the risk of reducing the computational capabilities of the block, we must therefore restrict the usage of the language. In particular, calculable upper bounds on loop iterations are required, circular method invocations are not allowed, and use of methods that may halt or indefinitely suspend thread execution is forbidden. Thus, `while` and `do while` loops may not be used, and the iteration variable in `for` loops cannot be modified within the loop.

Nondeterminism is easily to implement in a Java program, due to its integrated support for multithreading. For example, in Figure 8, threads *A*, *B*, and *C* share variable *X*. *A* and *B* write to *x*, while *C* reads the value from *x*. The order in which the three threads access *x* may differ between different executions of the program, and may produce different behaviors. We have not found a way to guarantee deterministic behavior in multithreaded programs without severely limiting the usability of Java's threads package. In the current policy of use, direct use of Java threads is prohibited, and concurrency is obtained through specification of separate functional blocks.

The restrictions given in the policy of use are conservative– they are sufficient to ensure that a Java program is consistent with the ASR model, but may not be necessary. That is, there are programs that violate our restrictions, but are expressible as ASR systems.

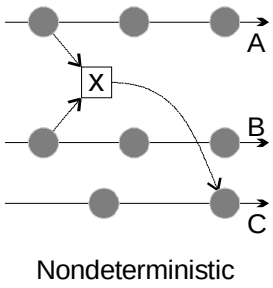


Figure 8. Nondeterministic thread interaction.

5 Experiments

We have applied our proposed methodology to a number of design examples, and found that restriction of Java programs is feasible and beneficial to performance. *JavaTime*, a set of tools to support the SFR methodology has been developed. *JavaTime* takes as input the program source code, and allows analyses, restrictions, transformations, and interpretations to be associated with a design.

A number of publicly available Java programs were taken as examples of unrefined designs to be implemented in an embedded system. The examples were incrementally transformed until they conformed to the restrictions set forth in the ASR model's policy of use. Identification of policy violations were aided by the *JavaTime* tools, which provide a flexible environment for searching and manipulating a program's abstract syntax tree. A mix of manual, semi-automated, and automated techniques was used for program transformation. The resulting restricted programs were compared to the original by execution on standard Java platforms.

In general, the number of transformations required to attain compliance with the policy of use was not unreasonable, and depended greatly on the original designer's programming style. For example, one designer might favor a particular control construct or data structure more than another. Transformations of `while` loops, which are prohibited in the policy of use, to restricted `for` loops was possible in the majority of cases, and limitation of new object instantiations to the initialization phase was found to be feasible for the examples as well.

		Unrestricted program	Restricted version	Restricted/ unrestricted
Sun jdk 1.0.2	Initialization time (s)	1.4	3.5	2.50
	Reaction time (s)	11.8	8	0.68
Café JIT	Initialization (s)	0.5	2.1	4.20
	Reaction time (s)	3.6	1.2	0.33
	Program size (k)	19.7	21.2	1.08

Table 1. Comparison of unrestricted vs. restricted versions of JPEG design example.

Table 1 shows results obtained from transforming our largest design example, a JPEG compression/decompression program. The JPEG algorithm was encapsulated using the ASR base class, and images were input as arrays of integers representing its red, green, and blue components. The *initialization time* reflects the amount of time spent in the object's constructor, while the *reaction time* is the average time required to process the 130x135 pixel test image. If multiple images are to be processed, the initialization time is incurred only once, while the reaction time is incurred for each image processed. For typical embedded systems, the reaction time is of primary concern, as initialization occurs relatively infrequently. These results were measured for the programs running on both a conventional Java Virtual Machine, and a Just-In-Time compiler. The program size given is the size of the Java `.class` files. The machine used is a 150MHz Pentium processor with 32 MB RAM and Windows 95.

Under both execution environments, the restricted version takes longer to initialize, but once initialization is complete, processes images faster than the unrestricted version. The performance differential reflects the original program's extensive use of dynamic data structures. The ASR policy of use disallows dynamic memory allocation, thus the restricted version we created uses only static data structures created during initialization. Hence, the restricted program is much slower to initialize, but performs no additional memory allocation once activated. While a significant performance improvement is obtained, the restricted program is roughly equal in size to the original.

These experiments show that a program can be modified to make it consistent with the ASR policy of use. Furthermore, execution of the modified example in standard Java environments resulted in

increased system initialization times but shorter response times. Such a characteristic is consistent with the needs of reactive embedded systems, which maintain an ongoing dialogue with their environments. Hence, higher startup overhead is tolerable in exchange for faster response times. This suggests that the SFR methodology might also be used as a directed optimization strategy for programs running on standard Java Virtual Machines.

6 Summary

We have presented successive formal refinement, a new methodology for design and specification of embedded systems This methodology allows systems to be designed initially using a general-purpose programming language and refined to a form consistent with a formal model of computation, within a single-language platform.

Refinement of a design with respect to the ASR model yields a program that is deterministic, uses bounded memory, executes in bounded time, and is compositional with other systems in the model. A program with these properties is suitable for use as a specification for embedded, reactive systems.

A system, called *JavaTime*, was created to aid in the program analyses and transformations required for SFR. The methodology was applied to a number of design examples, and found to be applicable and successful in improving performance on standard Java execution platforms.

Future development will be carried out along several lines. Development and implementation of more sophisticated Java program analysis techniques will enable a wider variety of transformations. Work on advanced user interface and system visualization tools will also better the quality of interaction a user has with the system in refining a design.

To support a variety of application domains, policies of use will be developed for additional models of computation, and the interaction of components specified with respect to different models explored. To provide a complete system design solution, the *JavaTime* system should be coupled with verification and synthesis tools.

Acknowledgments

This research was supported by DARPA, under contract DABT63-95-C-0074-NEWTON-06/96, NASA, under contract NCC 2-999, Synopsys, and Intel. Their support is gratefully acknowledged.

References

[1] F. Bauer, B. Moller, H. Partsch, and P. Pepper, "Formal Program Construction by Transformations– Computer-Aided, Intuition-Guided Programming," *IEEE Trans. Software Engineering*, vol. 15, no. 2, pp. 165-180, Feb. 1989.

[2] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proc. IEEE*, 79(9):1270-1282, September 1991.

[3] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, vol. 19, pp. 87-152, 1992.

[4] Stephen A. Edwards, *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. Ph.D. Thesis, University of California, Berkeley, 1997. Available as UCB/ERL M97/31.
`http://ptolemy.eecs.berkeley.edu/papers/97/sedwardsThesis/`

[5] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Englewood Cliffs, NJ: Prentice Hall, 1994.

[6] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Reading, MA: Addison-Wesley, 1996.

[7] CN. Halbwachs, *et. al.* The synchronous data flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305-1320, 1991.

[8] D. Harel, "Statecharts: a visual approach to complex systems," *Science of Computer Programming*, 8:231-274, 1987.

[9] D. Harel and A. Pnueli, "On the Development of Reactive Systems," vol. 13 of *NATO ASI Series. Series F, Computer and Systems Sciences*, pp. 477-498. Springer-Verlag, 1985.

[10] R. Helaihel and K. Olukotun, "Java as a Specification Language for Hardware-Software Systems," *Proc. ICCAD '97*, pp. 690-697, 1997.

[11] S. Liao, S. Tjiang, and R. Gupta, "An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment," *Proc. 34th DAC*, pp. 70-75, 1997.

[12] J. Rowson and A. Sangiovanni-Vincentelli, "Interface-Based Design," *Proc. 34th DAC*, pp. 178-183, 1997.

[13] D. Smith. "KIDS: A Semiautomatic Program Development System," *IEEE Trans. Software Engineering*, vol. 16, no. 9, pp. 1024-1043, Sept. 1990.