Copyright © 2000, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

# SYSTEM-LEVEL TYPES FOR COMPONENT-BASED DESIGN

by

Edward A. Lee and Yuhong Xiong

Memorandum No. UCB/ERL M00/8

29 February 2000

# SYSTEM-LEVEL TYPES FOR COMPONENT-BASED DESIGN

by

Edward A. Lee and Yuhong Xiong

Memorandum No. UCB/ERL M00/8

29 February 2000

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

# System-Level Types for Component-Based Design

Edward A. Lee and Yuhong Xiong

*University of California at Berkeley*
*eal@eecs.berkeley.edu*

*February 29, 2000*

## 1.0 INTRODUCTION

Type systems are one of the great practical triumphs of contemporary software. They do more than any other formal method to ensure correctness of software. Object-oriented languages, with their user-defined abstract data types, have had a big impact in both reusability of software (witness the Java class libraries) and the quality of software.

Type systems give us a vocabulary for talking about larger structure in software than lines of code and subroutines. However, type systems talk only about static structure. It is about the *syntax* of procedural programs, and says nothing about their concurrency or dynamics. Those properties are relegated to more informal descriptions, such as design patterns [10] and object modeling [9].

For example, it is not part of the type signature of an object that the initialize() method must be called before the go() method. Temporal properties of an object (method x() must be invoked every 10ms) are also not part of the type signature. Work with active objects and actors [1][2] move in the right direction by being a bit more explicit about dynamic properties of the interfaces of components. But they do not say enough about interfaces to ensure safety, liveness, consistency, or real-time behavior.

At its root, a type system constrains what a component can say about its interface, and how compatibility is ensured when components are composed. Mathematically, the more sophisticated type system techniques depend on a partial order of types, typically defined by a subtyping relation or by lossless convertibility (which can be thought of as ad hoc subtyping). They can be built from the robust mathematics of partial orders, leveraging for example fixed-point theorems to ensure convergence of type checking, type resolution, and type inference algorithms.

With this very broad interpretation of type systems, all we need is that the properties of an interface be elements of a partial order, preferably a complete partial order (CPO) or a lattice [29]. We suggest first that dynamic properties of an interface, such as the protocols used by a component to interact with other components, can be described using nondeterministic automata, and that the pertinent partial ordering relation is the simulation relation between automata. We also speculate that various timed automata extensions can perhaps be used in similar ways to define much more completely the temporal properties of an interface than what is common practice today.

### 1.1 Strongly typed languages

Type systems in modern languages serve to promote safety through static (compile time) and dynamic (run time) checking. In a computation environment, two kinds of run-time errors can occur, trapped errors and untrapped errors. Trapped errors cause the computation to stop immediately. A run-time handler can attempt to recover gracefully. Untrapped errors, which may go unnoticed (for a while) and later cause arbitrary behavior, can be disastrous for an embedded system. Moreover, they are less likely to be detected in testing. Examples of untrapped errors in many general purpose languages are jumping to the wrong address, or accessing data past the end of an array.

Strongly typed languages help prevent both trapped and untrapped errors. Many errors are detected at compile time, and run-time support for the type system can help ensure that the remaining errors are trapped. This helps prevent arbitrary behavior, but it only deals with certain aspects of program behavior. Moreover, run-time support for the type system, which can be provided systematically through preconditions and contracts, may incur substantial overhead.

Modern languages, such as Java and ML, emphasize avoiding untrapped errors. There is significant run-time overhead incurred in the required safety checks. Several researchers have shown that in many cases, this overhead can be eliminated through compile-time analysis (see for example [33]). The approach is to augment the type system to include such properties as array size, and then to annotate the generated code with assertions of safety. A run-time environment can thus bypass the safety checks.

Ousterhout [25] argues that strong typing compromises modularity and discourages reuse.

> "Typing encourages programmers to create a variety of incompatible interfaces, each interface requires objects of specific type and the compiler prevents any other types of objects from being used with the interface, even if that would be useful."

The alternative he advocates is languages without strong typing, such as Lisp and Tcl, where safety can only be achieved by extensive run-time checking. However, since type checking is postponed to the last possible moment, the system does not have fail-stop behavior, so a system may exhibit erroneous behavior only after running for an extended period of time after the error has occurred. Identifying the source of

the problem can be difficult, and guaranteeing the code may be impossible.

Ousterhout raises a valid point, but the solution is not to discard strong typing. Particularly for embedded systems, the extra degree of safety offered by strong typing overwhelms even the desire for modularity and reuse. How can we achieve modularity and reuse without discarding strong typing? One solution is to use polymorphism, reflection, and run-time type inference and type checking.

Strong typing and type resolution have other benefits in addition to the ones mentioned above. Strong typing helps to clarify the interfaces of components and makes libraries more manageable. Just as typing may improve run-time efficiency in a general-purpose language by allowing the compiler to generate specialized code, type information can be used for efficient synthesis of embedded hardware and software configurations. For example, if the type checker asserts that a certain polymorphic component will only receive integer arguments, and that component is to be implemented in configurable hardware, then only hardware dealing with integers needs to be synthesized.

In general-purpose strongly-typed languages, such as C++ and Java, static type checking done by the compiler can find a large fraction of program errors in object-oriented programs. However, with networked embedded systems where parallel execution, agents, migrating code, and software upgrades are all possibilities, static type checking does not do enough. Some of the type checking must be done at run time. Java's run-time type identification (RTTI) system together with its reflection package specifically addresses this problem by supporting run-time queries of type constraints and run-time verification of type compatibility.

Type systems in modern programming languages, however, do not go far enough. Many errors that in principle may be detectable at compile time are not within the scope of the type system. Several researchers have proposed extending the type system to handle such errors as array bounds overruns, which are traditionally left to the run-time system [33]. But many are still not dealt with. For example, the communication protocols between concurrent processes are not type checked. Yet failures in concurrency and synchronization are common causes of critical system failures in embedded systems.

### 1.2 Extended Types

Object-oriented programming promises software modularization, but has not completely delivered. The type system captures only static, structural aspects of software. It says little about the state trajectory of a program (its dynamics) and about its concurrency. Nonetheless, it has proved extremely useful, and through the use of reflection, is able to support distributed systems and mobile code.

Our proposal is to augment the type system to embrace dynamic properties of components. There is considerable precedent for such augmentations of the type system. For example, Lucassen and Gifford introduce state into functions using the type system to declare whether functions are free of side effects [20]. Martin-Löf introduces *dependent types*, in which types are indexed by terms [22]. Xi uses dependent types to augment the type system to include array sizes, and uses type resolution to annotate programs that do not need dynamic array bounds checking [33]. The technique uses singleton types instead of general terms [13] to help avoid undecidability. While much of the fundamental work has been developed using functional languages (especially ML [12]), there is no reason that we can see that it cannot be applied to more widely accepted languages when applied at higher levels of abstraction.

Another innovative use of type systems is that of Necula, who describes the use of proof-carrying code [24]. Here, a program includes with it a proof of validity or compliance to some requirement, such as safety. If the code type checks, then it is valid. This is used primarily for security. The main drawback appears to be in the difficulty of constructing the proofs. We may face a similar drawback in our use of dependent types for capturing real-time properties in that constructing the real-time properties may prove difficult.

## 2.0 PROCESS-LEVEL TYPE SYSTEMS

Extended type systems could, in principle, capture the following aspects of a system:

- protocols for communication between concurrent components (e.g. rendezvous, asynchronous message passing, streams, events);
- models of time (e.g. a continuum, discrete, clocked, partially ordered); and
- flow of control (e.g. synchronous, scheduled firings, process scheduling, real-time).

Components will have to declare their requirements in these dimensions as part of their interface definition. However, this must be done at minimal cost to modularity and reuse. How can this be done?

### 2.1 Polymorphism

In hardware design, there has been movement in the direction of interface synthesis. In [26], Passerone, et al., describe component interfaces by automata, and synthesize protocol translators to connect components with distinct interfaces. This is one possible approach. An alternative approach, however, is to define components with tolerant interfaces, and to specialize them at synthesis time. How can this be done systematically?

Our approach is also to define component interfaces using automata. However, these automata are as non-specific as possible. With judicious use of nondeterminism, we can declare interfaces abstractly so that they impose only minimal constraints on the implementation. Then our approach to synthesis is to use polymorphism.

Our polymorphic interfaces are nondeterministic automata that can be simulated by a variety of deterministic automata. A synthesized (hardware or software) realization will implement one of these deterministic automata. But which one is implemented depends on the context in which the component is used. Generally, when connecting a pair of components, we seek the lowest cost implementation that simulates the interface automata of the two components. This lowest cost solution is given by the least solution to a

system of inequalities on the type hierarchy. We are assured of the existence and uniqueness of this solution by theorems that guarantee the existence of a unique least fixed point of a monotonic function on a lattice. This approach will be made more concrete below.

### 2.2 Language Support

Embedded systems must provide assurance of various properties. The most important mechanism for assurance is that the designer understand the system. But generally the designer needs a great deal of help. Object-oriented programming, for example, helps a designer understand the static structure of a software architecture by providing syntactic features of the language supporting object-oriented design, and by providing a compiler that checks types. We suggest that extended types that include dynamic properties of an interface can help a designer to understand the dynamic interaction of components.

Object-oriented programming, however, only works in practice when the programming language provides syntactic support for it. While it is possible to build object-oriented programs in C, it is rare, and requires more discipline and a deeper understanding of object-oriented concepts than most programmers have.

There has recently been considerable progress in codifying larger-scale program structure than that directly expressed by standard object-oriented languages, for example using UML. Nonetheless, the best parts of UML (especially the static structure diagrams) are those that are supported by the language syntaxes used by designers (especially C++ and Java). The syntactic structure of a program reflects the object model, and the compiler assures consistency in the model. The weakest parts of UML (such as its sequence diagrams, its variant of Statecharts for state diagrams, and its modeling of concurrency) are those with no syntactic support in widely used languages. Few tools are available for ensuring consistency between programs and their models and for validating the models. Perhaps eventually code generation from these models will ameliorate this, although this really amounts to defining new languages with graphical syntaxes, a non-trivial challenge.

Our proposal, to augment type systems with dynamic properties of interfaces, will also require syntactic language support to succeed. However, introducing new languages is extremely risky; many useful and valid concepts in programming have failed to catch on because they were only expressed in entirely new languages. The community is reluctant (understandably) to sacrifice its fluency with existing languages, particularly if the benefits are unproven.

We suggest, however, that system-level types can be introduced without modifying the underlying languages, but rather by overlaying on them design patterns that make these types explicit. Such overlays have sometimes been called coordination languages [6].

## 3.0 FRAMEWORK

We assume a component-oriented software framework such as Ptolemy II [7]. In Ptolemy II, components communicate via method invocations on an object called a receiver. Such an object is an instance of a class that implements an interface called Receiver. A receiver is contained by an instance of the class IOPort. An object model is shown in figure 1.

### 3.1 Receivers

In Ptolemy II, a **channel** of communication between a producer and consumer is implemented by a **receiver**. A receiver implements the Receiver interface. The Receiver interface has six methods. Two of these support the association with the containing IOPort. The other four support communication.

The Receiver interface assumes a producer/consumer model for interaction between components. Communicated data is encapsulated in a class called Token. The put() method is used to deposit a token into a receiver. The get() method is used to extract a token from the receiver. The producer uses the put() method while the consumer uses the get() method.

The hasToken() method, which returns a boolean, indicates whether a call to get() will trigger a NoTokenException. The hasRoom() method indicates whether a call to put() will trigger a NoRoomException. Thus, these two methods can be used to query the receiver for its state. Does it contain a token? Does it have room for another token?

Aside from assuming a producer/consumer model, the Receiver interface makes no further assumptions. It does not, for example, determine whether communication between components is synchronous or asynchronous. Nor does it determine whether tokens that are deposited in a receiver replace ones that were previously deposited. Nor does it determine the capacity of a receiver. These properties of a receiver are determined instead by concrete classes that implement the Receiver interface. Some of these are shown in figure 1.

We concentrate in this paper on a few of these concrete classes, Mailbox (which is then extended by CTReceiver), DEReceiver, CSPReceiver, PNQueueReceiver, and SDFReceiver. These implement five different communication mechanisms between components. In Ptolemy II, we have built several others, but these suffice to illustrate the concepts of this paper.

### 3.2 Domains

In Ptolemy II, a model of computation is implemented by a collection of classes called a *domain*. The concrete receivers shown in figure 1 are parts of Ptolemy II domains; we will focus on the CT, CSP, DE, PN, and SDF domains.

The CTReceiver is a slightly specialized version of the Mailbox receiver. "CT" stands for "continuous time." The CT domain models continuous time systems. The receiver's get() method always delivers the most recent data token deposited by the put() method (if no token has been deposited, then it throws an exception). The hasToken() method, therefore, always returns true after the first token has been deposited. The hasRoom() method also always returns true, since the current token in the receiver can always be overwritten.

CSPReceiver, as the name suggests, implements a rendezvous-style communication (sometimes called synchronous message passing), as in Hoare's communicating sequential processes model [14]. In the Ptolemy II CSP domain, the producer and consumer are separate threads. Whichever thread calls put() or get() first blocks until the other thread calls put() or get(). Data is exchanged as an atomic action when both the producer and consumer are ready. In Ptolemy II, the CSP receiver also supports both conditional send and conditional receive, although we will ignore that feature in this paper, so the support for it is omitted in the UML diagram.

PNQueueReceiver supports the Kahn process networks model of computation [16] using an implementation like that by Kahn and MacQueen [17]. In that model, just like CSP, the producer and consumer are separate threads. Unlike CSP, however, the producer can send data and proceed without waiting for the receiver to be ready to receive the data. This is implemented by a non-blocking write to a FIFO queue with (conceptually) unbounded capacity. The put() method in PNQueueReceiver always succeeds and always returns immediately. The get() method, however, blocks the calling thread if no data is available. To maintain determinacy, it is important that processes not be able to test a receiver for the presence of data, so the hasToken() method always returns *true*. Indeed, this return value is correct, since the hasToken() method will never throw a NoTokenException. Instead, it will block the calling thread until a token is available.

SDFReceiver supports a synchronous dataflow model of computation [19]. This is different from the thread-based domains in that the producer and consumer are implemented as finite computations (firings of a dataflow actor) that are scheduled (typically statically, and typically in the same thread). In this model, a consumer assumes that data is always available when it calls get() because it assumes that it

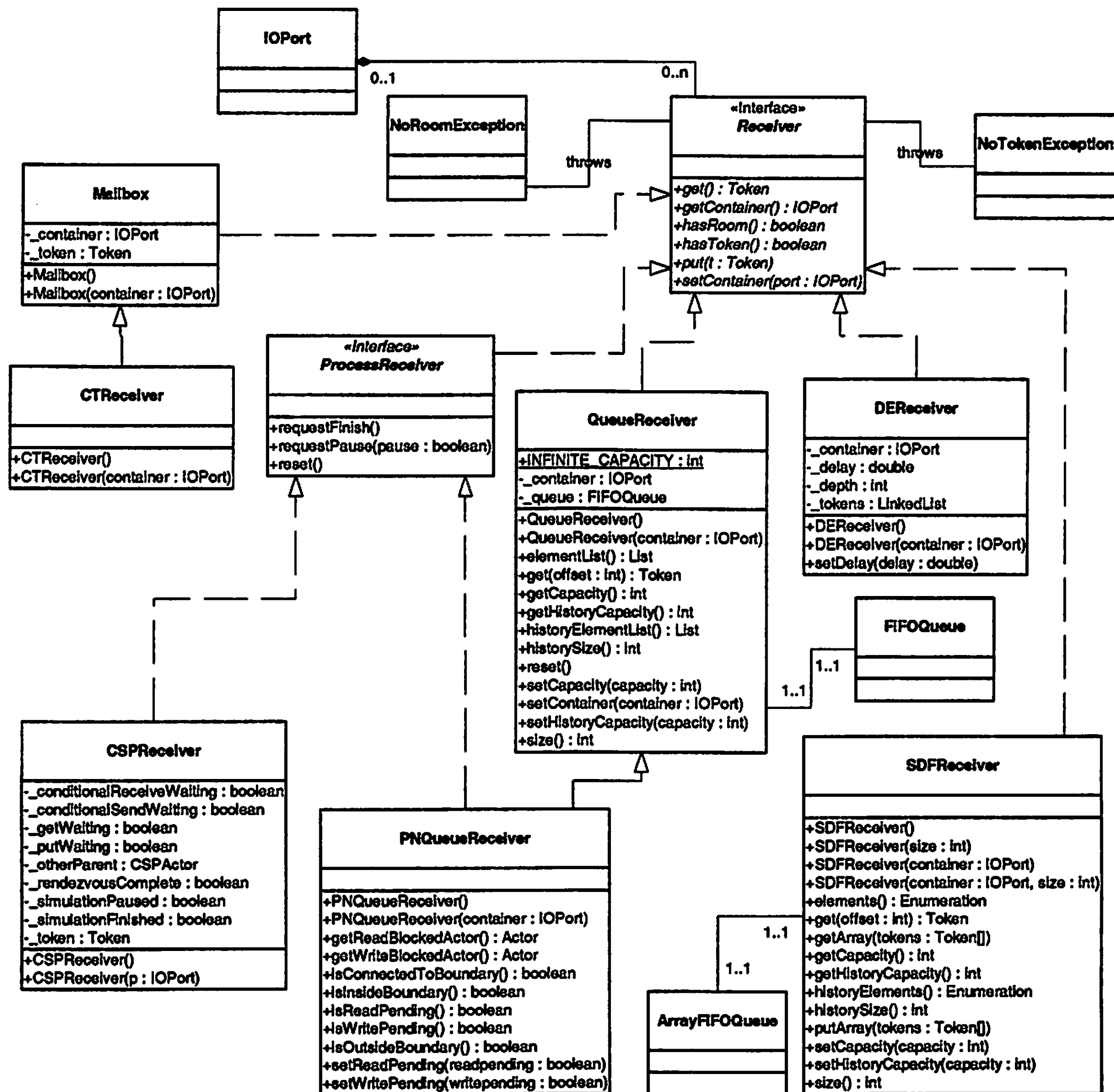


**Figure 1.** UML static structure diagram showing the Ptolemy II Receiver interface and some of the classes that implement that interface.

would not have been scheduled otherwise. The capacity of the receiver can be made finite, statically determined, but the scheduler ensures that when put() is called, there is room for a token. Thus, if scheduling is done correctly, both get() and put() succeed immediately and return.

The DE (discrete event) domain in Ptolemy II uses timed events to communicate between components. Before depositing a token in the receiver by calling put(), a producer might call the setDelay() method to indicate that the event should be made available to the consumer at some time in the future. Thus, an invocation of put() does not necessarily immediately result in a token being available. The scheduler may intervene, storing the token in a priority queue until its time arrives, and then reinvoke put() to deposit the token in the receiver and make it available to the consumer.

### 3.3 Domain Polymorphic Actors

In Ptolemy II, the domain determines what kind of receiver is used, not the actor. Thus, a natural question arises. Can we design actors that can work with a variety of receivers in different domains?

At the level of object modeling, the answer is clearly "yes" since Receiver is a common interface that is implemented by the receivers in all of the domains we are considering. However, the object model only captures some of the information about the interaction between actors. In particular, it does not say anything about the dynamics of the interaction. Thus, for example, just because an actor is written to use only methods defined by the Receiver interface does not mean it will work correctly with a CSPReceiver.

Suppose for example that an actor calls put() on a receiver followed by get() on the same receiver, with both calls occurring within the same thread. This actor will work fine in PN and SDF, but will fail in CSP. In particular, the actor will immediately deadlock, since put() will block the calling thread, which therefore will never reach the corresponding get().

## 4.0 INTERACTION TYPES

Our approach is to augment the type system so that components declare at their interface not just properties that are classically captured by the type system (their object model), but also dynamic properties, which we call **interaction types**. We describe these interaction types first informally (which is the way we originally developed them), and then formally using automata.

### 4.1 Informal Policies

When we first began constructing domain-polymorphic actors, we struggled to define a policy for their interaction that would yield reasonable and comprehensible behavior in all the domains we had built. With experience, we settled on some verbal descriptions of policies like the following:

> Upon firing, test each input channel to see whether it has a token by calling the hasToken() method of the receiver for that channel. If it returns *true*, then read one token from the channel by calling the get() method of the receiver.

This describes an actor behaving as a consumer of data. Such an actor, by the policy, consumes at most one token from each input channel. This policy represents a design choice. In certain circumstances, for example, we may wish for an actor to consume exactly one input token from each input channel. That would represent a different policy, and one that is not compatible with all the domains.

Consider for example a domain-polymorphic multi-channel adder. If it follows the above policy, its behavior is that, when fired, it will examine each input channel, and if it has a token, it will consume one token. It then adds together all the tokens it consumes and produces the sum at the output. In Ptolemy II, such an actor would also be data polymorphic, and thus can operate on any token type that supports addition.

Some domains, such as SDF and PN, will ensure that there is a token on every input channel. Others, such as DE, make no such assurance. Either way, the actor behavior is well defined.

A corresponding policy for a producer needs to be more restrictive in order to get reasonable behavior in all existing domains.[1] In particular, we assume:

> Upon firing, a domain-polymorphic actor will produce exactly one token on each output port.

This policy is necessarily very constraining. It makes it possible for domains such as SDF to construct schedules for the actor. The restrictiveness of this policy suggests that more than one polymorphic policy might be useful. For example, we might wish to define domain-polymorphic actors that are not required to produce an output on every channel, recognizing full well that these actors will not be usable in SDF. Somehow, the actors need to define their interfaces so that we can distinguish actors that will work in SDF from those that will not.

The above verbal descriptions of the policies followed by an actor are, in fact, part of its interface definition. But the verbal description is too informal to be used as part of a formal interface definition. Our objective is to convert the above informal description into a formal type signature. The way we will do that is by defining an automaton for the receiver such that given a receiver that behaves like that automaton, the actor will always behave "correctly." What we mean by "correctly" depends in part on what question we want to answer. We may wish to ensure, for example, that the actor will not throw an exception. Alternatively, we may wish to demonstrate that a composition will not deadlock.

In outline, our approach is as follows. We define a nondeterministic automaton for the receiver that this consumer can use. We define automata for the receivers in each of the domains. If the actor receiver automaton simulates[2] the domain receiver automaton, then the actor is "type compati-

1. It is generally true of polymorphic components that to maximize their utility they need to use the least specific types possible at their inputs and the most specific types possible at their outputs. Thus, the output policy is much more restrictive than the input policy.

2. We mean "simulates" here in the strong technical sense of automata theory.

ble" with the domain. Once these automata are constructed, we can use them to assert certain properties of the receivers, such as that if hasToken() returns *true*, then the next get() will return a token without throwing an exception. Moreover, once these automata are constructed, we can compose them with automata describing the actor and scheduler behaviors to assert properties of the composition.

### 4.2 Domain-Specific Receiver Automata

Consider the automaton in figure 2, which models a receiver in either the SDF or DE domain. Our automata are reactive machines with a single input and output stream. We depict them with bubble-and-arc diagrams where the bubbles represent states and the arcs represent state transitions. The initial state (or states, in case it is nondeterministic) are depicted by bold arrows. In figure 2, there are two states:

*hasToken*: Indicating the presence of one or more tokens available for consumption.

*noToken*: Indicating that there is no token available for consumption.

Each arc represents a state transition and is labeled "guard / output" where the guard is a member of the input alphabet (or a set of members of the input alphabet) and the output is a member of the output alphabet. The input alphabet for all our receiver automata is:

*g*: get
*p*: put
*h*: hasToken

These correspond precisely to the corresponding methods in the Receiver interface[1]. The output alphabet is

0: false
1: true
*t*: token
*v*: void
*e*: exception

These correspond to the return values of the methods of the Receiver interface. The *v* indicates that the method successfully returns, but returns no value ("v" is for "void"). The *e* indicates that the method throws an exception. An exception is thrown when the get() method is called and the automaton is in the *noToken* state.

The automaton in figure 2 is nondeterministic. Either state can be the initial state, reflecting the property of SDF that a receiver can be initialized with an initial token.

In addition, when the get() method is called and the automaton is in the *hasToken* state, the receiver may or may not become empty. Thus, the automaton does not fully reflect the current state of the receiver, in that it does not keep track of the number of tokens buffered by the receiver. This abstraction will serve for our purposes here, although for more detailed analysis, a more detailed automaton may be more convenient.

The automaton in figure 3 models a DE receiver, which is only slightly different. In particular, in DE, calling put() does not necessarily make the token available immediately to the consumer. Thus, there is an extra self-loop on the *noToken* state that permits the state to remain the same despite the call to put(). Notice that this automaton, which we name **DE1**, simulates the one in figure 2, which we name **SDF1**. The converse is not true. **SDF1** does not simulate **DE1**. This is indicated in figure 4, which depicts a partial order of automata determined by the simulation relation. **DE1** is above **SDF1** because it simulates **SDF1**. This ordering is our version of the subtyping relation in classical type systems.

Consider the automaton in figure 5, which models the receiver in the PN domain. Here, we use the shortcut notation of omitting the "/ output" from a transition if the transition produces no output. This somewhat reduces the clutter in our diagrams. So, for example, on the transition from *hasToken* to *stallcsmr*, an input *g* will produce no output.

The PN receiver always returns a token when get() is called (because it implements the Kahn-MacQueen blocking read [17]). Thus, it always returns *true* when hasToken() is called, since get() will always succeed. Thus, its initial state is called *hasToken*, suggesting that the receiver initially behaves as if it has a token.

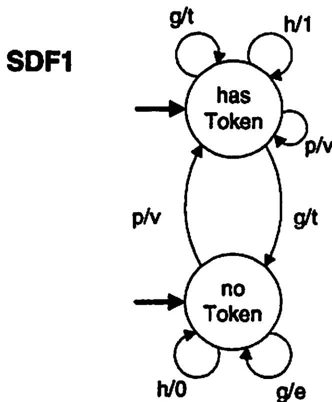


**Figure 2.** Receiver automaton for the SDF receiver.

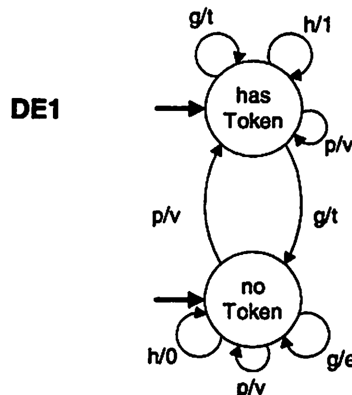


**Figure 3.** Receiver automaton for the DE receiver.

1. We ignore that hasRoom() method because all the domain-specific receivers that we consider in this paper always return *true* when that method is called, indicating that they always can accept a token.

When the receiver is in the *hasToken* state and get() is called, it may immediately return a token, or it may stall the calling the thread. We model the latter as a state transition to a *stallcsmr* state, where no output is produced by the transition. While in the *stallcsmr* state, any call to hasToken() returns *true*, as usual, and any call to get() does nothing (such calls would not normally occur, since the consumer thread is blocked, but our automaton reflects exactly the way the software is written, and it permits these calls to occur from some other thread).

The automaton in figure 5, which we name **PN1**, does not simulate **SDF1** nor **DE1**, nor do those simulate **PN1**. Thus, in the partial order of figure 4, it is shown to be incomparable.

Consider the automaton in figure 6, which is called **CSP1**. This one models the behavior of the receiver in the CSP (communicating sequential processes) domain. In CSP, communication occurs via rendezvous. The receiver starts in the state labeled *noToken*. If get() is called, the calling thread is stalled, as indicated in the automaton by transitioning to the *stallcsmr* state. In that state, if put() is called, then a token is transferred and the receiver returns to the *noToken* state. If the receiver is in the *noToken* state and put() is called, then the calling thread is stalled, and the automaton transitions to *stallpdcr*. It remains in that state until get() is called.

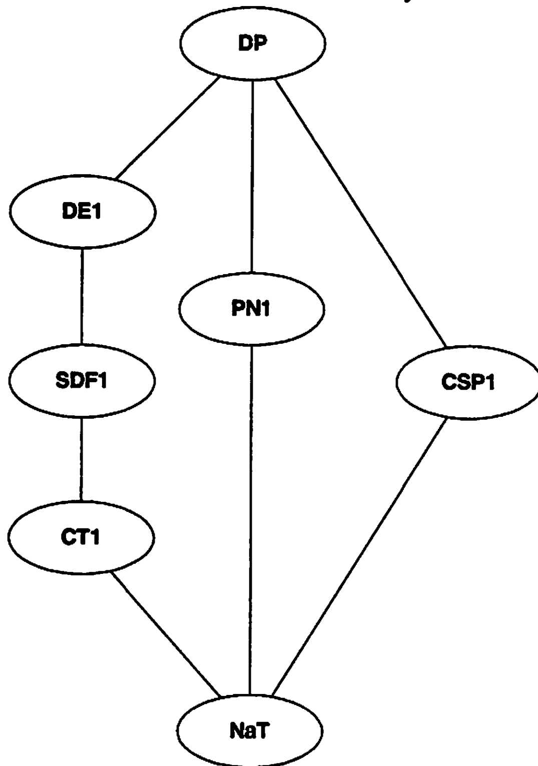

**Figure 4.** Partial order of automata determined by the simulation relation.

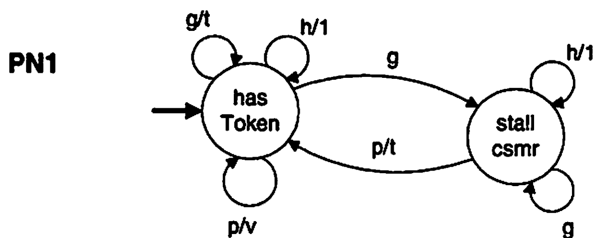

**Figure 5.** Receiver automaton for the PN receiver.

In the Ptolemy II CSP domain, the hasToken() method returns false unless there is a producer that is currently blocked. Thus hasToken() returns false (0) in both *noToken* and *stallcsmr*, and it returns true (1) in *stallpdcr*.

The CT (continuous-time) domain has automaton shown in figure 7. The receiver starts in the *noToken* state, but after the first put(), remains forever in the *hasToken* state. In that state, hasToken() always returns true, and the get() method returns the most recent token deposited with a put(). It is up to the scheduler to ensure that a call to put() occurs before the first call to get(). The **CT1** automaton is simulated by the **SDF1** automaton, as shown in figure 4.

## 4.3 Domain-Polymorphic Receiver Automaton

A domain polymorphic actor that can operate in the CSP, CT, DE, PN, and SDF domains must be able to operate with all the automata of these domains. It is sufficient for it to be able to operate with an automaton that simulates all these. Such an automaton is shown in figure 8. This automaton is fairly complex, but at least one useful property is evident. If hasToken() return true (1), then the next call to get() will not throw an exception. To assert more properties, it becomes necessary to couple this automaton, or one of the domain-

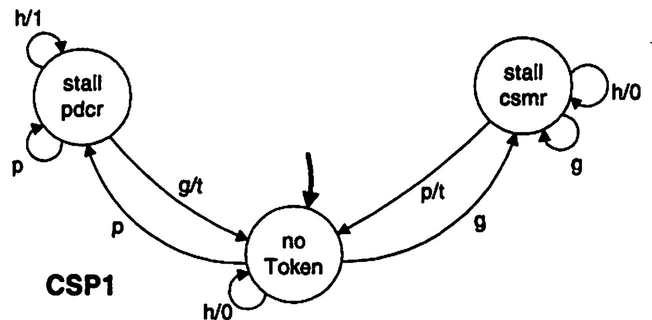

**Figure 6.** Receiver automaton for the CSP receiver.

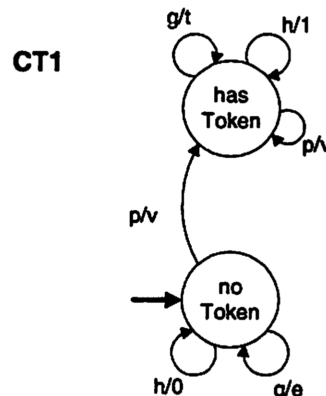

**Figure 7.** Receiver automaton for the SDF receiver.

specific ones, with automata that describe the actors and the domain schedulers.

## 5.0 TYPE RESOLUTION

### 5.1 Binding

In a Ptolemy II topology, each port has a type. For an input port (the port of a consumer), the type is the receiver automaton for the receiver contained in that port; for an output port (a producer), the type is the automaton for the receiver in the connected input port. For example, in figure 9, *t1* and *t2* are both the automaton for the receiver of the input port of *A2*. Since the type of a port may be assigned by the type resolution mechanism, it is also called the *resolved type*. Corresponding to this, each port also has a *declared type*. The default value for the declared type is the domain-polymorphic (DP) automaton. An actor writer can either specify the declared type, or use the default value.

In Ptolemy II, there is only one receiver per connection, so there can be only one type per connection. Thus, in figure figure 9, *t1* = *t2* and *t3* = *t4*.

A type constraint between the declared type and the resolved type of a port is that the resolved type must be less than or equal to the declared type:

$$t_{\text{resolved}} \le t_{\text{declared}} \qquad (1)$$

The ordering is that of figure 4, given by the simulation relation. The resolved type must be an automaton that the declared type simulates.

Another type constraint imposed in Ptolemy II is that all the ports on an actor must have the same type. For example, we require *t2* = *t3* in figure 9. This constraint is due to the philosophy of Ptolemy II that heterogeneity is achieved only through hierarchically mixing different models of computation, not through mixing different communication protocols on a flat hierarchy. Models of computation are mixed by encapsulating a subsystem in an **opaque composite actor,** which is a composite actor that contains a **director**. The director together with the receivers implements the model of computation. It is responsible for scheduling, for example.

Figure 9. A topology with types

Under this formulation, type constraints can be set up and solved. For example, if *t1* = *t4* = **PN1**, and *t2* and *t3* have the default declared type, then *t2* and *t3* will be resolved to **PN1**. Thus, their polymorphism is resolved statically by the type resolution mechanism.

### 5.2 Type Coercion

The partial order of figure 4 is augmented with a bottom element, labeled **NaT** (not a type) to become a lattice. Type constraints are given as inequalities and equalities between types in the lattice. The type system infrastructure of Ptolemy II [34] finds the least type in the lattice that satisfies all the constraints. If the solution is NaT for any port, then a type conflict has been detected. Type conflicts can be corrected by coercion.

Type conflicts indicates the need to mix different domains. In the above example, if *t2* = *t3* = **CSP1**, then *A2* should be wrapped in an opaque composite actor with a CSP director inside. Opaque composite actors in Ptolemy II provide the hierarchical domain mixing that is a signature Ptolemy trait. The ports at the boundary of the composite actor act as type converters (protocol converters) between PN and CSP.

It seems appealing to seek algorithms for automating the type coercion. However, this may not be possible. For one thing, the way the actors are wrapped may not be unique. For example, in figure 10, multiple ways of resolving the type of *A2* and assigning the outer and inner domains are possible, including:

- Resolve *A2* to **PN1**, and wrap *A4* in a composite actor;
- Resolve *A2* to **CSP1**, and wrap *A2* and *A4* in a composite actor;
- Resolve *A2* to **PN1**, and wrap *A1*, *A2*, and *A3* in a composite actor;
- etc.

It seems possible to make this decision automatically using

Figure 8. Receiver automaton for domain-polymorphic actors.

some criteria, such as minimum interconnection between the inner and outer domains. However, it probably makes more sense to require user intervention since the user has more knowledge on the functional partition of the system and can make more meaningful decisions. This is consistent with type systems in modern strongly typed languages, where automated coercion is largely avoided.

### 5.3 Type conversion

In Ptolemy II, a model in one domain may be wrapped in an opaque composite actor that is then used as an atomic actor in another domain. Our formulation gives a simple way to construct such a wrapping. The opaque composite actor simply needs to expose at its interface the behavior of a domain-polymorphic actor. Thus, its external ports should have type **DP**. Since **DP** simulates all receivers in domains considered here, such a composite actor can be used with predictable behavior in all such domains.

## 6.0 FURTHER WORK

The mechanisms we have described are applicable primarily to compile-time analysis of embedded software and to design-time analysis of hardware. These concepts, however, seem applicable in a run-time context, using concepts similar to run-time type system support in existing languages.

### 6.1 On-Line Type System

Embedded software is traditionally highly static. No mechanism exists in many embedded systems for modifying in any way the executing software. However, this is changing. At a minimum, software upgrades are becoming essential as the complexity of the applications increases. Even more challenging, however, are networked embedded systems, where migrating software components (agents, applets, etc.) considerably complicate the run-time environment.

One approach to supporting such adaptive software is to view the embedded software as having a dynamically changing software architecture. This view reconciles the wholistic view of the system that is necessary to achieve the levels of assurance required for embedded software with the desire to modify the software on-the-fly.

Static support for type systems give the compiler responsibility for the robustness of software [5]. This is not adequate if the software architecture is dynamic. The software needs to take responsibility for its own robustness [18]. This means that algorithms that support the type system need to be adapted to be practically executable at run time.

ML is an early and well known realization of a "modern type system" [12][30][32]. It was the first language to use type inference in an integrated way [15], where the types of variables are not declared, but are rather inferred from how they are used. The compile-time algorithms here are elegant, but it is not clear to me whether run-time adaptations are practical.

Many modern languages, including Java and C++, use declared types rather than type inference, but their extensive use of polymorphism still implies a need for fairly sophisticated type checking and type resolution. Type resolution allows for automatic (lossless) type conversions and for optimized run-time code, where the overhead of late binding can be avoided.

Type inference and type checking can be reformulated as the problem of finding the fixed point of a monotonic function on a lattice, an approach due to Dana Scott [28]. The lattice describes a partial order of types, where the ordering relationship is the subtype relation. For example, Double is a subtype of Number in Java. A typical implementation reformulates the fixed point problem as the solution of a system of inequalities [23]. Reasonably efficient algorithms have been identified for solving these systems of inequalities [27], although these algorithms are still primarily viewed as part of a compiler, and not part of a run-time system.

Iteration to a fixed point, at first glance, seems too costly for on-line real-time computation. However, there are several languages based on such iteration that are used primarily in a real-time context. Esterel is a notable one of these [4]. Esterel compilers synthesize run-time algorithms that converge to a fixed point at each clock of a synchronous system [3]. Such synthesis requires detailed static information about the structure of the application, but methods have been demonstrated that use less static information [8]. Although these techniques have not been proposed primarily in the context of a type system, we believe they can be adapted.

### 6.2 Reflecting Program Dynamics

Reflection, as applied in software, can be viewed as having an on-line model of the software within the software itself. In Java for example, this is applied in a simple way. The static structure of objects is visible through the Class class and the classes in the reflection package, which includes Method, Constructor, and various others. These classes allow Java code to dynamically query objects for their methods, determine on-the-fly the arguments of the methods, and construct calls to those methods. Reflection is

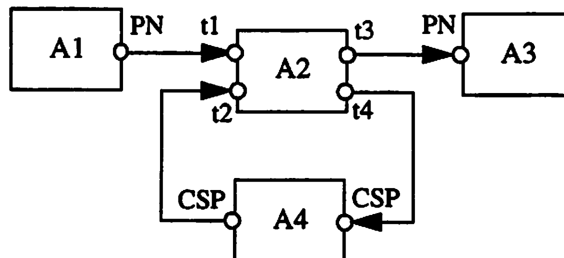


Figure 10. A topology with non-unique type resolution

an integral part of Java Beans, mobile code, and CORBA support. It provides a run-time environment with the facilities for stitching together components with relatively intolerant interfaces.

However, static structure is not enough. The interfaces between components involve more than method templates, including such properties as communication protocols. To get adaptive software in the context of real-time applications, it will also be important to reflect program state. Thus, we need reflection on the program dynamics.

The first question becomes at what granularity to do this. Reflection intrinsically refers to a particular abstracted representation of a program. E.g., in the case of static structure, Java's reflection package does not include finer granularity than methods, nor coarser granularity than objects.

Process-level reflection could include two critical facets, communication protocols and process state. The former would capture in a type system such properties as whether the process uses rendezvous, streams, or events to communication with other processes. By contrast, Java Beans defines this property universally to all applications using Java Beans. That is, the event model is the only interaction mechanism available. If a component needs rendezvous, it must implement that on top of events, and the type system provides no mechanism for the component to assert that it needs rendezvous. For this reason, Java Beans seem unlikely to be very useful in applications that need stronger synchronization between processes, and thus it is unlikely to be used much beyond user interface design.

Reflecting process state could be done with an automaton that simulates the program. That is, a component or its run-time environment can access the "state" of a process (much as an object accesses its own static structure in Java), but that state is not the detailed state of the process, but rather the state of a carefully chosen automaton that simulates the application. Designing that automaton is then similar (conceptually) to designing the static structure of an object-oriented program, but represents dynamics instead of static structure.

Just as we have object-oriented languages to help us develop object oriented programs, we would need state-oriented languages to help us develop the reflection automaton. These could be based on Statecharts, but would be closer in spirit to UML's state diagrams in that it would not be intended to capture all aspects of behavior. This is analogous to the object model of a program, which does not capture all aspects of the program structure (associations between objects are only weakly described in UML's static structure diagrams). Analogous to object-oriented languages, which are primarily syntactic overlays on imperative languages, a state-oriented language would be a syntactic overlay on an object-oriented language. The syntax could be graphical, as is now becoming popular with object models (especially UML).

Well-chosen reflection automata would add value in a number of ways. First, an application may be asked, via the network, or based on sensor data, to make some change in its functionality. How can it tell whether that change is safe? The change may be safe when it is in certain states, and not safe in other states. It would query its reflection automaton, or the reflection automaton of some gatekeeper object, to determine how to react. This could be particularly important in real-time applications. Second, reflection automata could provide a basis for verification via such techniques as model checking.

This complements what object-oriented languages offer. Their object model indicates safety of a change with respect to data layout. But they provide no mechanism for determining safety based on the state of the program.

When a reflection automaton is combined with concurrency, we get something akin to Statechart's concurrent, hierarchical FSMs, but with a twist. In Statecharts, the concurrency model is fixed. Here, any concurrency model can be used. We call this generalization "*charts," pronounced "starcharts", where the star represents a wildcard suggesting the flexibility in concurrency models [11]. Some variations of Statecharts support concurrency using models that are different from those in the original Statecharts [21][31]. As with Statecharts, concurrent composition of reflection automata provides the benefit of compact representation of a product automaton that potentially has a very large number of states. In this sense, aggregates of components remain components where the reflection automaton of the aggregate is the product automaton of the components. But the product automaton never needs to be explicitly represented.

Ideally, reflection automata would also inherit cleanly. For example, a component that derives from another inherits its automaton and refines the states of the automaton (similar to the hierarchy, or "or" states in Statecharts).

In addition to application components being reflective, it will probably be beneficial for components in the run-time environment to be reflective. The run-time environment is whatever portion of the system outlives all application components. It provides such services as process scheduling, storage management, and specialization of components for efficient execution. Because it outlives all application components, it provides a convenient place to reflect aspects of the application that transcend a single component or aggregate of closely related components.

## 7.0 REFERENCES


[1] G. A. Agha, "Concurrent Object-Oriented Programming," *Communications of the ACM*, 33(9), pp. 125-141, 1990.

[2] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[3] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language," *IEEE Trans. on Automatic Control*, vol. 35, no. 5, pp. 525-546, May 1990.

[4] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87-152, 1992.

[5] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, Vol. 17, No. 4, pp. 471-522, 1985.

[6] P. Ciancarini, "Coordination models and languages as software integrators," *ACM Computing Surveys*, Vol. 28, No. 2 (June 1996), Pages 300-302.

[7] J. Davis, R. Galicia, M. Goel, C. Hylands, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay and Y. Xiong, "Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java," Technical Report UCB/ERL No. M99/40, University of California, Berkeley, CA 94720, July 19, 1999. (http://ptolemy.eecs.berkeley.edu/papers/99/HMAD)

[8] S. A. Edwards, "The Specification and Execution of Heterogeneous Synchronous Reactive Systems," Ph.D. thesis, University of California, Berkeley, May 1997. Available as UCB/ERL M97/31. http://ptolemy.eecs.berkeley.edu/papers/97/sedwardsThesis/

[9] H.-E. Eriksson and M. Penker, *UML Toolkit*, Wiley, 1998.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

[11] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, Vol. 18, No. 6, June 1999.

[12] M. J. Gordon, R. Milner, L. Morris, M. Newey and C. P. Wadsworth, "A Metalanguage for Interactive Proof in LCF," *Conf. Record of the 5th Annual ACM Symp. on Principles of Programming Languages*, ACM, pp. 119-130, 1978.

[13] S. Hayashi, "Singleton, Union, and Intersection Types for Program Extraction," in A. R. Meyer (ed.), *Proc. of the Int. Conf. on Theoretical Aspects of Computer Science*, pp. 701-730, 1991.

[14] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978.

[15] P. Hudak, "Conception, Evolution, and Application of Functional Programming Languages," *ACM Computing Surveys*, Vol. 21, No. 3, September 1989.

[16] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," Proc. of the IFIP Congress 74, North-Holland Publishing Co., 1974.

[17] G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing 77*, B. Gilchrist, editor, North-Holland Publishing Co., 1977.

[18] R. Laddaga, "Active Software," position paper for the *St. Thomas Workshop on Software Behavior Description*, December, 1998.

[19] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proc. of the IEEE*, September, 1987.

[20] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems," in *Proc. 15-th ACM Symp. on Principles of Programming Languages*, pp. 47-57, 1988.

[21] F. Maraninchi, "The Argos Language: Graphical Representation of Automata and Description of Reactive Systems," in *Proc. IEEE Workshop on Visual Languages*, Kobe, Japan, Oct. 1991.

[22] P. Martin-Löf, "Constructive Mathematics and Computer Programming," in *Logic, Methodology, and Philosophy of Science VI*, pp. 153-175, North-Holland, 1980.

[23] R. Milner, *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences 17, pp. 384-375, 1978.

[24] G. Necula, "Proof-Carrying Code," in *Conf. Record of the 24th Annual ACM Symp. on Principles of Programming Languages*, pp. 106-119, ACM Press, 1997.

[25] J. K. Ousterhout, "Scripting: Higher Level Programming for the 21 Century," *IEEE Computer*, March 1998.

[26] R. Passerone, J. A. Rowson, and A. L. Sangiovanni-Vincentelli, "Automatic Synthesis of Interfaces between Incompatible Protocols," *Proceedings of the 35th Design Automation Conference*, 1998.

[27] J. Rehof and T. Mogensen, "Tractable Constraints in Finite Semilattices," *Third International Static Analysis Symposium*, pp. 285-301, Volume 1145 of Lecture Notes in Computer Science, Springer, Sept., 1996.

[28] D. Scott, "Outline of a mathematical theory of computation", *Proc. of the 4th annual Princeton conf. on Information sciences and systems*, 1970, 169-176.

[29] W. T. Trotter, *Combinatorics and Partially Ordered Sets*, Johns Hopkins University Press, Baltimore, Maryland, 1992.

[30] J. D. Ullman, *Elements of ML Programming*, Prentice-Hall, 1994.

[31] M. von der Beeck, "A Comparison of Statecharts Variants," in *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems, LNCS* 863, pp. 128-148, Sprinter-Verlag, Berlin, 1994.

[32] Å. Wikstrom, *Standard ML*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[33] H. Xi and F. Pfenning, "Eliminating Array Bound Checking Through Dependent Types," In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, pp. 249-257, Montreal, June 1998.

[34] Y. Xiong and E. A. Lee, "An Extensible Type System for Component-Based Design," *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS-2000, Berlin, Germany, 2000.