Automatic Specialization of Actor-oriented Models in Ptolemy II

by Stephen Neuendorffer

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Edward Lee

Research Advisor

* * * * *

Kurt Keutzer

Second Reader

Published as:

Technical Memorandum UCB ERL M02/41,

Electronics Research Laboratory,

University of California at Berkeley,

December 25, 2002.

1 Introduction

In the design of embedded systems, profitability and time-to-market are critical concerns. Since complete implementation can be expensive and time-consuming, making good high-level decisions early in the design process is crucial. These high-level decisions can then be used to guide system implementation, and the designer will have a reasonable expectation that the system will perform as expected. Many high-level, domain-specific tools have been built to help designers to more quickly arrive at a successful design. An important subset of these tools, including commercial embedded systems tools such as Simulink from The Mathworks, VCC from Cadence, and CoCentric System Studio from Synopsys, are often known as *actor-oriented* [32]. These tools emphasize the concurrent operation of the different parts of a design, making them a good match for designing embedded systems that interact concurrently with the physical world.

When using actor-oriented tools, designers typically construct models that are intended to capture some aspects of an embedded system. These models are constructed through the composition of primitive components, which may be drawn from an existing library or specified directly in the model. The behavior of a model corresponds, in some fashion, with the behavior of the embedded system being modeled. Such models are commonly used to formally analyze or simulate the behavior of a system prior to actually building the system. In fact, sufficiently detailed system models can be considered specifications of those systems [45]. These "golden models" are often used as a reference for testing the correctness of a manual system implementation in another language, such as C or Verilog.

Unfortunately, manual system implementation tends to be an error-prone process and it can be difficult to ensure that the behavior of an actor-oriented model is preserved when faced with many low-level implementation details. One solution to this problem is to provide facilities for automatic *code generation* from within an actor-oriented design tool [9, 10]. Code generation provides a correct-by-construction path from system modeling to both optimized simulation and embedded system implementation. The resulting im-

plementation may include both software portions (e.g., a program that will execute on a microprocessor) and hardware portions (e.g., a synthesized ASIC circuit, or an FPGA configuration).

A key difficulty with code generation is that embedded systems are often extremely resource constrained. An automatically generated implementation must efficiently use resources (memory, processor time, chip area, etc.) to be useful in a final system. Because of this efficiency requirement, a significant amount of research into optimizing generated code has been performed. There has been work in optimal execution scheduling [5, 6, 41, 17, 42], communication and control architectures for efficient software [43, 16] and hardware [51, 49], and design space exploration techniques [30, 40, 46]. This work is based in the formal definitions of components and the semantics of models of computation.

Unfortunately, the generation of optimized code from individual primitive components is rarely considered in the code generation literature. From a designers point of view, well-designed libraries of primitive components are often the most critical part of a design tool.[1] Such libraries depend on the generality of components, so that those components can be reused and reconfigured in different models. This report presents a series of techniques for automatic specialization of generic component specifications. These techniques allow the transformation of a generic component specifications into more compact and efficient ones.

We have integrated these techniques into a code generator for Ptolemy II, a software framework for actor-oriented design in Java [15]. Combining automatic code generation with actor specialization enables efficient implementation of models without sacrificing design flexibility. We call this approach to implementing an actor oriented model *co-compilation*, to emphasize how it integrates compiler optimization with automatic code generation. It is conceptually similar to concepts of Aspect-Oriented Design [28], since the style of component interaction is incorporated into the specification of a component.

[1] Kurt Keutzer often cites anecdotal evidence from his time as CTO of Synopsys. He found that users often compared system-level design tools solely based on the libraries that were available without considering the tools' broader technical merits.

It also resembles the application of partial evaluation in functional languages to improve execution speed [24].

The following section provides a summary of the relevant aspects of actor-oriented modeling, and the code generation problem. Sections 2 and 3 describe actor-oriented design in more detail, along with a short description of its implementation of Ptolemy II. Section 4 describes the specialization of Java actor specification, while section 5 describes how actor specialization is used to synthesize software system implementations.

2 Actor-oriented System Modeling

Actor-oriented modeling [32] is a methodology that is particularly effective for system-level design. *Actors* execute and communicate concurrently with other actors in a *model*. The ordering of actor execution and the style of actor communication is determined by a *model of computation* (MoC). Many different models of computation are possible. For instance, in a *dataflow* or *process network* [25, 26, 41] model, actors communicate by queues and execute concurrently whenever their input queues have sufficient data available. Alternatively, in a *discrete event* model, actors communicate by events in time and only the actor with the earliest event is enabled to execute.

Actors have a well defined interface. This interface abstracts the internal state and execution of an actor, and restricts how an actor interacts with its environment. The interface includes *ports* that represent points of communication for an actor and *parameters* which are used to configure the operation of an actor, increasing the contexts in which it can be used. Generally parameter values are part of the *a priori* configuration of an actor and do not change when a model is executed. The configuration of a model also contains explicit communication *channels* that pass data from one port to another.[2]

This “port and parameter” syntax has several nice properties:

[2]This terminology is inspired by [21, 2, 1], although the specifics are somewhat different.

- The interface to a component, i.e. its ports and parameters, is an explicit part of the component.
- It is easy to add external ports and parameters to a model, allowing models to be composed hierarchically with other actors and models. Such a model is externally indistinguishable from any other actor.
- The specification of an actor is independent of the model in which the actor is used, making it easy to reuse an actor specification in other contexts.

Individual models of computation govern the interaction between components in widely different ways. Each of these semantics is associated with a set of properties that are useful for the modeling or design of a certain class of designs [34]. These properties are generally recognized, often because they have been established through formal mathematical reasoning about the model of computation. For example, some models of computation expose a wide range of concurrency in the system, allowing for models to be executed in a distributed environment [25, 22, 12]. Other models of computation are more sequential, making it more difficult to distribute a model, but allowing efficient sequential execution [33, 4, 20]. Similarly, some models of computation are easily analyzed at design time, allowing static scheduling [7, 8], while others allow more dynamic component behavior and runtime reconfiguration [35, 44]. Because of these properties, models of computation [32] can be considered as *patterns of component interaction*, in the sense of Gamma *et. al.* [19].

2.1 Specifying Actor Behavior

The *behavior* of an actor is defined to be a possibly infinite set of operations. For the purposes of this paper, the only thing that we need to say about these operations is that they can represent the production and consumption of data from the ports of the actor, the computation of new data, and modification of the internal state of the actor. The behavior of an actor is primarily determined by an *actor specification*. The execution of this specification,

in the context of other actors and data from the environment results in an actor's observed behavior.

Actor specifications can be given in a variety of ways. For instance, actors are often specified by drawing finite-state machines where each transition corresponds to a particular sequence of operations [38]. An actor can also be specified by composing yet other actors in a *hierarchical model*. Yet another technique is to use a special purpose textual language that specifies what tokens to consume and what operations to compute on that data, such as CAL [50]. However, one of the most flexible ways to specify actor behavior is to embed the specification within a traditional programming language, such as Java or C, and use special purpose programming interfaces for specifying ports and sending and receiving data. This technique is very attractive, and has been widely used [40, 10, 18] since it allows for existing code to be integrated into an actor-oriented design tool and for programmers to quickly start using actor-oriented methodologies.

2.2 Generality of actors

We consider actors to be *abstract system components* in the following senses:

- They have a well-defined interface, given by ports and parameters, through which they can interact with the rest of the model. Other interactions are not allowed.
- They are generalizable across different uses of the same component. For instance, a component may be given new parameter values or may be connected to a different set of channels.
- They may have vastly different levels of granularity. For instance, an actor may represent algorithms ranging from the boolean OR of two bits to the boolean satisfiability solution of an arbitrary logic formula.
- Larger components may be assembled from smaller components. Component com-

position is the primary mechanism by which complex behavior is constructed.

- They are executable specifications of behavior. (i.e. Actor models have an operational semantics)

Of these, the generality of actors is something that often varies from one actor to the next. For instance, some actors are designed to be very specialized to a particular purpose, such as an actor that represents the controller for a particular model of brushless motor. Other actors are more general, such as an actor that has parameters that can be set to control different models of brushless motors. In the second case, such a parameterized controller enables *component reuse* of the controller in another model.

Component generality also allows for the possibility of *dynamic reconfiguration* during execution. For example, an actor that is not specialized to a connection with another actor has the potential to be dynamically reconnected to a different actor with no modification. Similarly, an actor that is not specialized to particular parameter values has the potential to be dynamically reconfigured with new values. Generally speaking, the use of generalized components increases the available design possibilities when using components.

Imperative programming languages, such as C or Pascal, primarily offer generality through the generality of function arguments. Any set of actual arguments that are compatible with the formal arguments of a function may be passed to that function. Furthermore, a function may be called from many places in a program using different arguments in each case. We call such generality *data polymorphism*. Object-oriented languages, such as C++ and Java, add the notion of *type polymorphism*, where the types of actual arguments may not be identical to the types of formal parameters. We identify several types of generality in actor-oriented models, all of which are exhibited in Ptolemy II:

- data polymorphism: An actor may be given data with different values.
- type polymorphism: An actor may be given data of different types.
- parameter generality: An actor may be given a different set of parameter values.

- connection generality: An actor may be connected to a different set of actors.
- domain polymorphism: An actor may interact with other actors according to different models of computation.

Unfortunately, extremely general components do have significant drawbacks. One drawback that can seriously limit the reuse of existing components is increased design complexity. A parameterizable component can be more difficult to use than an appropriately specialized one, since it may be easier for a designer to develop a new component than to determine the proper application of an existing one. A second drawback comes in the form of execution efficiency: generality is often gained through increasing abstraction. However, increased abstraction can have significant negative impacts on resource usage, e.g., increased processor cycles and memory usage for software systems or greater chip area for hardware systems. We find that managing the tradeoff between component generality and speciality becomes critical, especially for resource constrained systems.

2.3 Composition of Actors

As mentioned previously, actors with sophisticated behaviors are often specified by composing simple actors in a model. Since this model is itself an actor, it can be further composed to create an arbitrarily large hierarchical model. We define the *context* of an actor in a model to include the model of computation used for composition, as well as parameter values, port connections, and types of ports and parameters. Although actors may be specified in a general way, actors acquire specific context when composed in a model. This context may change if a model is further composed hierarchically with further actors or models. We say that an actor or model that can be further composed is an *open composition*. Conversely, a *closed composition* cannot be composed further. In a closed composition, it is often useful to distinguish the model that contains all other actors as the *toplevel* model of the hierarchical model.

Note that even in a closed composition, the context of actors may change dynamically through dynamic reconfiguration. For instance, ports may be reconnected and parameters may be assigned new expressions and values. Generally speaking, such modifications can be initiated either through external user interaction or internally by the model. External modifications are generally able to modify the model in entirely unpredictable ways. Internally triggered modification, on the other hand, can often be predicted since models often include descriptions of the modification to be performed.

3 Ptolemy II Summary

Ptolemy II [15] is a modeling and design tool written in Java that incorporates the concepts of actor-oriented modeling introduced in the previous section. This section describes some of the details of Ptolemy II necessary for understanding the process of actor specialization. The rest of this paper will exclusively consider actors and models built using Ptolemy II.

3.1 Ptolemy II models

The syntax of a Ptolemy II model consists of actors (represented by the `TypedAtomicActor` base class), communication ports (represented by the `TypedIOPort` class), and relations (represented by the `TypedIORelation` class) that mediate the channels between ports. Links between a port and a relation create the communication channels between ports. Actor models (represented by the `TypedCompositeActor` class) contain actors, ports, relations and the links between the ports and relations. Each of these Ptolemy II objects can also have arbitrary attributes (represented by the `Attribute` base class). The most interesting type of attribute is a parameter, which has a simple string expression that can be evaluated.

This syntax is commonly represented visually as in Figure 1. Ports are visible on the

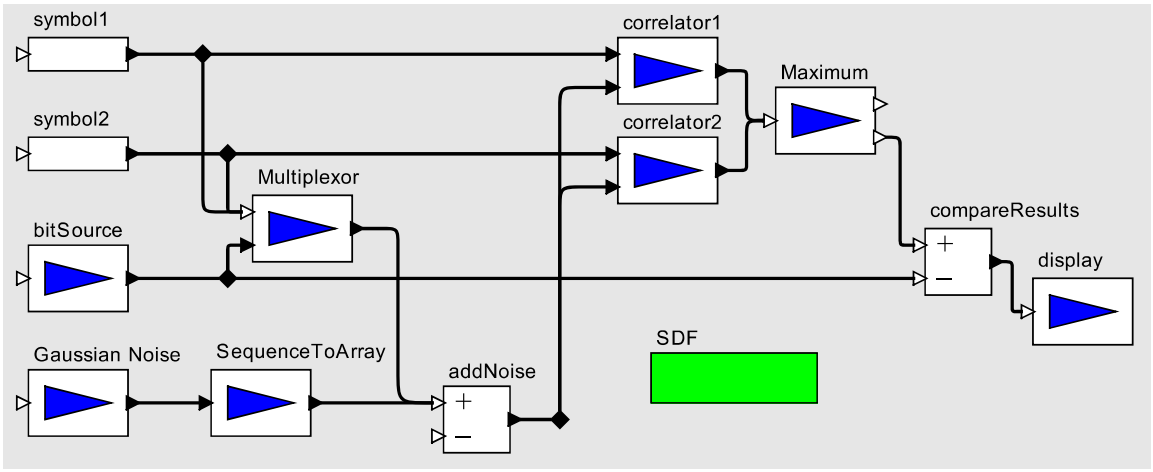

Figure 1: A model of a simple digital communication system, adapted from Jeff Tsay.[48]

boundaries of boxes representing actor interfaces and are connected to other ports by relations. Parameter values are usually not shown. The model of computation is specified by the box marked "SDF." In most cases, this representation of a model is sufficient, since it concentrates on the connections between actor interfaces. However, in some cases, it is useful to represent not only the connections between actors, but also the specifications of actor behavior. The specification of an actor's behavior is given by drawing an arrow from the actor interface to the specification, as in Figure 2. This figure shows the interface to the model, as well as a partial Java specification of the DotProduct actor. The format of this Java code will be explained in the following section.

In a Ptolemy II model, data communicated between actors is represented by instances of the `ptolemy.data.Token` base class, or simply *tokens*. The values of parameters are also represented by tokens, which can encapsulate any immutable value. Values with different data types are represented by different subclasses of the `Token` base class. We often refer to these subclasses as *token classes*. Most token classes, such as `ptolemy.data.IntToken` represent tokens with *numeric* types and serve simply as object wrappers for native Java types. For instance, an IntToken might represent the number 7. Other token class represent more complex numeric types, such as the the `ptolemy.data.DoubleMatrix` class, which could represent a square identity ma-

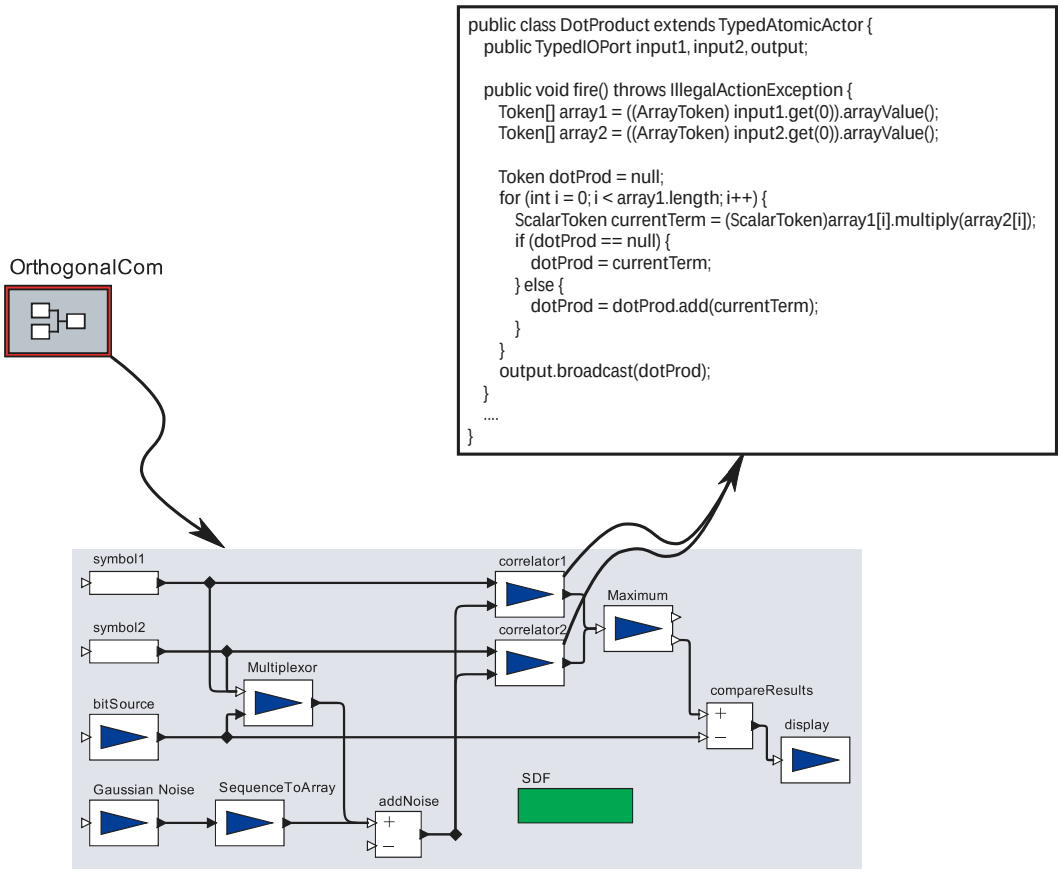

Figure 2: The previous example model, explicitly showing actor specifications.

trix of size 4. More unusual are token classes, such as `ptolemy.data.ArrayToken`, which represent tokens with *structured* types. Structured tokens aggregate other tokens into composite data structures.

Ptolemy II also includes a sophisticated type-inference mechanism that allows actors to be type polymorphic [52]. This mechanism infers the types of the ports of an actor based on constraints declared by actors and constraints implied by connections between ports. Automatic type conversions are performed when data is communicated between ports with different types. This type inference system emphasizes *exact* solutions the type constraints. That is, the inferred type of a port or parameter is, in most cases, exactly the type of the data that will be carried by that port or parameter. In cases where exact types cannot be determined, an *indeterminate type*, such as `general` is inferred. In such cases, a model will still execute as long there are no contradictory type constraints, but automatic type conversion will not be performed for the port or parameter with the indeterminate type.

As mentioned previously, model parameters are associated with string expression. These expressions operate on tokens and, when evaluated, result in a single token value. Additionally, expressions may reference identifiers, such as the name of another parameter in scope or the name of a globally defined constant. When the value referred to by the identifier has been determined, the identifier has been *bound*. The scoping rules allow access to the value of any parameter contained by same actor or any container of the actor. These scoping rules allow a designer to selectively hide most actor parameters deep in the hierarchy, and only expose a small number of model parameters at a high-level. An identifier is only bound to the value of a global constant if it cannot be bound to any parameter in the model, which allows new global constants to be added without shadowing the parameters of existing models. Expressions are evaluated by parsing the string into a parse tree and traversing the parse tree from the bottom-up computing intermediate results. The final value is cached to avoid expensive parse tree traversal.

3.2 Actor Specification in Java

Actors are commonly specified using a stylized form of a Java class. The class implements the `Executable` interface, and each actor with the same specification is represented by an independant instance of the class. The methods of the `Executable` interface break the overall set of operations of an actor into a sequence of *firings*. The `preinitialize()` and `initialize()` methods are generally executed once before the actor does any computation. The `prefire()`, `fire()`, and `postfire()` methods are executed to form a single firing. The `prefire()` method allows a precondition check to ensure that the actor can, in fact, be fired. In the models of computation considered in this report, the `fire()` and `postfire()` methods are invoked immediately after one another after the `prefire()` method returns true. The `wrapup()` and `terminate()` methods are called after computation is finished. For the purposes of this paper, we are primarily concerned with the fact that each of these methods corresponds to an independant set of operations specified using Java code. For more information about the operational semantics of this actor model, see [37].

Within each method, invocations of the `TypedIOPort` class methods `get()`, `send()`, and `broadcast()` correspond to operations that send and receive data. Similarly, invocations of the `Parameter` class methods `getToken()` and `getExpression()` query the values of parameters. The persistent state of the actor is stored in fields of the Java class.

Ptolemy II also defines other methods that are used to notify an actor that certain properties have changed. The `attributeChanged()` method of an actor is invoked when the value of an parameter of the actor has changed. Similarly, the `attributeTypeChanged()` method is called when the type of a parameter has changed. These methods are not part of the normal flow of execution control, but may contain code that is an important part of an actor specification.

3.3 Other Actor Specifications

We will consider several other actor specifications, which are useful in particular circumstances: token expressions, finite-state machines, and modal models. Token expressions are useful for specifying actors that compute stateless functions of several inputs. The expression is evaluated in the same way that parameter values are evaluated, except that identifiers in the expression may be bound to the last values received on the actor's input ports. The value of the expression is sent to the actor's single output port whenever the actor is fired.

Finite state machine specifications [38] are useful for expressing actors with state. Whenever the guard expression of a transition leaving the current state evaluates to true, the transition is executed. The execution of a transition results in the evaluation of the transition's output expressions, and the results are sent to the corresponding output port. Additionally, the state update expressions of the transition are evaluated, and the corresponding state variables updated. Lastly, the current state is updated to be the target state of the transition.

Modal models [38], however, are somewhat more interesting. In a modal model, each state of a finite-state machine is additionally associated with a *refinement* actor. In addition to checking for transitions, each firing of the modal model results in a firing of the refinement associated with the current state. Additionally, when a transition of a modal model is taken, the transition may update the parameter values of the refinement associated with the destination state. The possibility for a modal model to update parameter values in a refinement is a common source of dynamic reconfiguration in Ptolemy II models.

4 Actor Specialization

As mentioned previously, Java specifications of Ptolemy II actors define actor behavior in a generic way. In order to generate an efficient implementation from a specification, it is transformed into a new actor specification that is specialized to a particular context. Such a context includes, for instance, assignments of values to parameters and assignments of types to ports and parameters. While such a context could be specified explicitly, we instead concentrate on using the implicit context that actors acquire when composed in a model. In contrast with an explicit context, an implicit context is more difficult to use for specialization, since it may change through dynamic reconfiguration of the model.

This report considers four types of actor specialization: parameter specialization, type specialization, connection specialization and domain specialization. The following sections describe, for each type of specialization, the possibilities for determining whether or not the appropriate context of an actor can change. In each case, an actor specification can be specialized if the appropriate context does not change. Additionally, parameter specialization can be performed even if parameter values are dynamically reconfigured by a modal model. In all cases, it is assumed that the model is not dynamically reconfigured by any means external to the model.

Note that we do not consider specialization of the data generality from actors. In most cases, actors used in models of embedded systems operate on unknown data, since they are constantly receiving unknown data from sensors in the physical world. Hence, data generality seems crucial to the notion of an embedded system. However, in some cases it is useful to have actors internal to a model that produce sequences of constant or deterministic data. In such cases it seems possible that classical compiler optimizations, such as constant propagation and constant expression elimination [39] could be applied. We have not fully considered these kinds of specializations, but they seem straightforward to apply.

4.1 Parameter Specialization

In the context of a complete executable model, the expression of a parameter must always be *evaluatable*. Additionally, we distinguish expressions containing identifiers that have not been bound in the model. Such expressions are called *open* expressions, while expressions where all identifiers have been bound are called are called *closed* expressions. The identifiers in open expressions must be either bound later in composition, making the expression closed, or they must match the names of globally defined constants. Lastly, we distinguish those parameters that are *constant*, i.e., that have a closed expression and always evaluate to the same value, and those which may not be constant. Any parameter that is not definitely constant will be called *dynamic*, even though its value may actually not change. A parameter is dynamic if either its expression may be dynamically reconfigured, or its expression is constant and contains identifiers bound to other dynamic parameters. Note that open expressions may still be evaluatable, since identifiers may be bound to globally defined constants, and that closed expressions are not necessarily constant expressions, since parameter values may change during execution.

Parameter specialization is the transformation of an actor specification with unspecified parameter expressions into a specification where parameter expressions are fixed. In this context, two transformations are primarily of interest: replacing queries for the value of constant parameters with the constant value of the parameter, and replacing evaluations of dynamic parameters with generated code that avoids traversing the parse tree.

Unfortunately, these transformations cannot be performed in the context of an open composition. First of all, it is impossible to determine when parameters in an open composition are, in fact, constant, since the open composition may be additionally composed as part of a modal model. Secondly, it is impossible to determine whether identifiers in open expressions should be bound to globally defined constants or left to be bound with other parameters during later composition. For the rest of this section we ignore these difficulties and deal solely with the case of closed composition.

4.1.1 Analyzing for Constant Parameters in Closed Compositions

This section describes a technique for directly computing the set of dynamic parameters in a model, from which can be derived the set of constant parameters. In lieu of pseudocode, we give an abstract mathematical description that can be easily implemented as an algorithm. We will assumes that external modification of parameters is not allowed and that models do not change parameter values internally, except through assignments in the transitions of modal models. The scoping rules of parameters are described by a function $dependents : Parameters \rightarrow \mathcal{P}(Parameters)$, where $Parameters$ is the set of all parameters in the model, and $\mathcal{P}(X)$ denotes the power set of the set X. The set $dependents(p)$ contains every parameter whose expression includes an identifier bound to p. Parameter assignments are described by the set $assignedInModalModels \subseteq Parameters$ contains all parameters that can be modified in the transitions of modal models.

Given this information, we define a function

$$f : \mathcal{P}(Parameters) \rightarrow \mathcal{P}(Parameters)$$

where $\forall X \in \mathcal{P}(Parameters)$,

$$f(X) = X \cup \bigcup_{x \in X} dependents(x)$$

The set of dynamic parameters is the least fixed point

$$dynamicParameters = f(dynamicParameters)$$

starting from the initial point $assignedInModalModels$. Note that since $X \subseteq f(X)$ for all X, f is a *monotonic function under the subset order*. This property, along with the fact that the number of parameters in the model is finite, implies that the above fixed point exists and can be found by repeated application of f.[14]

$$dynamicParameters = \bigcup_{n \geq 0} f^n(assignedInModalModels)$$

Given the set of dynamic parameters, the set of constant parameters is simply $Parameters - dynamicParameters$. Note that this formulation seems straightforward to extend given

```
public TypedIOPort input, output;
public Parameter arrayLength;
public void fire() {
  int length = ((IntToken)arrayLength.getToken()).intValue();
  Token[] valueArray = input.get(0, length);
  output.send(0, new ArrayToken(valueArray));
}
```

Figure 3: Original Code from SequenceToArray.

other sources of dynamic parameter expressions, such as a user interface that can directly manipulate the model.

4.1.2 Replacing Constant Parameters

Parameters that are determined to be constant through the above algorithm can be specialized by replacing accesses to the parameter with the parameter's constant value. Primarily, this results in the replacement of invocations of the parameter's `getToken()` method. These invocations are replaced with references to token objects. Since tokens are immutable objects, the expense of runtime allocation is reduced by creating the tokens during initialization and storing a reference to the token in an automatically created field of the new specification.

As an example, consider the SequenceToArray actor in the example model. This actor consumes eight tokens in the model (the value of the `arrayLength` parameter) and aggregates them into a single array token, as shown in Figure 3. This specification, specialized to a constant `arrayLength` parameter value of 8, is shown in Figure 4.

As an additional optimization, the invocation of the `intValue()` method can also be replaced, since it is always called on the same object. The result is shown in Figure 5. In this case, since the parameter is not used elsewhere in the actor specification, the field and token creation are dead and can also be removed.

```
public TypedIOPort input, output;
public IntToken arrayLength_value = new IntToken(8);
public void fire() {
  int length = arrayLength_value.intValue();
  Token[] valueArray = input.get(0, length);
  output.send(0, new ArrayToken(valueArray));
}
```

Figure 4: The SequenceToArray actor, after specialization with `arrayLength` = 8.

```
Port input, output;
public void fire() {
  int length = 8;
  Token[] valueArray = input.get(0, length);
  output.send(0, new ArrayToken(valueArray));
}
```

Figure 5: The SequenceToArray actor, after additional specialization with `arrayLength` = 8.

Although the initialization of constant parameters is relatively uninteresting, since the values are known to be constant, actor specifications often contain code that is executed when the parameter is initialized. This code is found in the `attributeChanged()` and `attributeTypeChanged()` methods. These methods are used to cache parameter values or to take other action based on the value of a parameter. For instance, the `attributeChanged()` method of the SequenceToArray actor is shown in Figure 6.

After the value of the `arrayLength` parameter is set in the constructor of the actor, the `attributeChanged()` method is called. As part of specializing the actor specification, this method invocation is inlined. Additionally, by analyzing object references, the comparison `attribute == arrayLength` can be removed, since it will always be true after initializing the `arrayLength` parameter and false otherwise. The result is shown in Figure 7.

```
public void attributeChanged(Attribute attribute) {
  if (attribute == arrayLength) {
    int rate = ((IntToken)arrayLength.getToken()).intValue();
    if(rate < 0) {
      throw new IllegalActionException(this,
          "Invalid arrayLength: " + rate);
    }
    input.setTokenConsumptionRate(rate);
  } else {
    super.attributeChanged(attribute);
  }
}
```

Figure 6: The `attributeChanged()` method of the SequenceToArray actor.

4.1.3 Replacing Dynamic Parameters

The process of replacing dynamic parameters follows much the same pattern as replacing constant parameters. However, invocations of the `getToken()` method are replaced with a reference to a field containing the current value of the parameter. This field is updated to contain the new value of the parameter whenever the expression for the parameter is modified. Similarly, if the expression does not change, but depends on another value, then

```
IntToken arrayLengthValue;
public SequenceToArray(CompositeEntity container, String name) {
  super(container, name);
  // Initialize field for parameter.
  arrayLengthValue = new IntToken(8);
  // The result of calling attributeChanged(arrayLength).
  input.setTokenConsumptionRate(8);
}
```

Figure 7: The constructor of the SequenceToArray actor, after specialization with `arrayLength` = 8.

every modification of that value triggers the expression to be recomputed. In the second case, since the expression is constant, code is automatically generated from the expression in order to optimize computation of the new parameter value. This code is generated by traversing the parse tree of the expression in the same way that expressions are evaluated.

Unfortunately, one difficulty with this transformation is that the number of times that the expression is computed during simulation may be different from the number of times the expression is computed in generated code. This can have undesired effects when certain methods are invoked (such as a method that returns a random number). Unfortunately, this difficulty is somewhat inherent in the established semantics of parameter evaluation in Ptolemy II, and is unlikely to be solved solely in the code generator. We are considering ways of simplifying the semantics of the expression language to deal with this problem in ways that are still acceptable for generated code.

4.2 Type Specialization

Actor specifications in Ptolemy II are often type polymorphic, allowing them to operate equally well on integers, doubles, or even arrays of integers. This polymorphism is abstracted by the `ptolemy.data` package, which represents a set of common operations in a type-independent fashion, and a type inference algorithm that infers types where they are not completely specified. Although actors are generically typed, in most models actors only ever receive or produce data of a single type. *Type specialization* reduces a type-polymorphic specification to a specification that is only able to deal with a single type.

Currently, we deal only with closed composition, and assume that the types inferred for ports and parameters are exact. Based on inferred types for ports and parameters, the types for variables in the Java specification are inferred. The types inferred are generally much stronger than the types inferred by the regular Java type system and, when it is necessary to distinguish, they will be referred to as *token types*. Note, however, that token types almost always correspond exactly with a single token class and in most cases this distinction is not

necessary.

4.2.1 Inferring Token Types in Java Actor Specifications

The token type inference algorithm is based on a dataflow analysis. The types of ports and parameters of the actor are fixed, along with the types of newly created tokens, and these types are propagated through the Java code. Unlike standard Java type inference, the types of Java arrays of tokens and the types of fields that refer to tokens are considered type variables, and will be updated with the correct token types. Additionally, the token types of data contained in Java arrays are constrained to be the same.

A key distinction between this dataflow analysis, the Ptolemy II type system [52] and the Java type system occurs when a type variable has different types along different paths through the program. In Java, the variable is assigned a type based on object-oriented subtyping, that is, the type will be the greatest subtype of the types along the two paths. In the Ptolemy II type system, the type would be assigned according the Ptolemy type lattice and an automatic type conversion inserted to ensure that the assigned type is an exact type. In this dataflow analysis, exact types are highly desireable, since they enable type specialization and token unboxing, but automatically inserting a type conversion would change the behavior of the Java code. Hence we interpret the presence of different types along different control paths to be a type error.

As an example, consider the specification of the Ramp actor in Figure 8, where the value of the `init` parameter is an IntToken, and the `step` parameter contains a DoubleToken. The type constraints are such that the type of the output port must be double. On the first firing, the field `_state` refers to an IntToken, which is the value of the `init` parameter. Since the output port has type double, the IntToken is converted to a DoubleToken in the process of being sent. After the `postfire()` method is invoked, the `_state` field refers to a DoubleToken, which results from adding the initial IntToken to the value of the `step` parameter (a DoubleToken). The token type inference system will flag this as a type error.

```
TypedIOPort output;     // double
Parameter init;         // int
Parameter step;         // double
private Token _state;   // general
public void initialize();
  _state = init.getToken();
}
public void fire() {
  output.send(0, _state);
}
public void postfire() {
  _state = _state.add(step.getToken());
}
```

Figure 8: A ramp actor specification, which does not have exact token types.

There are actually two solutions to the above problem. The first solution is to strengthen the type constraints on the actor, declaring that the types of the parameters and the types of the output port must all be the same. The IntToken value of the `init` parameter will be automatically converted into a DoubleToken before being queried, and no type conflict will be found. Another solution, shown in figure 9, is to manually insert code into the `initialize()` method to convert the value of the initial token to the type of the output port. Unfortunately, both of the above solutions have the potential to change the behavior of the program and cannot be performed as automatic transformations.

4.2.2 Type Specialization Transformations

Inference of token types within Java code leads to several automatic transformations. Primarily, Java fields which maintain the state of the actor, such as the field `_state` in Figure 9, can be given new Java types that more accurately reflect the data they reference. Similar transformations can be performed on Java arrays of tokens. These transformations often require the insertion of Java casts to ensure that the specification is still properly typed under the Java type system. Although these transformations do not significantly modify the

```
TypedIOPort output;    // double
Parameter init;        // int
Parameter step;        // double
private Token _state;  // double
public void initialize();
  _state = step.getType().convert(init.getToken());
}
public void fire() {
  output.send(0, _state);
}
public void postfire() {
  _state = _state.add(step.getToken());
}
```

Figure 9: A ramp actor specification annotated with possible exact token types.

behavior of the code, they enable the unboxing of tokens, described in Section 5.1.

4.2.3 Type Inference and Type-Controlled Recursion

In the previous sections, we purposefully avoided describing the propagation of token types through method calls. In fact, there are rather fundamental difficulties with the inference of exact token types for such method calls. These difficulties are closely related to the fact that Java is a Turing-complete language, and arise in loops as well. An example is the actor specification shown in Figure 10. This actor takes an input, which can be either an ArrayToken or numeric token, and multiplies it by the parameter `factor`. If the input is an ArrayToken, then the output is also an ArrayToken. Note that in this case, the `_scaleOnLeft()` is called recursively with different token types.

It seems unsatisfying that such a simple actor and such an elegant use of recursion should not be a useful actor specification. We have tentatively approached this problem by inlining all methods in the actor specification that take tokens as arguments, or return to-

```
TypedIOPort input,output; // array(double)
Parameter factor;         // double
public void fire() {
  if (input.hasToken(0)) {
    Token in = input.get(0);
    Token factorToken = factor.getToken();
    Token result = _scaleOnLeft(in, factorToken);
    output.send(0, result);
  }
}

private Token _scaleOnLeft(Token input, Token factor) {
  if(input instanceof ArrayToken) {
    Token[] argArray = ((ArrayToken)input).arrayValue();
    Token[] result = new Token[argArray.length];
    for (int i = 0; i < argArray.length; i++) {
      result[i] = _scaleOnLeft(argArray[i], factor);
    }

    return new ArrayToken(result);
  } else {
    return factor.multiply(input);
  }
}
```

Figure 10: An actor that scales its input.

kens. Combining this approach with repeated application of type inference and recognition of dead code guarded by the `instanceof` check allows the troublesome method to be completely eliminated. This solution is certainly not perfect, since it may greatly increase code size and is only applicable to recursive function calls governed by the type of a token. However, in lieu of rewriting the actor more simply, there does not appear to be a better solution if exact types are required. The final inlined version is shown in Figure 11.

```
TypedIOPort input,output; // array(double)
Parameter factor;         // double
public void fire() {
  if (input.hasToken(0)) {
    Token in = input.get(0);
    Token factorToken = factor.getToken();
    Token[] argArray = ((ArrayToken)input).arrayValue();
    Token[] result = new Token[argArray.length];
    for (int i = 0; i < argArray.length; i++) {
      result[i] = factor.multiply(input);
    }

    Token result = new ArrayToken(result);
    output.send(0, result);
  }
}
```

Figure 11: An unrolled version of an actor that scales its input.

4.3 Connection and Domain Specialization

Connections between the ports of actors are the primary mechanism for inter-component communication in an actor-oriented model. However, actors are commonly specified independently of their actual connections. The connections between ports are specified separately in a model. This separation allows for Java actor specifications to be easily reused multiple times in a model, and for connections to be created and deleted between actors while a model is executing. However, many models are *statically connected*, i.e. the connections between ports are specified when the model is created and are not changed dynamically.

In this section, we will assume that models are not able to not modify their own connections, implying that all models are statically connected. *Connection specialization* is the process of transforming an actor specification into a new specification where the connections to other actors are fixed.

Additionally, most actors in Ptolemy II are specified by Java code that is domain polymorphic. These actors make relatively few assumptions about how they communicate and interact with other actors. Unlike other forms of actor generality, domain generality seems unique in that it does not enable useful dynamic reconfiguration of a model. That is, we have not seen instances where is it useful to change the model of computation of a model from, for instance, synchronous dataflow to Kahn process networks. Hence, we assume that actors in a statically connected model always operate according to the same model of computation. *Domain specialization* is the process of transforming an actor specification into a new specification where the model of computation is fixed.

Unlike parameter and type specialization, connection and domain specialization is possible in the context of both open compositions and closed compositions. This possibility arises because, in Ptolemy II models, a connection between two ports is always a local connection between ports in the same model. Communication that crosses the boundary of a model must travel through two channels, one that mediates the external connection to the model's port and the other that mediates the internal connection.[3] An example is shown in Figure 12. In contrast, parameters and type constraints have global scope, in the sense that modification of a parameter value or type can affect parameter values and types across hierarchical boundaries.

Connection specialization and domain specialization are dealt with in the same section for two reasons. Primarily, they both involve transformation on the method of ports, and after both of these transformations are performed, ports can be completely removed from the actor specification. Perhaps more importantly, for the synchronous dataflow model of computation we have considered, the `send()`, `get()`, and `broadcast()` methods of ports can be replaced with more efficient code if the domain and port connections are considered at the same time.

[3]This is true in the case of what Ptolemy II considers *opaque* hierarchy. In *transparent* hierarchy, the same is not true. In this paper we deal solely with opaque hierarchy in composition.

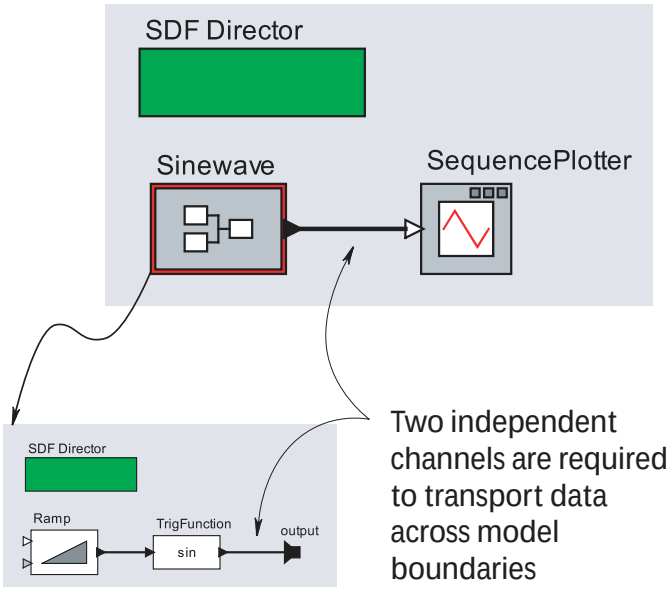

Figure 12: Communication across hierarchy requires multiple channels.

4.3.1 Connection and Domain Specialization Transformations

The specialization of static connections results in the removal of all method invocations on ports. Invocations of methods that query the connections to the ports (such as the `getWidth()` method) are replaced with actual values. At the same time, invocations of methods that send and receive data from ports (i.e. `get()`) are replaced with methods that operate directly on communication channels.

As an example of the implications of this transformation, Figure 13 shows an actor that transmits data received from any input port to its output port. This actor is specified in such a way that each of the three input ports may or may not be connected. If any of the inputs are not connected, then the corresponding loop will never actually consume any data. This specification can be specialized to remove the extraneous loop (since the width of the port is zero). Additionally, some domains, such as synchronous dataflow and Kahn process networks, ensure the presence of data when an actor is fired. In these cases, domain specialization results in the elimination of the unnecessary if statements.

```
Port input1, input2, input3, output;
public void fire() {
  for(i = 0; i < input1.getWidth(); i++) {
    if(input1.hasToken(i)) {
      output.send(0, input1.get(i));
    }
  }
  for(i = 0; i < input2.getWidth(); i++) {
    if(input2.hasToken(i)) {
      output.send(0, input2.get(i));
    }
  }
  for(i = 0; i < input3.getWidth(); i++) {
    if(input3.hasToken(i)) {
      output.send(0, input3.get(i));
    }
  }
}
```

Figure 13: A merge actor.

A source of complexity exists with any port method that takes a channel index, such as the `get()`, `send()`, and `hasToken()` methods in Figure 13. For these methods, it is often the case that the correct channel index can often not be statically determined from the Java specification of the actor. For instance, it is common practice to write actor specification code as in the Figure which iterates over all the channels of the port. To deal with this, an additional array of channels is created for each port of an actor that allows the channel to receive data to be determined at runtime. If all channel references can be determined, then no such array is necessary.

We have currently implemented domain specialization transformations only for the *synchronous dataflow* (SDF) model of computation in Ptolemy II. In an SDF model, all execution and communication can be statically scheduled [7]. This implies that the communication between ports can be implemented using fixed length arrays and circular addressing. Additionally, we notice that SDF buffers can be shared in cases where data is broadcast

```
// The buffer for the input port.
DoubleToken[] _relation_0_double;
// The current read index for the input port.
int[] _index_input;
// The buffer for the output port.
ArrayToken[] _relation3_0__double_;
public void fire() {
  // Code replacing the input.get() method
  DoubleToken[] doubletokens = _relation_0_double;
  int index = _index_input[0];
  Token[] tokens = new Token[8];
  for (int i = 0; i < 8; i++) {
    tokens[i] = doubletokens[i];
    index = ++index % 8;
  }
  _index_input[0] = index;
  ArrayToken arraytoken = new ArrayToken(tokens);
  // Code replacing the output.send() method
  ArrayToken[] arraytokens = _relation3_0__double_;
  arraytokens[0] = arraytoken;
}
```

Figure 14: The SequenceToArray actor, with optimized communication.

to multiple receiving port. Invocations of the `get()` and `send()` are replaced with array reads and writes and circular buffer addressing. The length of the arrays is statically computed by simulating the execution of the schedule.

The code in Figure 14 illustrates the resulting transformed code for the SequenceToArray actor in the example model. This code contains references to the arrays of tokens for the input and output buffers. It also contains a reference to the array of indices into the input buffer which is updated as data is read from the buffer. No array of indices is created for the output buffer, which contains only a single location.

5 Co-compilation in Ptolemy II

Considered alone, the specialization transformations in the previous section allow optimization of individual actors to a given context. This section describes how those transformations can be used in combination with automatic code generation to synthesize an efficient implementation from an actor-oriented model. When have built a tool, called Copernicus, that generates an implementation from Ptolemy II models using co-compilation.

The sequence of operations performed during co-compilation is:

1. Perform any analysis on the model, extracting information such as inferred types and scheduling information.
2. Duplicate each original Java actor specification for each actor in the model. Automatically generate a Java actor specification for actors that are specification using finite state machines, modal models, or expressions.
3. Specialize each specification according to the usage of the actor in the model.
4. Automatically generate code from the model to join the specialized actor specifications.
5. Unbox tokens, remove internal actor interfaces, obfuscate the generated code and perform other traditional compiler optimizations.

These transformations are represented by the large arrows in Figure 15. Note that after the transformations are completed, an actor originally specified using a model has been replaced with a Java actor specification. In this specification, the structure of the model has been completely removed, leaving only the original actor interface to the model. Also note that the generated specification still implements the Java `Executable` interface, even if it no longer contains ports and parameters.

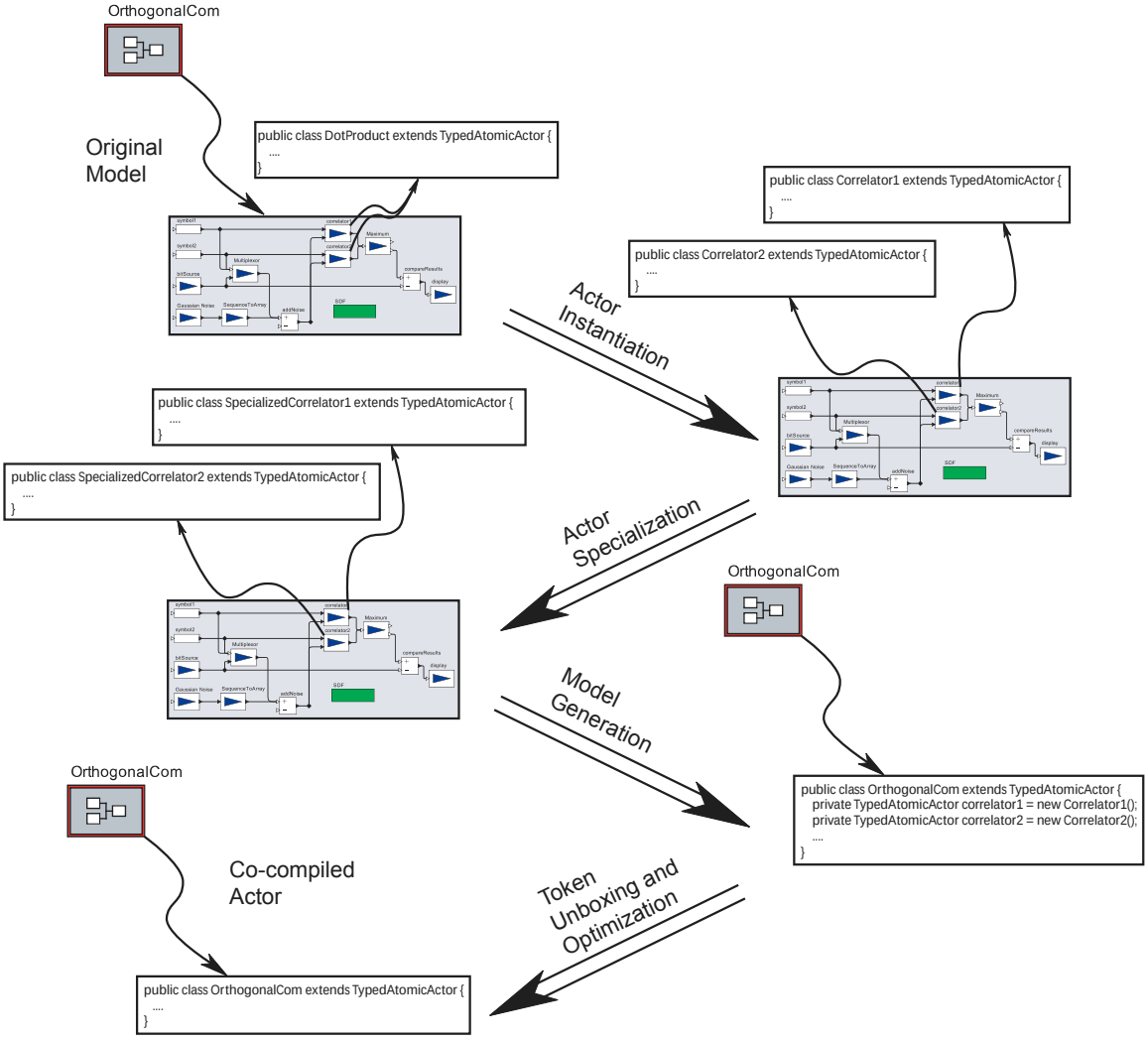

Figure 15: Co-compilation steps

The specializations in step three are exactly those specializations described in section 4. In step four, code is automatically generated to instantiate the correct communication buffers and schedule actor firings according the model of computation. Code is also generated to connect the specialized actor specifications to the correct buffers.

Step five consists of several optimizations that do not operate on the structure of the model, but are instrumental in the efficiency of the final actor specification. Token unboxing transformations replace abstract token objects with native Java types, reducing object allocation and garbage collection requirements. These transformations are possible because we restrict our actor specifications and models to those where exact types exist. This transformation will be described more completely in the next section.

Internal actor interfaces are also eliminated in step five. Since ports and parameters have been removed during specialization, the Java classes corresponding to the specialized actors can be modified to extend from the `java.lang.Object` base class. This transformation is possible because there is exactly one instance of each specialized actor specification.

Another optimization that occurs in step five is *obfuscation* of the generated code. Obfuscation replaces the names of all methods with shorter strings. This is important since, in Java bytecode, methods are referred to by the complete signature of the method. Hence, unlike in C or C++, the names of methods often have a significant impact on the size of compiled Java classes. We have applied the obfuscator in Jode [23], an open source decompilation tool.

To avoid undue complication, we do not specifically mention other optimizations that are performed in step five, since they are well described in the literature. These techniques include both peephole optimizations (common sub expression elimination, dead assignment elimination, etc.) [39], as well as optimizations that take into account more global information (loop unrolling, object flow analysis, local splitting [39], method inlining, etc.) Note that in some cases, the co-compiler has more information about the classes being operated on than a standard compiler, since automatically generated classes are not referenced

from outside of automatically generated code. This fact can be used to implement efficient transformations on the class hierarchy that are not normally performed by Java compilers.

5.1 Token Unboxing

The boxing and unboxing of data is a well-known technique used in functional languages, such as ML [36]. In functional languages, the goal of unboxing is to be able to pass numeric types to type-polymorphic functions. The functions themselves are written to handle arbitrary objects, but are unable to handle numeric values. *Boxing* refers to the process of automatically encapsulating a numeric value in a wrapper object so that it can be passed to such a type-polymorphic method. When the number is eventually passed to another method that requires the numeric value, it is automatically removed from its wrapper through *unboxing*. This transformation happens within the execution engine for the language and is totally transparent to the programmer.

The co-compiler performs a transformation similar to unboxing: it replaces token objects (an abstract wrapper for a data object) with the value that the token contains. Similarly, operations on the token (i.e., method calls) are replaced with native numeric operations. For instance, the `IntToken.add()` method, which adds the values contained in two integer wrapper objects, is replaced with a simple integer addition. In most Java implementations, this greatly reduces the overhead involved in the operation. More importantly, the overhead of allocating and garbage collecting the wrapper object for the result is also eliminated.

It is important to notice that token unboxing is not possible in the presence of type-polymorphic actor specifications. Token unboxing is possible during co-compilation because the Ptolemy II type system emphasizes models where types are exactly determined and type-polymorphic actor specifications have been specialized to those exact types.

So, for a particular type of token, which native numerical type and operations should it replaced with? One possibility is to use a fixed and hardcoded replacement relation

between a type of token and a native numerical type [48]. Unfortunately, this limits the ability to add new data types to the Ptolemy II framework, as the operations for each token must be essentially reimplemented in the code generation framework. We must also have some way of transforming structured token types that are not directly replaced with native types. This is not easily handled by a small set of hand-written rules.

We have implemented a technique for transforming tokens that does not rely on hand-written replacement rules. Instead of reimplementing each token operation, we make use of the specification of each token operation that already exists in the corresponding token class. Wherever a method is invoked on a token class, this method is inlined from the correct token class. Each token variable and field that refers to a token is replaced with variables and fields corresponding to the fields of the token class. Additionally, a boolean field is created that tracks whether the original token reference is `null`. This flag is used to properly replace comparisons between the token and `null`.

This technique is generally effective for all numeric token types. Furthermore, it does not preclude optimized transformations for specific numeric types, such as those described for fixed-point types in [27]. It is also applicable for structured types as well, such as arrays and records. For instance, the `ptolemy.data.ArrayToken` class aggregates a set of other tokens and indexes them using integers. Since one field of the class contains an array of other tokens, unboxing the array token replaces it with an array of tokens. These tokens (regardless of their type) can then be unboxed by applying the above procedure recursively.

6 Results

One of the key goals of our co-compiler is the generation of very efficient and optimized code. We have measured the code size, execution time, and memory allocation of the generated code at each step in the co-compilation process, to better understand how individual transformations change the generated code. We present this data for several exam-

ple models. The `orthoCom` model is the example shown in Figure 1, consisting of 12 actors. The `rijndael` model is a model of the AES encryption system and contains a total of 109 actors specified with expressions, modal models, as well as Java code. The `rijndaelKeyGen` model is a portion of the `rijndael` model that generates a pseudo-random key sequence, consisting of a total of 35 actors.[4]

We present performance data for all stages of co-compilation, from the original model through actor specializations to the final optimized implementation of the model. For comparison, we also compare performance with and without obfuscation of the generated code. Note, however, that the intermediate stages of specialization have not been independently optimized. For instance, the parameter specialization transformations have not been optimized to minimize the number of tokens created, since any tokens created are later unboxed. This data is simply intended to demonstrate the potential feasibility of using actor specialization to improve execution efficiency and provide a rough comparison between different techniques.

Primarily of interest is the execution time of the generated code. This can have a tremendous impact on the usefulness of code generation for embedded software systems. Programs that do not meet the real-time requirements of an embedded system are useless. To reduce measurement error, we report the execution time to process a reasonably large amount of data through each model, and average over several runs. The data is collected using the Java virtual machine in interpreted mode to avoid confusion from just-in-time compilation. The results shown in Figure 16 indicate that a large speedup (roughly 20x) of SDF models comes from specializing the communication between actors. Profiling suggests that this speedup largely arises from the removal of synchronization in Ptolemy II that protects modifications to models in multi-threaded environments. Some speedup also occurs after unboxing tokens in step 5, mainly due to reduced load on the garbage collection system.

Another important metric of the generated code is the size of the code that is generated.

[4]These models are available at `http://ptolemy.eecs.berkeley.edu/publications`.

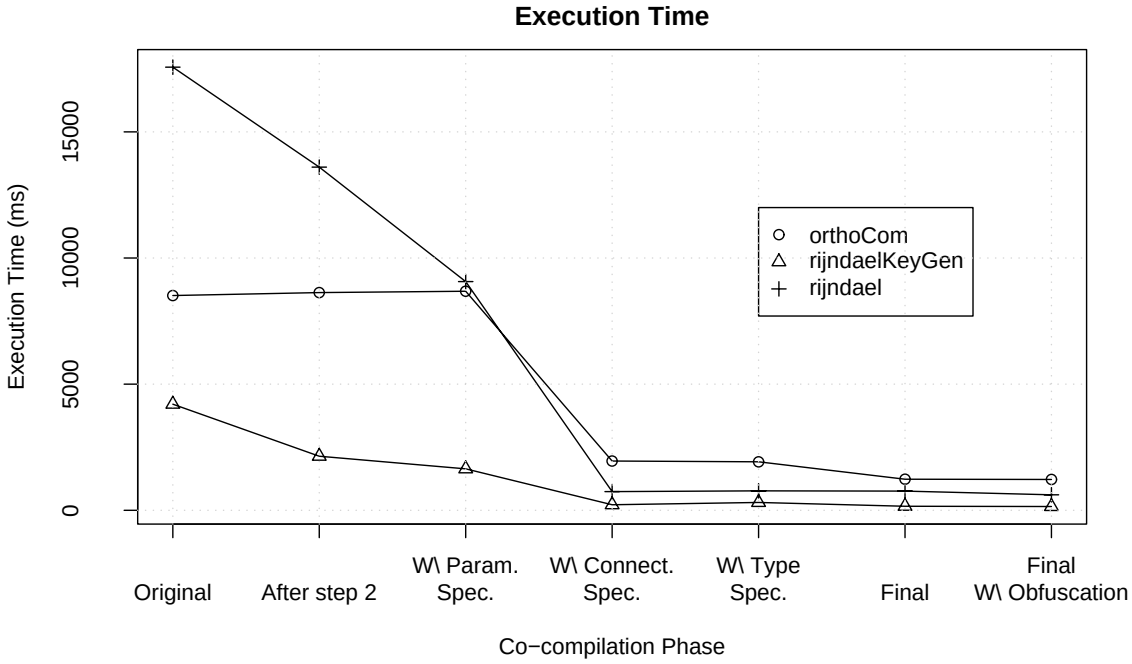

Figure 16: Total Execution Time

This size is primarily of importance for embedded software systems, where it determines the amount of non-volatile memory required. Figure 17 shows the size of the minimal `.jar` archive of Java classes necessary to execute the generated code. This file is generated by executing the generated code and tracking which classes are dynamically loaded. This archive does not include the size of the standard JVM.

The results in Figure 17 suggest that the code size of the Ptolemy II software framework greatly dominates the size of the generated code, even for the largest model. Actor specialization, combined with token unboxing, removes almost all references to the software framework. The data also suggests that even for modestly large models, like *rijndael*, the size of generated code is significant. Even a hundred thousand bytes of code is unsuitable for many embedded systems, and is roughly twelve times larger than a Java reference implementation for Rijndael.[5]

[5] `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/rijndael.zip`

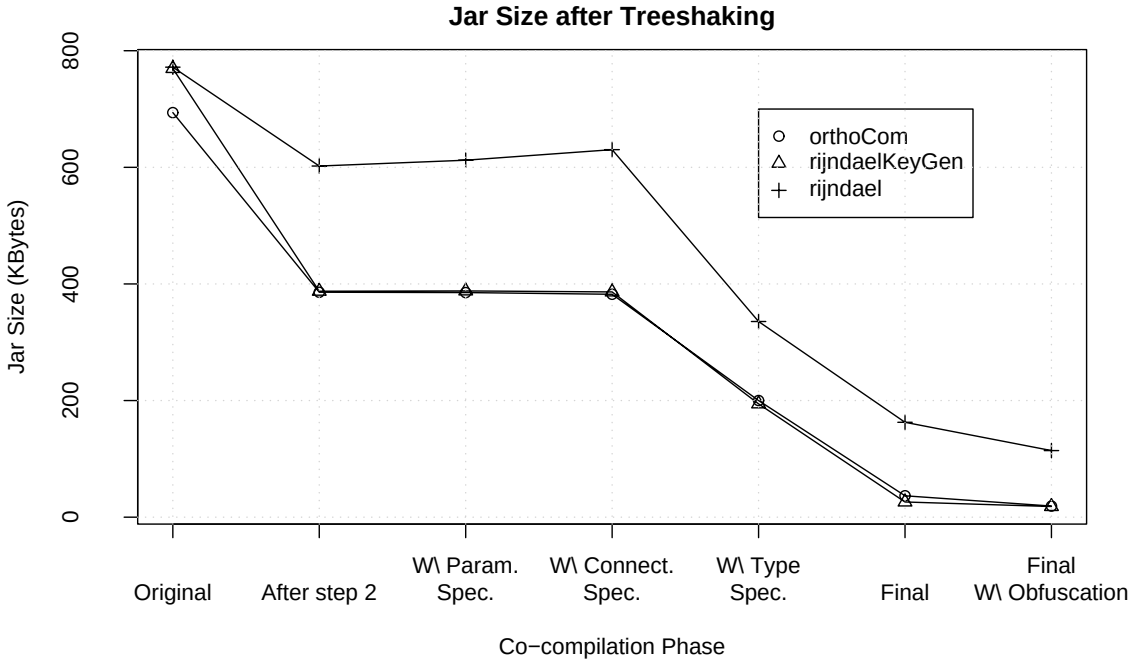

Figure 17: Jar File Size

The final metric we consider is the run-time memory requirements of the generated code. Here the primary concern is reducing load on the Java garbage collector, so we consider only memory that is allocated and available for recovery by the Java virtual machine. This information is collected by logging garbage collection in the Java virtual machine and totalling the number of bytes collected. Objects that are not available for recovery represent a relatively small fraction of total allocated memory. The results in Figure 18 suggest that significant improvements arise from communication specialization, since caches for performance improvement can be replaced with hardcoded relationships. Token unboxing also reduces the amount of data allocated, although this effect is relatively less significant for the profiled models. In particular, the textttorthoCom model contains many array tokens. Although token unboxing eliminates the token allocations, the arrays themselves are still allocated. Future work will hopefully address this difficulty.

Ultimately, these results seem to suggest that actor specialization is a promising technique for improving execution efficiency of actor-oriented models. While the current ap-

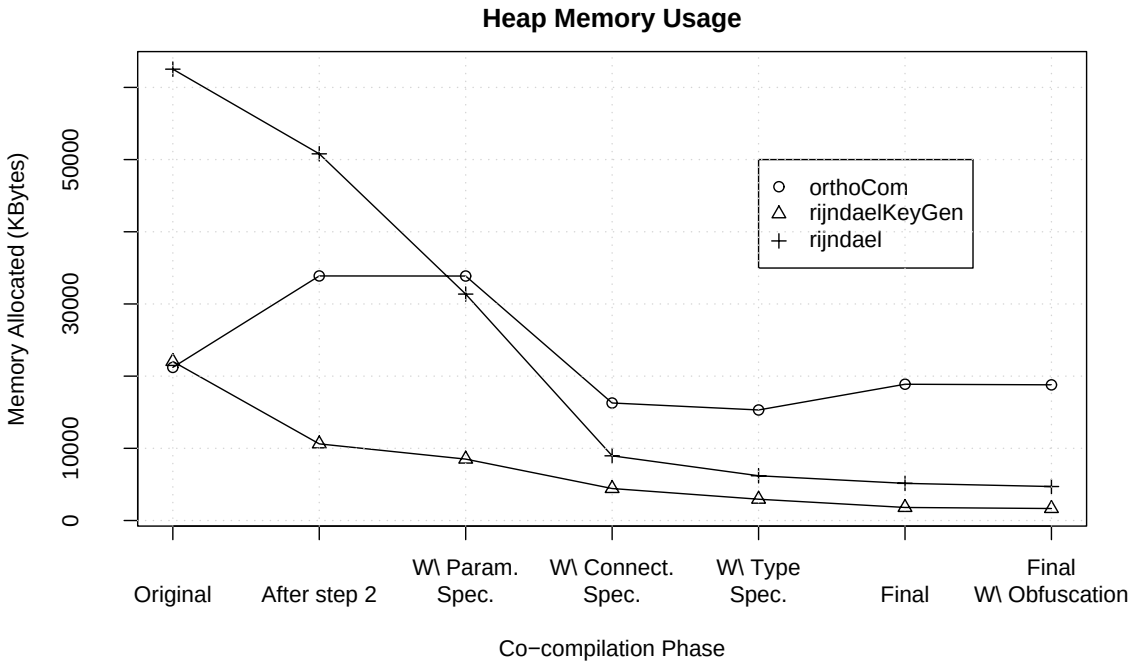

Figure 18: Heap memory usage of the generated code.

plication of these techniques in code generation seems useful for improving simulation speed, greater execution efficiency is still needed to use the generated implementations in resource constrained embedded systems. Currently none of these specialization techniques take memory usage or code size into account.

After removing actor generality through specialization, the next major performance bottleneck appears to be caused by actor safety. For instance, extra copies of data are often made when tokens are created, in order to ensure that the tokens themselves are immutable pieces of data. This ensures that the data is always used in a safe fashion. However, it is possible that safe use of tokens can be shown through formal analysis instead, eliminating this overhead. Future work will hopefully address this issue as well.

7 Summary

This paper presents a series of techniques that enable the automatic specialization of general component specifications. These components are specialized according to an implicit context that exists when the components are composed with other components. This form of specialization is particularly useful when synthesizing an optimized implementation of an aggregation of components. The specialization transformations have been implemented as part of a Ptolemy II co-compilation tool, called Copernicus.

An important aspect of our co-compiler is that it is not built for a particular set of token data types or actor specifications. Optimized code is generated by parsing Java code specifications of tokens and actors and manipulating those specification through their Java interfaces. This flexibility is critical, since it enables user-specified actors and data types to be easily used.

7.1 Related Work

The Ptolemy II co-compiler can also be viewed as an application of *aspect-oriented programming* and *aspect weaving* [28] to actor-oriented design. In aspect-oriented programming, different languages are used to express separate, orthogonal aspects of a program. Aspect weaving is the process of taking those orthogonal aspects and compiling them together into a single program.

In an actor-oriented system model, there are two separate aspects of the model that we are interested in: the behavior of individual actors and the composition of those actors. The behavior of an actor is usually specified using Java code and the Ptolemy II actor programming interfaces. The composition of actors can be specified using several semantically equivalent syntaxes, including an XML-based textual syntax and a block diagram visual syntax. The result of co-compiling a Ptolemy II model is the weaving of the communication and configuration aspect of the model into the actor code. The basis for the

combination is the precise definition of the programming interfaces summarized in section 3.

Unfortunately, co-compilation is not compatible with existing aspect-oriented design methodologies. The available tools, such as AspectJ [29], primarily provide syntactic support for weaving in a carrier language. Unfortunately, many steps in co-compilation require a significant amount of semantic analysis of the existing code, such as loop and dataflow analysis in order to be able to create an efficient implementation. The tools are also not designed with many facilities to dynamically control the weaving. For Ptolemy II, this would mean that a new set of aspects would have to be dynamically generated from each Ptolemy II model. For these reasons, we have more directly leveraged traditional compiler techniques and infrastructure, and primarily discuss co-compilation in those terms.

Another area of related work is Paul Hudak's notion of a *domain-specific embedded languages* (DSELs) [24]. A DSEL is a language that is implemented not as a traditional programming language such as C, Java or Haskell, but as a well-defined set of programming interfaces in a traditional programming language (the *carrier language*). Instead of having a specific compiler, syntax, and semantics, an embedded language inherits them from the carrier language.

Essentially, the style of Java actor specifications in section 3.2, combined with exact type inference of section 4.2.1 forms an embedded language for actor design. While Hudak concentrates on using the functional language Haskell as a carrier language, we use the object-oriented Java language instead. Additionally, Hudak describes how partial evaluation of Haskell functions can be used to improve performance of his DSELs. The specialization transformations described in section 4 describe the kinds of partial evaluation that can be performed on our actor specifications.

7.2 Future Work

There are several interesting areas of future work. Firstly, this paper has concentrated on generating code for closed compositions. It would be interesting to concentrate on implementing these techniques for open compositions as well. This would require recognizing open parameter expressions and leaving those as parameters. It would also be necessary to verify that no information about the actor interface of open compositions is lost when generating code. In the context of type specialization, open compositions can be approached by recognizing the presence of "tight" type constraints. Such type constraints ensure that a type cannot change, even in the presence of further composition.

One disadvantage of the type inference algorithm given in section 4.2.1 is that it assumes that the type constraints declared by an actor are, in fact, correct. However, it is relatively easy to write actor code for which the correct type constraints are non-trivial to write by hand. I anticipate that these constraints can actually be automatically extracted from Java actor specifications or, at the very least, checked for inconsistency.

We have currently approached co-compilation with an eye towards maximizing the opportunities for specialization. This approach can lead to the creation of large amount of duplicated code. For instance, an actor may be reused multiple times in the model, with the same types and parameter values in each case. Recognizing such cases has the potential to greatly reduce the size of generated code.

Another area of future work surrounds how generated code can be integrated into an embedded systems. For generated Java code, in addition to difficulties with resource constraints, library management is also a concern. Embedded Java virtual machines inevitably have limited libraries which must be augmented with generated code. This problem is often described as application extraction [47]. Additionally, embedded systems often require a hardware implementation or mixed hardware and software systems are the norm. I have begun investigating how to synthesize such implementations from Ptolemy II models and Java actor specifications. While there has been some work dealing with communication

refinement from actor-oriented models to such systems [11, 3], there appears to be much work yet to be done.

I intend to apply this research to the Caltech robotic vehicle testbed [13]. This vehicle has complex dynamics that resemble the two-dimensional dynamics of airplanes. I anticipate implementing autonomous control algorithms as well as pilot-involved control algorithms. Of particular interest is a "softwalls" system, which is capable of preventing the vehicle's pilot from driving into a predefined unsafe area [31].

7.3 Acknowledgments

I'd like to thank the authors of Soot,[6] a library for optimizing bytecode, and Jochen Hoenicke, the author of Jode. Their effort building usable tools made my task easier. I'd also like to thank Christopher Hylands, for his assistance in scripting the extraction of performance data.

References

[1] G. Agha, I. A. Mason, S. F.Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 1993.

[2] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, 1986.

[3] F. Balarin, L. Lavagno, C. Passerone, and Y. Watanabe. Processes, interfaces, and platforms: Embedded software modeling in Metropolis. In *Proceedings of EMSOFT 02: Embedded Software*, Lecture Notes in Computer Science 2491, pages 407–421. Springer, 2002.

[6] http://www.sable.mcgill.ca/soot/

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[5] S. S. Bhattacharyya and E. A. Lee. Scheduling synchronous dataflow graphs for efficient looping. *Journal of VLSI Signal Processing*, 6, 1993.

[6] S. S. Bhattacharyya and E. A. Lee. Memory management for dataflow programming of multirate signal processing algorithms. *IEEE Transactions on Signal Processing*, 1994. To appear.

[7] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer, 1996.

[8] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

[9] J. Buck, S. Ha, E. A. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulations*, 4:155–182, April 1995.

[10] J. Buck and R. Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES), San Diego, California*, May 2000.

[11] W. O. Cesário, G. Nicolescu, L. Gauthier, D. Lyonnard, and A. A. Jerraya. Colif: A design representation for application-specific multiprocessor SOCs. *IEEE Design and Test of Computers*, 18(65):8–19, Sept. 2001.

[12] K. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[13] L. Cremean et al. The caltech multi-vehicle wireless testbed. In *Proceedings of the Conference on Decision and Control (CDC)*. IEEE, Dec. 2002.

[14] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[15] J. Davis et al. Ptolemy II - Heterogeneous concurrent modeling and design in Java. Memo M01/12, UCB/ERL, EECS UC Berkeley, CA 94720, Mar. 2001.

[16] S. Edwards. Compiling Esterel into sequential code. In *Proceedings of International Symposium on Hardware/Software Codesign (CODES)*. SIGDA, ACM, May 1999.

[17] S. A. Edwards. *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. PhD thesis, EECS Department, University of California at Berkeley, CA, 1997.

[18] D. D. Gajski, editor. *SpecC: Specification Language and Methodology*. Kluwer, 2000.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[20] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1321, September 1991.

[21] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence*, 8(3):323–363, June 1977.

[22] C. A. R. Hoare. *Communicating sequential processes*. Computer Science. Prentice Hall International, 1985.

[23] J. Hoenicke. Jode: Java optimizer and decompiler. http://jode.sourceforge.net.

[24] P. Hudak. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse*, pages 134–142, IEEE Computer Society, June 1998.

[25] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, pages 471–475, Paris, France, 1974. International Federation for Information Processing, North-Holland Publishing Company.

[26] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Proceedings of the IFIP Congress 77*, pages 993–998, Paris, France, 1977. International Federation for Information Processing, North-Holland Publishing Company.

[27] H. Keding, M. Coors, O. Luethje, and H. Meyr. Fast bit-true simulation. In *Proceedings of the 38th Design Automation Conference (DAC'2001)*, June 2001.

[28] G. Kiczales et al. Aspect-oriented programming. In *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, number 1241 in Lecture Notes in Computer Science, pages 220–242. Springer-Verlag, 1997.

[29] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

[30] B. A. Kienhuis. *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools*. PhD thesis, TU Delft, Jan. 1999.

[31] E. A. Lee. Soft Walls - Modifying flight control systems to limit the flight space of commercial aircraft. Technical Memorandum UCB/ERL M01/31, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California Berkeley, CA 94720, USA, Oct. 2001.

[32] E. A. Lee. Embedded software. *Advances in Computers*, 56, 2002.

[33] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, pages 55–64, September 1987.

[34] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, scheduled for publication June 2003.

[35] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–798, May 1995.

[36] X. Leroy. Effectiveness of type-based unboxing. Technical Report BCCS-97-03, Boston College Computer Science Department, June 1997. In Workshop on Types in Compilation '97.

[37] J. Liu. *Responsible Frameworks for Heterogenous Modeling and Design of Embedded Systems*. PhD thesis, EECS Department, University of California at Berkeley, CA, 2001.

[38] J. Liu, X. Liu, T. J. Koo, B. Sinopoli, S. Sastry, and E. A. Lee. A hierarchical hybrid system model and its simulation. In *38th IEEE conference on Decision and Control, Phoenix, AZ,* December 1999.

[39] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[40] P. K. Murthy, E. G. Cohen, and S. Rowland. System Canvas: A new design environment for embedded DSP and telecommunication systems. In *Proceedings of International Symposium on Hardware/Software Codesign (CODES)*. SIGDA, ACM, Apr. 2001.

[41] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, EECS Department, University of California at Berkeley, CA, 1995.

[42] C. Passerone, Y. Watanabe, and L. Lavagno. Generation of minimal size code for schedule graphs. In *Proceedings of Design, Automation and Test in Europe (DATE)*. SIGDA, ACM, Mar. 2001.

[43] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck. Software synthesis for DSP using Ptolemy. *Journal on VLSI Signal Processing*, 9(1):7–21, Jan. 1995.

[44] D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. on Software Engineering*, 23(12):759–776, Dec. 1997.

[45] J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE Computer*, pages 110–112, Apr. 1997.

[46] J. Teich, E. Zitzler, and S. Bhattacharyya. 3D exploration of software schedules for DSP algorithms. In *Proceedings of International Symposium on Hardware/Software Codesign (CODES)*. SIGDA, ACM, May 1999.

[47] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. Technical Report 21451(96813), IBM Research, Oct. 1999.

[48] J. Tsay, C. Hylands, and E. Lee. A code generation framework for Java component-based designs. In *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems(CASES)*, pages 18–25, ACM, Nov. 2000.

[49] P. van der Wolf, P. Lieverse, M. Goel, D. L. Hei, and K. Vissers. An MPEG-2 decoder case study as a driver for a system level design methodology. In *Proceedings of International Symposium on Hardware/Software Codesign (CODES)*. SIGDA, ACM, May 1999.

[50] L. Wernli. Design and implementation of a code generator for the CAL actor language. Technical Memorandum UCB/ERL M02/5, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California Berkeley, CA 94720, USA, March 2002.

[51] M. Williamson. *Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications*. PhD thesis, EECS Department, University of California at Berkeley, CA, 1998.

[52] Y. Xiong and E. Lee. An extensible type system for component-based design. In *Proceedings of 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 1785 in Lecture Notes in Computer Science. Springer-Verlag, Mar. 2000.