An Extensible Type System for Component-Based Design

Yuhong Xiong and Edward A. Lee
{yuhong, eal}@eecs.berkeley.edu

Abstract. We present the design and implementation of the type system for Ptolemy II, which is a tool for component-based heterogeneous modeling and design. This type system combines static typing with run-time type checking. It supports polymorphic typing of components, and allows automatic lossless type conversion at run-time. To achieve this, we use a lattice to model the lossless type conversion relation among types, and use inequalities defined over the type lattice to specify type constraints in components and across components. The system of inequalities can be solved efficiently, with existence and uniqueness of a solution guaranteed by fixed-point theorems. This type system increases the safety and flexibility of the design environment, promotes component reuse, and helps simplify component development and optimization. The infrastructure we have built is generic in that it is not bound to one particular type lattice. The type system can be extended in two ways: by adding more types to the lattice, or by using different lattices to model different system properties. Higher-order function types and extended types can be accommodated in this way.

1 Introduction

Ptolemy II [5] is a system-level design environment that supports component-based heterogeneous modeling and design. The focus is on embedded systems. In component-based design, each component has an interface, which includes the data type of the messages sent or received by the component, and the communication protocols used by the component to exchange information with others. In Ptolemy II, the interconnection of components is represented by hierarchical clustered graphs. Interconnections imply type constraints. In addition, components themselves may have constraints on their interface and internal state variables.

A good type system is particularly important for embedded systems. A type system can increase safety though type checking, promote component reuse through polymorphic typing, provide services such as automatic type conversion, and help optimize the design by finding low cost typing for polymorphic components.

Ptolemy II supports heterogeneous design by providing a variety of models of computation (MoCs) [5]. It can be viewed as a coordination language where it manages the communication among independent components without much knowledge about the computation they carry out. In this regard, it is similar to other coordination languages like Manifold [2]. In different MoCs, component interaction obeys different semantics. However, this level of detail can be ignored in data level type system design, and a general message passing semantics can be assumed. This abstraction enables the same type system to work with widely differing models. Fig.1 shows a simplified graph representation of a Ptolemy II model. In Ptolemy II terminology, each of the components A, B, and C is an *actor*, and actors contain *ports*, denoted by the

small circles on the actors. Actors send and receive messages through ports. Messages are encapsulated in *tokens*, which are typed.

In general-purpose languages, there are two approaches for type system design: static typing and dynamic typing. Research in this area is driven to a large degree by the desire to combine the flexibility of dynamically typed languages with the security and early error-detection potential of statically typed languages. Polymorphic type systems of modern languages have achieved this goal to a large extent [14]. Since Ptolemy II is intended for large, complex, and possibly safety-critical system design, we choose static typing for its obvious advantages. To do this, we give each actor port a type. This type restricts the type of tokens that can pass though the port. Based on the port types and the graph topology, we can check the type consistency in the model statically, before it is executed. In Ptolemy II, static checking alone is not enough to ensure type safety at run-time because Ptolemy II is a coordination language, its type system does not have detailed information about the operation of each actor, except the declared types of the ports and the type constraints provided by the actors. In fact, Ptolemy II places no restriction on the implementation of an actor. So an actor may wrap a component implemented in a different language, or a model built by a foreign tool [11]. Therefore, even if a source actor declares its port type to be *Int*, no static structure prevents it from sending a token containing *Double* at run-time. The declared type *Int* in this case is only a promise from the actor, not a guarantee. Analogous to the run-time type checking in Java, the components are not trusted. Static type checking checks whether the components can work together as connected based on the information given by each component, but run-time type checking is also necessary for safety. With the help of static typing, run-time type checking can be done when a token is sent from a port. I.e., the run-time type checker checks the token type against the type of the port. This way, a type error is detected at the earliest possible time, and run-time type checking (as well as static type checking) can be performed by the system infrastructure instead of by the actors.

Another benefit of static typing is that it allows the system to perform lossless type conversion. For example, if a sending port with type *Int* is connected to a receiving port with type *Double*, the integer token sent from the sender can be converted to a double token before it is passed to the receiver. This kind of run-time type conversion is done transparently by the Ptolemy II system (actors are not aware it). So the actors can safely cast the received tokens to the type of the receiving port. This makes actor development easier. As a design principle of Ptolemy II, the system does not implicitly

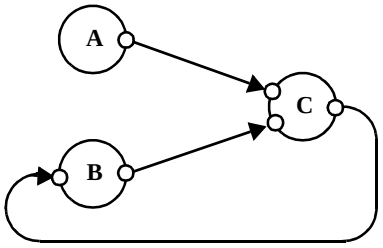

Fig. 1. A simplified Ptolemy II model.

perform data type conversions that lose information. The lossless type conversion relation among different types is modeled as a partially ordered set, called the *type lattice*.

In Ptolemy II, polymorphic actors are actors that can accept multiple types on their ports. In general, the types on some or all of the ports of a polymorphic actor are not rigidly defined to specific types when the actor is written, so the actor can interact with other actors having different types. The acceptable types on polymorphic actors are described by a set of *type constraints*, which have the form of inequalities defined over the type lattice. The static type checker checks the applicability of a polymorphic actor in a topology (an interconnection of components) by finding specific types for them that satisfy the type constraints. This process, called *type resolution*, can be done by a very efficient algorithm.

In addition to maintaining type consistency for data transfer, our type system plays a larger role. In a component-based architecture, there are two ways to get data to components: static configuration (via parameters) and dynamic message passing (via ports). Our system allows constraints on the types of parameters, as well as the types of ports. In addition, Ptolemy II permits state variables that are local to a component to be typed, so type constraints between ports, parameters, and state variables can all be expressed.

Besides the models based on message passing, Ptolemy II also supports control oriented models, such as finite state machines (FSM), which represent a system as a sequence of state transitions in response to events. In this model, type constraints can link the transition guard and the event of the state machine. Hierarchical FSMs can be mixed with other concurrency models [7]. In these mixed models, type constraints can be propagated between the events of the control model and the data of the other concurrency models. Section 4.2 below shows an example of this.

Our type system is related to the work of Fuh and Mishra [6] that extended polymorphic type inference in ML [12] with subtypes. The lossless type conversion relation is a subtype relation. However, there are several key differences between our approach and the ML type system and the system of Fuh and Mishra. First, the ML type inference algorithm produces principal types. Principal types are the most general types for a program in that any other legal type assignment is a substitution instance of it. In our system, the type resolution algorithm finds the most specific type rather than the most general type. This specific type is the least fixed point solution for the type constraints rather than the greatest fixed point. As we will see, using the most specific type may help optimize the system under design, as the most specific type usually has a lower implementation cost. Second, the ML type system does all the checking statically, while our system combines static and run-time checking. As discussed above, we assume that the system components are opaque to the type system. The type system does not have detailed knowledge of the operation of the components, so static checking alone cannot guarantee run-time safety. Our combined approach can detect errors at the earliest possible time and minimize the computation of run-time checking. Third, the system of Fuh and Mishra allows arbitrary type conversion, represented by a coercion set, while our system concentrates on lossless conversion. This focus permits the conversion relation to form a lattice structure, and the type constraints to be expressed as inequalities on the lattice. As a result, the type constraints can be solved

by a linear time algorithm, which is more efficient than the algorithm to check the consistency of a coercion set.

The advantage of a constraint-based approach, like ours, is that constraint resolution can be separated from constraint generation, and resolution can employ a sophisticated algorithm. Although the users need to understand the constraint formulation, they do not have to understand the details of the resolution algorithm in order to use the system. In addition, the constraint resolution algorithm can be built as a generic tool that can be used for other applications. Even more important in Ptolemy II, the types are not aware of the constraints, so more types can be added to the type lattice, resulting in an extensible type system.

2 Ptolemy II

Ptolemy II offers a unified infrastructure for implementation of a number of models of computation. It consists of a set of Java packages. The key packages relevant to the type system are the kernel, actor, data, and graph packages.

2.1 The Kernel Package

The kernel package defines a small set of Java classes that implement a data structure supporting a general form of uninterpreted clustered graphs, plus methods for accessing and manipulating such graphs. These graphs provide an abstract syntax for netlists, state transition diagrams, block diagrams, etc. A graph consists of *entities* and *relations*. Entities have *ports*. Relations connect entities through ports. Relations are multiway associations. Hierarchical graphs can be constructed by encapsulating one graph inside the composite entity of another graph. This encapsulation can be nested arbitrarily.

2.2 The Actor Package

The actor package provides basic support for executable entities, or actors. It supports a general form of message passing between actors. Messages are passed between ports, which can be inputs, outputs or bidirectional ports. Actors can be typed, which means that their ports have a type. The type of the ports can be declared by the containing actor, or left undeclared on polymorphic actors; type resolution will resolve the types according to type constraints. Messages are encapsulated in tokens that are implemented in the data package or in user-defined classes extending those in the data package.

A subpackage of the actor package contains a library of (currently) about 40 polymorphic actors.

2.3 The Data Package

The data package provides data encapsulation, polymorphism, parameter handling, and an expression language. Data encapsulation is implemented by a set of token classes. For example, IntToken contains an integer, DoubleMatrixToken contains a two-dimensional array of doubles. The tokens can be transported via message passing

between Ptolemy II objects. Alternatively, they can be used to parameterize Ptolemy II objects. Such encapsulation allows for a great degree of extensibility, permitting developers to extend the library of data types that Ptolemy II can handle.

One of the goals of the data package is to support polymorphic operations between tokens. For this, the base Token class defines methods for the primitive arithmetic operations, such as add(), multiply(), subtract(), divide(), modulo() and equals(). Derived classes override these methods to provide class specific operations where appropriate.

Parameter handling and an extensible expression language, including its interpreter, are supported by a subpackage inside the data package. A parameter contains a token as its value. This token can be set directly, or specified by an expression. An expression may refer to other parameters, and dependencies and type relationships between parameters are handled transparently.

2.4 The Graph package

This package provides algorithms for manipulating and analyzing mathematical graphs. Mathematical graphs are simpler than Ptolemy II clustered graphs in that there is no hierarchy, and arcs link exactly two nodes. Both undirected and directed graphs are supported. Acyclic directed graphs, which can be used to model complete partial orders (CPOs) and lattices [4], are also supported with more specialized algorithms. This package provides the infrastructure to construct the type lattice and implement the type resolution algorithm. However, this package is not aware of the types; it supplies generic tools that can used in different applications.

3 Type System Formulation

3.1 The Type Lattice

A lattice is a partially ordered set in which every subset of elements has a least upper bound and a greatest lower bound [4]. This mathematical structure is used to represent the lossless type conversion relation in a type system. An example of a type lattice is shown in Fig.2. This particular lattice is constructed in the data package using the infrastructure of the graph package. In the diagram, type α is *greater than* type β if there is a path upwards from β to α. Thus, ComplexMatrix is greater than Int. Type α is *less than* type β if there is a path downwards from β to α. Thus, Int is less than ComplexMatrix. Otherwise, types α and β are *incomparable*. Complex and Long, for example, are incomparable. The top element, *General*, which is "the most general type," corresponds to the base token class; the bottom element, *NaT* (Not a Type), does not correspond to a token. Users can extend a type lattice by adding more types.

In the type lattice, a type can be losslessly converted to any type greater than it. For example, an integer can be losslessly converted to a double. Here, we assume an integer is 32 bits long and a double is 64 bits using the IEEE 754 floating point format, as in Java. This hierarchy is related to the inheritance hierarchy of the token classes in that a subclass is always less than its super class in the type lattice, but some adjacent types in the lattice are not related by inheritance. So this hierarchy is a combination of

the subtyping relation in object oriented languages, and ad hoc subtyping rules, such as *Int* ≤ *Double* [13]. Organizing types in a hierarchy is fairly standard. For example, Abelson and Sussman [1] organized the coercion relation among types in a hierarchy. However, they did not deliberately model the hierarchy as a lattice. Long ago, Hext [9] experimented with using a lattice to model the type conversion relation, but he was not working with an object oriented language and did not intend to support polymorphic system components. This work predates the popular use of those concepts.

Type conversion is done by a method convert() in the token classes. This method converts the argument into an instance of the class implementing this method. For example, DoubleToken.convert(Token token) converts the specified token into an instance of DoubleToken. The convert() method can convert any token immediately below it in the type hierarchy into an instance of its own class. If the argument is several levels down the type hierarchy, the convert() method recursively calls the convert() method one level below to do the conversion. If the argument is higher in the

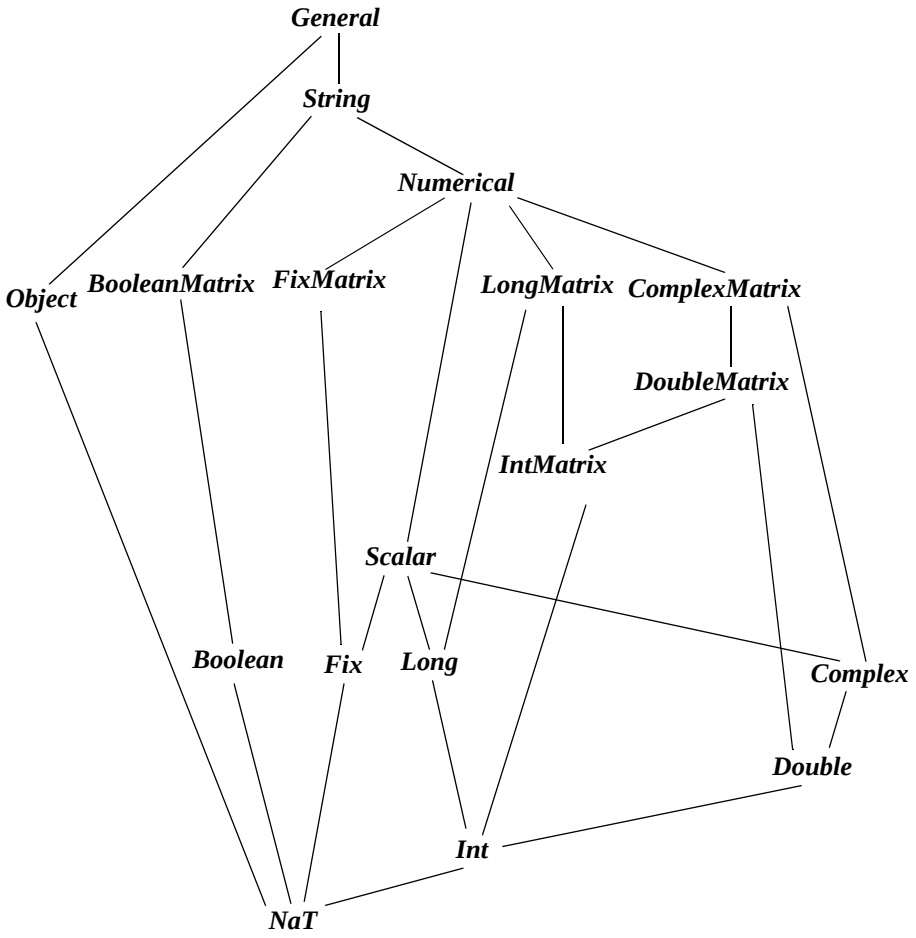

Fig. 2. An example of a type lattice.

type hierarchy, or is incomparable with its own class, convert() throws an exception. If the argument to convert() is already an instance of its own class, it is returned without any change.

3.2 Type Constraints

In Ptolemy II, to guarantee that information is not lost during data transfer, we require the type of a port that sends tokens to be the same as or lower than the type of the receiving port:

$$sendType \leq receiveType \tag{1}$$

If both the *sendType* and *receiveType* are declared, the static type checker simply checks whether this inequality is satisfied, and reports a type conflict if it is not.

In addition to the above constraint imposed by the topology, actors may also impose constraints among ports and parameters. For example, the Ramp actor in Ptolemy II, which is a source actor that produces a token on each execution with a value that is incremented by a specified step, stores the first output and the step value in two parameters. This actor will not declare the type of its port, but will specify the constraint that the port type is greater than or equal to the types of the two parameters. As another example, a polymorphic Distributor actor, which splits a single token stream into a set of streams, will specify the constraint that the type of a sending port is greater than or equal to that of the receiving port. This Distributor will be able to work on tokens of any type. In general, polymorphic actors need to describe the acceptable types through type constraints.

All the type constraints in Ptolemy II are described in the form of inequalities like the one in (1). If a port or a parameter has a declared type, its type appears as a constant in the inequalities. On the other hand, if a port or a parameter has an undeclared type, its type is represented by a type variable in the inequalities. The domain of the type variable is the elements of the type lattice. The type resolution algorithm resolves the undeclared types in the constraint set. If resolution is not possible, a type conflict error will be reported. As an example of a constraint set, consider Fig.3.

The port on actor A1 has declared type *Int*; the ports on A3 and A4 have declared type *Double*; and the ports on A2 have their types undeclared. Let the type variables for the undeclared types be α, β, and γ; the type constraints from the topology are:

$Int \leq \alpha$
$Double \leq \beta$
$\gamma \leq Double$

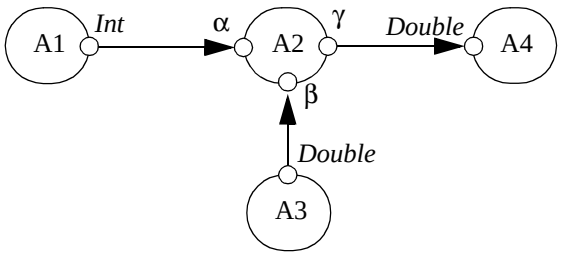

Fig. 3. A topology (interconnection of components) with types.

Now, assume A2 is a polymorphic adder, capable of doing addition for integer, double, and complex numbers. Then the type constraints for the adder can be written as:

$\alpha \leq \gamma$
$\beta \leq \gamma$
$\gamma \leq$ *Complex*

The first two inequalities constrain the precision of the addition result to be no less than that of the summands, the last one requires that the data on the adder ports can be converted to *Complex* losslessly. These six inequalities form the complete set of constraints and are used by the type resolution algorithm to solve for α, β, and γ.

This inequality formulation is inspired by the type inference algorithm in ML [12]. There, type equations are used to represent type constraints. In Ptolemy II, the lossless type conversion hierarchy naturally implies inequality relations among the types instead of equalities. In ML, the type constraints are generated from program constructs. In a heterogeneous graphical programming environment like Ptolemy II, where details of the components are hidden, the system does not have enough information about the function of the actors, so the actors must present their type information by either declaring the type on their port, or by specifying a set of type constraints to describe the acceptable types on the undeclared ports. The Ptolemy II system also generates type constraints based on (1).

This formulation converts type resolution into a problem of solving a set of inequalities defined over a finite lattice. An efficient algorithm for doing this is given by Rehof and Mogensen [15]. The appendix of this paper describes this algorithm through an example. Essentially, the algorithm starts by assigning all the type variables the bottom element of the type hierarchy, *NaT*, then repeatedly updating the variables to a greater element until all the constraints are satisfied, or until the algorithm finds that the set of constraints are not satisfiable. This process can be formulated as the search for the least fixed point of a monotonic function on the lattice. The least fixed point is the set of most specific types. It is unique [4], and satisfies the constraints if it is possible to satisfy the constraints.

If the set of type constraints are not satisfiable, or some type variables are resolved to *NaT*, the static type checker flags a type conflict error. The former case can happen, for example, if the port on actor A1 in figure Fig.3 has declared type *Complex*. The latter can happen if an actor does not specify any type constraints on an undeclared sending port. If the type constraints do not restrict a type variable to be greater than *NaT*, it will stay at *NaT* after resolution. To avoid this, any sending port must either have a declared type, or some constraints to force its type to be greater than *NaT*.

A solution satisfying the constraints may not be unique. In fact, the algorithm given in [15] can be used to find either the most specific solution (least in the lattice) or the most general solution (greatest in the lattice). The ML type inference algorithm finds the most general types for a given program, which allows maximal reuse of compiled code. In our case, multiple occurrences of an actor in a topology are treated as different actors, even though they specify the same set of type constraints, so we do not need to use the most general type. In fact, our choice of using the most specific types has a key advantage: types lower in the type lattice usually have a lower implementation cost. For example, in embedded system design, hardware is often synthesized

from a component-based description of a system. If a polymorphic adder is going be synthesized into hardware, and it receives *Int* tokens and sends the addition result to a *Double* port, our scheme will resolve the types of all the ports on the adder to *Int*, rather than *Double*. Using an integer adder will be more economical than a double adder. This is analogous to using types to generate more optimized code in compilers.

3.3 Run-time Type Checking and Lossless Type Conversion

The declared type is a contract between an actor and the Ptolemy II system. If an actor declares that a sending port has a certain type, it asserts that it will only send tokens whose types are less than or equal to that type. If an actor declares a receiving port to have a certain type, it requires the system to only send tokens that are instances of the class of that type to that port. Run-time type checking is the component in the system that enforces this contract. When a token is sent from a sending port, the run-time type checker finds its type, and compares it with the declared type of the port. If the type of the token is not less than or equal to the declared type, a run-time type error will be reported.

As discussed before, type conversion is needed when a token sent to a receiving port has a type less than the type of that port but is not an instance of the class of that type. Since this kind of lossless conversion is done automatically, an actor can safely cast a received token to the declared type. On the other hand, when an actor sends tokens, the tokens being sent do not have to have the exact declared type of the sending port. Any type that is less than the declared type is acceptable. For example, if a sending port has declared type *Double*, the actor can send IntToken from that port without having to convert it to a DoubleToken, since the conversion will be done by the system. So the automatic type conversion simplifies the input/output handling of the actors.

Note that even with the convenience provided by the type conversion, actors should still declare the receiving types to be the most general that they can handle and the sending types to be the most specific that includes all tokens they will send. This maximizes their applications. In the previous example, if the actor only sends IntToken, it should declare the sending type to be *Int* to allow the port to be connected to a receiving port with type *Int*.

If an actor has ports with undeclared types, its type constraints can be viewed as both a requirement and an assertion from the actor. The actor requires the resolved types to satisfy the constraints. Once the resolved types are found, they serve the role of declared types at run time. I.e., the type checking and type conversion system guarantees to only put tokens that are instances of the class of the resolved type to receiving ports, and the actor asserts to only send tokens whose types are less than or equal to the resolved type from sending ports.

3.4 Discussion of Type Resolution and Polymorphism

Rehof and Mogensen proved that their algorithm for solving inequality constraints is linear time in the number of occurrences of symbols in the constraints, which in our case, can be translated into linear time in the number of constraints. This makes type resolution very efficient. On the other hand, one might be tempted to extend the formu-

lation to achieve more flexibility in type specification. For example, it would be nice to introduce a *OR* relation among the constraints. This can be useful, in the case of a two-input adder, for specifying the constraint that the types of the two receiving ports are comparable. This constraint will prohibit tokens with incomparable types to be added. As shown in [15], this cannot be easily done. The inequality constraint problem belongs to the class of meet-closed problems. Meet-closed, in our case, means that if A and B are two solutions to the constraints, their greatest lower bound in the lattice is also a solution. This condition guarantees the existence of the least solution, if any solution exists at all. Introducing the OR relation would break the meet-closed property of the problem. Rehof and Mogensen also showed that any strict extension of the class of meet-closed problems solved by their algorithm will lead to an NP-complete problem. So far, the inequality formulation is generally sufficient for our purpose, but we are still exploring its limitations and workarounds.

We have been using the term polymorphic actor broadly to mean the actors that can work with multiple types on their ports. In [3], Cardelli and Wegner distinguished two broad kinds of polymorphism: universal and ad hoc polymorphism. Universal polymorphism is further divided into parametric and inclusion polymorphism. Parametric polymorphism is obtained when a function works uniformly on a range of types. Inclusion polymorphism appears in object oriented languages when a subclass can be used in place of a superclass. Ad hoc polymorphism is also further divided into overloading and coercion. In terms of implementation, a universally polymorphic function will usually execute the same code for different types, whereas an ad-hoc polymorphic functions will execute different code.

In an informal sense, Ptolemy II exhibits all of the above kinds of polymorphism. The Distributor actor, discussed in section 3.2 shows parametric polymorphism because it works with all types of tokens uniformly. If an actor declares its receiving type to be *General*, which is the type of the base token class, then that actor can accept any type of token since all the other token classes are derived from the base token class. This is inclusion polymorphism. The automatic type conversion during data transfer is a form of coercion; it allows an receiving port with type *Complex*, for example, to be connected to sending ports with type *Int*, *Double* or *Complex*. An interesting case is the arithmetic and logic operators, like the Add actor. In most languages, arithmetic operators are overloaded, but different languages handle overloading differently. In standard ML, overloading of arithmetic operators must be resolved at the point of appearance, but type variables ranging over equality types are allowed for the equality operator [16]. In Haskell, type classes are used to provide overloaded operations [8]. Ptolemy II takes advantage of data encapsulation. The token classes in Ptolemy II are not passive data containers, they are active data in the sense that they know how to do arithmetic operations with another token. This way, the Add actor can simply call the add() method of the tokens, and work consistently on tokens of different type. An advantage of this design is that users can develop new token types with their implementation for the add() method, achieving an effect similar to user defined operator overloading in C++.

4 Examples

This section provides two examples of type resolution in Ptolemy II.

4.1 Fork Connection

Consider two simple topologies in Fig.4. where a single sending port is connected to two receiving ports in Fig.4(a) and two sending ports are connected to a single receiving port in Fig.4(b). Denote the types of the ports by *a1, a2, a3, b1, b2, b3,* as indicated in the figure. Some possibilities for legal and illegal type assignments are:

- In Fig.4(a), if *a1 = Int, a2 = Double, a3 = Complex*. The topology is well typed. At run-time, the IntToken sent out from actor A1 will be converted to DoubleToken before transferred to A2, and converted to ComplexToken before transferred to A3. This shows that multiple ports with different types can be interconnected as long as the sender type can be losslessly converted to the receiver type.
- In Fig.4(b), if *b1 = Int*, b2 = *Double*, and *b3* is undeclared. The the resolved type for *b3* will be *Double*. If *b1 = Int* and *b2 = Boolean*, the resolved type for *b3* will be *String* since it is the lowest element in the type hierarchy that is higher than both *Int* and *Boolean*. In this case, if the actor B3 has some type constraints that require *b3* to be less than *String*, then type resolution is not possible, a type conflict will be signaled.

A Java applet that demonstrates the situation in Fig.4(b) and shows the type resolution process is available at the URL: *http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII0.3/ptII0.3/ptolemy/domains/sdf/demo/Type/Type.htm*

4.2 Mixing FSM and SDF

In [7], Girault, Lee and Lee showed how to mix finite-state machines (FSMs) with other concurrency models. For example FSM can be mixed with synchronous dataflow (SDF) [10], as shown in Fig.5. In this figure, the top of the hierarchy is an SDF system. The middle actor B in this system is refined to a FSM with two states, each of which is further refined to a SDF subsystem. One type constraint on the receiving port of B is that its type must be less than or equal to the types of both of the receiving ports of the SDF subsystems D and E, because tokens may be transported from the receiving port of B to the receiving ports of D or E. Assuming the types of the receiving ports on D and E are *Int* and *Double*, respectively, type resolution will resolve the type of the receiving port of B to *Int*. Similarly, a type constraint for the sending port of B is that

Fig. 4. Two simple topologies with types.

its type must be greater than or equal to the types of both of the sending ports of D and E, and its resolved type will be *Double*.

Note that this result is consistent with function subtyping [13]. If we consider the order in the type lattice as subtype relations, and the actors as functions, then D: *Int*→*Int*, E: *Double*→*Double*, and B: $\alpha \rightarrow \beta$ before type resolution. Since D and E can take the place of B during execution, their types should be considered as subtypes of the type of B. Since function subtyping is contravariant for function arguments and covariant for function results, the type α should be a subtype of *Int* and *Double* and β should be a super type of *Int* and *Double*. This is exactly what the type constraints specify, and the resulting type for B: *Int*→*Double* is indeed a supertype of both of the types of D and E.

5 Conclusion and Future Work

In the design of the Ptolemy II type system, we have taken the approach of polymorphic static typing combined with run-time type checking. We use a lattice structure to model the lossless type conversion relation and provide automatic type conversion during data transfer. Polymorphism is supported by allowing the system components to specify type constraints, and a linear time algorithm is used for constraint resolution. This type system increases the safety and usability of the component-based design environment, promotes component reuse, and helps with design optimization. The infrastructure is built to operate on any type lattice, and so can be used to experiment with extended type systems.

Currently, we are working on extending this system to support structured types such as array and record types. The goal is to allow the elements of arrays and records to contain tokens of arbitrary types, including structured types, and to be able to specify type constraints on them. One of the major difficulty with this extension is that the type lattice will become infinite, which raises questions on the convergence of type resolution.

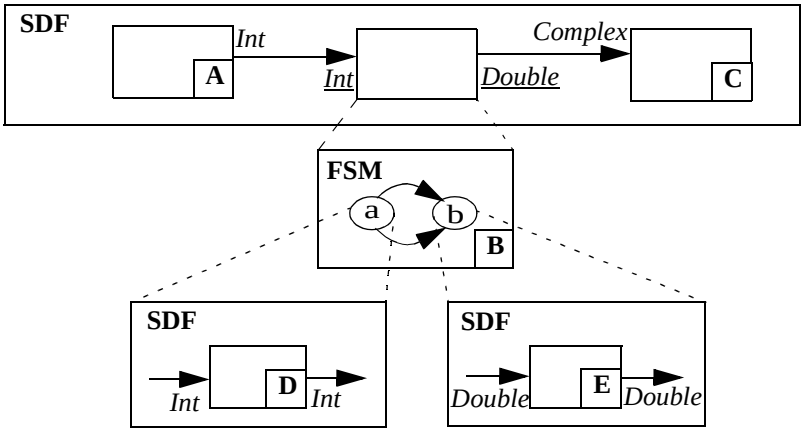

Fig. 5. Mixing FSM with SDF.

In the longer term, we will try to characterize the communication protocols used between system components, or some of the real-time properties of the system as types, and design a process-level type system to facilitate heterogeneous real-time modeling. This may potentially bring some of the benefit of data typing to the process level.

Acknowledgments

This work is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: The Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, Motorola, NEC, and Philips.

Appendix. The Type Resolution Algorithm

The type resolution algorithm starts by assigning all the type variables the bottom element of the type hierarchy, *NaT*, then repeatedly updating the variables to a greater element until all the constraints are satisfied, or until the algorithm finds that the set of constraints are not satisfiable. This iteration can be viewed as repeated evaluation of a monotonic function, and the solution is the least fixed point of the function. The kind of inequality constraints for which the algorithm can determine satisfiability are the ones with the greater term being a variable or a constant. By convention, we write inequalities with the lesser term on the left and the greater term on the right, as in $\alpha \leq \beta$, not $\beta \geq \alpha$. The algorithm allows the left side of the inequality to contain monotonic functions of the type variables, but not the right side. The first step of the algorithm is to divide the inequalities into two categories, *Cvar* and *Ccnst*. The inequalities in *Cvar* have a variable on the right side, and the inequalities in *Ccnst* have a constant on the right side. In the example of Fig.3, *Cvar* consists of:

$Int \leq \alpha$
$Double \leq \beta$
$\alpha \leq \gamma$
$\beta \leq \gamma$

And *Ccnst* consists of:

$\gamma \leq Double$
$\gamma \leq Complex$

The repeated evaluations are only done on *Cvar*, *Ccnst* are used as checks after the iteration is finished, as we will see later. Before the iteration, all the variables are assigned the value *NaT*, and *Cvar* looks like:

$Int \leq \alpha(NaT)$
$Double \leq \beta(NaT)$
$\alpha(NaT) \leq \gamma(NaT)$
$\beta(NaT) \leq \gamma(NaT)$

Where the current value of the variables are inside the parenthesis next to the variable.

At this point, *Cvar* is further divided into two sets: those inequalities that are not currently satisfied, and those that are satisfied:

Not-satisfied	*Satisfied*
$Int \le \alpha(NaT)$	$\alpha(NaT) \le \gamma(NaT)$
$Double \le \beta(NaT)$	$\beta(NaT) \le \gamma(NaT)$

Now comes the update step. The algorithm selects an arbitrary inequality from the *Not-satisfied* set, and forces it to be satisfied by assigning the variable on the right side the least upper bound of the values of both sides of the inequality. Assuming the algorithm selects $Int \le \alpha(NaT)$, then

$$\alpha = Int \vee NaT = Int \tag{2}$$

After α is updated, all the inequalities in *Cvar* containing it are inspected and are switched to either the *Satisfied* or *Not-satisfied* set, if they are not already in the appropriate set. In this example, after this step, *Cvar* is:

Not-satisfied	*Satisfied*
$Double \le \beta(NaT)$	$Int \le \alpha(Int)$
$\alpha(Int) \le \gamma(NaT)$	$\beta(NaT) \le \gamma(NaT)$

The update step is repeated until all the inequalities in *Cvar* are satisfied. In this example, β and γ will be updated and the solution is:

$\alpha = Int$, $\beta = \gamma = Double$

Note that there always exists a solution for *Cvar*. An obvious one is to assign all the variables to the top element, *General*, although this solution may not satisfy the constraints in *Ccnst*. The above iteration will find the least solution, or the set of most specific types.

After the iteration, the inequalities in *Ccnst* are checked based on the current value of the variables. If all of them are satisfied, a solution to the set of constraints is found.

As mentioned earlier, the iteration step can be seen as a search for the least fixed point of a monotonic function. In this view, the computation in (2) is the application of a monotonic function to type variables. Let L denote the type lattice. In an inequality $r \le \alpha$, where α is a variable, and r is either a variable or a constant, the update function f: $L^2 \rightarrow L$ is $\alpha' = f(r, \alpha) = r \vee \alpha$. Here, α represents the value of the variable before the update, and α' represents the value after the update. f can easily be seen to be monotonic and non-decreasing. And, since L is finite, it satisfies the ascending chain condition, so f is also continuous. Let the variables in the constraint set be $\alpha_1, \alpha_2, \ldots, \alpha_N$, where N is the total number of variables, and define $A = (\alpha_1, \alpha_2, \ldots, \alpha_N)$. The complete iteration can be viewed as repeated evaluation of a function F: $L^N \rightarrow L^N$ of A, where F is the composition of the individual update functions. Clearly, F is also continuous. The iteration starts with the variables initialized to the bottom, $A = \perp^N$, and computes the sequence $F^i(\perp^N)$ $(i \ge 0)$, which is a non-decreasing chain. By the fixed point theorem in [4], the least upper bound of this chain is the least fixed point of F, corresponding to the most specific types in our case.

Rehof and Mogensen [15] proved that the above algorithm is linear time in the number of occurrences of symbols in the constraints, and gave an upper bound on the number of basic computations. In our formulation, the symbols are type constants and type variables, and each constraint contains two symbols. So the type resolution algorithm is linear in the number of constraints.

References

1. H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, 1985.

2. F. Arbab, *MANIFOLD Version 2.0*, CWI, Software Engineering Cluster, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, June, 1998.

3. L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, Vol.17, No.4, Dec. 1985.

4. B. A. Davey and H. A. Priestly, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

5. J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay and Y. Xiong, *Overview of the Ptolemy Project*, ERL Technical Report UCB/ERL No. M99/37, Dept. EECS, University of California, Berkeley, CA 94720, July 1999. (http://ptolemy.eecs.berkeley.edu/publications/papers/99/HMAD/)

6. Y-C. Fuh and P. Mishra, "Type Inference with Subtypes," *Second European Symposium on Programming*, Nancy, France, 1988.

7. 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.

8. C. V. Hall, K. Hammond, S.L. Peyton Jones, and P. L. Wadler, "Type Classes in Haskell," *ACM Transactions on Programming Languages*, Vol.18, No.2, Mar. 1996.

9. J. B. Hext, "Compile-Time Type-Matching," *Computer Journal*, 9, 1967.

10. E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transaction on Computer*, Jan. 1987.

11. J. Liu, B. Wu, X. Liu, and E. A. Lee, "Interoperation of Heterogeneous CAD Tools in Ptolemy II," *Symposium on Design, Test, and Microfabrication of MEMS/MOEMS*, Paris, France, Mar. 1999.

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

13. J. C. Mitchell, *Foundations for Programming Languages*, The MIT Press, 1998.

14. M. Odersky, "Challenges in Type Systems Research," *ACM Computing Surveys*, Vol.28, No.4es, 1996.

15. J. Rehof and T. Mogensen, "Tractable Constraints in Finite Semilattices," *Third International Static Analysis Symposium*, LNCS 1145, Springer, Sept., 1996.

16. J. D. Ullman, *Elements of ML Programming*, Prentice Hall, 1998.