The Design and Application of Structured Types in Ptolemy II

Xiaojun Liu
Lizhi C. Zhong
Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-21
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-21.html

January 30, 2007

The Design and Application of Structured Types in Ptolemy II

Yang Zhao, Yuhong Xiong, Edward Lee, *Fellow, IEEE,* Xiaojun Liu, and Lizhi C. Zhong

***Abstract*— Ptolemy II is a component-based design and modeling environment. It has a polymorphic type system that supports both base types and structured types, such as arrays, records and unions. This paper presents the extensions to the base type system that support structured types. In the base type system, all the types are organized into a type lattice, and type constraints in the form of inequalities can be solved efficiently over the lattice. We take a hierarchical and granular approach to add structured types to the lattice, and extend the format of inequality constraints to allow arbitrary nesting of structured types. We also analyze the convergence of the constraint solving algorithm on an infinite lattice after structured types are added. To show the application of structured types, we present two Ptolemy II models that have direct real-world background. The first one describes the workflow of a charity organization, and the second one implements part of the IEEE 802.11 specifications. These models make extensive use of record and union types to represent structured information.**

***Index Terms*— IEEE 802.11, lattice, structured types, type system.**

I. Introduction

EMBEDDED systems have become ubiquitous. They can be found in cars, consumer electronics, appliances, networking equipment, aircraft, security systems, etc. Due to the complexity of modern systems, their design poses many challenges [6]. In recent years, component-based design has been established as an important approach in coping with the complexity. By dividing a complex system into a set of interacting components, the design problem is converted to the design of individual components and their interaction model. This divide and conquer approach is consistent with the abstract model of granular computing [15] and can be viewed as one of its embodiments. In granular computing, granules reside at different levels and a granule at a higher level can be decomposed into multiple granules at a lower level. Similarly, in component-based design, components can be described at different levels of abstraction, and a component modeled at a high level can be refined into multiple components at a lower level. This hierarchical decomposition naturally supports both the top-down and bottom-up design approaches.

A good type system is very important for component-based design. By detecting mismatches at component interfaces and ensuring compatibility, a type system can greatly increase the robustness of a system. This is more valuable for embedded software systems as they are generally held to a much higher reliability standard than general purpose software [6]. In addition, polymorphic type systems can increase the flexibility of the design environment by allowing a component to have more than one type, so that it can be reused in different settings.

Ptolemy II is a component-based design and modeling environment [3]. It supports hierarchical design decomposition, provides many models for component interaction, and includes a visual interface for model construction and execution. Ptolemy II is developed in Java, and it has a polymorphic type system that supports both the base types, such as integers and floating point numbers, and structured types, such as arrays and records. In the base type system, which was reported in [14], all the types are organized into a lattice to model the subtyping relation among types. Type constraints in components and across components are described as inequalities over the lattice. These inequality constraints can be solved efficiently, with existence and uniqueness of a solution guaranteed by fixed-point theorems.

This paper describes the extensions to support structured types, such as arrays, records and unions. In [13], we described our design for array and record types. This paper adds the union type, and discusses the design of structured types in more detail. There are some interesting technical challenges when structured types are introduced. In particular, the type lattice becomes infinite, and an extension on the format of the inequality constraints is required. We present an analysis on the issue of convergence on an infinite lattice, and add an unification step in the constraint solving algorithm to handle the new inequality format. Our extension allows structured types to be arbitrarily nested, and supports type constraints that involve the elements of structured types.

The rest of the paper is organized as follows. Section II reviews the Ptolemy II base type system. Section III describes the technical problems and solutions for adding structured types. Section IV presents two applications that use structured types. Section V concludes the paper and suggests future work.

II. Ptolemy II Base Type System

We review the base type system in Ptolemy II below. In addition to the material drawn from [14], we put the system into a larger context by relating it to set constraints and rough set theory.

A. Actor-Oriented Design in Ptolemy II

Ptolemy II is an actor-oriented language [7]. In Ptolemy II, components are actors, and the interface to actors consists

Manuscript received March 15, 2006. This paper describes work that is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from NSF and the following companies: Agilent, General Motors, Hewlett-Packard, Honeywell, Infineon, Samsung, and Toyota.

Y. Xiong is with the HP Laboratories, Palo Alto, CA 94304 USA (phone: 650-236-4741; e-mail: yuhong.xiong@hp.com).

E. Lee, X. Liu and Y. Zhao are with the EECS Department, University of California at Berkeley, Berkeley, CA 94720 USA (e-mail: {eal,liuxj,ellen_zh}@eecs.berkeley.edu).

L. Zhong is with the LSI logic Corporation (e-mail: czhong@yahoo.com).

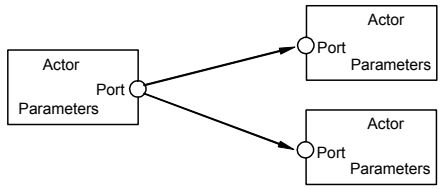

Fig. 1. An abstract representation of a Ptolemy II model.

of parameters and ports. Figure 1 shows a simplified graph representation of a Ptolemy II model. Here, we ignore the detailed interaction semantics and simply assume that actors communicate with one another through message passing. Messages are encapsulated in *tokens*. The ports that send out tokens are called output or sending ports, and the ports that receive tokens are called input or receiving ports.

To express the typing requirements inside the actors and across connections, each port and each parameter is given a type. The type constraint can be set up based on the topology of the block diagram and the specification of actors. Static type checking can be performed before the model executes to check whether the components can work together as connected. Run time type checking, which can be performed when a token is sent from a port, is also necessary to ensure type safety. 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.

B. Type Lattice

To support polymorphic typing of actors and automatic run-time type conversion, all the base types of Ptolemy II are organized into a *type lattice*, as shown in figure 2. The ordering relation in the lattice is a combination of the lossless type conversion relation among primitive types, and the subclass relation of the Java classes that implement those types. Since the type conversion relation among primitive types can be viewed as ad hoc subtyping [8], we can say that the relation in the type lattice represents two kinds of subtyping relations.

C. Type Constraints and Type Resolution

As mentioned earlier, each port has a type. The type of the port restricts the type of the token that can pass through it. These types can be declared by the actor writer, or left undeclared for polymorphic actors. Undeclared types are denoted by type variables, and they are solved during type resolution.

In a model, the interconnection of components naturally implies type constraints. In particular, the type of a sending port must be the same or less than the type of the connected receiving port:

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

This means that the sending port can send out tokens of the same type as *receiveType*, or a subtype.

In addition to the above constraint imposed by topology, actors may also impose constraints. For example, an actor may

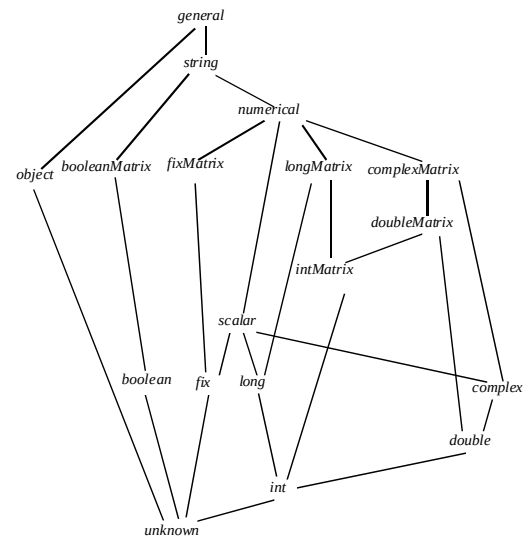

Fig. 2. An example of a type lattice.

specify that the type of a port is no less than the type of a parameter. In general, polymorphic actors need to describe the acceptable types through type constraints.

All the type constraints are described in the form of inequalities like the one in (1). For example, the set of constraints for the model in figure 3 is:

$$\begin{gathered} int \leq \alpha \\ double \leq \beta \\ \gamma \leq double \\ \alpha \leq \gamma \\ \beta \leq \gamma \\ \gamma \leq complex \end{gathered}$$

The first three inequalities are constraints from the topology, the last three are from the adder. This adder is polymorphic, capable of doing addition for integer, double, and complex numbers.

The inequality constraints can be solved by a linear-time algorithm given by Rehof and Mogensen [10]. This algorithm is an iterative procedure. It starts by assigning all the type variables the bottom element of the type hierarchy, *unknown*, 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 least fixed point is the set of most specific types.

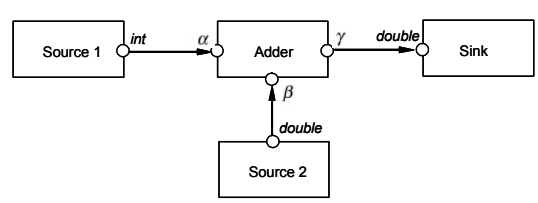

Fig. 3. A topology (interconnection of components) with types.

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 $\beta \leq \alpha$. 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 figure 3, *Cvar* consists of:

$$\begin{gathered} int \leq \alpha \\ double \leq \beta \\ \alpha \leq \gamma \\ \beta \leq \gamma \end{gathered}$$

And *Ccnst* consists of:

$$\begin{gathered} \gamma \leq double \\ \gamma \leq complex \end{gathered}$$

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 *unknown*, and *Cvar* looks like:

$$\begin{gathered} int \leq \alpha(unknown) \\ double \leq \beta(unknown) \\ \alpha(unknown) \leq \gamma(unknown) \\ \beta(unknown) \leq \gamma(unknown) \end{gathered}$$

where the current values of the variables are inside the parentheses 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
$int \leq \alpha(unknown)$	$\alpha(unknown) \leq \gamma(unknown)$
$double \leq \beta(unknown)$	$\beta(unknown) \leq \gamma(unknown)$

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 $int \leq \alpha(unknown)$, then:

$$\alpha = int \vee unknown = int$$

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
$\alpha(int) \leq \gamma(unknown)$	$int \leq \alpha(int)$
$double \leq \beta(unknown)$	$\beta(unknown) \leq \gamma(unknown)$

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 for the set of constraints is found.

D. Discussion

In addition to the simple inequalities that involve only the constant types and type variables, the type resolution algorithm also admits monotonic functions on the left hand side of the inequality:

$$f(\alpha) \leq \alpha \tag{2}$$

Monotonic functions are order preserving. That is, $\alpha \leq \beta \Rightarrow f(\alpha) \leq f(\beta)$. They can be used to express complicated type constraints. Examples can be found in [12].

The inequality type constraints can be generalized to set constraints [1]. Set constraints are more expressive, but the resolution algorithm is more expensive, and is often exponential depending on the type of set operations admitted [1]. We have found that inequalities are generally enough to express the desired type constraints, particularly when augmented with monotonic functions. Therefore, the lattice formulation represents a good trade-off between expressiveness and computation cost.

The inequality type constraints can also be related to rough set theory [9]. The inequalities $Const \leq \alpha$ or $\alpha \leq Const$, where $Const$ is a constant type, define either a lower bound or an upper bound of the set of acceptable types for the type variable α, so we can view the set of acceptable types as a rough set. When the type of a port is resolved to a specific type, say *string*, it does not mean that the tokens sent to that port have to contain a string. In fact, they can also be any token whose type is less than *string* in the type lattice. Therefore, type resolution is a process of refining the initial rough set to a specific set. In Ptolemy II, if an input port receives a token with a type less than the type of the port, a run-time type system will convert the token to the type of the port automatically. This conversion makes the system easier to use.

III. Structured Types

A. Goals and Problems

Structured types are very useful for organizing related data and making programs more readable. In a block diagram based design environment, we want to support tokens that contain structured data, such as array tokens and record tokens. For example, a record type is a named collection of types, like the structure in the C language. It can be used to bundle multiple pieces of information in one token and transfer them in one round of communication, making the execution more efficient. In addition, it can be used to reduce the number of ports on certain actors, which simplifies the topology of the block diagram.

Another very useful structured type is the union type, which allows the user to create a token that can hold data of various types, but only one at a time. This is like the union construct in C. Union types are also called variant types in type system literature [4].

In our system, the elements of structured tokens are also tokens. For example, an integer array token contains an array of integer tokens. This allows structured types to be arbitrarily nested, so we can have, for instance, an array of arrays, or records containing arrays. Another desired feature for structured types is to be able to set up type constraints between the element type and the type of another object in the system. For example, we want to be able to specify that the element type of an array is no less than the type of a certain port.

To support these features in the framework of our type system, we need to answer the following questions:

- Ordering relation. What is the ordering relation among various structured types?
- Type constraints on structured types. Can the simple format of inequalities express type constrains on structured types? If not, how can we extend the format to do so?
- Infinite lattice. Since the element type of structured types can be arbitrary, the type lattice becomes infinite. Will type resolution always converge on this infinite lattice? If not, can we detect and handle the cases that do not converge?

The rest of this section will answer these questions for array, record and union types. To express the values and types of structured data, we will use the syntax of the expression language of Ptolemy II. In this syntax, structured values and types are enclosed in braces, elements are separated by comma, and the equal sign is used to link the label with the element type or value of record types or union types. For example:

- $\{1.4, 3.5\}$: An array containing two double values.
- $\{double\}$: The type of the above array.
- $\{\{1, 2\}, \{3, 4\}\}$: An array of arrays.
- $\{\{int\}\}$: The type of the above array.
- $\{name = \texttt{"}xyz\texttt{"}, value = 1\}$: A record with two fields. One has label *name* and string value *xyz*, the other has label *value* and integer value `1`.
- $\{name = string, value = int\}$: The type of the above record.
- $\{|name = \texttt{"}xyz\texttt{"}, value = 1|\}$: A union with two alternative fields. One has label *name* and string value `xyz`, the other has label *value* and integer value `1`. Since a union value can only hold one field at a time, we define the value of the above expression to be the first field in Ptolemy II. That is, this expression evaluates to a union value with label *name* and string value `xyz`.
- $\{|name = string, value = int|\}$: The type of the above union.

B. Ordering Relation

In most general purpose languages, arrays are mutable. That is, they can be modified after construction. It is known in type system research that subtyping for mutable arrays cannot be defined to allow compile-time type checking to find all type errors (see [12] and the references therein). There are two ways to define subtyping for arrays. In the covariant definition, $\{\tau_1\} \leq \{\tau_2\}$ if $\tau_1 \leq \tau_2$. In the contravariant definition, $\{\tau_1\} \leq \{\tau_2\}$ if $\tau_2 \leq \tau_1$. In either case, programs can be written on mutable arrays to cause type errors that the compiler cannot detect.

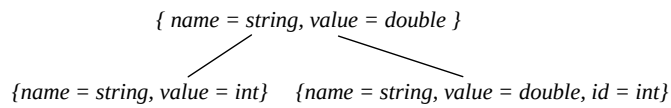

Fig. 4. Subtyping of record types

One way to obtain subtyping for arrays is to use a run-time check, as is done in Java. Java arrays are covariant. For example, a reference for *Object* array can point to a *string* array. To ensure type safety, Java performs run-time checking when the array elements are set, and the exception `java.lang.ArrayStoreException` is thrown if the check fails.

Another way to obtain array subtyping, which we have adopted, is to make arrays immutable. This restriction is usually not acceptable for general purpose imperative languages. But for actor-oriented languages, making the arrays immutable is justifiable, or even desirable. In our case, array tokens are mostly used for passing messages between actors. As a message carrier, we usually do not need to modify the contents of the array. Furthermore, if arrays are mutable, when we send an array token to multiple actors, we will need to make copies of the array and send each receiving actor a new copy. This incurs significant performance penalty. Without copying, multiple actors will share the same mutable array and the modification by one actor will affect the operation of the others. This is analogous to the use of global variables in programming, which is regarded as one of the main sources of program errors, particularly in concurrent software. In fact, this problem is not only limited to arrays, it applies to any type of token. Because of this, most tokens in Ptolemy II, including array, record and union tokens, are immutable. For immutable arrays, we define subtyping in a covariant way in the type lattice. That is, if $\tau_1 \leq \tau_2$, then $\{\tau_1\} \leq \{\tau_2\}$.

There are two kinds of subtyping relations among record types, depth subtyping and width subtyping. In depth subtyping, the element types of a sub record type are subtypes of the corresponding elements in the super record type. For example, $\{name = string, value = int\} \leq \{name = string, value = double\}$. In width subtyping, a longer record is a subtype of a shorter one. For example, $\{name = string, value = double, id = int\} \leq \{name = string, value = double\}$. Figure 4 shows the two subtyping relations.

Subtyping among union types is also defined by depth subtyping and width subtyping. The depth subtyping for union types is similar to that of record types, i.e. the element types of a sub union type are subtypes of the corresponding elements in the super union type. However, the width subtyping relation

for union types is opposite to that of record types. That is, a shorter union is a subtype of a longer one. An example of the union subtyping is shown in figure 5, where $\{|a = string, b = int|\} \leq \{|a = string, b = double|\}$ based on depth subtyping, and $\{|a = string, b = int|\} \leq \{|a = string, b = int, c = boolean|\}$ based on width subtyping.

To add structured types to the type lattice, we take a hierarchical and granular approach by treating each kind of structured type as a very coarse granule that contains an infinite number of types of that kind. For example, the array type granule contains an infinite number of array types. Figure 6 shows the organization of the type lattice of Ptolemy II after adding array, record and union types. These granules are incomparable in the type lattice, For example, any array type is incomparable with any record type. All the structured types are less than the type *general* and greater than *unknown*, but they are not comparable with other base types.

C. Inequality Constraints

The inequality solving algorithm we described in the last section admits definite inequalities, which are the ones having the following form:

$$\begin{matrix} Const \\ \alpha \\ f(\alpha) \end{matrix} \quad \leq \quad \begin{matrix} Const \\ \alpha \end{matrix}$$

That is, the left hand side of the inequality can be a constant, a variable, or a monotonic function, and the right hand side can be either a constant or a variable. Notice that the right hand side cannot be a function. This is because during type resolution based on the algorithm of Rehof and Mogensen [10], we need to update the right hand side to the least upper bound of both sides, and in general, we cannot update the value of a function to an arbitrary value.

When structured types are added, we may have inequality constraints with the right hand side being a variable structured type, such as:

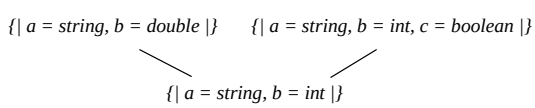

Fig. 5. Subtyping of union types

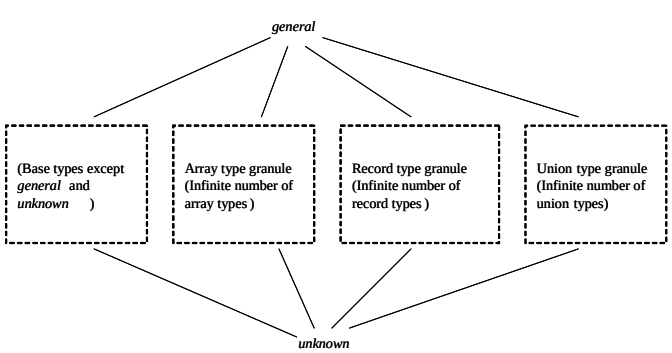

Fig. 6. The type lattice of Ptolemy II with array, record and union types added.

$$\tau \leq \{\alpha\}$$

In this inequality, the right hand side is neither a constant nor a simple variable. It can be viewed as a function that takes α and returns an array type $\{\alpha\}$:

$$f : \alpha \rightarrow \{\alpha\}$$

Strictly speaking, this inequality cannot be admitted by the algorithm of Rehof and Mogensen. However, it is possible to extend this algorithm if the function satisfies a certain property. During type resolution, we may need to compute this function in two directions. In the forward direction, we update the result of the function to a new array type when the variable α is updated. In the reverse direction, we compute the least upper bound of both sides, and update α such that the result of the function equals to the least upper bound. To be able to perform these updates, it is sufficient for the function to be a bijection with known domain and range. In the above example, the domain of the function is all the types, and the range is array types. It is interesting that this is a bijection even though $arraytypes \subset types$. This means that the two sets, $types$ and $arraytypes$ have the same cardinality. This is analogous to Cantor's result that the set of integers and the set of rationals have the same cardinality even though the set of integers is a strict subset of the set of rationals.

The reverse update is conceptually a process of unification [11] for the right hand side of the inequality and the least upper bound of the two sides. In the above inequality, if τ is $\{int\}$ and the current value of α is *unknown*, the least upper bound of both sides is $\{int\} \vee \{unknown\} = \{int\}$. So we need to unify $\{\alpha\}$ with $\{int\}$ by updating α to int. For this unification to succeed, the least upper bound must be a substitution instance of the right hand side variable structured type. That is, we must be able to obtain the least upper bound by substituting the variables on the right hand side with type expressions. If the function is a bijection and the least upper bound is in the range of the function, this unification is possible. If the least upper bound is not in the range of the function, we have a type conflict in the model.

D. Infinite Lattice

After structured types are added, the type lattice becomes infinite. Type resolution on this lattice, unfortunately, does not always converge. To see this, let's look at a simplified lattice, with only array types added, and include only seven base types: *general*, *string*, *boolean*, *long*, *double*, *int*, and *unknown*. This lattice is shown in figure 7.

Notice that there is an infinite chain in this lattice:

$$unknown, \{unknown\}, \{\{unknown\}\}, ...$$

This chain may cause problem in type resolution. For example, if we try to solve the inequality $\{\alpha\} \leq \alpha$, we will encounter an infinite iteration:

$$\begin{gathered} \{unknown\} \leq unknown \\ \{\{unknown\}\} \leq \{unknown\} \\ \{\{\{unknown\}\}\} \leq \{\{unknown\}\} \end{gathered}$$

This kind of infinite iteration can be detected. Observe that:

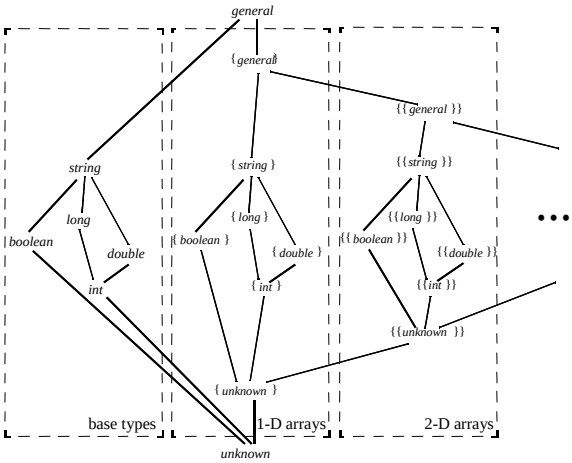

Fig. 7. An example of a type lattice with arrays.

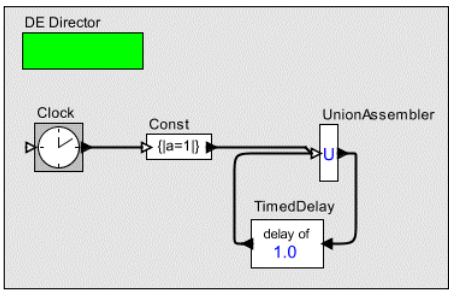

Fig. 8. A model showing that type resolution does not always converge.

- The infinite iteration only happens along the chain that involves *unknown*.
- From any type not including *unknown* as an element, all chains to the top of the lattice have finite length.

These two conditions are true not only after array types are added, but also true after record types are added. According to the subtyping rules for records, a super record type cannot have more fields than a sub record type, so any upward chain starting from a record type that does not involve *unknown* will have a finite number of elements before reaching the top of the lattice.

If we want to detect the infinite iteration shown above, we can simply set a bound on the depth of structured types that contain *unknown*. The depth of a structured type is the number of times a structured type contains other structured types. For example, an array of arrays has depth 2, and an array of arrays of records has depth 3. By setting the bound to a large enough number, say 100, the infinite iterations can be detected in practice.

However, these two conditions will not hold after union types are added, since the width subtyping relation for union types is different from record types, i.e. a super union type may have more fields than a sub union type. This means that there are infinite number of types from a particular union type to the top of the type lattice.

To show this, we give a concrete example as shown in figure 8. In this example, the *UnionAssembler* actor is polymorphic, and its input type is left undeclared. We can denote this input type by a type variable α. Given some input type α and the name of the input port, say b, the *UnionAssembler* actor outputs data with union type $\{|b = \alpha|\}$. The output port of the *Const* actor is a union type of $\{|a = int|\}$. The input and output type of the delay actor is the same. After a little bit of simplification, the type constraint of this model can be represented as:

$$\begin{aligned} & f(\alpha) = \{|b = \alpha|\} \\ & f(\alpha) \leq \alpha \\ & \{|a = int|\} \leq \alpha \end{aligned} \tag{3}$$

When we apply the algorithm discussed in section III-C, we get the following iterations:

$$\begin{aligned} & \alpha_1 = \{|a = int|\} \\ & f(\alpha_1) = \{|b = \{|a = int|\}|\} \\ & \alpha_2 = \{|a = int, b = \{|a = int|\}|\} \\ & f(\alpha_2) = \{|b = \{|a = int, b = \{|a = int|\}|\}|\} \\ & \alpha_3 = \{|a = int, b = \{|a = int, b = \{|a = int|\}|\}|\} \end{aligned} \tag{4}$$

As we can see, it will not converge, and there are infinite number of types between the particular union type $\{|a = int|\}$ and the top of the type lattice *general*.

Fortunately, in practice we can still apply the same technique used for detecting infinite iterations of array and record types. We can set a bound on the width of union types, which is the total number of fields a union type has, and the infinite iterations due to width subtyping of union types can then be detected based on this bound.

IV. Application

We have implemented the array, record and union types in Ptolemy II, and a set of actors using these types. Our implementation supports type checking on hierarchical models. The implementation details can be found in [3] [12]. These structured types are widely used in the modeling of the systems requiring component-based design. In the following, we first discuss a simple "proof of concept" model to demonstrate how record and union types can be used in Ptolemy II. Then we present a more practical application, a network protocol, that intensively uses record types and union types for the communications between blocks.

A. A Simple Example

The demo in figure 9 describes the workflow of a charity organization as a Ptolemy II model. This charity accepts various kinds of donations, such as clothing, car, or cash. It realizes the value of non-cash donations by selling them in stores, and adds up all the income to compute the total revenue. The *Clothing Donation*, *Car Donation* and *Money Donation* components model the above three kind of donations. Information about each donation is represented as a record token. The type of the clothing donation is $\{category = int, kind = int, size = int\}$, the type of the car donation is $\{kind = int, model = string, year = int\}$, and the type of the cash donation is $\{amount = double\}$. The *UnionAssembler* actor combines

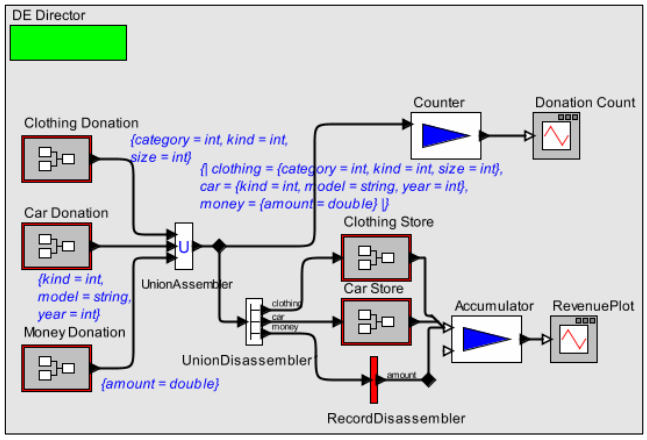

Fig. 9. Modeling the workflow of a charity organization using record and union types.

these three record types to a union type $\{|clothing = \{category = int, kind = int, size = int\}, car = \{kind = int, model = string, year = int\}, money = \{amount = double\}|\}$. The *UnionDisassembler* separates the donations and sends the value of the union token to the corresponding output whose name matches its field name. For example, the value $\{category = 1, kind = 0, size = 4\}$ in a union token $\{|clothing = \{category = 1, kind = 0, size = 4\}|\}$ will be sent to the output port named cloth. The *Clothing Store* actor and the *Car Store* actor model the stores that sell clothes or cars, and output the amount of money each cloth or car is sold for. The *RecordDisassembler* actor extracts the cash value number from the record token representing cash donation. The *Accumulator* computes the revenue of the charity by adding up the cash values of all the donations. This revenue is plotted by the *Revenue Plot* actor. The other plotter in the model, *Donation Count* plots the total number of donations received, regardless of its type.

In this example, the use of record and union types makes the model much easier to understand. In addition, this model also demonstrates how types propagate through record and union composition and decomposition. Several actors in the model, such as the *UnionAssembler* and *UnionDisassembler*, are polymorphic, i.e. their types are undeclared in the actor code. After type resolution, the type of each port is automatically resolved. Figure 9 shows the types of the input and output ports of the *UnionAssembler* actor after type resolution.

Another advantage of using union types is that the model becomes more scalable. For example, if the charity decides to accept a new kind of donation, say boats, we can just add a *Boat Donation* and a *Boat Store* actor in the model. We do not need to change the part of the model that counts the total number of donations, nor the Accumulator that computes the total revenue.

B. Modeling Wireless Protocol

The model shown in figure 10 is based on the IEEE 802.11 media access control (MAC) and physical (PHY) specifications [2], which defines an over-the-air interface between a wireless client and an access point (base station) or between two wireless clients. The physical layer model deals with receiving and transmitting messages, sensing medium status, and detecting collisions. The MAC layer model implements the distributed coordinator function (DCF), including carrier-sense multiple access with collision avoidance (CSMA/CA), inter frame space (IFS), persistent back offs, and exponential back off, to coordinate with other nodes in sharing the communication medium. Before transmission, a node first senses the medium to determine whether another node is transmitting. The transmission will proceed if the medium is idle, otherwise the node defers its transmission and starts back off by randomly choosing the amount of time the node will wait before trying to transmit again. To reduce congestion, the size of the back off window is increased exponentially if a packet transmission has failed. IFS specifies the duration between contiguous frame sequences, and a node needs to ensure the medium is idle for this required amount of time before transmission. IFS can be used to assign priorities to different packets: the shorter IFS, the higher priority. A request to send (RTS) message and clear to send (CTS) message are used to implement the virtual CSMA. A node initiating the packet transmission will send a RTS message to its destination. The destination node will respond with the CTS message if it detects that the medium is idle. All other nodes that hear the RTS or CTS message should defer their transmission. This way, the probability of having hidden nodes can be greatly reduced.

The MAC model is decomposed into four blocks (modeled as composite actors in Ptolemy II), each of which is further decomposed into several processes (modeled as actors). Figure 10 shows the model at different levels. Every process is essentially a finite state machine. They interact with one another by the messages, each of which has a collection of fields of various types. Record types are the natural choice for representing the messages. They make the execution more efficient and the block diagram more readable. Without record types, an actor needs to either use a string to encode the fields and values, or provide a different port for each field. The first choice loses type information for type checking, and requires the receiving actor to parse the received string to get the fields, which can be time consuming. The second choice will significantly increase the number of ports and connections in the model, reducing readability, and complicating actor design as the designer needs to effectively manage tokens from multiple ports in order to process one message. In figure 10, the *ChannelState* actor would need 9 ports without record types. The other two actors in the *Reception* block would suffer more since some of their ports have nested record types, which will require more ports to represent separately. One can imagine how messy the graph will become with about 40 connections and how difficult the actor design can be when the designer has to manage tokens from more than a dozen ports.

One can see from figure 10 the use of record types really makes the modeling of IEEE 802.11 MAC easier. The bottom portion of Figure 10 shows the record types on each connection in the *Reception* block. For instance, the

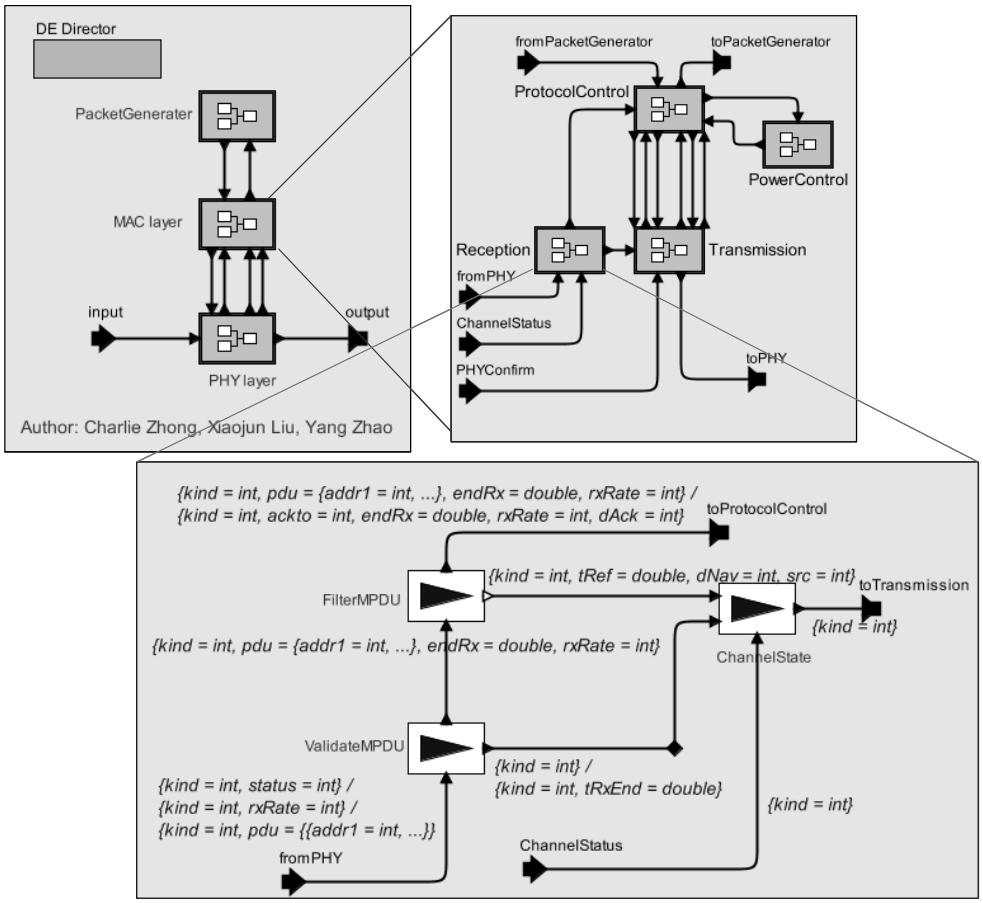

Fig. 10. An wireless protocol model using record types

ValidateMPDU actor in this block gets messages from the physical layer model through the fromPHY port. One of the messages is represented as a record type: $\{kind = int, pdu\}$, where *kind* is the message type and *pdu* field is another record type. The protocol data unit (pdu) can be of any data type. In the case of a RTS packet, it is represented as: $\{addr1 = int, addr2 = int, length = int\}$, where the *addr1* field encodes the address of the sending node, *addr2* encodes the address of the destination node, and *length* field is the length of the packet. In this model, several connections have more than one record type, which means that the messages have a number of different formats. For these messages, union types could be used to encode them easily. For example, the *PHY layer* component in this model can define the type of its fromPHY output to be a union type of $\{|m1 = \{kind = int, status = int\}, m2 = \{kind = int, rxRate = int\}, m3 = \{kind = int, pdu = \{addr1 = int, ...\}\}|\}$. The *ValidateMPDU* actor has its input type undeclared, which will be resolved by type resolution to the same union type as the type of the fromPHY port.

V. Conclusion

We have presented some extensions to the Ptolemy II base type system to support structured types, including array, record and union. In particular, we extended the format of inequality constraints to admit variable structured types, and added a unification step in the constraint solving algorithm to handle these types. We have also analyzed the convergence of the type resolution algorithm on an infinite type lattice and proposed a method to detect rare non-convergent cases. Structured types provide more flexibility and clarity in modeling. We demonstrated their usefulness though two Ptolemy II applications that model a business process and an engineering system. More demos using structured types are available by downloading the Ptolemy II software (open source).

In the type system literature, a union is also called a variant type, formed from an unordered set of labeled types [5]. We adopted this kind of union type in the design of union types in Ptolemy II because it is relatively straightforward to add the type granule of this kind of union types to the current type lattice. Later we noticed that there is some disadvantage with the labeled union types; specifically, it makes the actor library design more complex. Since union types are not comparable with base types and other structured types, we need special actors in the library to mediate the data interaction between union types and other types. For example, in figure 9, the *UnionAssembler* actor is needed to combine the three record types to a union type, and the *UnionDisassembler* actor is needed to disassemble the union type to different record types before we can send the data to the down-stream actors. Thus, we need a new actor library, including some domain-specific actors, to provide usable functionality with union

types. This complication of actor design with labeled union types motivated us to think about union types without labels. This kind of union types is formed from an unordered set of types, and is denoted by a sequence of types, separated by commas and enclosed in $\{| \ |\}$. For example, a union type can be defined as $\{|int, boolean|\}$, and a value of this type can either be an integer (e.g. 2), or a boolean (e.g. $true$). With unlabeled union types, we can simply broaden the semantics of multiports (ports that accept an indeterminate number of connections) so that if they are given distinct types, then they resolve to a union of these type. All polymorphic actors in the library suddenly work with union types without any changes to the actor code and without the addition of special actors. We are currently exploring the design of this kind of union types in Ptolemy II.

REFERENCES

[1] A. Aiken. Set constraints: Results, applications and future directions. In *Proceedings of the Second Workshop on the Principles and Practice of Constraint Programming*, May 1994.

[2] I.-S. S. Board. ANSI/IEEE std 802.11, part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. Technical report, 1999 Edition.

[3] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java: Volume 1: Introduction to Ptolemy II. Technical Report Technical Memorandum UCB/ERL M04/27, University of California, July 29 2004.

[4] L. Cardelli. *Type Systems*. CRC Press, the computer science and engineering handbook edition, 1997.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. In *ACM Computing Surveys, Vol.17, No.4*, 1985.

[6] E. A. Lee. What are the key challenges in embedded software? In *Guest Editorial, in System Design Frontier, Volume 2, Number 1*, January 2005.

[7] E. A. Lee and S. Neuendorffer. Classes and subclasses in actor-oriented design. In *Invited Paper in Proc. of the Conference on Formal Methods and Models for Codesign (MEMOCODE)*, June 22-25, 2004.

[8] J. C. Mitchell. Coercion and type inference. In *Proc. of 11th Annual ACM Symp. on Principles of Programming Languages*, 1984.

[9] Z. Pawlak. Granularity of knowledge, indiscernibility and rough sets. In *In Proceedings of The 1998 IEEE International Conference on Fuzzy Systems.*

[10] J. Rehof and T. Mogensen. *Tractable Constraints in Finite Semilattices*. Springer, third international static analysis symposium, lncs 1145 edition, Sept., 1996.

[11] J. A. Robinson. A machine-oriented logic based on the resolution principle. In *J. ACM 12(1)*, 1965.

[12] Y. Xiong. An extensible type system for component-based design. Technical Report Ph.D. thesis, Technical Memorandum UCB/ERL M02/13, University of California, May 1 2002.

[13] Y. Xiong, E. Lee, X. Liu, Y. Zhao, and L. Zhong. The design and application of structured types in ptolemy ii. In *IEEE Int. Conf. on Granular Computing (Grc 2005)*, July 25-27, 2005.

[14] Y. Xiong and E. A. Lee. An extensible type system for component-based design. In *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, March/April 2000. LNCS 1785.

[15] Y. Y. Yao. A partition model of granular computing. In *Transaction on Rough Sets I, LNCS Vol. 3100*, 2004.