IEEE Int. Conf. on Granular Computing (GrC 2005), Beijing, China, July 25-27, 2005.

The Design and Application of Structured Types in Ptolemy II

Yuhong Xiong† Edward Lee Xiaojun Liu Yang Zhao Lizhi C. Zhong‡
Department of EECS, University of California, Berkeley
†HP Laboratories, Palo Alto ‡ STMicroelectronics Central R&D Berkeley Labs
†yuhong.xiong@hp.com {eal,liuxj,ellen_zh}@eecs.berkeley.edu ‡czhong@yahoo.com

***Abstract*— Ptolemy II is a component-based design and modeling environment. It has a polymorphic type system that supports both the base types and structured types, such as arrays and records. The base type support was reported in [12]. This paper presents the extensions 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 a Ptolemy II model that implements part of the IEEE 802.11 specifications. This model makes extensive use of record types to represent the protocol messages in the system.**

I. Introduction

Embedded systems have become ubiquitous. They can be found in cars, consumer electronics, appliances, networking equipments, aircrafts, security systems, etc. Due to the complexity of modern systems, their design poses many challenges [5]. 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 [13] 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 as they are generally held to a much higher reliability standard than general purpose software [5]. 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 [12], all the types are organized into a lattice, and type constraints are decribed as inequalities over the lattice.

This paper describes the extensions to support structured types, and a practical application model that takes advantage of the type system. By using structured types, data that are naturally related can be grouped together, and the application model becomes more readable. 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 a Ptolemy II application that uses structured types. Section V concludes the paper and points out future works.

II. Ptolemy II Base Type System

We review the base type system in Ptolemy II below. In addition to the material drawn from [12], 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 [6]. In Ptolemy II, components are actors, and the interface to actors are 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.

B. Type Lattice

Ptolemy II supports automatic run-time type conversion. To do this, all the base types are organized into a `type lattice`, as shown in figure 2. The ordering relation in

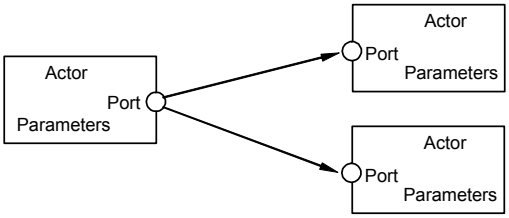

Fig. 1. An abstract representation of a Ptolemy II model.

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 [7], we can say that the relation in the type lattice represents two kinds of subtyping relations.

C. Type Constraints and Type Resolution

In Ptolemy II, each port and each parameter 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 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 [9]. 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. The solution, if it exists, is the set of most specific types.

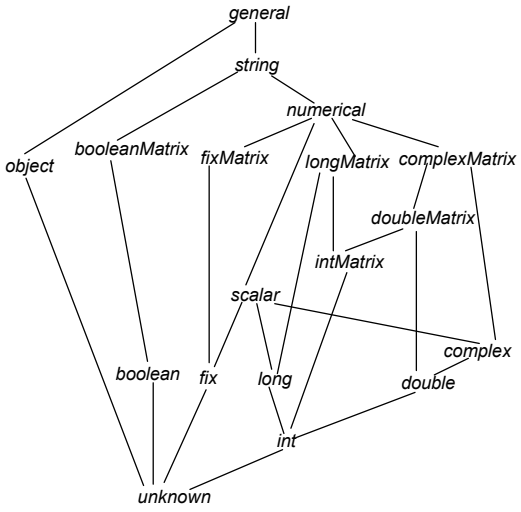

Fig. 2. An example of a type lattice.

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 [11].

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.

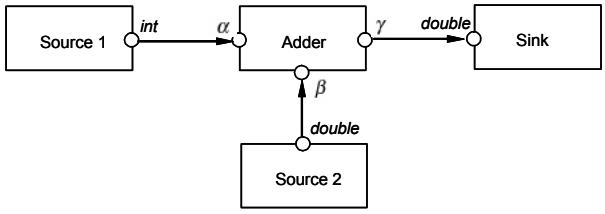

Fig. 3. A topology (interconnection of components) with types.

The inequality type constraints can also be related to rough set theory [8]. 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. For example, a record type is a named collection of types, like the structure in the C language. They 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, they can be used to reduce the number of ports on certain actors, which simplifies the topology of the block diagram.

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 and record 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 record label with the element type or value. 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.
- $\{name = \text{“}foo\text{”}, value = 1\}$: A record with two fields. One has label `name` and string value `foo`, the other has label `value` and integer value `1`.
- $\{name = string, value = int\}$: The type of the above record.

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 [11] 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 compile cannot detect.

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 text based languages. But for block diagram based 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 source 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 and record tokens, are immutable. For immutable arrays, we define subtyping in a covariant way in the type lattice.

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

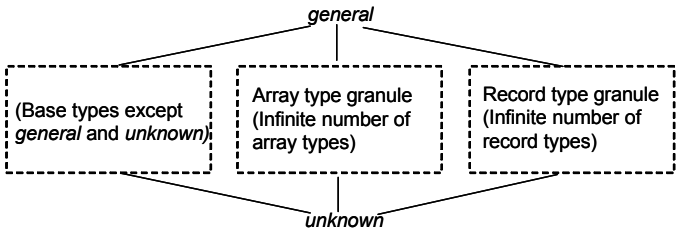

Fig. 4. The type lattice of Ptolemy II with array and record types added.

example, $\{name = string, value = double, id = int\} \leq \{name = string, value = double\}$.

To add structured types to the type lattice, we take a hierarchical and granular approach by treating each kind of structured types 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. These granules are incomparable in the type lattice, as shown in figure 4. 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, 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:

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

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 would be sufficient if the function is 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 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.

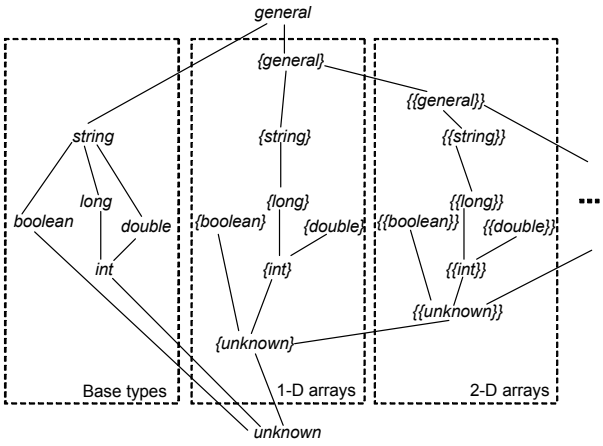

Fig. 5. An example of a type lattice with arrays.

The reverse update is conceptually a process of unification [10] 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 5.

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:

$$\{unknown\} \leq unknown$$
$$\{\{unknown\}\} \leq \{unknown\}$$
$$\{\{\{unknown\}\}\} \leq \{\{unknown\}\}$$

Fortunately, this kind of infinite iteration can be detected. Observe that:

- 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 the array types are added, but also after the 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.

IV. Application

We have implemented the array and record 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] [11]. These structured types are widely used in the modeling of the systems requiring component-based design. As an example, in the following, we present a network protocol model built in Ptolemy II that intensively uses record types for the communications between blocks.

The model shown in figure 6 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, exponential back off, to coordinate with other nodes in sharing the communication medium. Before transmission, a node first senses the medium to determine if 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 media 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 6 shows the model at different levels. Every process is essentially a finite state machine. It interacts with one another by the messages, each of which has a collection of fields of various types. Record types are the nature choice for representing the messages. They make the execution more efficient and the block diagram moe readable as mentioned in Section 3.1. 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 6, 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 6 the use of record types really makes the modeling of IEEE 802.11 MAC easier. The bottom portion of Figure 6 shows the record types on each connection in the *Reception* block. For instance, the ValidateMPDU process 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. One can also see from this figure that the fromPHY port receives three types of messages, calling for the support of union types in Ptolemy II.

V. Conclusion and Future Work

We have presented some extensions to the Ptolemy II base type system to support two structured types, array and record.

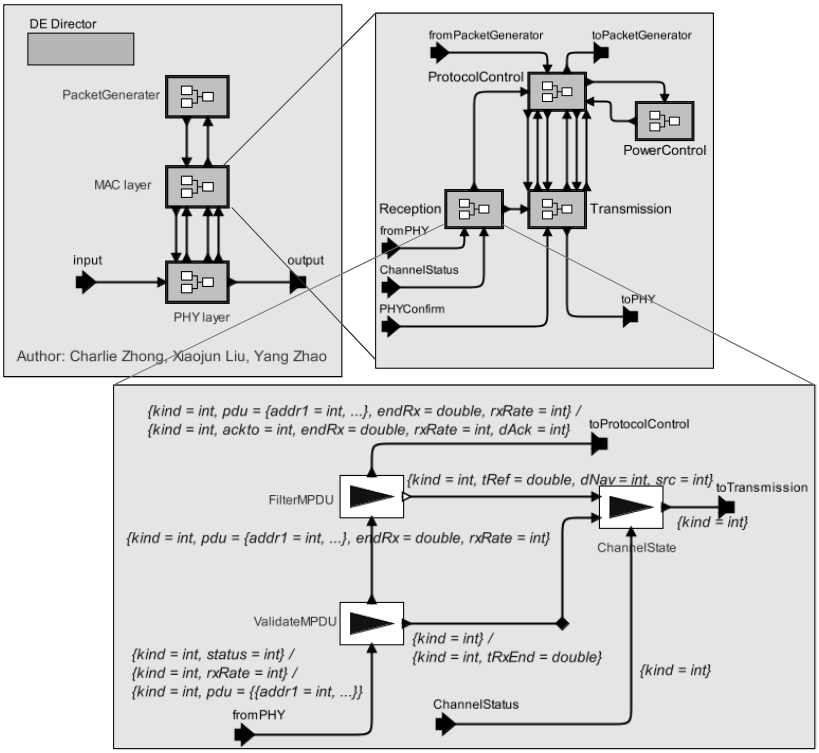

Fig. 6. An wireless protocol model using record types

In particular, we extended the format of inequality constraint 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.

The union type is another structured type that is very useful, it 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. The union type is also called variant types in the type system literature [4]. In the Ptolemy II application above, the union type can be used to allow protocol messages to have different formats. The width subtyping relation for union type is the opposite to that of the record type. That is, a shorter union is a subtype of a longer one. This means that there is an infinite number of types from a particular union type to the top of the type lattice, so the convergence of the type resolution algorithm is not immediately obvious. We are currently addressing this issue and working on the implementation of the union type in Ptolemy II.

ACKNOWLEDGMENT

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.

REFERENCES

[1] A. Aiken. Set Constraints: Results, Applications and Future Directions. In *Proc. of the Second Workshop on the Principles and Practice of Constraint Programming*, May 1994.

[2] IEEE-SA Standards 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 Memorandum UCB/ERL M04/27, University of California, July 29 2004.

[4] L. Cardelli. Type Systems. *The Computer Science and Engineering Handbook*, CRC Press, 1997.

[5] E. A. Lee. What Are The Key Challenges in Embedded Software? *Guest Editorial, in System Design Frontier, Volume 2, Number 1*, January 2005.

[6] E. A. Lee and S. Neuendorffer. Classes and Subclasses in Actor-Oriented Design. Invited Paper in *Proc. of the Conference on Formal Methods and Models for Codesign (MEMOCODE)*, June 22-25, 2004.

[7] J. C. Mitchell. Coercion and Type Inference. In *Proc. of 11th Annual ACM Symp. on Principles of Programming Languages*, 1984.

[8] Z. Pawlak. Granularity of Knowledge, Indiscernibility and Rough Sets. In *Proceedings of the 1998 IEEE International Conference on Fuzzy Systems*, May 4-9, 1998.

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

[10] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. In *J. ACM 12(1)*, 1965.

[11] Y. Xiong. An Extensible Type System for Component-Based Design. Ph.D. thesis, Technical Memorandum UCB/ERL M02/13, University of California, May 1 2002.

[12] 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*, LNCS 1785, March/April 2000.

[13] Y. Y. Yao. A Partition Model of Granular Computing. In *Transaction on Rough Sets I*, LNCS 3100, 2004.