An Extensible Type System for Component-Based Design

by

Yuhong Xiong

B.E. (Tsinghua University) 1990
M.S. (University of Washington) 1993

A dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Engineering — Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Edward A. Lee, Chair
Professor Thomas A. Henzinger
Professor Stuart E. Dreyfus

Spring 2002

The dissertation of Yuhong Xiong is approved:

Chair Date

Date

Date

University of California at Berkeley

Spring 2002

An Extensible Type System for Component-Based Design

by

Yuhong Xiong

Abstract

An Extensible Type System for Component-Based Design

by

Yuhong Xiong

Doctor of Philosophy in Engineering — Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Edward A. Lee, Chair

Component-based design has been established as an important approach to designing complex embedded systems, which often have many concurrent computational activities and mix widely differing operations. A good type system is particularly important for component-based design. A type system can improve the safety and flexibility of the design environment, promote component reuse, and help simplify component development and optimization. Although type systems have been studied extensively in the programming language community, its research in component-based design is not enough.

In this thesis, we present an extensible type system for component-based design. Fundamentally, a type system detects incompatibilities at component interfaces. Incompatibility may happen at two different levels: data types and dynamic behavior. Accordingly, the type system presented in this thesis also has two parts. For data types, our 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 subtyping relation among types, and use inequalities defined over the type lattice to specify type constraints in components and across components. By requiring the types to form a lattice, we can use a very efficient algorithm to solve the inequality type constraints, with existence and uniqueness of a

solution guaranteed by fixed-point theorems. This 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.

Our type system supports both the primitive types and structured types, such as arrays and records. The addition of structured types makes the type lattice infinite, and requires an extension on the format of the inequality constraints. 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 data-level type system has been implemented in Ptolemy II, which is a component-based design environment. Our implementation is modular. In particular, the CPO and lattice support, including the algorithm for solving inequality constraints, are implemented as a generic infrastructure that is not bound to one particular type lattice. Type definition and type checking are implemented in separate packages and have been fully integrated with Ptolemy II.

To describe the dynamic behavior of components and perform compatibility check, we extend the concepts of conventional type system to behavioral level and capture the dynamic interaction between components, such as the communication protocols the components use to pass messages. In our system, the interaction types and the dynamic behavior of components are defined using a light-weight formalism, interface automata. Type checking, which checks the compatibility of a component with a certain interaction type, is conducted through automata composition. Our system is polymorphic in that a component may be compatible with more than one interaction types. We show that a subtyping relation exists among various interaction types and this relation can be described using a partial order. This behavioral type order provides significant insight into the relation among the interaction types. It can be used to facilitate the design of polymorphic components and simplify type checking. In addition to static type checking, we also propose to extend the use of interface automata to the on-line reflection of

component states and to run-time type checking. We illustrate our framework using the Ptolemy II environment, and discuss the trade-offs in the design of behavioral type system.

Professor Edward A. Lee, Chair Date

To my wife Bei, my son Michael
and my parents Erke Mao and Rumei Xiong

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am deeply grateful to Professor Edward Lee for being my advisor for this thesis. He led me into this research area and his vision has directly guided my work over the past five years. This thesis would have never been written without his support and encouragement. Furthermore, his way of using intuition to understand complicated problems and hands-on approach for research will have a lasting impact on my career in the future.

I would also like to thank Professor Thomas Henzinger and Professor Stuart Dreyfus for serving on my thesis and qualifying exam committee. They have provided constructive comments during my qualifying exam, which directly influenced the writing of this thesis. In addition, Professor Henzinger has helped me understand some issues in automata theory. I am also grateful to Professor David Messerschmitt for chairing my qualifying exam committee and providing valuable comments on issues related to component-based design.

I had the privilege of working with exceptional colleagues in the Ptolemy group, including Adam Cataldo, Chris Chang, Elaine Cheong, Jörn Janneck, Bart Kienhuis, Bilung Lee, Jie Liu, Xiaojun Liu, Steve Neuendorffer, John Reekie, Win Williams, Yang Zhao, and Haiyang Zheng. Our stimulating discussions and the fun group activities have enriched my life at Berkeley. I would like to gratefully acknowledge their help on my work and their friendship. In particular, Xiaojun helped me in the implementation of interface automata. He is a great partner to work with. Jie has influenced my work in several areas. Steve is always quick in suggesting improvements. Discussions with Jörn were always stimulating. The software practice shaped up by John has set a high standard for me to follow and brought my understanding for software development to a new level. I would also

like to thank Christopher Hylands and Mary Stewart for managing an excellent computing facility.

Many people outside the Ptolemy group have also helped a great deal with my research. Among them, Freddy Mang assisted in clarifying the equivalence relations of automata. The discussions with Luca de Alfaro on interface automata were very fruitful. Manuel Fahndrich pointed out the paper with the constraint solving algorithm to me. Zoltan Kemenczy, Sean Simmons, and Ed Willink have suggested improvements to the type system.

During my internship at IBM Almaden Research Center, I benefited invaluably from working with the people in the TSpaces project, including Marcus Fontoura, Toby Lehman, Tom Truong, and Dwayne Nelson. They gave me the opportunity to work in an industry research environment, and greatly influenced my attitude toward research.

Lastly, but most important, I thank my family — my wife Bei Wang, my son Michael Xiong, my father Erke Mao, my mother Rumei Xiong, and my brother Yuxing Mao. I am extremely grateful for Bei's support over the years. Michael brings me happiness and makes me proud. My parents always believe in me and have done all they can to support the endeavor I choose. Yuxing is the only adult in the family who has escaped from a profession in electrical engineering. He showed me other possibilities in life.

1 Motivation

1.1 Embedded System Design

Over the past 50 years, the center stage of computing has shifted from mainframe to PC and is about to shift again to embedded computing. The invention of ENIAC in 1947 marked the start of mainframe era and computing was focused on information processing. In the 1980's, personal computers emerged and took over mainframes as the driving force in computer industry. In recent years, many people believe that we are at the cross line of a *post-PC era*, in which communication and pervasive interaction are becoming the focus. In this era, the role of desktop computing will be greatly diminished by new computing paradigms, described by such terms as *invisible computing*, *pervasive computing* [38], or *handheld computing*. From our perspective, all of these new forms of computing fall into the category of *embedded computing*.

Embedded systems are computing systems that are not first-and-foremost computers [61]. They are everywhere, residing in cars, consumer electronics, appliances, networking equipments, aircrafts, security systems, etc. In addition to the dozens of embedded systems we interact with everyday, networked embedded systems will play a larger role in our life in the post-PC era. As painted vividly by media, many of the devices in our home and office will be connected and will cooperate with each other. For example, our sprinkler will compute the optimal watering schedule based on the sensor data and weather forecast, our PDA or mobile phone may display a coupon for Big Mac if we walk by a McDonald at lunch time. Our cars will also become more intelligent; they can help us find out the best route according to real-time traffic information, and the mechanical steering system will be replaced by drive-by-wire technology, making the ride safer and more pleasant. These pre-

dictions may not become true in the near future, but at least one thing is evident: the design and development of embedded systems will continue to evolve for many years to come.

Besides consumer applications, embedded systems also play a big role in industry and safety critical systems. For example, embedded software is an important part of the flight control system in aircrafts. Research is also under way to use embedded software to prevent aircrafts from entering restricted space [63]. There is little doubt that embedded systems are taking the center stage of computing.

Embedded systems often mix technologies, such as hardware and software, analog and digital circuits, and mechanical devices. They are heterogeneous in that they frequently perform diverse operations, including signal processing, feedback control, sequential decision making, and user interfaces. In addition, they often have many parts that are working concurrently, and must meet some real-time requirements [62]. Unlike PCs, where most systems follow de facto industry standards for system architecture, CPU instruction sets, bus protocol, and operation systems, the vast majority of embedded systems are custom designed. With the increasing complexity, and time to market and cost pressures, the design of embedded systems has become a challenging task.

In recent years, *component-based design* has shown great promise in coping with the complexity in modern systems. At the system level, component-based design amounts to wiring up pre-designed components to form the complete system. This approach has the potential of increasing design productivity by reusing the same components in multiple designs. At the same time, it also poses some new challenges. One of the fundamental questions is: when we connect components to form a system, how can we ensure that they will work together?

1.2 Type Systems for Component-Based Design

To make components work together, one of the prerequisites is that their interfaces must be compatible in some sense. This thesis focuses on techniques to ensure interface compatibility. For software components, interface mismatch can happen at (at least) two different levels. One is the data type level. For example, if a component expects to

receive data encoded as integers, but another components sends it a string, then the first component may not be able to function correctly. The second level of mismatch is the dynamic interaction behavior, such as the communication protocols the components use to exchange data. Since embedded systems often have many concurrent computation activities and mix widely differing operations, components may follow widely different communication protocols. Therefore, ensuring the compatibility of dynamic behavior is also an important and non-trivial problem.

In the programming language community, the data type problem has been studied extensively over several decades. However, most of the type systems proposed are for conventional text based languages, and type systems for component-based design environments, which are usually block diagram based languages, have not been studied enough. In the first part of this thesis, we propose a type system for component-based design environment that addresses the data type compatibility issue.

To address the compatibility of communication behavior, we extend the concept of type systems to capture the dynamic aspects of component interaction. We call the result *behavioral types* [69][69]. We will describe our framework for capturing and checking the dynamic behavior of component interface, and show that it offers some of the same benefits as data typing.

By detecting mismatch at component interfaces and ensuring component compatibility, a type system can greatly increase the robustness of a system. This is particularly valuable for embedded software. Unlike desktop computers, many embedded systems do not enjoy the luxury of being able to be rebooted when things go wrong. Once the system is deployed, it must continue to work without human intervention.

In addition to providing safety guarantees, many modern type systems are also very flexible in that they support polymorphic typing of programs. That is, they allow a program to have more than one type so that it can be used in different settings. Our type system, at both the data level and system level, is also polymorphic. This feature enables component reuse, which is a key benefit of component-based design.

The rest of this thesis is organized as follows. Chapter 2 reviews the background material, including type systems for conventional languages, the issues in component-based design, and the mathematical tools used in this thesis. Chapter 3 presents our data type system. Chapter 4 presents behavioral types. Chapter 5 discusses our implementation of the type system in Ptolemy II - a component-based design environment. Chapter 6 concludes the thesis and points out future directions.

2 Background

On a high level, the main goal of this thesis is to present techniques that help improve the quality and productivity of embedded software development. As such, the techniques presented in this thesis are closely related to other research in the general area of software. As witnessed by the frequent crashes of PCs and frequent delays in software projects, software quality and development productivity have not reached the desired level. To solve this problem, many solutions have been proposed by researchers and practitioners. Roughly speaking, these solutions fall into three categories: *good software practice*, *new languages and tools*, and *formal method*.

Good software practice includes design and code reviews, code rating, thorough testing, good source code repository management, and complete documentation. They are very effective in improving software quality. For example, in the Ptolemy II project, all of the above are practiced and they have greatly improved the quality of software [111]. The number of bugs in the core parts of Ptolemy II is small and most bugs can be tracked down within hours. Since the adoption of good practice does not require developers to learn new languages and tools based on formal methods, good practice can make quicker impact than other approaches. In addition, it also helps by boosting the experience of novice programmers in a team setting. However, human factors may be the main hurdle in their adoption. Some programmers may not be willing to have their code reviewed by peers. And some business strategies give early availability of product (even with a lot of bugs) higher priority than software quality, making some managers reluctant to enforce good practice throughout the development cycle. We will not make further discussion on software practice since it is outside the scope of this thesis.

At the beginning of their life span, widely accepted languages were usually adopted in a new application domain, where they offer unique advantage over existing languages. For example, Fortran offers the convenience of high level programming while maintaining efficiency close to direct assembly language programming. Since this language was originally designed for the IBM 704 computer, which had built-in floating point capabilities [10], it provides good support for floating point computation. Due to the ease of high level programming and floating point support, Fortran has been widely used in the science and engineering community for decades after its introduction in 1954.

Another popular language is C, which was devised as a system implementation language for the UNIX operating system [114]. C is "close to the machine" in that the abstractions it introduces are readily grounded in the concrete data types and operations supplied by conventional computers. This makes it simple and small, and allows engineers who understand how computers work to generate time- and space-efficient programs. This attribute makes C suitable for system programming.

In recent years, Java has become the language of choice for many software developers. While Java has many good features, the most important one that popularized the language was the capability of Java applets to "move behavior" [19]. This makes it uniquely positioned for web programming.

As can be seen, new languages can often enable or boost the massive software development activities in a new application area, because they offer an abstraction level and a set of features that hit the bull's eye in that area. The high level notation and floating point support in Fortran, the closeness to the machine of C, and the capability of moving behavior in Java are all examples of this. Now if we consider embedded software, the characteristics are quite different from other applications. As argued by Lee in [62], the principal role of embedded software is the interaction with the physical world, so factors including timeliness, concurrency, liveness, interfaces, heterogeneity, and reactivity need to be considered. The prevailing software engineering methods do not address these issues well, and a new language is in order. This new language should take a component-based approach and be based on formal Models of Computations. The type system proposed in this thesis is

designed for such languages. Later in this chapter, we will discuss component-based design and models of computation in more detail, in section 2.2 and 2.3.

Formal methods include theorem proving and model checking for verification, and formal languages for specification [28]. Broadly, formal methods also include type systems since many type systems have precise mathematical formulations. Theorem proving can be used to verify properties or prove the correctness of programs. For example, it can be proved that a division program computes the correct remainder and quotient. However, verifying specific properties of non-trivial programs is difficult. Very often, program proofs are more detailed than the programs being verified, and the proving process requires human intervention. In practice, theorem proving is generally applied to safety-critical applications by skilled experts [47].

Compared with theorem proving, traditional type systems in programming languages are not used to verify arbitrary properties, but to prevent the occurrence of *execution errors*, which means that during the execution of the program, the machine encounters a meaningless instruction, such as jumping to the wrong address, or adding an integer to a boolean [21]. Although the prevention of execution errors does not ensure the correctness of a program, it does eliminate a large percentage of the errors, or "bugs," in the program. Furthermore, many program errors are multi-faceted, so some aspects of the problem are exposed as type errors. This makes type systems one of the most successful formal methods in software design. The next section of this chapter reviews some of the results in type system research.

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model [28]. It has been applied successfully in hardware and protocol verification. In recent years, it has also been used in software. Model checking is related to our behavioral type system. Both use automata as a basic tool. In section 2.4.2, we will present an overview of a particular form of automata used in behavioral types - interface automata [34].

The above categorization of techniques is not clear cut. For example, type systems belong to both language design and formal methods. Also, major advancement in software devel-

opment requires a successful combination of the techniques. As Meyer puts it, “Every Little Bit Counts” [81].

2.1 Data Types

Research in type systems for programming languages dates back many decades. As early as 1954, real and integer types are distinguished in Fortran. Due to the vastness of the field, it is beyond reasonable hope to mention even a moderate part of the literature. In this section, we will examine some basic issues in type system design and some notable type systems that have inspired our work. Section 2.1.1 discusses the role of type systems as catching execution errors. Section 2.1.2 examines design issues for type systems. And section 2.1.3 presents some notable type systems in programming languages.

2.1.1 Role of Type Systems

The Dictionary of Computing [57] defines *type*, or *data type* as:

> *An abstract set of possible values that an instance of the data type may assume, given either implicitly, e.g. INTEGER, REAL, STRING, or explicitly as, for example, in Pascal:*
>
> *TYPE color = (red, green, orange)*
>
> *The data type indicates a class of implementations for those values.*

Other sources [21][55][109] give similar definitions. A *type system* is the component of a language that keeps track of the types of objects in a program. In general, the type of an object implicitly specifies a set of admitted operations of the object, and the type system ensures that only those operations are applied during the execution of the program, which prevents the occurrence of execution errors. To make this statement more precise, let’s look at figure 2.1, which shows a simple categorization of program errors. This figure

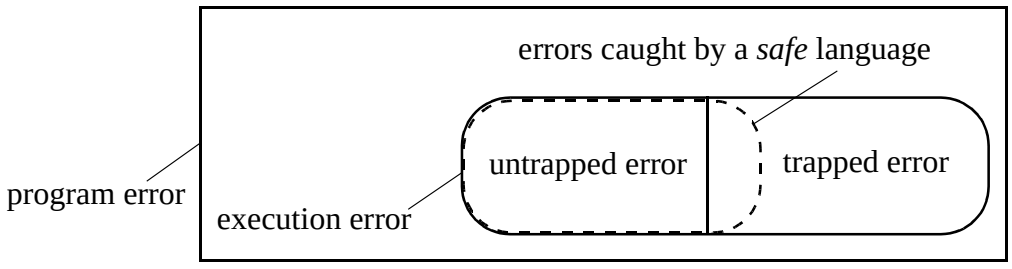


Figure 2.1 Venn diagram of program errors.

is generated based on the definitions in [21]. In this figure, execution error is a subset of all program errors, or bugs. Very often, execution errors cause the program to crash, while non-execution errors do not crash a program, but may cause the program to produce a wrong result. Execution errors can be divided into two categories: trapped errors and untrapped errors. Trapped errors cause the execution to stop immediately, but untrapped errors may go unnoticed (for a while) and later cause arbitrary behavior. Examples of trapped errors include divide by zero or dereferencing `null`. Examples of untrapped errors include going over the boundary of an array or jumping to the wrong address. Obviously, untrapped errors are the most insidious form of execution errors, so the type system of a languages should strive to detect as many untrapped errors as possible. Languages that rule out all untrapped errors are called *safe* languages. Examples of safe languages are ML [124] and Java, and examples of unsafe languages include C and various assembly languages. The lack of safety in language design is usually driven by performance considerations, as the safety guarantee requires extra computation. However, safe languages can catch a larger fraction of routine programming errors and eliminate many lengthy debugging sessions. In practice, many bugs reveal themselves in multiple ways. For example, a buggy sorting routine may go over array boundary and produce the wrong result. If this routine is written in a safe language, this error will be caught in the first execution, at the latest. However, if this routine is written in an unsafe languages, this error may not be caught until the result of the program is examined.

In addition to catching execution errors, type information can also be used for program optimization. For example, type information was introduced in Fortran to improve code generation and run time efficiency for numerical computations. More recently, type information has been used extensively in program analysis. Program analysis includes various techniques to extract information from programs so that compilers can take advantage of this information to produce optimized code. For example, *common subexpressions* are expressions that were previously computed, and not later modified. The common subexpression analysis identifies such expressions for each program point so that the compiler can avoid recomputing them. Another example of program analysis is *alias analysis*. In

languages with side effects, a memory location has *aliases* if it is denoted by more than one expression, such as pointed to by more than one pointer. Alias information is useful in compilers because, for instance, if a memory location does not have aliases, the compiler can assume that it will not be modified through other pointers, so its value does not need to be loaded every time it is used. This is called *redundant-load elimination* (*RLE*). Common subexpression analysis, alias analysis, and many other kinds of program analyses have been studied and used in optimizing compilers for a long time (see for example [5]). In recent years, type information is exploited to make such analyses simpler and more efficient [105]. For example, type-based alias analysis was performed on C++ [89] and Modula-3 [36]. For object-oriented languages, *class hierarchy analysis* was used to identify the virtual method calls that can be resolved at compile time, and possibly replace those calls with method inlining [35]. The survey paper of Palsberg [105] examines the state of the art of type-based analysis and contains many references of work in this area.

Type annotations in programs also serve as important documentation and help clarify program interfaces. Compared with other form of annotation, such as informal comments and formal specifications for theorem proving, types are more precise than program comments, and easier to check than formal specifications.

The design of a good type system is not a trivial task. To improve safety, a type system defines a set of rules, or type rules, regarding how objects with various types may interact. These rules are enforced by a *type checking algorithm*. Ideally, the type rules should enable the type checking algorithm to separate out all the “bad” expressions from the “good” ones in a program. However, since type-checking must be theoretically decidable and practically feasible, this perfect separation is usually not possible. As a result, some good expressions are sacrificed. For example, as we will see in section 2.1.3.2, the ML program

```
fun h(f) = (f(3), f(true));
```

is rejected by the type system, even thought it can be a good function definition if the argument `f` is always a function that can be applied to both integer and boolean. In unsafe languages, some bad expressions are also admitted. One direct consequence of rejecting good expressions is that the reusability of programs is compromised. So the trade-off between

safety guarantees and the flexibility of the language needs to be carefully evaluated in type system design.

Furthermore, from the programmers' point of view, type rules should be easily understandable so that they can predict easily whether a program will typecheck. If it fails to typecheck, the compiler or the run-time environment should be able to provide a clear explanation of the problem. Also, type checking should not require excessive type annotation. To meet these requirements, we need to look into some issues in type system design.

2.1.2 Issues in Type System Design

2.1.2.1 Static vs. Dynamic Type Checking

Type checking can be performed at either compile time or run time. Compile time checking is also called static checking. Static checking has the key advantages of early error detection and not incurring run-time overhead. However, static checking often requires the programmer to declare the type of variables[1], slows down compilation, and limits the reuse of programs to some degree. On the other hand, run time, or dynamic type checking usually requires much less annotation in the program and permits more program reuse, but slows down the execution of program. Full dynamic type checking is often employed in languages that are intended for interactive use, such as scripting languages. These languages are often labeled typeless since the source program does not contain type annotation. The choice between static and dynamic type checking is still subject to some debate. For example, Ousterhout argues that system programing languages should be typeless to support a compact coding style and reuse [103]. However, for languages that are intended for developing large scale applications, static type checking is favored by most researchers. Much of the research in this area has been driven by the desire to combine the flexibility of dynamically typed languages with the security and early error-detection potential of statically typed languages [100]. Modern type systems have achieved this goal to a large extent by making program components *polymorphic*.

1. Of course, there are exceptions. For example, the language ML uses *type inference* to automatically infer the type of variables. The ML type system will be discussed in section 2.1.3.2.

2.1.2.2 Polymorphism

In polymorphic languages, some values and variables may have more than one type. Polymorphic functions are functions whose operands can have more than one type [22]. Cardelli and Wegner [22] 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. Examples of these kinds of polymorphism will be discussed in section 2.1.3. In terms of implementation, a universally polymorphic function will usually execute the same code for different types, whereas an ad hoc polymorphic function will execute different code.

Ad hoc polymorphism is also further divided into overloading and coercion. Overloading refers to the reuse of the same operator or function name to denote different operations, and the context is used to decide which operation should be invoked. This kind of polymorphism can be viewed as a convenient syntactic abbreviation, since the compiler can eliminate overloading by giving different names to different operations. On the other hand, coercion is instead a semantic operation that converts an argument to the type expected by a function.

The distinction between overloading and coercion is not always clear. For example, the operation `3.0 + 4` can be interpreted as either overloading or coercion. That is, we can either view the + operator as overloaded for integer and floating point addition, or only used for floating point addition but the integer 4 is converted to a floating point number through coercion. Obviously, coercion is one kind of type conversion.

2.1.2.3 Type Conversion

Frequently, a value of one type needs to be converted to another type before it can be used. The most common case is the conversion between integer and floating point numbers. Type conversion can be implicit, in which case it is carried out automatically by the compiler or the run-time system, or explicit, in which case it is specified by the programmer. Many languages provide ways to *cast* values or variables of one type to another, which is

a form of explicit conversion. The coercion discussed in the last section is the same as implicit type conversion. Type conversions can also be categorized into lossless and lossy conversions. For example, converting the integer 3 to the floating point number 3.0 is lossless, but converting the floating point number 3.5 to an integer by removing the fractional part loses information. In practice, the amount of implicit conversion varies widely among languages, reflecting the different view on type conversion by different language designers. Some people believe that type conversions should be explicit [10], or at least that complicated rules that define implicit conversions should be avoided [53]. Other people implement complicated conversion rules in languages. For example, in ALGOL 68, a type *proc ref bool* can be coerced to *proc [] union (real, proc union (int, bool))* [73][1]. Since the conversion rules in ALGOL 68 are very complex, some of the coercions in a program may actually arise from programming errors, rather than intention. After the ALGOL 68 experience, many later languages have greatly reduced the amount of implicit coercions [22].

Type conversion can also be performed on objects in systems with subtyping. In this context, the conversion not only affects data layout, but also the behavior of the objects. An example of the object type converter is the respectful type converter [125]. In a type hierarchy where T is a super type of both A and B, a converter C: $A \rightarrow B$ that converts an object of type A to type B respects T if a type A object and a type B object have the same behavior when both are viewed as a type T object, i.e., from T's point of view, the objects A and B look the same. Intuitively, T captures the information preserved after type conversion. The formal definition of *respect* is based on behavioral subtyping [125]. Respectful converters are used in the Typed Object Model (TOM) that is used for document type conversion. For example, PNG image and GIF image types are both subtypes of bitmap image type. Converters that respect bitmap image type can be designed to convert PNG to GIF image.

1. Types are called modes in [73].

2.1.2.4 Type Inference

Type inference is the process of finding a type for all the expressions and variables in a program without requiring the programmer to declare the types. Type inference reduces the burden of programmers by allowing them to omit most of the type annotations while still providing the benefit of static type checking. Type inference in polymorphic type systems is in general a difficult problem. The most successful inference system is the one used in the ML family of languages [85]. This system will be discussed in section 2.1.3.2.

2.1.2.5 Extended Type Systems

In most languages, the types defined by the type system include the primitive types such as integer and float, function types, structured types such as arrays and records, and user defined types. Many researchers have proposed various extended type systems to broaden the information carried by types, so as to provide more guarantee through type checking or more opportunity for program optimization through type-based analysis. The following are some examples of this work.

In [127], Xi and Pfenning proposed to assign more accurate types to programs, with the objective of catching more errors at compile time. The more accurate type information is represented by dependent types, which allows types to be indexed by terms. The index object is drawn from a certain domain, and type checking for the refined property of a program is reduced to constraint satisfaction in that domain. For example, to do array bounds checking statically, an integer index object is attached to an array type. The index object is a singleton type, such as *int(n)* for the type of integer n. Array bounds checking is reduced to linear inequality solving in the integer domain. Their approach requires some additional annotation by programmers.

Although most of the work on type checking is done on high-level languages, some is aimed at safety guarantees at the binary executable level. An example of these is the proof-carrying code (PCC) [96], which can also be viewed as work on type system extensions. PCC is a mechanism by which a system can verify that an executable provided by an untrusted source adheres to some safety policy. The safety policy can be viewed as typing rules and the validation processing as typing checking. For this to be possible, the

untrusted code producer must supply with the code a safety proof, either manually or through a certifying compiler. The host can then easily and quickly validate the proof without using cryptography and without consulting any external agents. In the implementation of PCC that ensures memory safety for the DEC Alpha machine code, the safety proof is represented in such a way that the validity of a proof is implied by the well typedness of the proof representation. Thus, proof validation amounts to type checking [95]. The PCC approach is promising for distributed and web computing, when mobile code is deployed, or when an operating system kernel needs to determine the safety of user supplied applications.

A more drastic extension to conventional type systems is to capture some of the dynamic behavior of programs. One area of research in this direction is the *effect systems* [77]. In conventional type systems, types abstract the values that an expression may return, but they do not carry any information about the execution behavior of the program. The effect systems augment types with effects, which describe the side-effects that an expression may have, such as read/write effects on the store, or exceptions that may be raised by the expression. The scope of effects are represented by *regions*, which abstract a set of memory locations in which side-effects may occur. The effects of expressions can be analyzed and inferred using techniques similar to the one used in the ML type system, with some extensions. One application of effect analysis is in parallel computers. If two expressions do not have interfering effects, then a compiler can schedule them in parallel. Effect systems were first proposed in the Ph.D. thesis of Lucassen [76]. Since then, a lot of research was carried out in this area. A summary of more recent research can be found in [98], and in Fahndrich's lecture notes [43], which provides a very accessible tutorial.

Another example of an extended type system that captures the dynamic behavior of programs is the work of Nielson, *et al.*, who analyzed the communication topology of a concurrent language [99]. They extended the effect system to capture the communication *behavior* of programs, such as the sending of a value, the receiving of a value, the allocation of a new communication channel, or the spawning of a new process. Their work is based on the language Concurrent ML (CML) [113], which is an extension of the functional language Standard ML. In their system, behaviors can be included in the type infer-

ence system, and the inference result indicates whether a program only spawns a finite number of processes, or only creates a finite number of channels. In these cases, the compiler may allocate the processes to available processors or allocate communication resources statically.

In addition to the ML family of languages, extended type systems have also been proposed for object and actor based languages [3]. In [29], Colaco *et al.* presented a type inference system for a primitive actor calculus. In the actor model, the communication topology is dynamic, so some messages sent out by actors may never be handled. The aim of the inference system is to detect these orphan messages. This system is based on set constraints [6]. It can detect many orphan messages statically and the remaining messages dynamically based on the static type information.

The issues discussed in this section represent a set of trade-offs in type system design. Each type system represents a set of decisions on these issues. The following section discusses some notable type systems that have influenced the design of our system.

2.1.3 Notable Type Systems

2.1.3.1 Typed Lambda Calculus

To study type system features without being encumbered by non-type system details, such as the syntax of a language, researchers often use an abstract, or bare bone language. λ-calculus [11] is such a language that has been used widely in literature. The original λ-calculus is untyped, but type annotation can be added to form typed λ-calculus. In this section, we will first review the basic syntax and operational semantics of λ-calculus, then add type annotations and type rules to build progressively more complex type systems.

Lambda Calculus

Lambda calculus was invented by Church in the 1930s. Since the 1960s, it has been used extensively in the programming language community for the specification of language features and the study of type systems. Lambda calculus is a mathematical system that defines a syntax for terms and a set of rewrite rules for transforming terms. It captures one's intu-

ition about the behavior of functions. A comprehensive treatment of lambda calculus can be found in [11]. The following introduction is mainly drawn from [7], [106] and [56].

Let x, y, z denote variables, e, e_1, e_2 denote *lambda expressions*. The syntax of λ-calculus is:

$$e ::= x \mid \lambda x.e \mid e_1 e_2$$

A variable x by itself is a lambda expression. Expressions of the form $\lambda x.e$ are called abstractions. It denotes a function with argument x and function body e. $e_1 e_2$ is an *application* of e_1 to e_2. A lambda expression is also called a *term*.

Informally, the meaning of applying a function $\lambda x.e_1$ to a term e_2, $(\lambda x.e_1)e_2$, is to bind x to e_2, evaluate e_1, and return the result of this evaluation. To define this operational semantics more precisely, we need the notions of free variables and substitution.

The free variables of an expression e, denoted $fv(e)$, is defined by:

$$\begin{aligned} fv(x) &= \{x\} \\ fv(e_1 e_2) &= fv(e_1) \cup fv(e_2) \\ fv(\lambda x.e) &= fv(e) - \{x\} \end{aligned}$$

The substitution $[e_1/x]e_2$ is then defined inductively by:

$$[e/x_i]x_j = \begin{cases} e, & \text{if } i = j \\ x_j, & \text{if } i \neq j \end{cases}$$

$$[e_1/x](e_2 e_3) = ([e_1/x]e_2)([e_1/x]e_3)$$

$$[e_1/x_i](\lambda x_j.e_2) = \begin{cases} \lambda x_j.e_2, & \text{if } i = j \\ \lambda x_j.[e_1/x_i]e_2, & \text{if } i \neq j \text{ and } x_j \notin fv(e_1) \\ \lambda x_k.[e_1/x_i]([x_k/x_j]e_2), & \text{otherwise,} \\ & \text{where } k \neq i, k \neq j, x_k \notin fv(e_1) \cup fv(e_2) \end{cases}$$

The last rule handles the situation where the variable x_j has a name conflict with a free variable in e_1. This conflict is resolved by renaming x_j to x_k.

In general, bound variables can be renamed by an operation called *α-conversion*:

$$\lambda x_i.e = \lambda x_j.[x_j/x_i]e, \quad \text{where } x_j \notin fv(e)$$

Intuitively, α-conversion says that the names of the formal parameters in a function in a programming language do not affect the operation. For example, the following two Java programs are the same:

```
int plusOne (int foo) {
     return (foo + 1);
}

int plusOne (int bar) {
     return (bar + 1);
}
```

Computation in λ-calculus is carried out by function application. The rule for function application is called *β-reduction*:

$$(\lambda x.e_1)e_2 \Rightarrow [e_2/x]e_1$$

For example, (λ*x.xy*)(*uv*) reduces to *uvy*.

Given the above simple definition of λ-calculus, much more involved computation can be carried out. For example, we can define multi-argument functions using *higher order functions*, which are functions that yield functions as results. A function *F* with arguments *x* and *y* can be written as $F = \lambda x.\lambda y.e$. That is, *F* is a function that given a value for *x*, yields a function that, given a value for *y*, yields the desired result. This transformation of multi-argument functions into higher order functions is called *currying*.

The pure λ-calculus does not have constants. But they can be encoded as λ-expressions. For example, the boolean values *true* and *false* can be encoded as:

$$true = \lambda t.\lambda f.t$$
$$false = \lambda t.\lambda f.f$$

We can also define a λ-expression *if* so that $if\ e_1\ e_2\ e_3$ reduces to e_2 if e_1 is *true* and reduces to e_3 if e_1 is *false*:

$$if = \lambda l.\lambda m.\lambda n.lmn$$

This function takes three arguments. When applying it to e_1, e_2, and e_3, the result is ($e_1e_2e_3$). When e_1 is *true* or *false*, we have ($true\ e_2\ e_3$) and ($false\ e_2\ e_3$), respectively. In

fact, *true* and *false* are conditionals, they each take two arguments, and choose the first or the second, respectively. To show the computation in term of β-reduction, let's compute *if true* e_2 e_3:

$$
\begin{aligned}
if\ true\ e_2\ e_3 &= (\lambda l.\lambda m.\lambda n.\ l\ m\ n)\ true\ e_2\ e_3 && \text{definition of } if \\
&\Rightarrow (\lambda m.\lambda n.true\ m\ n)\ e_2\ e_3 && \beta\text{-reduction} \\
&\Rightarrow (\lambda n.true\ e_2\ n)\ e_3 && \beta\text{-reduction} \\
&\Rightarrow true\ e_2\ e_3 && \beta\text{-reduction} \\
&= (\lambda t.\lambda f.t)\ e_2\ e_3 && \text{definition of } true \\
&\Rightarrow (\lambda f.e_2)\ e_3 && \beta\text{-reduction} \\
&\Rightarrow e_2 && \beta\text{-reduction}
\end{aligned}
$$

This computation assumes that *n*, *t*, and *f* are not free in e_2 and e_3. Otherwise, some extra renaming steps are required.

We can also encode numbers using the *Church Numerals* C_0, C_1, C_2, etc., as follows:

$$
\begin{aligned}
C_0 &= \lambda z.\lambda s.z \\
C_1 &= \lambda z.\lambda s.sz \\
C_2 &= \lambda z.\lambda s.s(sz) \\
&\dots \\
C_n &= \lambda z.\lambda s.s(s(\dots(sz))\dots)
\end{aligned}
$$

That is, each number *n* is represented by a function with two arguments, *z* and *s* ("zero" and "successor"), and applies *n* copies of *s* to *z*. Given these numerals, we can define some common arithmetic operations as follows:

$$
\begin{aligned}
plus &= \lambda m.\lambda n.\lambda z.\lambda s.m(nzs)s \\
times &= \lambda m.\lambda n.mC_0(Plus\ n) \\
isZero &= \lambda m.m\ True\ (\lambda x.False)
\end{aligned}
$$

For the function *isZero*, it applies a Church numeral to two arguments *true* and (*λx.false*). If the numeral is C_0, it will choose the first argument *true*. If the numeral is C_n (n>0), it will apply n copies of the second argument (*λx.false*) to *true*. Since the function (*λx.false*) just throws away its argument and always returns *false*, the result is *false*. For the intuition

behind *plus* and *times*, as well as the encoding for other data types and operations, please see [106] and the reference therein.

In addition to β-reduction, there is also an *η-reduction* and a notion of *normal form* in λ-calculus. A λ-expression is in normal form if it cannot be further reduced using β- or η-reduction. The normal form is what we intuitively think of as the value of an expression. The Church-Rosser theorems ensure that the normal form is unique, and there is a way to find it whenever it exists. Another important result is Church's thesis, which establishes the equivalence between Turing computability and λ-definability. These results have enabled λ-calculus to be the foundation of functional languages and have profound impacts on programming languages in general. However, since these results are not central to the discussion of the type systems below, they are skipped here. Details can be found in [56].

First-Order Type System

In the above introduction to λ-calculus, we have actually defined a tiny programming language with booleans and natural number constants. For example, we can write a program:

$$\lambda y.if\ (isZero\ y)\ C_0\ C_1$$

that has an argument *y*. If *y* is zero, the program returns 0; otherwise, it returns 1.

Notice however, that we can also write programs that do not make sense, such as:

$$\lambda y.if\ plus\ (true\ y)\ times$$

When provided with an argument, this program can be translated into a λ-expression, but the result is meaningless because some operations are applied to unintended arguments. To solve this problem, we need to introduce a type system to make the intended interpretations explicit. Type systems can be formalized by defining a syntax of types and a set of type rules using that syntax. In the rest of this section, we will add types to the untyped λ-calculus and build three type systems, including a *first-order type system*, a *second-order type system*, and a system with *subtyping*. First-order type systems include a set of base types, structured types, and function types. Second-order type systems add type parameterization

and type abstraction. The material in this section is based on [21], [7], and [97]. We now start with the first-order type system.

Each language has a set of *base types*. For example, the base types for our tiny language are *boolean* and *natural number*. Using K to denote the base types, and τ to denote a type, we have the following in a first order type system:

$$\tau ::= K \mid \tau \to \tau$$

A type $\tau_1 \to \tau_2$ stands for the set of functions that map arguments of type τ_1 to results of type τ_2.

To add types to the untyped λ-calculus, we assign types to bound variables:

$$e ::= x \mid \lambda x{:}\tau.e \mid e_1 e_2$$

This is analogous to declaring the type of function parameters in a real programming language. To express "e has type τ", we write $e{:}\tau$. For example,

$$\lambda x{:}\tau.x{:}\ \tau \to \tau$$
$$\lambda x{:}\tau_1.\lambda y{:}\tau_2.x{:}\ \tau_1 \to \tau_2 \to \tau_1$$
$$true{:}\ Bool$$
$$C_0{:}\ Nat$$

Where *Bool* and *Nat* stand for the boolean and natural number data types, respectively. They are names for sets of λ-expressions.

To perform type checking, we also need the types for free variables. This is given in a *type environment*, which is a function from variables to types. The syntax of the environments is:

$$\Gamma ::= \varnothing \mid \Gamma, x{:}\tau$$

where $\varnothing$ is the empty environment. The meaning of $\Gamma, x{:}\tau$ is:

$$(\Gamma, x{:}\tau)(y) = \begin{cases} \tau & \text{if } x = y \\ \Gamma(y) & \text{if } x \neq y \end{cases}$$

In real compilers, the type environment is implemented by a symbol table.

A *typing judgment*, which asserts that an expression e has a type τ with respect to a type environment for the free variables of e, has the form:

$$\Gamma \vdash e{:}\tau$$

For example,

$\varnothing \vdash true{:}Bool$	$true$ has type $Bool$
$\varnothing, x{:}Nat \vdash plus\ x\ C_1{:}\ Nat$	$x + 1$ has type Nat, provided that x has type Nat

Type rules assert the validity of certain judgments based on the validity of other judgments. The general form of a type rule is:

$$\frac{\Gamma_1 \vdash \Im_1 \ \ldots\ \Gamma_n \vdash \Im_n}{\Gamma \vdash \Im}$$

The judgments above the line are the *premises*, the one below the line is the *conclusion*. Two fundamental type rules regarding function abstraction and application are:

$$\frac{\Gamma, x{:}\tau_1 \vdash e{:}\tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e{:}\ \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1{:}\tau_1 \to \tau_2 \quad \Gamma \vdash e_2{:}\tau_1}{\Gamma \vdash e_1 e_2{:}\ \tau_2}$$

The first rule says that if we can deduce that the expression e has type τ_2, given that its free variable x has type τ_1, then the expression $\lambda x{:}\tau_1.e$ is a function of type $\tau_1 \to \tau_2$. The second rule says that if e_1 is a function of type $\tau_1 \to \tau_2$, and e_2 has type τ_1, then the result of applying e_1 to e_2 has type τ_2.

Using the type rules, we can perform type checking by deriving the types of expressions under a type environment. A *derivation* is a tree of judgments with leaves at the top and a root at the bottom, where each judgment is obtained from the ones immediately above it by some type rule. For example, given that e has type $\tau_1 \to \tau_2$, and x has type τ_1, we can derive that $\lambda x{:}\tau_1.(ex)$ has type $\tau_1 \to \tau_2$ as follows:

$$\frac{\dfrac{e{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash e{:}\tau_1 \to \tau_2 \qquad e{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash x{:}\tau_1}{e{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash (ex){:}\tau_2}}{e{:}\tau_1 \to \tau_2 \vdash \lambda x{:}\tau_1.(ex){:}\tau_1 \to \tau_2}$$

The first step uses the function application rule, the second step uses the function abstraction rule.

A type rule is associated with each basic construct of a language. For example, the type rules for the *if* construct and the *isZero* function are:

$$\frac{\Gamma \vdash e_1 : Bool \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash if\ e_1\ e_2\ e_3 : \tau} \qquad \frac{\Gamma \vdash e : Nat}{\Gamma \vdash isZero\ e : Bool}$$

Now that we have established the basic structure of a simple first-order type system, we can enriching it with more type constructs. We will add another base type *Unit* and two structured types *Record* and *Array*.

The *Unit* type is called *void* in languages like C and Java. It is used as a filler for uninteresting arguments and results. There is no operation on this type, and it only has one legal value, *unit*. So *unit*:*Unit*. We will use this type in one of the type rules for array type.

A record type is a named collection of types, like the structure in the C language. A record is denoted by $record(l_1=x_1, ..., l_n=x_n)$, where $l_1, ..., l_n$ are *labels*, and $x_1, ..., x_n$ are values. The type of this record is denoted by $Record(l_1:\tau_1, ..., l_n:\tau_n)$. The operation $e.l_i$ extracts the field whose label is l_i from the record e. Two rules for record type are:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash record(l_1 = e_1, \ldots, l_n = e_n) : Record(l_1:\tau_1, \ldots, l_n:\tau_n)}$$

$$\frac{\Gamma \vdash e : Record(l_1:\tau_1, \ldots, l_n:\tau_n) \quad i \in 1..n}{\Gamma \vdash e.l_i : \tau_i}$$

The record type is immutable (although mutable record can be defined). That is, once a record is constructed, its contents cannot be changed. Compared with record, Array type is mutable. We use $array(n, e_1)$ to denote an array of length n with all the elements set to the value e_1. The type of this array is denoted by $Array(\tau)$. The operation *bound e* returns the length of the array e, the operation $e[n]$ return the n'th element of the array, and $e[n]=e_1$ assigns the n'th element of the array to value e_1. The rules for array construction and the operations are:

$$\frac{\Gamma \vdash n : Nat \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash array(n, e_1) : Array(\tau)} \qquad \frac{\Gamma \vdash e : Array(\tau)}{\Gamma \vdash bound\ e : Nat}$$

$$\frac{\Gamma \vdash n : Nat \quad \Gamma \vdash e : Array(\tau)}{\Gamma \vdash e[n] : \tau} \qquad \frac{\Gamma \vdash n : Nat \quad \Gamma \vdash e : Array(\tau) \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e[n] = e_1 : Unit}$$

In addition to record and array types, other structured types, such as product, union, variant, and list types, can also be added. Details can be found in [21]. The resulting first-order typed λ-calculus is called system F_1. This system is *monomorphic* in that each expression has only one type. For example, the identity function $\lambda x{:}Bool.x$ has type $Bool \rightarrow Bool$, and the function $\lambda x{:}Nat.x$ has type $Nat \rightarrow Nat$. If we want an identity function for another type, we need to write a new function. Obviously, this is inconvenient. To reuse the same function on different types, we can parameterize the type of the variable *x*, and instantiate the type parameter to different types. By doing this, we obtain a second-order type system F_2.

Second-Order Type System

To accommodate type parameters, we need to add a new kind of expression in our typed λ-calculus: $\lambda\alpha.e$. This expression can be viewed as a program *e* that is parameterized with respect to a *type variable* α. In this thesis, we will use α, β, and γ to denote type variables. By using type variables, we can turn some monomorphic functions to *polymorphic* ones. For example, the identity function for a fixed type τ, $\lambda x{:}\tau.x$, can be turned into a polymorphic identity function by abstracting over τ: $\lambda\alpha.\lambda x{:}\alpha.x$. This is parametric polymorphism discussed in section 2.1.2.2.

In parametric polymorphism, type variables can be instantiated to any given type. So we use *universally quantified types* to denote the type of the expression $\lambda\alpha.e$: $\forall\alpha.\tau$. This means that *forall* α, the body *e* has type τ. Here *e* and τ may contain occurrences of α. For example, the type of the polymorphic identity function is $\forall\alpha.\alpha\rightarrow\alpha$.

To complete the treatment of type parameterization, we need to add two new rules in the second-order system F_2 to handle the abstraction and application of type variables:

$$\frac{\Gamma, \alpha \vdash e{:}\tau}{\Gamma \vdash \lambda\alpha.e{:}\ \forall\alpha.\tau} \qquad \frac{\Gamma \vdash e{:}\forall\alpha.\tau \quad \Gamma \vdash \tau_1}{\Gamma \vdash e\ \tau_1{:}\ [\tau_1/\alpha]\tau}$$

In the second rule for the application of type expression, $[\tau_1/\alpha]\tau$ stands for the substitution of τ_1 for all the free occurrences of α in τ.

In addition to universally quantified types, the second-order type system also has *existentially quantified types*, which is used to model data abstraction. This extension is skipped here since it is not directly related to the system that we will present in chapter 3.

Subtyping

Subtyping is a feature found in almost all of the object-oriented languages. In these languages, an element of a type can be considered also as an element of any of its supertypes, thus allowing an object to be used wherever a supertype element is expected.

One of the simplest type systems with subtyping is an extension of F_1 called $F_{1\leq}$. In $F_{1\leq}$, we add a new judgment

$$\Gamma \vdash \tau_1 \leq \tau_2$$

stating that τ_1 is a subtype of τ_2.

We also need some additional type rules regarding subtyping. The following two rules say that the subtyping relation is reflexive and transitive:

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \leq \tau} \qquad \frac{\Gamma \vdash \tau_1 \leq \tau_2 \quad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3}$$

The *subsumption* rule says that if an expression *e* has type τ_1, and τ_2 is a supertype of τ_1, then e also has τ_2:

$$\frac{\Gamma \vdash e{:}\tau_1 \quad \Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash e{:}\ \tau_2}$$

The subtyping relation for function types is a little bit involved. If we want a function *e* with type $\tau_1 \to \tau_2$ to be a subtype of $\tau_1' \to \tau_2'$, *e* must be able to accept all the arguments of type τ_1', so the argument type of *e* must be a supertype of τ_1'. Also, the return type of *e* must be acceptable when a value of type τ_2' is expected, so τ_2 must be a subtype of τ_2'. Therefore, the subtyping rule for function types is:

$$\frac{\Gamma \vdash \tau_1' \leq \tau_1 \quad \Gamma \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

Notice that the subtyping relation is inverted (contravariant) for function arguments, while it goes in the same direction (covariant) for function results.

In addition to the subtyping relation between objects in object-oriented languages, ad hoc subtyping for base types can be defined. For example, if we add two base types *Int* and *Double* to the typed λ-calculus, we can define $Int \leq Double$. If we consider the four function types whose argument and result types are *Int* or *Double*, the subtyping relation among them can be shown in figure 2.2, where a type at the lower end of a line is a subtype of the one at the upper end.

Subtyping rules can also be defined for some structured types. The rule for record subtyping is:

$$\frac{\Gamma \vdash \tau_1 \leq \tau_1' \quad \ldots \quad \Gamma \vdash \tau_n \leq \tau_n' \qquad \Gamma \vdash \tau_{n+1} \quad \ldots \quad \Gamma \vdash \tau_{n+m}}{\Gamma \vdash Record(l_1{:}\tau_1, \ldots, l_{n+m}{:}\tau_{n+m}) \leq Record(l_1{:}\tau_1', \ldots, l_n{:}\tau_n')}$$

This rule actually specifies two kinds of subtyping for record. A subtype record may have more fields than a super type record, and the field types of the subtype record may be subtypes of the corresponding fields in the supertype record. The former condition is called *width subtyping*, and the latter *depth subtyping*.

Subtyping for mutable types, such as array, is neither covariant nor contravariant. For example, given $\tau_1 \leq \tau_2$, we cannot define $Array(\tau_1) \leq Array(\tau_2)$ or $Array(\tau_2) \leq Array(\tau_1)$ and ensure type consistency through static checking alone. In both cases, we can write programs that cause functions to be applied to wrong type of argument. To illustrate this, let's define a function *f* with argument type τ_1, two arrays A_1 and A_2 with

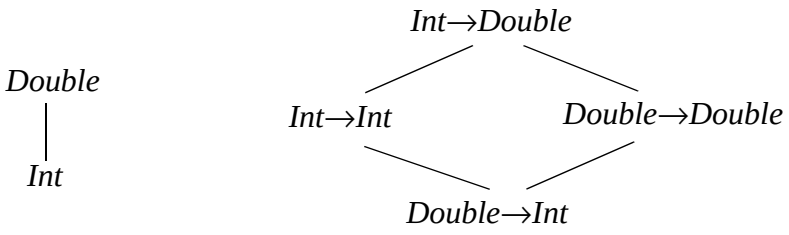


Figure 2.2 Function subtyping among *Int* and *Double* functions.

element types τ_1 and τ_2, and two values x_1 and x_2 of types τ_1 and τ_2, respectively. That is:

$$f: \tau_1 \rightarrow \tau$$
$$A_1: Array(\tau_1)$$
$$A_2: Array(\tau_2)$$
$$x_1: \tau_1$$
$$x_2: \tau_2$$

Now consider these two cases:

Case 1: Assuming that array type is covariant with the element type:

$$\frac{\Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash Array(\tau_1) \leq Array(\tau_2)}$$

We can write an illegal program that assigns x_2 to the first entry of A_2 (assuming array index starts from 0): $A_2[0] = x_2$. Since the type of A_1 ($Array(\tau_1)$) is a sub-type of that of A_2 ($Array(\tau_2)$), we can replace A_2 on the left side of the above assignment with A_1, which results in the assignment: $A_1[0] = x_2$. This causes the first entry of A_1 to contain an element of type τ_2! Now if we apply f to $A_1[0]$, we cause the function to be applied to the wrong type.

Case 2: Assuming that array type is contravariant with the element type:

$$\frac{\Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash Array(\tau_2) \leq Array(\tau_1)}$$

We can write a program that applies f to the first element of A_1: $f(A_1[0])$. Since the type of A_2 is a subtype of that of A_1 now, we can replace A_1 with A_2 in this application: $f(A_2[0])$. Once again, we have caused the function f to be applied to the wrong type.

In Java, arrays are covariant. For example, an Object array reference can point to a String array. To avoid the problem in case 1 above, Java performs run-time checking when the

array elements are set, and the exception `java.lang.ArrayStoreException` is thrown if the check fails.

In addition to the type systems discussed in this section, more complex ones can be defined. For a classification of type systems, see [22]. The systems in this section have ample type annotations, so type checking is relatively simple. If type declaration is omitted in the program, type inference must be used to reconstruct type information. In the second-order type system, type parameters are declared explicitly in polymorphic functions. This is called *explicit polymorphism*. If we omit all the type parameters and type applications, we achieve *implicit polymorphism*. Type inference for polymorphic type systems is in general a hard problem. So far, the most successful inference algorithm is the Hindly-Milner algorithm used in ML.

2.1.3.2 The ML Type System

ML (Metalanguage) was originally conceived as an interactive programming language for conducting proofs in a logical system [49], and later became one of the most popular functional languages. ML supports parametric polymorphism and type inference. In most programs, type declarations can be omitted and the type inference algorithm will infer the types for program expressions. The following overview of the ML type system is mostly based on [7][20][85].

Let's start with a ML program that computes the length of a list:

```
fun length(x) = if (x=nil) then 0 else 1+length(tl(x));
```

Here, `fun` can be viewed as the λ binder. In λ-calculus syntax, the above function definition can be viewed as *length = λx.if (x=nil) then 0 else 1+length(tl(x)).* Notice that this is a recursive function. In this program, *nil* is a constant representing an empty list of any type, *tl()* is a function that returns the tail of a list. To resolve the types in this program, a type variable is assigned to each unknown type. For example, we can denote the type of *length* by α, and the type of *x* by β. By inspecting this program, we can write down the expressions and their types as:

$$
\begin{aligned}
\mathit{length}&: \alpha \\
x&: \beta \\
\mathit{nil}&: \gamma\ \mathit{list} \\
x = \mathit{nil}&: \delta \\
0&: \mathit{Int} \\
1&: \mathit{Int} \\
\mathit{tl}&: \kappa\ \mathit{list} \rightarrow \kappa\ \mathit{list} \\
\mathit{tl}(x)&: \varepsilon \\
\mathit{length}(\mathit{tl}(x))&: \zeta \\
1 + \mathit{length}(\mathit{tl}(x))&: \eta \\
\mathit{if}\ (x = \mathit{nil})\ \mathit{then}\ 0\ \mathit{else} 1 + \mathit{length}(\mathit{tl}(x))&: \theta
\end{aligned}
$$

Then a system of type constraints can be set up according to the type rules of language constructs. For example, by the function abstraction rule, the type of the function *length* is $\beta \rightarrow \theta$. By the function application rule, the type of *x* in *tl(x)* must be the same as the argument type of *tl*, and the type of *tl(x)* must be the same as the resulting type of *tl*. That is, $\beta = \kappa\ list$, and $\varepsilon = \kappa\ list$. By the rule of the *if ... then ... else* construct, the type of *x=nil* is δ=*Boolean*, and the type of `0` and `1+length(tl(x))` must be the same. The important type constraints in this program can be summarized as follows:

$$
\begin{aligned}
\alpha &= \beta \rightarrow \theta \\
\beta &= \gamma\ \mathit{list} \\
\beta &= \kappa\ \mathit{list} \\
\varepsilon &= \kappa\ \mathit{list} \\
\varepsilon &= \beta \\
\zeta &= \theta \\
\zeta &= \mathit{Int} \\
\eta &= \mathit{Int} \\
\delta &= \mathit{Boolean} \\
\mathit{Int} &= \eta
\end{aligned}
$$

Now the problem of type resolution has become the problem of solving a set of type equations. What we want is to find substitutions for variables so that all the equations are satisfied. This is a *unification* problem, which was first studied by Robinson [115]. For our program, we can solve the problem by repeatedly applying five simple rules. Let *S* denote

the set of equations, and *C* denote a constant, such as *Int*, *Boolean*, and *v* denote the variables, such as α, β, and τ denote either a constant or a variable. The five rules are:

$$S \cup \{C = C\} \Rightarrow S \quad (1)$$

$$S \cup \{v = v\} \Rightarrow S \quad (2)$$

$$S \cup \{v = \tau\} \Rightarrow S[\tau/v] \cup \{v \cong \tau\} \quad (3)$$

$$S \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\} \Rightarrow S \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \quad (4)$$

$$S \cup \{\alpha\ list = \beta\ list\} \Rightarrow S \cup \{\alpha = \beta\} \quad (5)$$

Rules 1 and 2 eliminate trivial constraints. In rule 3, $[\tau/v]$ means replacing all occurrences of *v* with τ, and $v \cong \tau$ means marking this equation as solved. This rule eliminates a variable from all equations but one. Rules 4 and 5 apply structural equality to function and list types.

For the set of type equations for the *length* function, we can apply rule 3 to the equations ζ=*Int*, η=*Int*, and δ=*Boolean*, we obtain a new set of equations:

$$\begin{aligned} \alpha &= \beta \rightarrow \theta \\ \beta &= \gamma\ list \\ \beta &= \kappa\ list \\ \varepsilon &= \kappa\ list \\ \varepsilon &= \beta \\ \theta &= Int \end{aligned}$$

We can apply rule 3 again on θ=*Int*, and ε=β, and obtain:

$$\begin{aligned} \alpha &= \beta \rightarrow Int \\ \beta &= \gamma\ list \\ \beta &= \kappa\ list \\ \beta &= \kappa\ list \end{aligned}$$

Here, we can remove one of the duplicate constraints $\beta = \kappa$ *list*, and apply rule 3 again on the remaining one, we obtain:

$$\begin{aligned} \alpha &= \kappa\ list \rightarrow Int \\ \kappa\ list &= \gamma\ list \end{aligned}$$

Now, applying rule 5, we obtain:

$$\begin{aligned} \alpha &= \kappa\ list \rightarrow Int \\ \kappa &= \gamma \end{aligned}$$

By applying rule 3 one more time, we obtain the type of the *length* function α:

$$\alpha = \gamma\ list \rightarrow Int$$

In the process, we have also found the solution for other type variables: $\zeta=\eta=\theta=$*Int*, $\delta=$*Boolean*, $\varepsilon=\beta=\gamma$ *list*, and $\kappa=\gamma$. This solution is a substitutor from variable to expression that satisfies the equations. This substitutor is also called a *unifier*.

Notice that the solution for the type of the *length* function contains a type variable γ. This means that this function is polymorphic. Type variables can be viewed to be universally quantified. So the function *length* can be applied to a list of any type, and it returns an integer. This is parametric polymorphism.

Also notice that the solution for the type constraints is not unique. Other solutions include $\alpha =$ *Int List* $\rightarrow$ *Int*, $\alpha =$ *Boolean List List* $\rightarrow$ *Int*, and $\alpha = \kappa$ *List* $\rightarrow$ *Int*. In fact, there are infinite number of solutions. Nevertheless, the solution we have obtained is the *most general unifier* in that all the other solutions are *substitution instances* of the most general solution. For example, by substituting γ with *Int*, *Boolean List*, and κ, we obtain the above three solutions, respectively. In this sense, the ML type system computes the most general types. This most general type is also called the *principal type*. It is unique up to a renaming of type variables.

The algorithm we just used to find the most general unifier is not the optimal one. A faster algorithm can be found in [7]. For a comprehensive discussion on unification theory, please see [8].

The type system discussed above was first proposed by Hindley and later independently rediscovered by Milner [85][20]. In addition, Milner introduced a crucial extension to Hindley's work: the notion of generic and non-generic type variables. To understand this notion, consider the following program:

```
fun h(f) = (f(3), f(true));
```

In ML syntax, `(a, b)` is a tuple with elements *a* and *b*. Tuples can be viewed as records without labels. The above program defines a function *h* that takes an argument *f*, which is also a function. The body of *h* applies *f* to two arguments, *3* and *true*, and returns the result

as a tuple. This program cannot be typed in ML. For example, in the Standard ML of New Jersey compiler[1], the following error is reported:

```
- fun h(f) = (f(3), f(true));
stdIn:13.12-13.27 Error: operator and operand don't agree [liter-
al]
  operator domain: int
  operand:         bool
  in expression:
    f true
```

This is because the type inference algorithm cannot unify the types for the two occurrences of `f`. Suppose the type of the function `f` is denoted by $\alpha \rightarrow \beta$. Based on the first occurrence of `f` in `f(3)`, the type inference algorithm unifies α with *Int*. However, the second occurrence `f(true)` requires α to be *Bool*, which cannot be unified with *Int*.

Type variables appearing in the type of a `fun`-bound identifier like `f` are called *non-generic*. In this example, α is non-generic. It is shared among all the occurrences of `f` and its instantiations may conflict. Therefore, heterogeneous occurrences of `fun`-bound identifiers cannot be typed in ML.

However, if we know what `f` is, we should be able to do better. Indeed, ML has a construct `let ... in ... end` that can be used for this purpose. Suppose we first define an identity function `g`:

```
fun g(x) = x;
```

Then we can write:

```
let
     val f=g
in
     (f(3), f(true))
end;
```

Here, we specify that `f` is equal to the identity function `g`, then apply `f` to `3` and `true`. This program can be complied and executed with the correct result `(3, true)`. In this case, `f` has type $\alpha \rightarrow \alpha$. Type variables which, like this α, occur in the type of `let`-bound identifiers are called *generic*. They can assume different values for different instantiations

1. Standard ML of New Jersey, version 110.0.7. (http://cm.bell-labs.com/cm/cs/what/smlnj/)

of the `let`-bound identifier. This is achieved by making a copy of the type of `f` for every distinct occurrence.

The `let` polymorphism discussed above is considered one of the most important advances in many years. The type system in ML in general has made great impact. Many people have proposed extensions. For example, Mitchell extended the system with coercion [87], Hall, *et al.* extended the system with type classes to handle overloading [50], and Fuh and Mishra extended the system with subtypes [45]. The technique is also applied outside functional languages. In the visual language area, the type systems of several languages, including Forms/3 [18], ESTL [91], and CUBE [92] are based on the ML type system. A survey of the type systems for visual languages can be found in [15]. The basic idea has also been applied to the logic programming language Prolog [90].

2.2 Component-Based Design

2.2.1 Component

Component-based design has been established as an important approach to designing complex systems. In hardware design, people have long been assembling systems from commercial off-the-shelf (COTS) components. In recent years, with the adoption of system-on-a-chip (SoC) design, systems are often assembled from virtual components, which are intellectual property (IP) blocks. Some industry organizations, such as the Virtual Socket Interface Alliance (VSIA) [71] were formed during this trend to standardize the specification, interface, and protection of the virtual components.

In software, an industry that produces commercial software components (routines) was envisioned as early as 1968 [80]. However, the concept of software components did not catch on until recent years, after the widespread acceptance of object-oriented programming and the emergence of industry backed component architectures. Today there are three major forces in industry in the component software arena: the CORBA-based standards backed by the Object Management Group, the COM-based standards backed by Microsoft, and the JavaBeans-based standards backed by Sun. Component software also attracted many researchers in academia. The book by Szyperski [119] provides an excellent cover-

age on various aspects of component software, from both the technology and market perspective.

Although component-based design has become a widely used term in both industry and academic research community, there is no standard definition for the word component. It seems that every group has their own definition. In the Workshop on Component-Oriented Programming [122], a software component is defined as “a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” The OMG UML specification [101] defines component as “a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of a system’s implementation, including software code (source, binary or executable) or equivalents, such as scripts or command files.” The component description model in Microsoft Repository [13] defines component as “a software package that offers services through interfaces.” A list of other definitions can be found in [119]. People have also discussed the importance of various aspects of components. For example, Meyer and Szyperski engaged in a series of discussions on topics related to software components, including information hiding, binary vs. source, contracts between components [82][83][84][120][121]. Meyer also gave seven criteria for components [83]. Other people have given definitions for components with slightly different emphasis [37].

Although these definitions emphasize different aspects of components, at an abstract level, most of these definitions boil down to two basic points:

- Encapsulation: a component encapsulates behavior and state.
- Interface: a component interact with its environment through an interface.

In this thesis, we will view components at this abstract level, and largely ignore the other aspects, such as their function, complexity, implementation, and source. For example, a component can be as simple as an adder, or as complex as a video encoder. Components can be implemented either by software, or hardware, and can be developed in-house or provided by a third party.

In the Ptolemy II software, which is the experimental platform for the type system presented in this thesis, components are called actors. In this context, component and actor are used interchangeably.

2.2.2 Advantages of Component-Based Design

In component-based design, a system is developed by composing components. Some well recognized benefits of this approach include:

- Reuse. By reusing pre-designed components, companies can save development cost, significantly reduce the time to market, and address the shortage of good software and hardware developers.
- Clarify system structure.
- Simplify verification.
- Platform or language independence (for some software components).
- Permit dynamic system re-configuration.
- Protect the intellectual property associated with the components through encapsulation.

Fundamentally, components raise the level of abstraction in the design process. If we look at the evolution of programming languages that predates the adoption of software components, we see a process of increasing the levels of abstraction. In the early days of computers, people performed all the programming tasks using assembly languages, which reflect the structure of the underlying machines. This low level of abstraction makes programming tedious and error-prone. High-level languages such as FORTRAN raised the abstraction level from machine instruction to algebraic formulae, greatly improved the productivity. However, the FORTRAN family of imperative languages still follow the von Neumann model of computation. Backus and many other researchers proposed to use functional abstraction [9][56], and argue that functional programs are easier to understand, and their correctness can be justified by strict mathematics. In recent years, object abstraction, embodied in object-oriented languages, has received widespread adoption. However, neither functional nor object abstraction alone is enough for the design of heterogeneous concurrent systems. Lee proposed to raise the abstraction level by adopting *actor-oriented design* [62]. His actor model emphasizes concurrency and communication abstractions, and admits time as a first-class concept.

In component-based design, a system is often represented as a block diagram. One key advantage of the block diagrams is that they reflect the topology of component communication. In other approaches, such as implementing the whole system using a concurrent language, the communication topology may not be readily available. For example, if a system is implemented in Concurrent ML (CML) [113], a sophisticated algorithm must be used to infer the topology from the source program [99].

2.2.3 Challenges of Component-Based Design

When assembling components to form a system, an obvious question arises: "Can the components work together?"

By analogy, if we assemble a stereo system from components such as CD players, tuners, and amplifiers, we need to first ensure that the connectors between components have matching shape and size. And then we need to ensure that the connected boxes use the same signal protocol. For analog systems, this means that the source component must supply a signal within the voltage range expected by the receiver, and other circuit specifications, such as impedance, must be compatible. For digital systems, the communication protocol, encoded by the bit sequence flowing through the connection, must also be compatible between the source and the receiver.

For software components to work together, they also have to be compatible in at least two levels. One is the data type level. For example, if a component expects to receive an integer at its input, but another component sends it a string, then the first component may not be able to function correctly. The other level of mismatch is the dynamic interaction behavior, such as the communication protocol the components use to exchange data. Since embedded systems often have many concurrent computational activities and mix widely differing operations, components may follow widely different communication protocols. Ensuring compatibility at component interfaces is one of the major challenges in component-based design, and is the main goal of this thesis. These two levels of compatibility are also observed by other researchers [30].

The existing component software standards, such as CORBA, COM, and JavaBeans, are not good match for embedded system design. CORBA is oriented toward corporate

enterprise computing, COM was born out of a desktop environment, and Java-based standards are oriented toward internet computing. These standards are basically distributed object-oriented models, and they do not address the heterogeneity and concurrency issues in embedded systems. We should point out that the development of CORBA services such as event service, is moving in the right direction. Another reason that these standards are not good match for embedded systems is that they are mostly wiring standards that define the interconnection of components [119]; and they do not directly support the specification of the dynamic properties and constraints of component interface. As Meyer says, "we badly need more expressive Interface Definition Languages for both CORBA and COM to support the expression of semantic constraints" [81]. Because of this, many desired properties for embedded systems, such as determinacy, bounded memory usage, and deadlock freedom, are hard to verify in these standards. To design systems with some of these properties, a good approach is to use computation models that inherently offer these properties, or allow easy verification of them.

2.3 Models of Computation

In [61], Lee defines Model of Computation as:

> *A model of computation is the "laws of physics" of concurrent components, including what they are (the ontology), how they communicate and how their flows of control are related (the protocols), and what information they share (the epistemology).*

For embedded system design, we need to use models of computation that support concurrency and combine different models in a structured way to cope with the heterogeneity and complexity of the system. In the rest of this section, we will review some examples of models of computation that are suitable for embedded system design. This material is drawn from [62] and the references therein.

As discussed earlier, systems designed using a component-based approach are often depicted using block diagrams, like the one in figure 2.3. We will use this figure as the common syntax for the models of computation below. In the spirit of actor-oriented design [62], the components A, B, and C are also called actors.

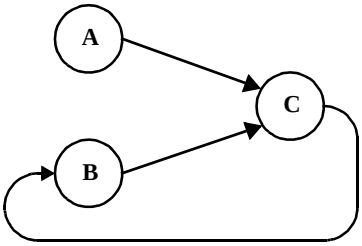


Figure 2.3 Multiple MoCs can share the same block diagram syntax.

Dataflow

In dataflow models, actors implement atomic computations that are driven by the availability of input data, and the connections between actors represent data streams. Certain restrictions on the general dataflow models can yield extremely useful properties. In synchronous dataflow (SDF), the number of data samples produced or consumed by each actor on each invocation is specified a priori, and the actors can be scheduled statically onto single or multiple processors [64][65]. As a result, deadlock and boundedness are decidable. These properties make SDF an excellent model for specifying digital signal processing systems. Boolean dataflow (BDF) models sometimes yield to deadlock and boundedness analysis, although fundamentally these questions are undecidable [17]. Dynamic dataflow (DDF) uses only run-time analysis, and thus makes no attempt to statically answer questions about deadlock and boundedness. Dataflow Process Networks [66] encompass both possibilities.

Discrete Events

In discrete-event (DE) models of computation, the actors communicate via sequences of events placed along a real time line. An event consists of a value and time stamp. This and several other models can be formally described using the tagged signal model [60][67]. The DE model is the basis for many simulation environments and hardware description languages, including VHDL and Verilog.

Although DE models are excellent for specifying digital hardware and for simulating telecommunication systems, the notion of global time in the model often causes difficulty in describing distributed systems. It is also relatively expensive to implement in software.

Asynchronous Message Passing

In this model, the actors are processes that communicate with each other through asynchronous message passing. The connections between actors are channels that can buffer messages. After an actor sends out a message to the channel, it can start other computations without waiting for the receiver to receiver the message. Kahn process networks [58] are a particular case of asynchronous message passing, where the connections represent sequences of data values, and the actors implement functions that map input sequences into output sequences. With certain technical restrictions on these functions, this model is deterministic, meaning that the sequences are fully specified. The dataflow models discussed above are special cases of process networks [66].

Process network (PN) models are natural for describing signal processing applications [72]. They can be implemented efficiently in both hardware and software. However, they are weak in specifying complicated control logic.

Synchronous Message Passing

In this model, the actors are processes that communicate with each other through rendezvous. If a process wants to send a message, it blocks until the receiving process is ready to accept it. Similarly, if a process wants to receive a message, it blocks until the sending process is ready to send it. When both processes are ready, the communication is conducted in a single uninterrupted step. An example of a rendezvous-based model is Hoare's communicating sequential processes (CSP) [54]. This model is non-deterministic as it includes conditional communication constructs. This model of computation is well suited for resource management problems.

Synchronous/Reactive

In the synchronous/reactive (SR) model of computation [12], the connections between the actors represent signals whose values are aligned with global clock ticks. The actors represent relations between the input and output signals at each clock tick. A signal need not be present at every clock tick, and the actors are usually implemented as partial functions with certain technical constraints to ensure determinacy.

The SR models are good for applications with concurrent and complex control logic. The tight synchronization in the model makes it suitable for designing safety-critical real-time applications. However, the tight synchronization requirement makes it hard to model distributed systems, and compromises modularity in the design.

In addition to the models discussed above, there are many other models, such as time triggered, publish and subscribe, continuous time and finite state machines. They are described in [62].

2.4 Mathematical Tools

2.4.1 CPO, Lattice, and Fixed Point Theorems

The type system that will be presented in the next few chapters is based on the mathematics of lattices, continuous functions, and fixed point theorems. This is a standard set of mathematical tools in the study of programming language semantics and type systems. For example, in the denotational semantics of programming languages [117][126], the denotation of a command is the least fixed point of a continuous function on a CPO. The elements of the CPO are functions that map one state to another. In concurrent programming models, the process network [58] and the synchronous reactive [39] models also have fixed point semantics. Fixed point theorems are handy in dealing with "circular systems", such as the `while` construct in general purpose languages, and circular graphs in PN and SR. We briefly review some basic definitions and results here, mostly for establishing notation. This review is based on [32][39].

2.4.1.1 CPOs and Lattices

Let P be a set. A *partial order* relation on P is a binary relation $\leq$ such that, for all $x, y, z \in P$,

- $x \leq x$ (reflexive)
- $x \leq y$ and $y \leq x$ imply $x = y$ (antisymmetric)
- $x \leq y$ and $y \leq z$ imply $x \leq z$ (transitive)

A set P equipped with such a relation is said to be a *partially ordered set*, or *poset*.

A finite partially ordered set can be depicted by a *Hasse diagram*, such as the ones in figure 2.4. The lines in the diagram represent the order relation, where the element at the lower end of the line is *less than* the one at the higher end of the line. For example, in figure 2.4(a), *A* is less than *B* and *C*, and *E* is greater than *B* and *C*. Two elements can be *incomparable*. For example, *D* and *E* are incomparable.

If *P* is a poset and $S \subseteq P$, an element $x \in S$ is the *least element* of *S* if $x \leq s$ for all $s \in S$. The *greatest element* is defined dually. The least and greatest elements of a set may not exist, but if they do, they are unique. For example, for the set {*B*, *D*, *E*} in figure 2.4(a), the least element is *B*, and there is no greatest element.

The *bottom element* of a partially ordered set, if it exists, is the least element of the whole set. Similarly, the *top element*, if it exists, is the greatest element of the whole set. The bottom element is denoted by $\perp$, and the top element is denoted by $\top$. In figure 2.4(a), *A* is the bottom element, and there is no top element. In 2.4(b), *A* is the bottom element, and *G* is the top element.

Lattices and CPOs are posets with some special structure. Before we give their definitions, we need to define the upper bound and lower bound of a subset.

Let *S* be a subset of a partially ordered set *P*. An element $x \in P$ is an *upper bound* of *S* if $s \leq x$ for all $s \in S$. The *least upper bound* of *S*, denoted by $\vee S$, is an upper bound *l* of *S* such that $l \leq u$ for all upper bounds *u* of *S*. The *lower bound* and *greatest lower bound* are defined dually. The greatest lower bound of a set S is denoted by $\wedge S$. The least upper bound and greatest lower bound of two elements *x* and *y* can also be denoted by $x \vee y$

(a) (b)

Figure 2.4 Hasse diagrams for two partially ordered sets.

and $x \wedge y$, respectively. In figure 2.4(a), D and E are upper bounds of the subset $\{B, C\}$, but there is no least upper bound for this subset. In 2.4(b), F and G are upper bounds of the subset $\{B, C\}$, and F is the least upper bound.

Consider the case where the set P has a bottom and a top element. If the set S in the above definition is P itself, it is easily seen that the least upper bound of S is the top element of P, and the greatest lower bound of S is the bottom element. Now let S be the empty subset of P. Then every element $x \in P$ vacuously satisfies $s \leq x$ for all $s \in S$. Therefore, every element of P is an upper bound of S, and the least upper bound is the bottom element of P. Dually, the greatest lower bound of the empty set S is the top element of P.

A non-empty partially ordered set P is a *lattice* if $x \vee y$ and $x \wedge y$ exist for all $x, y \in P$. If $\vee S$ and $\wedge S$ exist for all $S \subseteq P$, then P is a *complete lattice*.

A *chain* is a totally ordered set. That is, a set S is a chain if, for all $x, y \in S$, either $x \leq y$ or $y \leq x$. A chain appears as an upward path in a Hasse diagram.

A *complete partially ordered set* (*CPO*) is a poset P in which every chain in P has a least upper bound in P. All the posets discussed in this thesis are CPOs.

2.4.1.2 Fixed Point Theorem

Let P and Q be posets. A function f: $P \rightarrow Q$ is *monotonic* if $x \leq y$ in P implies $f(x) \leq f(y)$ in Q. A monotonic function is order-preserving.

A function f: $P \rightarrow Q$ between posets P and Q is *continuous* if for all chains $S \subseteq P$, $f(\vee S) = \vee f(S)$, where $f(S)$ is $\{f(s) \mid s \in S\}$. If we view the least upper bound of a set as its limit, continuous functions are limit-preserving. All continuous functions are monotonic.

For functions whose domain and range are the same poset, we can define the fixed point of the function.

Let P be a poset, $f{:}P \rightarrow P$ be a function, and $x, y \in P$. If $f(x) = x$, then x is a *fixed point*. In the set formed by all the fixed points of f, the least element, if it exists, is called the *least fixed point*. Since the least element of a set is unique, the least fixed point is unique.

The following fixed point theorem gives a way to find the least fixed point of a continuous function on a CPO.

Let P be a CPO with a bottom, and f: $P \rightarrow P$ be a continuous function. Then $\vee\{\perp, f(\perp)$**,** $f(f(\perp))$, ..., $f^k(\perp)$, ...$\}$ exists and is the unique least fixed point of f.

2.4.2 Interface Automata

In chapter 4, we will use a formalism called interface automata [34] to describe the interaction of components. Interface automata were proposed by de Alfaro and Henzinger. They are a light-weight formalism for the modeling of components and their environments. We give a high-level overview of interface automata in this section. Details can be found in [34].

2.4.2.1 An Example

As other automata models, interface automata consist of states and transitions[1], and are usually depicted by bubble-and-arc diagrams. There are three different kinds of transitions in interface automata: input, output, and internal transitions. When modeling a software component, input transitions correspond to the invocation of methods on the component, or the returning of method calls from other components. Output transitions correspond to the invocation of methods on other components, or the returning of method calls from the component being modeled. Internal transitions correspond to computations inside the component.

For example, figure 2.5 shows an interface automaton model of a software component called *Comp* that provides a message-transmission service. This and the other examples in this section are drawn from [34]. The automaton was constructed in the Ptolemy II software, and the figure is a screen shot of Ptolemy II. The convention in interface

1. Transitions are called actions in [34].

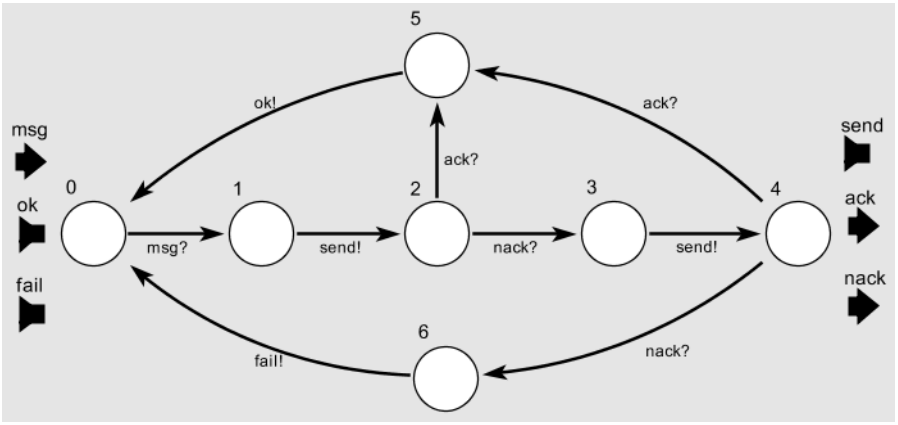


Figure 2.5 An interface automaton modeling a communication component.

automata is to label the input transitions with an ending "?", the output transitions with an ending "!", and internal transitions with an ending ";". Figure 2.5 does not contain any internal transitions. The *Comp* component has a method *msg*, used to send messages. When this method is called, the component calls a *send* method on a low-level communication channel to send the message. The send method may return either *ack*, indicating a successful transmission, or *nack*, indicating a failure during transmission. If *ack* is returned, *Comp* returns from the *msg* method with a return value *ok*. If *nack* is returned, *Comp* calls *send* once more to re-send the message. If the second attempt is successful, *Comp* returns *ok*, otherwise, it returns *fail*. The block arrows on the sides of figure 2.5 denote the inputs and outputs of the automaton. The three arrows on the left side correspond to the interface with the users of *Comp*, namely, the *msg* method and its return values *ok* and *fail*. The three arrows on the right side correspond to the interface with the channel, which is the *send* method and its return values *ack* and *nack*.

This example illustrates an important characteristic of interface automata. That is, they are not input enabled. In another words, they do not require all the states to accept all inputs. In figure 2.5, the input *msg* is only accepted in the initial state 0, but not in any other states. In fact, the illegal inputs are used to encode assumptions about the environment. These assumptions state, among other things, that once the *msg* method is called, the environment should not call this method again until an *ok* or *fail* is returned. This way of encoding environment assumptions also eliminates the need of using

explicit states to model error conditions. This example shows that interface automata take an optimistic approach for modeling, and they reflect the intended behavior of components under a good environment. As a result, interface automata models are usually more concise than other automata-based formalisms, such as I/O automata [78], where every input must be enabled at every state.

2.4.2.2 Composition and Compatibility

Two interface automata can be composed if their transitions do not overlap, except that an input transition of one may coincide with an output transition of the other. These overlapping transitions are called shared transitions. Shared transitions are taken synchronously, and they become internal transitions in the composition. Figure 2.6 shows an automaton that can be composed with *Comp*. This automaton models an user of the communication component. It always expects successful transmission of the messages. When composed with *Comp*, *msg*, *ok*, and *fail* become shared transitions, and the composition result is shown in figure 2.7.

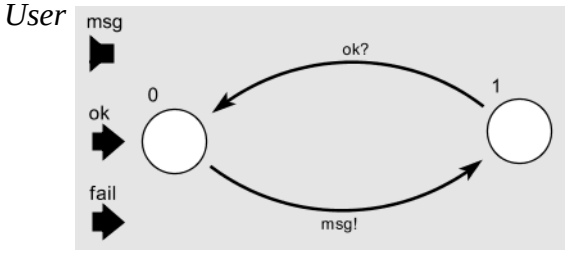


Figure 2.6 An interface automaton modeling a user of the component *Comp*.

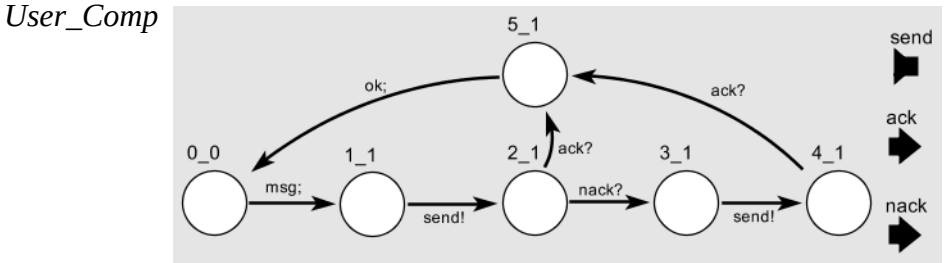
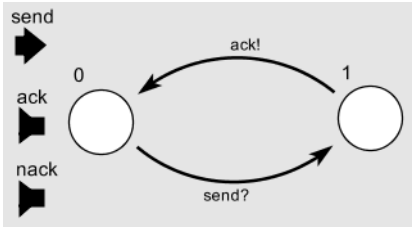


Figure 2.7 Composition of *User* and *Comp*.

Notice that the composition is smaller than the product of *User* and *Comp*. This is also due to the optimistic approach of interface automata. Under this approach, error conditions are not explicitly modeled. For example, if the *User* automaton is in state 1 and *Comp* is in state 6, *Comp* may make the transition *fail*. However, since *fail* is not accepted by *User*, the pair of states (1, 6) is *illegal* in the composition (*User*, *Comp*). In interface automata, illegal states are pruned out in the composition. Furthermore, all states that can reach illegal states through output or internal transitions are also pruned out. This is because the environment cannot prevent the automaton from entering illegal states from these states. The resulting composition reflects the environment assumption that no two consecutive transmissions fail.

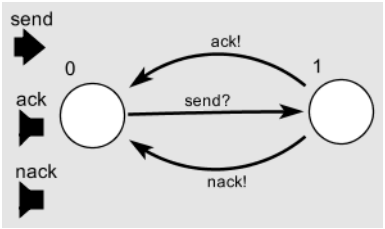
We can further compose the automaton *User_Comp* with a model of the low-level channel. Figure 2.8 shows two channel models. The good channel always returns *ack* on each send request, but the bad one nondeterministically returns *ack* or *nack*. When composing *User_Comp* with the good channel in figure 2.8(a), we obtain the composition in figure 2.9(a). This is a closed system in which all of the transitions are internal transitions. This model describes the behavior that communication is always successful on the first attempt. If we compose *User_Comp* with *BadChannel*, we obtain an empty automaton shown in figure 2.9(b). This is because the bad channel does not satisfy the assumption that no two consecutive transmissions fail. In particular, the pair of states (4_1, 1) in (*User_Comp, BadChannel*) is illegal because the bad channel may issue *nack* in state 1, which cannot be accepted by *User_Comp* in state 4_1. Since this illegal

GoodChannel



(a)

BadChannel



(b)

Figure 2.8 Two channel models.

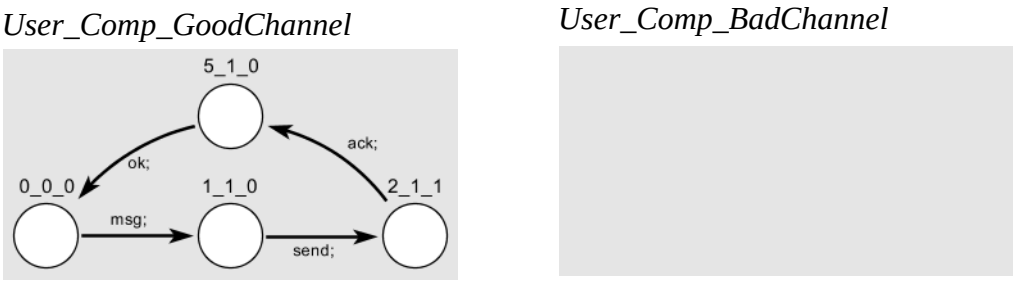


Figure 2.9 Composition of *User_Comp* with two channel models.

state is reachable by the initial state of the composition through internal transition, the initial state is prune out. As a result, the whole composition is empty.

The above examples illustrate the key notion of *compatibility* in interface automata. Two automata are compatible if their composition is not empty. This notion gives a formal definition for the informal statement “two components can work together”. The composition automaton defines exactly how they can work together.

Under the optimistic approach to composition in interface automata, two components are compatible if there is some environment that can make them work together. In the traditional pessimistic approach, two components are compatible if they can work together in all environments. Because of this difference, the composition of interface automata is usually smaller than the composition in other automata models.

2.4.2.3 Alternating Simulation

In conventional automata settings, several relations between automata have been studied, such as trace equivalence, simulation, and bisimulation [51][68]. In the area of system design, these relations are sometimes used as the refinement relations between the specification and implementation of systems. The simulation relation ensures that the output behaviors of the implementation are behaviors that are allowed by the specification. It also requires that the set of legal inputs of the implementation is a subset of the inputs allowed by the specification. In the non-input-enabled setting, such as interface automata, this requirement is not appropriate, because it could restrict the implementation to work in

fewer environments than the interface specification. This problem motivated the authors of interface automata to use *alternating simulation* to define refinement. Informally, for two interface automata *P* and *Q*, there is an alternating simulation relation from *Q* to *P* if all the input steps of *P* can be simulated by *Q*, and all the output steps of *Q* can be simulated by *P*. The formal definition also involves internal transitions, and is given in [34]. Under the alternating simulation relation, one interface refines another if it has weaker input assumptions, and stronger output guarantees.

Refinement and compatibility are related. A theorem states that if a third automaton *R* is compatible with *P*, then *Q* and *R* are also compatible, provided that *P* and *Q* are connected to *R* by the same inputs. The formal statement of this theorem can be found in [34]. Essentially, this theorem states that a component *P* can be replaced with a more refined version *Q* in the environment *R*.

3 Data Types

In this chapter, we present a type system for a block diagram based design environment. This system has been implemented in the Ptolemy II software [33]. We focus on the formulation and the design issues in this chapter and discuss the implementation details in chapter 5.

3.1 Introduction

3.1.1 Abstract Syntax and High-level Semantics

Before we start to design a type system, we must have a syntax for the language on which the type system can be implemented. For text based languages, most research papers use the notation of λ-calculus. For the discussion in this chapter, we use a syntax adapted from the abstract syntax for actor-oriented designs [62]. As shown in figure 3.1, each of the components is an *actor*, and actors contain *parameters* and *ports*. Ports are denoted by the small circles on the actors, and they are connected through *connections*.

The abstract syntax in [62] is more general than the one shown in figure 3.1. Among other things, the connections in figure 3.1 have directions. This implies a high-level

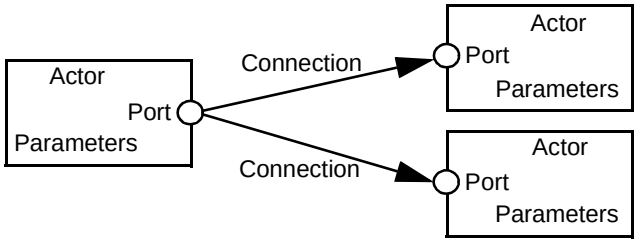


Figure 3.1 An abstract syntax for block diagram based language.

semantics of message passing. In this chapter, we assume that actors send and receive messages through ports, and 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. Here, the detailed interaction semantics in various MoCs is ignored. This abstraction enables the same type system to work with widely differing models.

The abstract syntax in [62] can also be used as the basis for state transition systems. There are some interesting type system issues when mixing state transition systems with message passing systems, such as in modal models [48]. These issues will be discussed later in this chapter.

3.1.2 Design Goal

In figure 3.1, the interconnections imply type constraints. For example, if an actor expects to receive integers from one of its input ports, then the output port that is connected to that input should not send out strings. In addition, actors themselves may have constraints among their ports and parameters. For example, an actor designer may want to specify that the type of the tokens sent out from a port is the same as the type of an internal parameter. The primary role of a type system is to support the specification of these kinds of constraints, and to ensure their satisfaction.

As discussed in chapter 2, many type systems play a larger role than just ensuring type safety. For example, many type systems provide type conversion services, most modern type systems support program reuse through certain kind of polymorphism, and type information can be used for program optimization. For component-based design, we also want our type system to do more than type checking. In particular, we want to support the following features:

- *Type conversion*: Conversions between primitive types, such as *Int* to *Double*, happen frequently in programs. We want to perform these kinds of conversions automatically in the system so that the actor designers do not need to be burdened with the conversion tasks. As a design principle, we will only support conversions that do not lose information. For example, using the IEEE 754 standard, a 32 bit integer can be losslessly converted to a double, but the reverse is not true.

- *Subtyping*: In recent years, a large percentage of software development has been done using object-oriented languages, such as Java or C++. For example, Ptolemy II is developed in Java, and many commercial system design tools are developed in C++. Subtyping is one of the main features of these languages. To support this feature, we want our type system to recognize the subtyping relation among types.
- *Polymorphism*: since the primary benefit of component-based design is reuse, the type system should be polymorphic, so that some actors can be reused in different settings with different types. We call these actors *data polymorphic actors*.
- *Supporting design optimization*: Polymorphic actors may have more than one possible type assignment. In this case, the type system should find a typing that has a lower cost of implementation.
- *Structured types*: In addition to primitive types, we want to support structured types such as arrays and records.
- *Extensibility*: Since the application areas and the technology used in designing embedded systems are diverse, designers may sometimes want to add new types to the design environment. Therefore, the type system should be extensible to accommodate this.

3.1.3 Our Approach

Type conversion among primitive types and subtyping naturally imply an ordering relation among all the types. In our system, we organize all the types into a *type lattice*. The reason to restrict the partial order to be a lattice will become clear later.

To express the typing requirements across connections and inside the actors in figure 3.1, we give each port and each parameter a type. The type of the port restricts the type of the token that can pass through it. Inspired by ML, we take a constraint solving approach in our type system. In particular, we use type variables to denote the type of polymorphic actors and set up type constraints in term of the variables and constant types. The format of type constraints in our system is different from that in ML. In ML, type constraints are type equations. In our system, they are inequalities defined over the type lattice. The constraint-base approach handles recursion or feedback loops well, because once the constraints are set up, the constraint solving process does not take the program structure into consideration anymore. Another advantage of this approach is that constraint resolution can be separated from constraint generation, and resolution can employ a sophisticated algorithm. Although the actor designers and tool 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, the types are not aware of the constraints, so more types can be added to the type lattice, resulting in an extensible type system.

Type constraints can be set up based on the topology of the block diagram and the specification of actors. The collection and resolution of the constraints can be performed statically before the model executes. This has obvious advantages. However, static checking alone is not enough to ensure type safety at run-time. In general, many block diagram based environment, such as Ptolemy II, can be viewed as coordination languages [27]. Their type systems do 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, many design tools place 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 [75]. Therefore, even if a source actor declares its port type to be *Int*, no static structure prevents it from sending a token containing a 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. Therefore, we combine both static and run-time checking in our system. 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.

Static type information can also be used to perform 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 can be done transparently by the type 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.

The rest of this chapter is organized as follows. Section 3.2 presents the formulation of our type system, including the type lattice, type constraints, and type checking. Section 3.3 extends the system to include structured types, and section 3.4 describes ways to express more involved type constraints. The last section discusses several design decisions.

3.2 Formulation

3.2.1 Type Lattice

As mentioned above, the type lattice represents the type conversion relation among primitive types and the subtype relation among other types. In systems with subtyping, such as $F_{1\leq}$ discussed in section 2.1.3.1, the subtyping relation is usually reflexive and transitive. We add two additional requirements in our system. First, we require the relation among types to be antisymmetric so that the set of types form a CPO. Secondly, we require the least upper bound and the greatest lower bound of each pair of types to exist so that the CPO becomes a lattice. As will be shown later, type constraints over a lattice can be solved using a very simple and efficient algorithm.

In each language, there is a set of base types. This set is often slightly different from language to language. The type lattice in Ptolemy II, which includes all the base types, is shown in figure 3.2. In Ptolemy II, data are encapsulated in a set of Java classes. The base class is called *Token*, and all the other token classes, such as *StringToken*, *IntToken*, are derived from it. Following the convention of the Hasse diagram discussed in section 2.4.1, a type α in figure 3.2 is greater than a type β if there is a path upwards from β to α. Thus, *ComplexMatrix* is greater than *Int*, and *Int* is less than *ComplexMatrix*. Two types can be 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, *UNKNOWN*, does not correspond to a token. Users can extend a type lattice by adding more types.

The ordering relation of the type lattice is a combination of the lossless type conversion relation among primitive types, such as $Int \leq Double$, and the subclass relation of the token classes, such as $String \leq General$. Since the type conversion relation among primitive

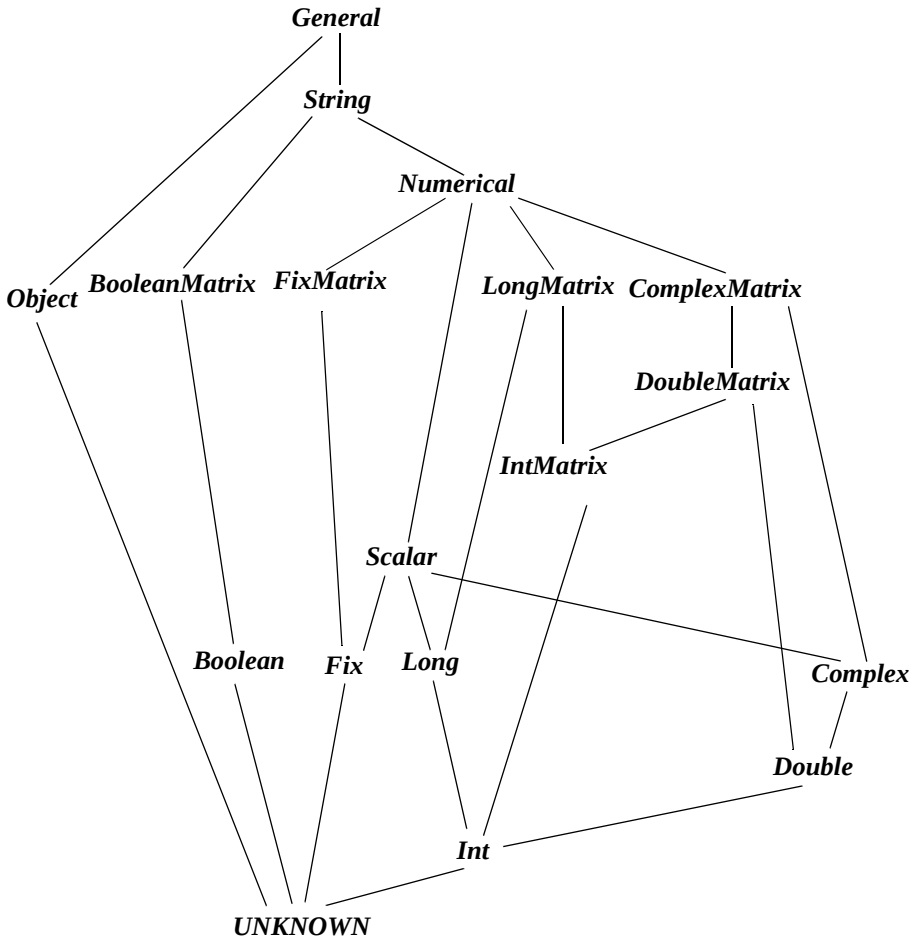


Figure 3.2 An example of a type lattice.

types can be viewed as ad hoc subtyping [87], we can say that the relation in the type lattice represents two kinds of subtyping relations.

3.2.2 Type Constraints

As mentioned earlier, each port has a type. This type can be declared by the actor writer, or left undeclared, in which case the type system will resolve the type when solving the type constraints. Type resolution can be viewed as a form of type inference.

In object-oriented languages, a subtype object can be used in place of a supertype object. Similarly, in block diagram based languages, if an input port expects to receive tokens of a certain type *receiveType*, the output port that is connected to that input

should be allowed to send out tokens of a subtype, as well as the same type as *receiveType*. Therefore, across any connections, we require the type of the port that sends tokens to be the same as or less than the type of the receiving port:

$$sendType \leq receiveType \quad (1)$$

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 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 figure 3.3.

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:

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

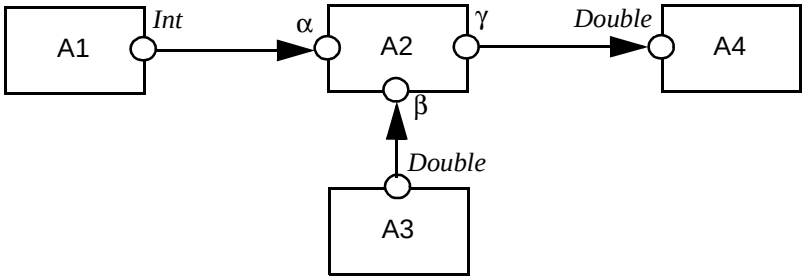


Figure 3.3 A topology (interconnection of components) with types.

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 that 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.

In the Ptolemy II implementation, the adder can also work on structured types such as arrays and records. These types will be discussed in section 3.3.

3.2.3 Type Resolution Algorithm

The above 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 [112]. Essentially, the algorithm 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. It is unique [32], and satisfies the constraints if it is possible to satisfy the constraints.

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 figure 3.2, *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 *UNKNOWN*, and *Cvar* looks like:

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

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

$$\alpha = Int \vee UNKNOWN = Int \qquad (2)$$

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*
$Double \le \beta(UNKNOWN)$	$Int \le \alpha(Int)$
$\alpha(Int) \le \gamma(UNKNOWN)$	$\beta(UNKNOWN) \le \gamma(UNKNOWN)$

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.

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. The function 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, $A = \perp^N$, where $\perp = UNKNOWN$, and computes the sequence $F^i(\perp^N)$ $(i \ge 0)$, which is a non-decreasing chain. By the fixed point

theorem in [32], 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 [112] 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.

If the set of type constraints is not satisfiable, or some type variables are resolved to *UNKNOWN*, the static type checker flags a type conflict error. The former case can happen, for example, if the port on actor `A1` in figure 3.3 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 *UNKNOWN*, it will stay at *UNKNOWN* 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 *UNKNOWN*.

The type constraints discussed in this section only involve constant types and type variables. In section 3.3 and 3.4, we will see more complicated constraints that involve structured types and monotonic functions.

3.2.4 Run-time Type Checking and Lossless Type Conversion

The declared type is a contract between an actor and the type 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. That is, 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.

3.3 Structured Types

3.3.1 Goals and Problems

Structured types are very useful for organizing related data and make 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. Both kinds of tokens allow multiple pieces of information to be transferred in one round of communication, making the execution more efficient. In addition, record tokens can be used to reduce the number of ports on certain actors, which simplifies the topology of the block diagram.

In our type 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. For example, we can have an array token whose elements are also array tokens, that is, an array of arrays. Also, we can have an array of records, 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 two features in the framework of our type system, we need to overcome some technical difficulties. In particular, 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 will become 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 examples:

- {*1.4, 3.5*}: An array containing two double values, 1.4 and 3.5.
- {*Double*}: The type of the above array.
- {{*1*, *2*}, {*3*, *4*}}: An array of arrays.
- {{*Int*}}: The type of the above array.
- {*name*="*foo*", *value=1*}: A record with two fields. One field 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.

3.3.2 Ordering Relation

In section 2.1.3.1, we discussed that subtyping for mutable arrays is neither covariant nor contravariant. This means that if a subtyping relation is defined on mutable arrays, static checking alone is not enough to ensure type consistency. One way to obtain subtyping in arrays is to use a run-time check, as is done in Java. Another way, which we use in our type system, is to disallow the contents of the arrays to be changed after they are initialized. That is, to make arrays immutable.

By making the arrays immutable, the elements of the arrays must be specified when constructing the array. Once the array is constructed, the elements cannot be changed. This restriction is usually not acceptable for general purpose text based languages. However, for block diagram based languages, making the arrays immutable is justifiable, or even desirable. In our case, the arrays are encapsulated in array tokens, which 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 the arrays are mutable, then when we send an array token to multiple actors, we will want to make copies of the array and send each receiving actor a new copy. Otherwise, multiple actors will share the same mutable array and the modification by one actor will affect the operation of the other. 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, it is desirable to make all tokens immutable. This way, no copying is necessary during token passing, and the communication between actors become more efficient.

For immutable arrays, we can define subtyping in a covariant way. That is, if $\tau_1 \leq \tau_2$, then $\{\tau_1\} \leq \{\tau_2\}$. This is the ordering relation we use in our type lattice. We do not need to perform run-time checking upon assignment because the elements cannot be changed.

Record types are immutable in most languages that support them. In our system, they are certainly immutable. According to the discussion in section 2.1.3.1, 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 correspond-

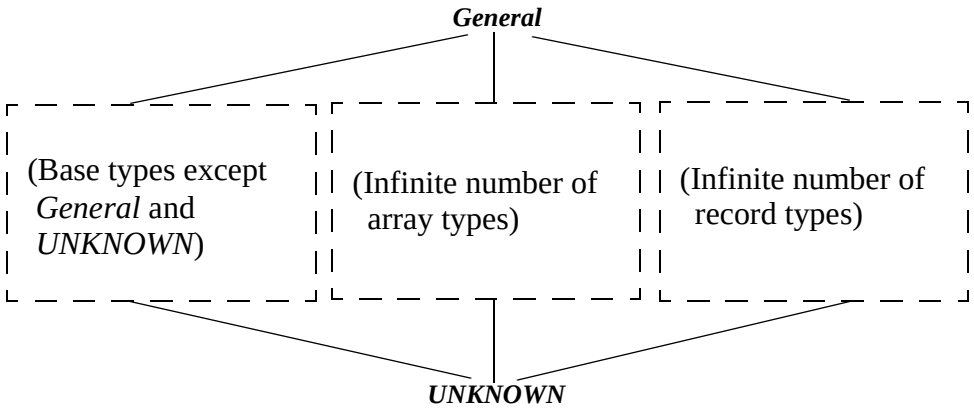


Figure 3.4 The type lattice of Ptolemy II with array and record types added.

ing elements in the super record type. For example, {*name=String, value=Int*} ≤ {*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*} ≤ {*name=String, value=Double*}.

Different kinds of structured types are mutually incomparable. For example, any array type is incomparable with any record type. Figure 3.4 shows the organization of the type lattice of Ptolemy II after adding array and record types. All the structured types are less than the type *General* and greater than *UNKNOWN*, but they are not comparable with other base types. As indicated by the type lattice in figure 3.2, the base types in Ptolemy II include matrix types. Matrices and arrays are different. Matrices contain primitive data, such as integers or doubles, while arrays contain tokens that may have arbitrary type. In Ptolemy II, matrix types are comparable with the corresponding element types. For example, *Int* is less than *Int Matrix*, but array types are not. This is largely a design decision on the construction of the type lattice.

3.3.2.1 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 side of the inequality can be a constant, a variable, or a monotonic function, and the right side can be either a constant or a variable. Notice that the right side cannot be a function. This is because that during the update step, 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 side is neither a constant nor a simple variable. It can be viewed as a function that takes α and returns an array type $\{\alpha\}$. Strictly speaking, this inequality cannot be admitted by the algorithm of Rehof and Mogensen since the right side is a function. However, in the case of structured type, we know exactly the definition of the function, so that during the update step of the algorithm, we can attempt to update the arguments of this function such that the value of the function is the least upper bound of the two sides. In the above inequality, if τ is $\{Int\}$ and the current value of α is *UNKNOWN*, then the least upper bound of both sides is $\{Int\} \vee \{UNKNOWN\} = \{Int\}$. By matching the structure of the right side $\{\alpha\}$ with $\{Int\}$, we can update α to *Int*, which makes the value of the right side $\{Int\}$. Conceptually, this matching and updating is a process of unification for the right side of the inequality and the least upper bound of the two sides. For this unification to succeed, the least upper bound must be a substitution instance of the right side variable structured type. If this is not the case, we have a type conflict in the model. The implementation of this process in Ptolemy II will be discussed in chapter 5.

3.3.2.2 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 lat-

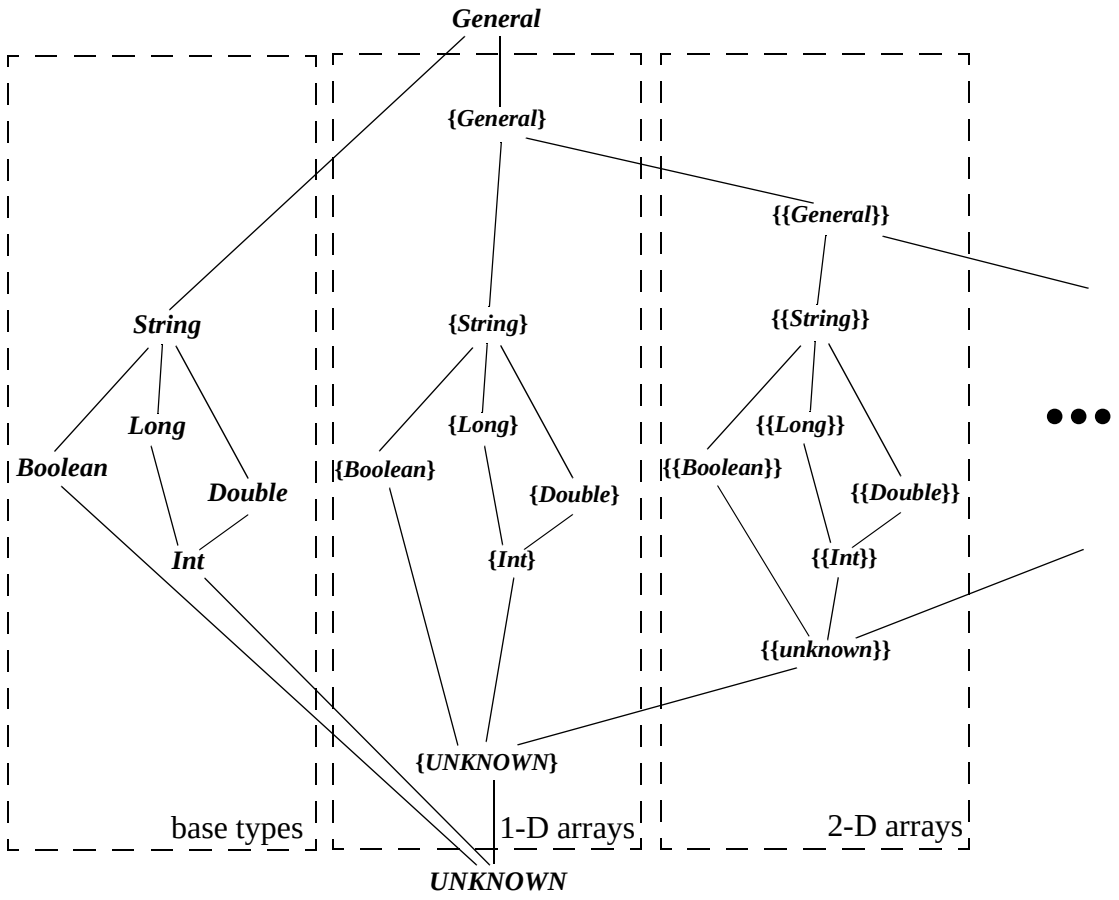


Figure 3.5 An example of a type lattice with arrays.

tice, 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 3.5.

Notice that there is an infinite chain in this lattice:

UNKNOWN, {*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:

$$\{UNKNOWN\} \leq UNKNOWN$$
$$\{\{UNKNOWN\}\} \leq \{UNKNOWN\}$$
$$\{\{\{UNKNOWN\}\}\} \leq \{\{UNKNOWN\}\}$$
$$\ldots$$

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 that does not include *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 without limiting the flexibility of the design environment in practice.

3.3.3 Actors Operating on Structured Types

To simplify the usage of structured types, some actors can be designed to construct and manipulate them. In the Ptolemy II software, the actors that construct arrays and records are called `SequenceToArray` and `RecordAssembler`. `SequenceToArray` bundles a certain number of input tokens into an array token. *RecordAssembler* assembles the tokens from multiple input ports into a record token. The actors `ArrayToSequence` and `RecordDisassembler` perform the reverse operation. Figure 3.6 shows a model that uses the above actors to construct a record whose elements are arrays, and disassembles the elements.

It is interesting to note that the above actors can be viewed as a typed version of some of the canonical SDF actors described by Reekie [110]. He showed that any SDF actor can be implemented as a network containing delays and instances of just five canonical actors. Four of these five actors are `group`, `concat`, `zip`, and `unzip`. The actors discussed in this section, `SequenceToArray`, `ArrayToSequence`, `RecordAssembler`, and

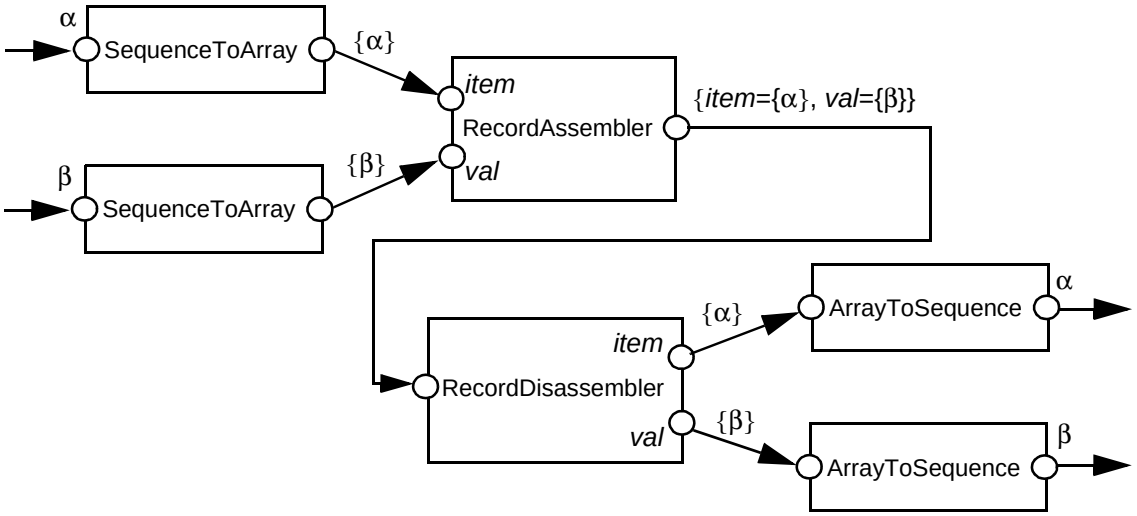


Figure 3.6 Using actors to construct and disassemble structured data.

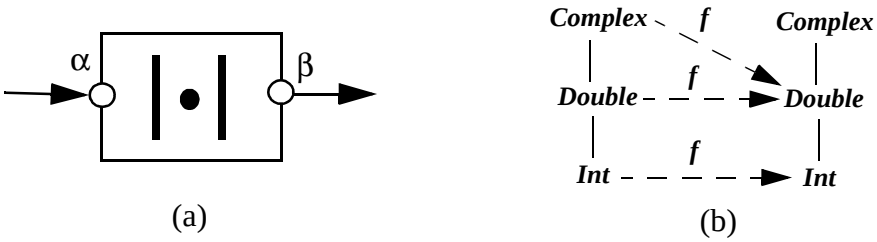
`RecordDisassembler`, can be viewed as the typed version of the four canonical SDF actors, respectively.

3.4 Using Monotonic Functions in Constraints

So far, all of the type constraints we have seen are simple inequalities that involve only the constant types, type variables, and variable structured types. These kinds of inequalities may not be able to express more complicated type constraints. In section 3.3.2.1 above, we mentioned that the type resolution algorithm admits monotonic functions on the left side of the inequality. It turns out that monotonic functions can be used to express complicated type constraints. We show this using the type constraints in three actors. These actors are implemented in Ptolemy II by the Ptolemy research group and some outside contributors.

AbsoluteValue

Suppose we want to implement an `AbsoluteValue` actor shown in figure 3.7(a) that computes the absolute value of the input. We want this actor to be polymorphic and work with several scalar types, including *Int*, *Double*, and *Complex*. If the input type is not *Complex*, the output type should be the same as the input. However, if the input type is *Complex*, the output type is *Double*. This type constraint cannot be expressed by



(a) (b)

Figure 3.7 The `AbsoluteValue` actor and the monotonic function expressing its type constraint.

simple inequalities, but can be expressed with the help of a function. Let the input type be α and the output type be β. We can express the above constraint using

$$f(\alpha) \leq \beta \quad \text{where} \quad f(\alpha) = \begin{cases} Double, & \text{if } \alpha = Complex \\ \alpha, & \text{otherwise} \end{cases}$$

The function $f(\alpha)$ if monotonic. Figure 3.7(b) shows the input and output of this function for a portion of the type lattice.

RecordUpdater

A `RecordUpdater` has a `record` input port that receives record tokens, and a number of `update` input ports. Upon each firing, this actor updates the fields of the record received from the `record` input using the tokens received from the `update` inputs. Since tokens are immutable, this actor does not actually modify the received token, but creates a new token with the correct fields. Figure 3.8 shows such an actor with an example input and output. This particular actor has two update inputs, named

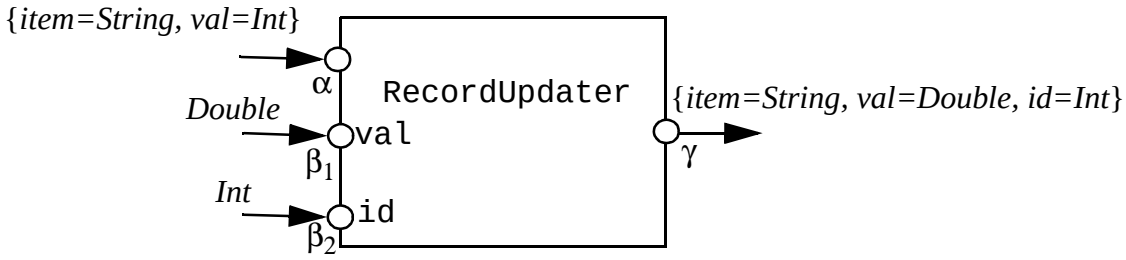


Figure 3.8 A `RecordUpdater` actor with an example input and output.

val and *id*. These names are used as labels in the output record. In figure 3.8, the input record has a field with label *val*, so the input token from the `val` port, which has type *Double*, replaces the one in the original record, which has type *Int*. The original record does not have a field with label *id*, so a new field is added in the output record.

In general, the type constraint for the `RecordUpdater` actor can be expressed as

$$f(\alpha, \beta_1, \beta_2, \ldots, \beta_n) \leq \gamma$$

Where α is the type of the `record` input port, and β_1, β_2, ..., β_n are the types of the `update` inputs, and `n` is the number of update inputs. This function can be computed as follows:

- If α=*UNKNOWN*, return *UNKNOWN*;
- If α={l_1=t_1, l_2=t_2, ..., l_m=t_m}
  Let $\gamma = \alpha$
  For each β_i (`i` = `1`, `2`, `...`, `n`)
  Let l = name of the port for β_i
  If ∃`j` such that $l = l_j$ (`j` = `1`, `2`, `...`, `n`)
  set $t_j = \beta_i$ in γ
  Else add a new field $l = \beta_i$ in γ
  return γ;
- If $\alpha \neq$ *UNKNOWN* and α is not a record type, report type error;

In the first line above, the function value is *UNKNOWN* when α is *UNKNOWN*. This helps make the function monotonic. Some example function arguments and results for the actor in figure 3.8 are shown in table 3.1.

α	β_1	β_2	$f(\alpha, \beta_1, \beta_2)$
UNKNOWN	*UNKNOWN*	*UNKNOWN*	*UNKNOWN*
UNKNOWN	*Double*	*Int*	*UNKNOWN*
{*item=String, val=Int*}	*Double*	*Int*	{*item=String, Val=Double, id=Int*}

Table 3.1. Some example arguments and results for the monotonic function that expresses the type constraint in `RecordUpdater`.

Scale

The `Scale` actor has an input port, an output port, and a parameter called `factor`, as shown in figure 3.9. The operation of this actor is to multiply the value of the input token by the `factor` parameter and send the result to the output. The `factor` parameter has one of the scalar types, such as *Int*, *Double*, *Complex*, *Long*. The input type can be either a scalar or an array. If the input type is a scalar, the output type will be the higher type of the input and the `factor` parameter. If the input type is array, the scaling operation is performed on the elements of the array so the output type will also be an array whose element type is the higher type of the input array element and the `factor` parameter. To maximize the usage of this actor, we want to allow the input array to have arbitrary dimension. For example, it can be an array of *Int*, or an array of arrays of *Double*. This latter requirement can be supported by a recursive monotonic function. That is, we can express the type constraint as

$$f(\alpha, \beta) \leq \gamma$$

and the function $f(\alpha, \beta)$ can be computed as follows:

- If α=*UNKNOWN*
  return *UNKNOWN*;
- If α=a scalar type
  return the higher type of α and β;
- If α=an array type
  Let ε=element type of α
  Let η=f(ε, β)
  return a new array type with element type η;

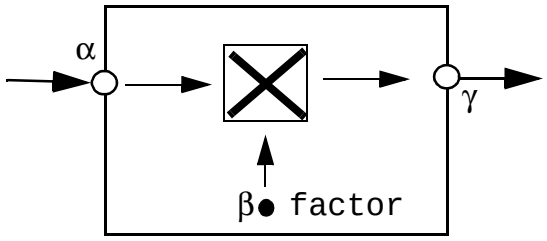


Figure 3.9 The `Scale` actor.

Some of the example arguments and results for this function are shown in table 3.2.

α	β	$f(\alpha, \beta_1, \beta_2)$
UNKNOWN	*UNKNOWN*	*UNKNOWN*
{*UNKNOWN*}	*Double*	{*UNKNOWN*}
{*Int*}	*Double*	{*Double*}
{{*Double*}}	*Double*	{{*Double*}}

Table 3.2. Some example arguments and results for the monotonic function that expresses the type constraint in `Scale`.

3.5 Discussion

In this section, we discuss some of the design decisions in our type system and compare them with some of the related work.

3.5.1 Type System for Block Diagram Based Languages

In the area of system-level design, some block diagram based environments have implemented type systems with different approaches from ours. In the Ptolemy Classic software [16], most components (called stars) are monomorphic. But it supports a rudimentary form of polymorphism by allowing the component to declare that their ports can work with ANYTYPE. For example, the `fork` star in Ptolemy Classic, which copies an input value to multiple outputs, declares the types of all of its ports to be ANYTYPE. This is analogous to parametric polymorphism. This approach is simple and straightforward, but more sophisticated type constraints cannot be expressed, and type checking is ad hoc.

The type system of Simulink [79] is similar to that of the Ptolemy Classic. Simulink has *virtual blocks*, which are polymorphic components that can work with any type. Examples of virtual blocks include multiplexer and demultiplexer. Type conversion is supported through a *Data Type Conversion block*. This component converts an input signal to the data type specified by a parameter. Type checking is guided by a set of ad hoc rules. For example, the output type of a block is generally the same as the input type, except the constant blocks and the data type conversion block. Simulink does not have built-in structured types, but it allows the user to define new types.

Another design tool, the CoCentric system studio from Synopsys [118] uses a template approach, where a component has type variable that the user must set. By setting the type variables to different types, the component can be configured to work on different types. This approach is similar to C++ templates, and it allows a component to have multiple types. This template approach achieves a similar effect as parametric polymorphism, but different code is executed for different types. One disadvantage of this approach is that the user has the burden to set the type, and type constraints cannot be propagated across a topology. In some components, a single type variable is used to control the types of several objects, such as multiple ports, then all of those port must have the same type, which makes components less reusable.

Compared with the above systems, our system and its implementation in Ptolemy II support more kinds of polymorphism. In section 2.1.2.2, we discussed four kinds of polymorphism: parametric, inclusion, overloading and coercion. In Ptolemy II, many flow control actors, such as the `Distributor`, which splits a single token stream into a set of streams, show parametric polymorphism because they work 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 performed 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 [124]. In Haskell, type classes are used to provide overloaded operations [50]. 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 types. 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++. The detailed implementation of our type system in Ptolemy II will be discussed in chapter 5.

In many strongly typed text based languages, such as ML, type checking is done almost entirely at compile time. Once static type checking passes, type information can be discarded at run time. For block diagram based languages, complete static checking is often not possible. In component-based design, we assume that the components are opaque in that the detailed operation of the components are encapsulated, and only the type declaration and constraints at the interface are exposed. As a result, the type systems for such environments can only check type consistency at the interface of the components, but not inside the components. Therefore, static checking alone usually cannot ensure type safety. At run-time, some components may send out tokens with a wrong type. To enforce that the actors obey the interface types, we combine static typing with run-time type checking. This is one of the differences between our system and the ML type system. Our combined approach can detect errors at the earliest possible time and minimize the computation of run-time checking.

3.5.2 Type Lattice and Type Constraints

In our type system, we organize all the types into a lattice using the subtyping relations. 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 [52] 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.

In some other type systems, the subtyping relation does not form a hierarchy. One representative example is the system of Fuh and Mishra, which extends polymorphic type inference in ML with subtypes [45]. Their system allows arbitrary type conversion, represented by a coercion set. This approach makes the system more expressive, since type con-

version is not limited to those that do not lose information. However, because of the lack of structure among the types, the algorithm for type matching and checking the consistency of a coercion set becomes more costly than the inequality solving algorithm we use in our system.

In [112], 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 formulation to achieve more flexibility in type specification. For example, one may be tempted to introduce an *OR* relation among the constraints. This can be useful, for example 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 [112], 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. This implies heavier reliance on run-time checking. However, we have found that our system is generally sufficient to express most of the type constraints, particularly when augmented with monotonic functions.

The inequality type constraints can be generalized to set constraints [6]. Set constraints have been used widely in program analysis. They are more expressive than the inequality constraints based on a lattice. For example, for the type lattice in figure 3.2, the *OR* relation that a type α is either less than or equal to *Boolean*, *OR* less than or equal to *Double*, can be expressed using a set constraint $X \subseteq \{Boolean, Double, Int, UNKNOWN\}$, where X is the set of types that can be assigned to α. The main disadvantage of set constraints is that the resolution algorithm is generally expensive. For the basic set constraint problem, which allows set expressions to involve set constants, set variables, set

union, intersection, negation, and set constructors, deciding satisfiability has exponential complexity [6]. In light of this, the formulation of our type system represent a good trade-off between expressiveness and computation cost.

3.5.3 Most Specific Type

The type resolution algorithm we described in section 3.2.3 computes the most specific types (least in the lattice) that satisfy the type constraints. This least solution may not be the only solution. In the example in figure 3.3, another solution is to resolve all the ports on A2 to *Double*. In fact, the inequality solving algorithm can be used to find either the least fixed point or the greatest fixed point. We choose to use the least solution for a practical reason. In general, types lower in the type lattice 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 to 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.

In section 2.1.3.2, we discussed that the type resolution algorithm in ML computes the principal types. These principal types can be viewed as the most general solution for the type constraints in that all the other solutions are substitution instances of the principal types. In ML, functions can be separately compiled, so resolving the types of polymorphic functions to the most general types allows maximal reuse of compiled code. In our case, we perform type resolution on a complete model, and 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 to achieve reuse. Notice that on multiple occurrences of the same polymorphic actor, different type variables are used to denote the types of their ports and parameters. These variables are analogous to the generic type variables in ML.

It is interesting to note that some other researchers have also attempted to infer more precise types for text based programs. For example, Plevyak and Chien observed that principal types are inadequate for program optimization, and proposed an algorithm to infer concrete types in programs [107]. They use different type variables for different runs of the same program, and the inference is based on set constraints. To guide the type inference effort to where it is fruitful, they propose to do coarse level inference first, then zoom in on particular areas to find more precise types. Their algorithm works well for many programs, but there are still some program structures which will require run time type checks.

3.5.4 Type Resolution in Modal Models

Besides the models based on message passing, our type system can also be used in *modal models* that mix finite-state machines (FSMs) with other concurrency models [48]. In these mixed models, type constraints can be propagated between the events of the control model and the data of the other concurrency models. For example, Girault, Lee and Lee showed how to mix FSM with synchronous dataflow (SDF) in [48]. Figure 3.10 is such an example. 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

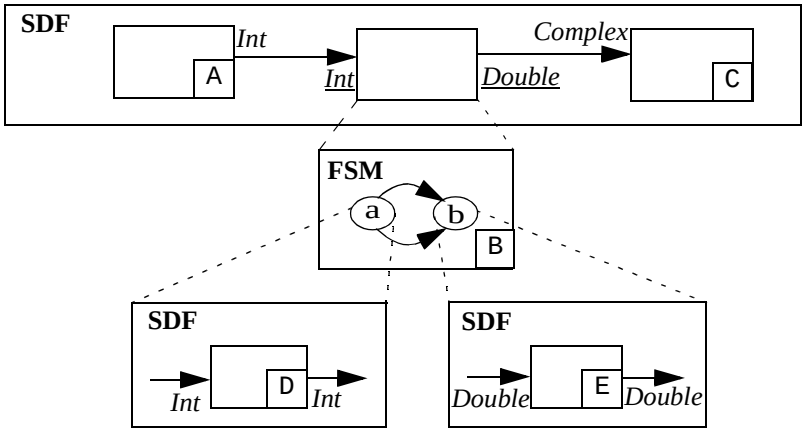


Figure 3.10 Mixing FSM with SDF.

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 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 discussed in section 2.1.3.1. If we consider the actors as functions, then the types of the actors are 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 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.

4 Behavioral Types

4.1 Capturing the Dynamic Behavior of Components

In the last chapter, we presented a type system for component-based design. Fundamentally, a type system detects mismatches at component interfaces and ensures component compatibility. As discussed in section 2.2.3, interface mismatch can happen at (at least) two different levels: data exchange and the dynamic interaction. The system presented in the last chapter is a data-level type system. In this chapter, we present a system that captures the dynamic behavior of components. We call the result *behavioral types*.

At the interface of components, the communication protocol and execution control are two of the most important aspects of the dynamic behavior. They essentially determine the model of computation the components use to interact with each other. Our approach is to describe the communication protocol types and the component behavior using interface automata, and perform compatibility checking through automata composition. From a type system point of view, this compatibility check amounts to type checking.

Traditionally, automata models are used to perform model checking at design time. Here, our emphasis is not on model checking to verify arbitrary user code, but rather on compatibility of the composition of pre-defined types. As such, the scalability of the methods is much less an issue, since the size of the automata in question is fixed. We also propose to extend the use of automata to on-line reflection of component state, and to do run-time type checking.

As has been discussed several times earlier, polymorphism is a very desirable feature in the design of type systems. At the data level, research has been 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 [100], and modern polymorphic type systems have achieved this goal to a large extent. At the behavioral-level, type systems should also be polymorphic to support component reuse while ensuring component compatibility.

In our data-level type system, we organize all the types into a hierarchy using the subtyping relation. We form a polymorphic type system at the behavioral level through an approach similar to subtyping. Using the alternating simulation relation of interface automata, we organize all the interaction types in a partial order. Given this hierarchy, if a component is compatible with a certain type *A*, it is also compatible with all the subtypes of *A*. This property can be used to facilitate the design of polymorphic components and simplify type checking.

Even with the power of polymorphism, no type system can capture all the properties of programs and allow type checking to be performed efficiently while keeping the language flexible. So the language designer always has to decide what properties to include in the system and what to leave out. Furthermore, some properties that can be captured by types cannot be easily checked statically before the program runs. This is either because the information available at compile time is not sufficient, or because checking those properties is too costly. Hence, the designer also needs to decide whether to check those properties statically or at run time. Any type system represents some compromise.

Type systems at the behavioral level have similar trade-offs. Among all the properties in a component-based design environment, we choose to check the compatibility of communication protocols as the starting point. This is because communication protocols are the central piece in many models of computation [62] and determine many other properties in the models. Our type system is extensible, so other properties, such as deadlock in concurrent models, can be included in type checking. Another reason we choose to check the compatibility of communication protocols is that it can be done efficiently, when a component is inserted in a model. More complicated checking may need to be postponed to run time.

Our system is based on the Ptolemy II environment [33]. Ptolemy II supports multiple models of computation so it is ideal for studying the dynamic interaction of components. Some researchers have proposed extended type systems for other languages, including π-calculus [86] and the actors model [3]. Some of these systems also capture the dynamic behavior of components and are closely related to ours. We will discuss related work in section 4.4.

In addition to the benefit of improved safety through type checking, our type system also has an intangible benefit. The process of describing communication protocols and component behavior formally can help designers gain a deeper understanding of the system being specified. Through this specification process, developers may uncover design flaws, inconsistencies, ambiguities, and incompleteness.

The rest of this chapter is organized as follows. Section 4.2 describes component interaction in Ptolemy II. Section 4.2 presents our behavioral type system, including the type definition, the type hierarchy and some type checking examples. Section 4.4 discusses the trade-offs in the design of behavioral types and compares our approach with related work.

4.2 Component Interaction in Ptolemy II

Ptolemy II [33] is a system-level design environment that supports component-based heterogeneous modeling and design. The focus is on embedded systems. In Ptolemy II, components are called *actors*, and the channel of communication between actors is implemented by an object called a *receiver*, as shown in figure 4.1. Receivers are contained in *IOPorts* (input/output ports), which are in turn contained in actors. Ptolemy II

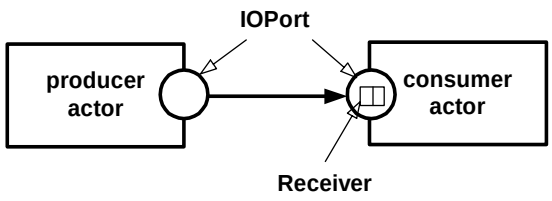


Figure 4.1 A simple model in Ptolemy II.

is implemented in Java. The methods in the receiver are defined in a Java interface `Receiver`. This interface assumes a producer/consumer model, and communicated data is encapsulated in a class called `Token`. The `put()` method is used by the producer to deposit a token into a receiver. The `get()` method is used by the consumer to extract a token from the receiver. The `hasToken()` method, which returns a boolean, indicates whether a call to `get()` will trigger a `NoTokenException`.

Aside from assuming a producer/consumer model, the `Receiver` interface makes no further assumptions. It does not, for example, determine whether communication between actors is synchronous or asynchronous. Nor does it determine the capacity of a receiver. These properties of a receiver are determined by concrete classes that implement the `Receiver` interface. Each one of these concrete classes is part of a Ptolemy II *domain*, which is a collection of classes implementing a particular model of computation. In each domain, the receiver determines the communication protocol, and an object called a *director* controls the execution of actors. From the point of view of an actor, the director and the receiver form its execution environment.

Each actor has a `fire()` method that the director uses to start the execution of the actor. During the execution, an actor may interact with the receivers to receive or send data. The following are some of the domains in Ptolemy II. The models of computation implemented by these domains are discussed in section 2.3.

- *Communicating Sequential Processes (CSP)*: As the name suggests, this domain implements a rendezvous-style communication (sometimes called synchronous message passing), as in Hoare's communicating sequential processes model [54]. In this domain, the producer and consumer are separate threads executing the `fire()` method of the actors. Whichever thread calls `put()` or `get()` first blocks until the other thread calls `get()` or `put()`. Data is exchanged in an atomic action when both the producer and consumer are ready.
- *Process Networks (PN)*: This domain implements the Kahn process networks model of computation [58]. The Ptolemy II implementation is similar to that by Kahn and MacQueen [59]. In that model, just like CSP, the producer and consumer are separate threads executing the `fire()` method. Unlike CSP, however, the producer can send data and proceed without waiting for the receiver to be ready to receive data. This is implemented by a non-blocking write to a FIFO queue with (conceptually) unbounded capacity. The `put()` method in a PN receiver always succeeds and always returns immediately. The `get()` method, however, blocks the calling thread if no data is

available. To maintain determinacy, it is important that processes not be able to test a receiver for the presence of data. So the `hasToken()` method always returns *true*. Indeed, this return value is correct, since the `get()` method will never throw a `NoTokenException`. Instead, it will block the calling thread until a token is available.

- *Synchronous Data Flow (SDF)*: This domain supports a synchronous dataflow model of computation [65]. This is different from the thread-based domains in that the producer and consumer are implemented as finite computations (firings of a dataflow actor) that are scheduled (typically statically, and typically in the same thread). In this model, a consumer assumes that data is always available when it calls `get()` because it assumes that it would not have been scheduled otherwise. The capacity of the receiver can be made finite, statically determined, but the scheduler ensures that when `put()` is called, there is room for a token. Thus, if scheduling is done correctly, both `get()` and `put()` succeed immediately and return.
- *Discrete Event (DE)*: This domain uses timed events to communicate between actors. Similar to SDF, actors in the DE domain implement finite computations encapsulated in the `fire()` method. However, the execution order among the actors is not statically scheduled, but determined at run time. Also, when a consumer is fired, it cannot assume that data is available. Very often, when an actor with multiple input ports is fired, only one of the ports has data. Therefore, for an actor to work correctly in this domain, it must check the availability of a token using the `hasToken()` method before attempting to get a token from the receiver.

As can be seen, different domains impose different requirements for actors. Some actors, however, can work in multiple domains. These actors are called *domain-polymorphic* actors. One of the goals of the behavioral type system is to facilitate the design of domain-polymorphic actors.

4.3 Behavioral Types

4.3.1 Type Definition

We use interface automata to describe the behavior of actors and the interaction type. Figure 4.2 shows the interface automata model for an implementation of the consumer actor in figure 4.1. This figure is a screen shot of Ptolemy II. The block arrows on the two sides denote the inputs and outputs of the automata. They are:

- *f*: the invocation of the `fire()` method of the actor.
- *fR*: the return from the `fire()` method.
- *g*: the invocation of the `get()` method of the receiver at the input port of the actor.
- *t*: the token returned in the `get()` call.

SDFActor

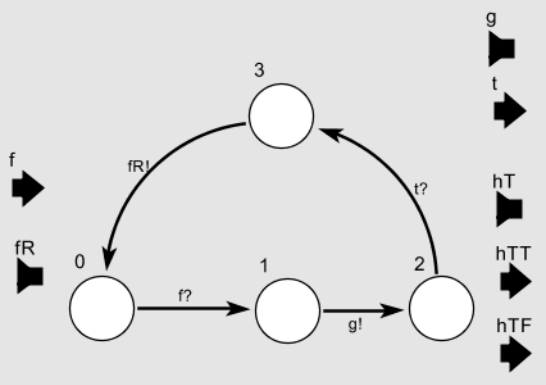


Figure 4.2 Interface automaton model for an SDF actor.

- *hT*: the invocation of the `hasToken()` method of the receiver.
- *hTT*: the value *true* returned from the `hasToken()` call, meaning that the receiver contains one or more tokens.
- *hTF*: the value *false* returned from the `hasToken()` call, meaning that the receiver does not contain any token.

The initial state is state 0. When the actor is in this state, and the `fire()` method is called, it calls `get()` on the receiver to obtain a token. After receiving the token in state 3, it performs some computation, and returns from `fire()`. Following the optimistic approach of interface automata, this model only encodes the behavior of the actor under a good environment, namely, the SDF domain. In this domain, there is only one thread of execution, so the actor assumes that its `fire()` method will not be called again if it is already inside this method. Therefore, the input *f* is only accepted in state 0, but not in any other states. Also, the scheduler guarantees that data is available when a consumer is fired, so the transition from state 2 to state 3 assumes that the receiver will return a token. An error condition, such as the receiver throws `NoTokenException` when `get()` is called, is not explicitly described in the model.

Figure 4.3 describes another actor that can operate in a wider variety of domains. Since this actor is not designed under the assumption of the SDF domain, it does not assume that data are available when it is fired. Instead, it calls `hasToken()` on the receiver to check the availability of a token. If `hasToken()` returns false, it immediately returns from `fire()`. This is a simple form of domain-polymorphism.

PolyActor

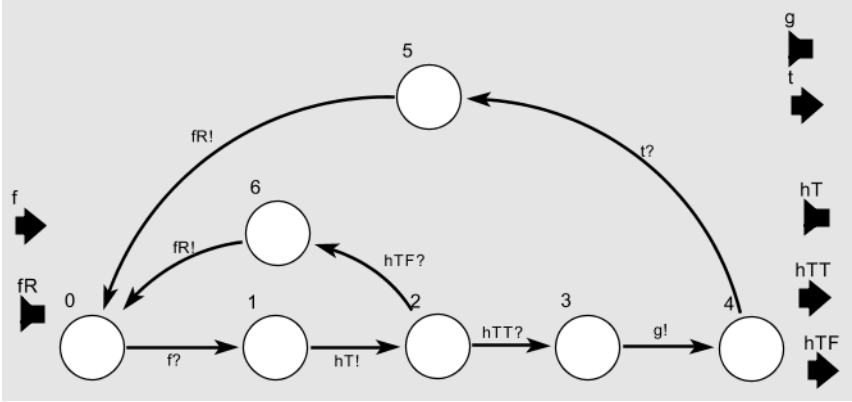


Figure 4.3 Interface automaton model for a domain-polymorphic actor.

In Ptolemy II, actors interact with the director and the receivers of a domain. In figures 4.2 and 4.3, the block arrows on the left side denote the interface with the director, and the ones on the right side denote the interface with the receiver. As discussed in section 4.2, the implementation of the director and the receiver determines the semantics of component interaction in a domain, including the flow of control and the communication protocol. If we use an interface automaton to model the combined behavior of the director and the receiver, this automaton is then the type signature for the domain. Figure 4.4 shows such an automaton for the SDF domain. Here, *p* and *pR* represent the

SDFDomain

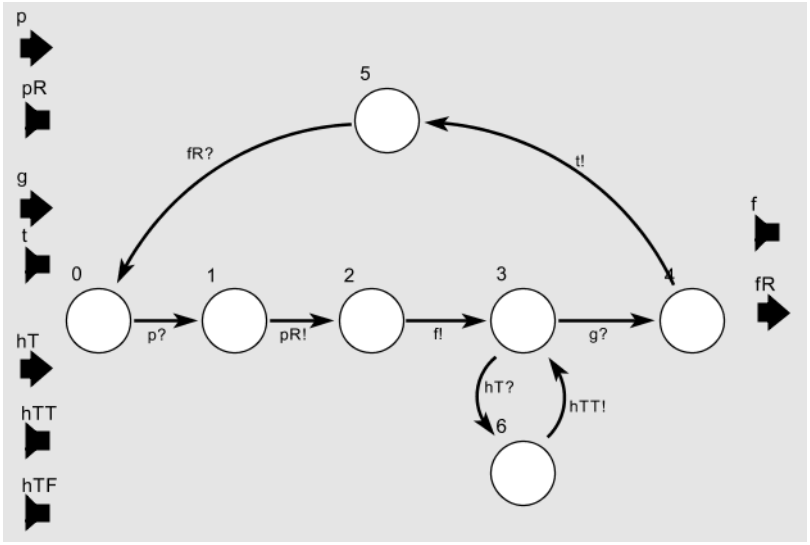


Figure 4.4 Type signature of the SDF domain.

call and the return of the `put()` method of the receiver. This automaton encodes the assumption of the SDF domain that the consumer actor is fired only after a token is put into the receiver[1].

The type signature of the DE domain is shown in figure 4.5. In DE, an actor may be fired without a token being put into the receiver at its input. This is indicated by the transition from state 0 to state 7. Figures 4.4 and 4.5 also reflect the fact that both of the SDF and the DE domains have a single thread of execution, so the `hasToken()` query may happen only after the actor is fired, but before it calls `get()`, during which time the actor has the thread of control.

CSP and PN are two domains in Ptolemy II in which each actor runs in its own thread. Figures 4.6 and 4.7 give the type signature of these two domains. These automata are simplified from the true implementation in Ptolemy II. In particular, *CSPDomain* omits conditional rendezvous, which is an important feature in the CSP model of computa-

DEDomain

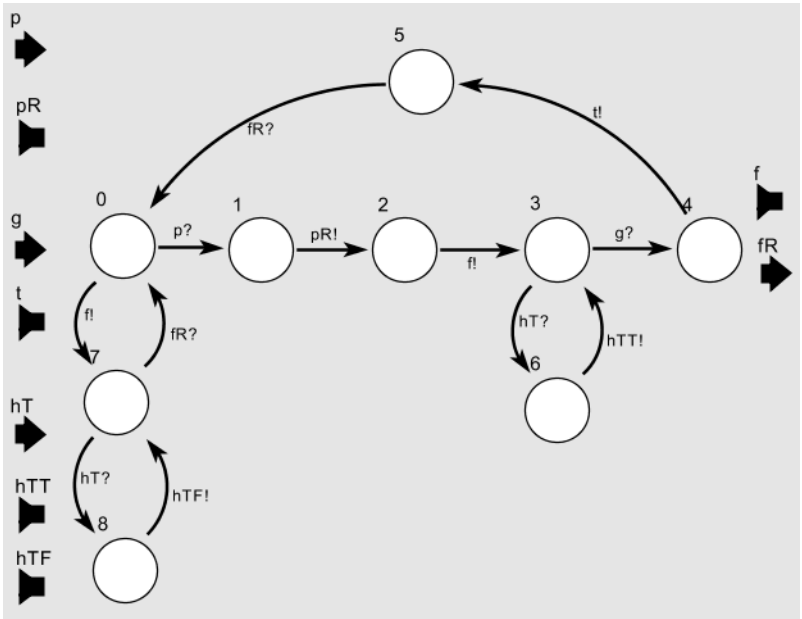


Figure 4.5 Type signature of the DE domain.

1. This is a simplification of the SDF domain, since an actor may require more than one token to be put in the receiver before it is fired. This simplification makes our exposition clearer. Modeling this fully would require dependent types.

CSPDomain

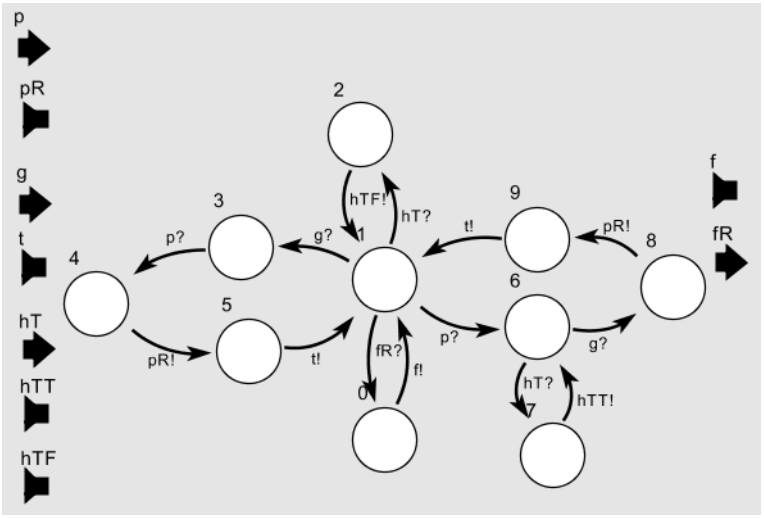


Figure 4.6 Type signature of the CSP domain.

PNDomain

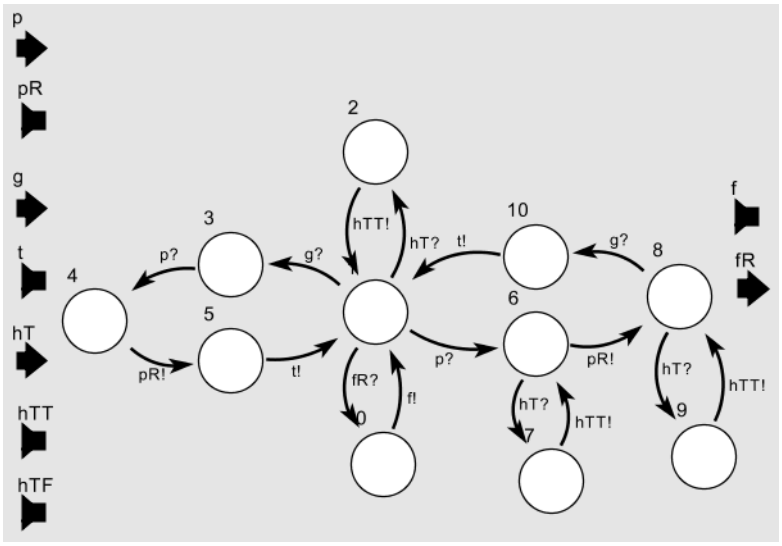


Figure 4.7 Type signature of the PN domain.

tion. In the CSP and PN domains, an actor is fired repeatedly by its thread, as modeled by the transitions between state 0 and 1.

In CSP, the communication is synchronous; the first thread that calls `get()` or `put()` on the receiver will be stalled until the other thread calls `put()` or `get()`. The case where `get()` is called before `put()` is modeled by the transitions among the states

1, 3, 4, 5, 1. The case where `put()` is called before `get()` is modeled by the transitions among the states 1, 6, 8, 9, 1.

In PN, the communication is asynchronous. So the `put()` call always returns immediately, but the thread calling `get()` may be stalled until `put()` is called. The case where `get()` is called first in PN is modeled by the transitions among the states 1, 3, 4, 5, 1 in figure 4.7, while the case where `put()` is called first is modeled by the transitions among the states 1, 6, 8, 10, 1.

Given an automaton modeling an actor and the type signature of a domain, we can check the compatibility of the actor with the communication protocol of that domain by composing these two automata. Type checking examples will be shown below in section 4.3.3.

4.3.2 Behavioral Type Order and Polymorphism

If we compare the SDF and DE domain automata, we can see that they are closely related. This relationship can be captured by the alternating simulation relation of interface automata. In particular, there is an alternating simulation relation from SDF to DE.

In the set of all domain types, the alternating simulation relation induces a partial order, or *behavioral type order*. An example of this partial order is shown in figure 4.8. From a

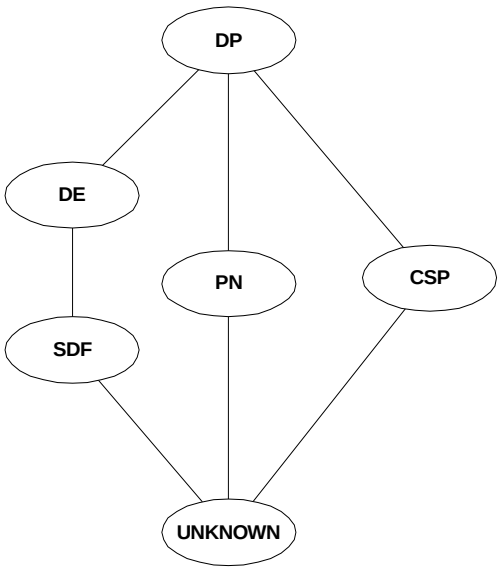


Figure 4.8 An example of behavioral type order.

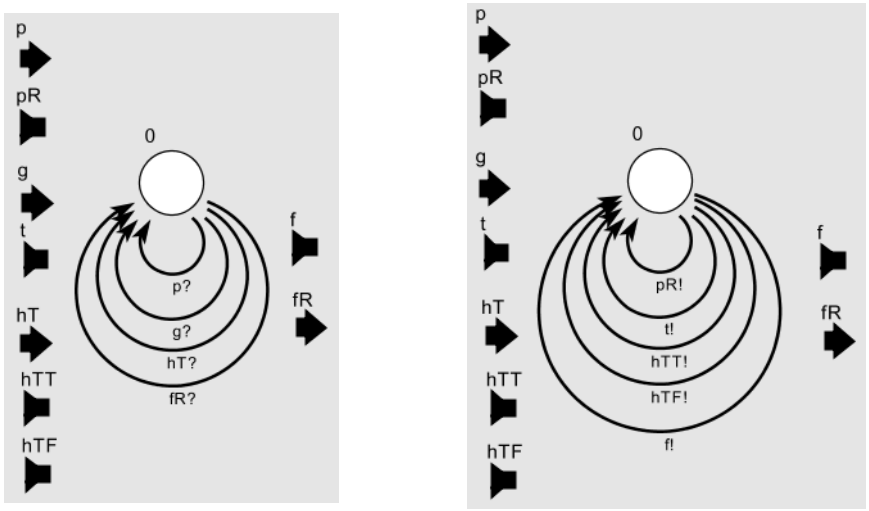
type system point of view, this order is the subtyping hierarchy for the domain types. If we view the automata as functions with inputs and outputs, then the alternating simulation relation is exactly analogous to the standard function subtyping relation in data type systems. According to the definition of alternating simulation, the automaton lower in the hierarchy can simulate all the input steps of the ones above it, and the automaton higher in the hierarchy can simulate all the output steps below it. In function subtyping, if a function $A \rightarrow B$ is a subtype of another function $A' \rightarrow B'$, then A' is a subtype of A and B is a subtype of B'. Note that in both relations, the order is inverted (contravariant) for the inputs and goes in the same direction (covariant) for the outputs.

In [34], alternating simulation is used to capture the refinement relation from the specification to the implementation of components. Our use of this relation is not directly for component design, but for capturing the relation between interaction types. In the behavioral type order, *SDFDomain* is not a refinement of *DEDomain*, but a subtype of *DEDomain*. In fact, *SDFDomain* has fewer states than *DEDomain*. This subtyping relation can help us design actors that can work in multiple domains. According to the theorem discussed in section 2.4.2, if an actor is compatible with a certain domain *D*, and there are other domains below *D* in the behavioral type order, then the actor is also compatible with those lower domains. Therefore, this actor is domain polymorphic.

In figure 4.8, we added a bottom element *UNKNOWN* and a top element *DP*. *DP* stands for "domain polymorphic". There is an alternating simulation relation from *UNKNOWN* to any other element, and from all the elements to *DP*. One possible design of these two automata is shown in figure 4.9. In this figure, both automata have a single state. The *UNKNOWN* automaton has all the input transitions, and the *DP* automaton has all the output transitions. We will discuss these two automata further in section 4.4.1.

4.3.3 Type Checking Examples

Let's perform a few type checking operations using the actors and domains in the earlier sections. To verify that the *SDFActor* in figure 4.2 can indeed work in the SDF domain, we compose it with the *SDFDomain* automaton in figure 4.4. The result is shown in



(a) bottom (*UNKNOWN*) (b) top (*DP*)

Figure 4.9 The bottom and top elements of the behavioral type order.

figure 4.10. As expected, the composition is not empty so *SDFActor* is compatible with *SDFDomain*. This composition is a new type definition for the composed components.

Due to the optimistic approach of interface automata, the above composition is much smaller than the product automaton. Before adopting interface automata, we also attempted to describe behavioral types using a more traditional finite state machine model [69]. Compatibility checking in that setting proved to be more difficult.

Now let's compose *DEDomain* with *SDFActor*. The result is an empty automaton shown in figure 4.11. This is because the actor may call `get()` when there is no token in the receiver, and this call is not accepted by an empty DE receiver. The exact sequence that leads to this condition is the following: first, both automata take a shared transition *f*.

SDFDomain_SDFActor

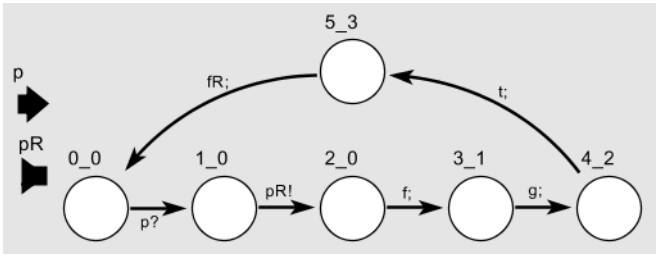


Figure 4.10 Composition of *SDFDomain* in figure 4.4 and *SDFActor* in figure 4.2.

DEDomain_SDFActor

Figure 4.11 Composition of *DEDomain* in figure 4.5 and *SDFActor* in figure 4.2.

In this transition, *DEDomain* moves from state 0 to state 7, and *SDFActor* moves from state 0 to state 1. At state 1, *SDFActor* issues *g*, but this input is not accepted by *DEDomain* at state 7. So the pair of states (7, 1) in (*DEDomain*, *SDFActor*) is illegal. Since this state can be reached from the initial state (0, 0), the initial state is pruned out from the composition. As a result, the whole composition is empty. This means that the SDF actor cannot be used in DE Domain.

The *PolyActor* in figure 4.3 checks the availability of a token before attempting to read from the receiver. By composing it with *DEDomain*, we verify that this actor can be used in the DE Domain. This composition is shown in figure 4.12. Since *SDFDomain* is below *DEDomain* in the behavioral type order of figure 4.8, we have also verified that *PolyActor* can work in the SDF domain. Therefore, *PolyActor* is domain polymor-

DEDomain_PolyActor

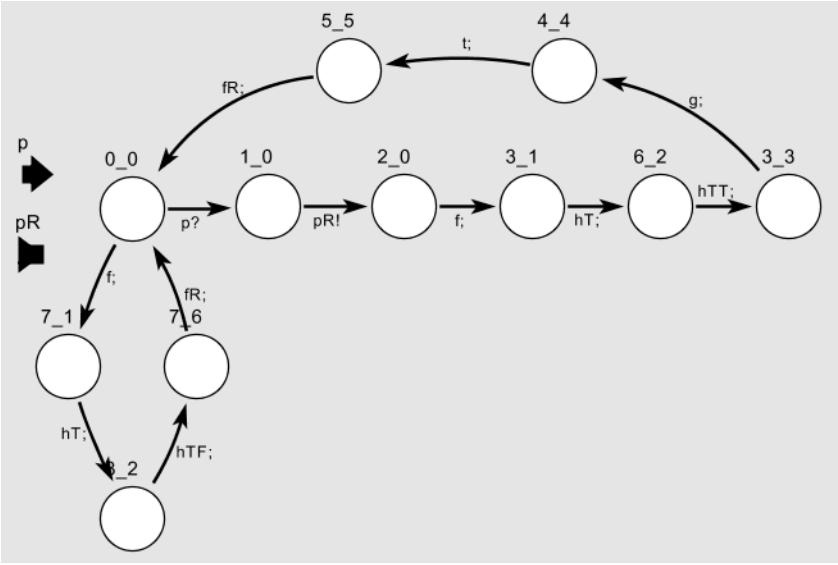


Figure 4.12 Composition of *DEDomain* in figure 4.5 and *PolyActor* in figure 4.3.

SDFDomain_PolyActor

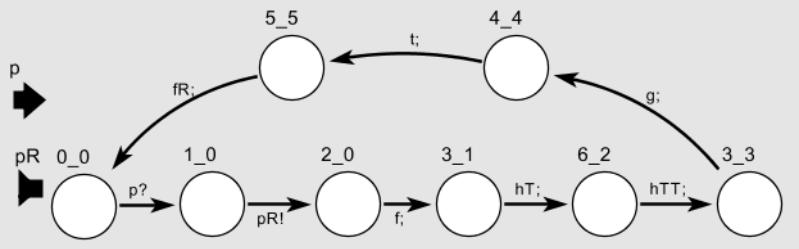


Figure 4.13 Composition of *SDFDomain* in figure 4.4 and *PolyActor* in figure 4.3.

phic. As a sanity check, we have composed *SDFDomain* with *PolyActor*, with the result shown in figure 4.13.

In Ptolemy II, there is a library of about 100 domain-polymorphic actors. The communication behavior for many of these actors can be modeled by the *PolyActor* automaton.

4.3.4 Reflection

So far, interface automata have been used to describe the operation of Ptolemy II components. These automata can be used to perform compatibility checks between components. Another interesting use is to reflect the component state in a run-time environment. For example, we can execute the automaton *SDFActor* of figure 4.2 in parallel with the execution of the actor. When the `fire()` method of the actor is called, the automaton makes a transition from state 0 to state 1. At any time, the state of the actor can be obtained by querying the state of the automaton. Here, the role of the automaton is reflection, as realized for example in Java. In Java, the Class class can be used to obtain the static structure of an object, while our automata reflect the dynamic behavior of a component. We call an automaton used in this role a *reflection automaton*.

4.4 Discussion

4.4.1 Top and Bottom

We have shown one possible design for the top and bottom elements of the behavioral type order in figure 4.9. These two automata are very general in that they are not only the top and bottom elements of the partial order in figure 4.8, but also the top and bottom of the partial orders formed by any set of automata with the same set of input and output

transitions. In another word, there is an alternating simulation relation from any automaton to the *DP* automaton in figure 4.9(b), and an alternating simulation relation from the *UNKNOWN* automaton in figure 4.9(a) to any automaton with the same inputs and outputs.

If we can design an actor that is compatible with the *DP* automaton, then that actor will be maximally polymorphic in that it will be able to work in any domain that may be created. However, it is easy to see that this is almost impossible. Since the *DP* automaton may issue any output at any time, no non-trivial actor can be compatible with it. This means that we cannot hope to design a non-trivial actor that will be able to work in any environment.

On the other hand, the *UNKNOWN* automaton is compatible with any actor automaton. For example, the compositions of *UNKNOWN* with the *SDFActor* or the *PolyActor* are shown in figure 4.14. The two compositions are the same. Intuitively, since the *UNKNOWN* automaton does not have any output transition, it does not call the `fire()` method of the actor, so there is no interaction between the *UNKNOWN* automaton and the actor automaton. The only transition is the input *p* from outside the composition.

The *DP* and *UNKNOWN* automata represent two extremes of the possible environments for actors. The *DP* is the most stringent environment in which no non-trivial actor can work, while *UNKNOWN* is the laxest environment in which an actor is not asked to do anything.

UNKNOWN_SDFActor or *UNKNOWN_PolyActor*

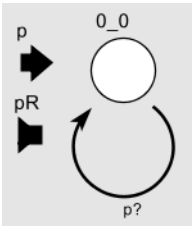


Figure 4.14 Composition of *UNKNOWN* in figure 4.9(a) and the *SDFActor* in figure 4.2 or the *PolyActor* in figure 4.3.

4.4.2 Trade-offs in Type System Design

The type checking examples in section 4.3.3 focus on the communication protocol between a single actor and its environment. This scope can be broadened by including the automata of more actors and using a more detailed director model in the composition. Also, properties other than the communication protocol, such as deadlock freedom in thread-based domains, can be included in the type system. However, these extensions will increase the cost of type checking. So there is a trade-off between the amount of information carried by the type system and the cost of type checking.

Another dimension of the trade-offs is static versus run-time type checking. The examples in the last section are static type checking examples. If we extend the scope of the type system, static checking can quickly become impractical due to the size of the composition. An alternative is to check some of the properties at run time. One way to perform run-time checking is to execute the reflection automata of the components in parallel with the execution of the components. Along the way, we periodically check the states of the reflection automata, and see if something has gone wrong.

These trade-offs imply that there is a big design space for behavioral types. In this space, one extreme point is complete static checking by composing the automata modeling all the system components, and check the composition. This amounts to model checking. To explore the boundary in this direction, we did an experiment by checking an implementation of the classical dining philosophers model implemented in the CSP domain in Ptolemy II. Each philosopher and each chopstick is modeled by an actor running in its own thread. The chopstick actor uses conditional send to simultaneously check which philosopher (the one on its left or the one on its right) wants to pick it up. We created interface automata for the Ptolemy II components *CSPReceiver*, *Philosopher*, and *Chopstick*, and a simplified automaton to model conditional send. We are able to compute the composition of all the components in a two-philosopher version of the dining philosopher model, and obtain a closed automaton with 2992 states. Since this automaton is not empty, we have verified that the components in the composition are compatible with respect to the synchronous communication protocol in CSP. We also checked for deadlock inherent in the implementation by looking for states in the composition that do not have any outgoing transitions,

and are able to identify two deadlock states in the composition. These two states correspond to the situation where all the philosophers are holding the chopsticks on their left and waiting for the ones on their right, and the symmetrical situation where all the philosophers are waiting for the chopsticks on their left.

Our goal here is not to do model checking, but to perform static type checking on a non-trivial models. Obviously, when the model grows, complete static checking will become intractable due to the well-known state explosion problem.

Another extreme point in the design space for behavioral types is to rely on run-time type checking completely. For deadlock detection, we can execute the reflection automata in parallel with the Ptolemy II model. When the model deadlocks, the states of the automata will explain the reason for the deadlock. In this case, the type system becomes a debugging tool. The point here is that a good type system is somewhere between these extremes. We believe that a system that checks the compatibility of communication protocols, as illustrated in sections 4.3.3, is a good starting point.

4.4.3 Behavioral Typing

In the concurrent object-oriented language community, there is a lot of ongoing work on type systems for parallel object languages and calculi. Some of the proposed systems have very similar objectives as ours, namely, capturing the dynamic behavior of components. In particular, the type model of Puntigam [108] and the behavioral type system of Najm and Nimour [93][94] both attempt to capture the communication behavior of components, and both systems have a notion of subtyping that is conceptually similar to the alternating simulation relation.

The type model of Puntigam is designed for a language that is based on a combination of the actor model [3] and a process calculus with trace semantics. Similar to our model, objects communicate by message passing. A message has the form $c(o_1, ..., o_m; v_1, ..., v_n)$. This can be viewed as a method call with method name c, input parameters $o_1, ..., o_m$, and output parameters $v_1, ..., v_n$. A *type trace* is a sequence $p_1 ... p_n$ of message prototypes, and a *type trace set T* is a prefix-closed non-empty set of type traces. Here, the type trace sets

are the type specifications of active objects. It defines the sequences of messages that an object is prepared to handle, and the clients of the objects are allowed to send message only according to exactly one type trace selected from the set. The type trace set of a type τ is denoted trace(τ). Under this formulation, a subtype is defined as:

A type σ is a subtype of a type τ (denoted by $\sigma \leq \tau$) if and only if for each type trace $p_1 \ldots p_n \in$ trace(τ) there is a $p_1' \ldots p_n' \in$ trace(σ) so that (for each $1 \leq i \leq n$; with $p_i = c_i(\phi_{i,1}, \ldots, \phi_{i,k_i}; \varphi_{i,1}, \ldots, \varphi_{i,l_i})$ and $p'_i = c'_i(\phi'_{i,1}, \ldots, \phi'_{i,k'_i}; \varphi'_{i,1}, \ldots, \varphi'_{i,l'_i})$

- $c_i = c'_i$ (equal message identifiers);
- $k'_i \leq k_i$ and $\phi_{i,j} \leq \phi'_{i,j}$ for $1 \leq j \leq k'_i$ (contravariant input parameter types);
- $l_i \leq l'_i$ and $\varphi'_{i,j} \leq \varphi_{i,j}$ for $1 \leq j \leq l_i$ (covariant output parameter types).

As the alternating simulation relation we use for defining subtypes in our system, this definition has contravariant input types and covariant output types.

However, there are several differences between this formulation and ours. First, a trace is a global property in that a trace specifies a complete run of an object, while the simulation relation is local in that it is a relation for each step of the run. Second, the subtyping definition here mixes data typing issues with behavior. In particular, the last two conditions in the definition is essentially the record subtyping rules we use for our data type system. In our system, we separate the data typing issues from behavioral typing and handle them in different ways. Third, the trace set, which defines a language, is more general than an automaton. If the trace set is constrained to be a regular set, then it is equivalent to an automaton.

The behavioral type system presented in [93] is closer to our system. This system is designed for an object calculus which is a variant of the π-calculus [86], with syntactic sugar for method definition. Here, behavioral types specify the set of methods (services) an objects supports. This set is dynamic since the set of supported methods may change after each method call. For example, an object implementing a one place buffer has a `put()` and a `get()` method for writing and reading data into and from the buffer. When

the buffer is empty, the set of supported methods includes only the `put()` method. After `put()` is called, the set includes only the `get()` method, and so on. This dynamic behavior is specified using a labeled transition system, where each transition is a method signature. Similar to our system, the definition of subtyping distinguishes the sending and receiving of messages. If a type X_2 is a subtype of a type X_1, then all the receiving actions of X_1 can be performed by X_2, and all the sending actions of X_2 can be performed by X_1. This is analogous to the alternating simulation relation of interface automata. The formal definition of behavioral types, the transition system, and subtyping can be found in [93][94]. In this system, the requirements for type compatibility are defined by complicated type rules.

In both of the above two systems, the basic goal of typing is to ensure that an object does not receive a method call that is not supported. This error condition is analogous to the error condition that results in illegal states in the composition of interface automata. Compared with them, our interface automata based system permits much easier type checking. Also, since interface automata can be easily described in bubble and arc diagrams, the type representation in our system is easier to understand than the algebraic form used in both approaches. Another difference is that the above two systems concentrate mostly on the communication between objects through message passing, while our system also takes the execution control into consideration. Finally, it is interesting to note the different terminologies used to describe the dynamic behavior of components. Inspired by [93], we call our description behavioral types, while it is called process types in [108].

4.4.4 Component Interfacing

In hardware design, many people have proposed techniques of *protocol synthesis* to connect components with different interfaces [25][26][40][41][102][104]. There are two approaches to protocol synthesis. One is library or template based. For example, Eisenring and Platzner [40] develop a tool that provides a template and a corresponding generator method for each interface type. The other is to generate a converter from the two interfaces to be connected. For example, Passerone *et al.* [104] describe the communication protocols of the two components to be interfaced by two finite state machines, and the converter is

essentially the product machine, with invalid states removed. Compared with this approach of component interfacing, our approach is to design polymorphic components with tolerant interfaces, so that they can be used in different settings. Besides, there are two additional differences between our system and the protocol synthesis techniques.

One difference is that behavioral types cover multiple models of computation, while protocol synthesis usually concentrates on interfacing different implementations of one model of computation. For example, Passerone *et al.* [104] focus on synchronous model (shared clock); Eisenring and Platzner [40] study dataflow models implemented by queues between component; in [41], Eisenring *et al.* design a system using synchronous dataflow; and in [102], Ortega and Borriello use a communication protocol with a non-blocking write behavior, which is similar to the one in process networks.

Another difference is on the level of abstraction. Since design is a process of refinement, the description of a component may exist at different levels. In [41], Eisenring, *et al.* divide the possible abstractions into two levels: *abstract communication types* and *physical communication types*. Abstract communication types includes buffered versus non-buffered, blocking versus non-blocking, synchronous versus asynchronous communication. Physical communication types includes memory-mapped I/O, interrupt or DMA-transfer. In [14], Borriello, *et al.* gave a more contiguous categorization of interface levels: electrical, logical, sequencing, timing, data transaction, packet, and message. The behavioral type work addresses the highest level in this classification: different mechanisms for message passing. It covers the abstract communication types. On the other hand, most work on protocol synthesis is at the hardware or architecture levels. For example, reconfigurable computing with FPGA is targeted in [40]; [104] is about RTL level interface synthesis; the problem of mapping a high-level specification to an architecture is considered in [102]; and a system to generate interface between a set of microprocessors and a set of devices is described in [25][26].

The differences between our type system and the work in protocol synthesis make them complementary to each other. They may be used at the different stages of the design process.

5 Implementation in Ptolemy II

In this chapter, we describe our implementation of the type system presented in the previous two chapters. This implementation is done in the Ptolemy II [33] software. We will first give an overview of Ptolemy II, then describe the implementation of a generic graph package that supports the construction of partial orders and the solution of inequality constraints over a lattice, followed by the implementation of the data level type system and the support for interface automata. Part of the material in this chapter is drawn from the Ptolemy II design document [33].

5.1 Overview of Ptolemy II

Ptolemy II offers unified infrastructure for implementation of a number of models of computation. It consists of a set of Java packages that are listed in figure 5.1. The core packages provide a basic infrastructure and tools that are shared by all the models of computation. Each model of computation is implemented in a separate package as a Ptolemy II domain. The user interface (UI) packages provide a graphical environment for building and executing Ptolemy II models. The name of the graphical environment is called vergil. Vergil stores models in text files using an XML schema called MoML, which stands for Modeling Markup Language. Some details are omitted here. For example, the actor package contains a subpackage `actor.gui` that is part of the user interface.

The key packages relevant to the implementation of the data-level type system are in the core. They are the kernel package, the data package, the actor package, and the graph package.

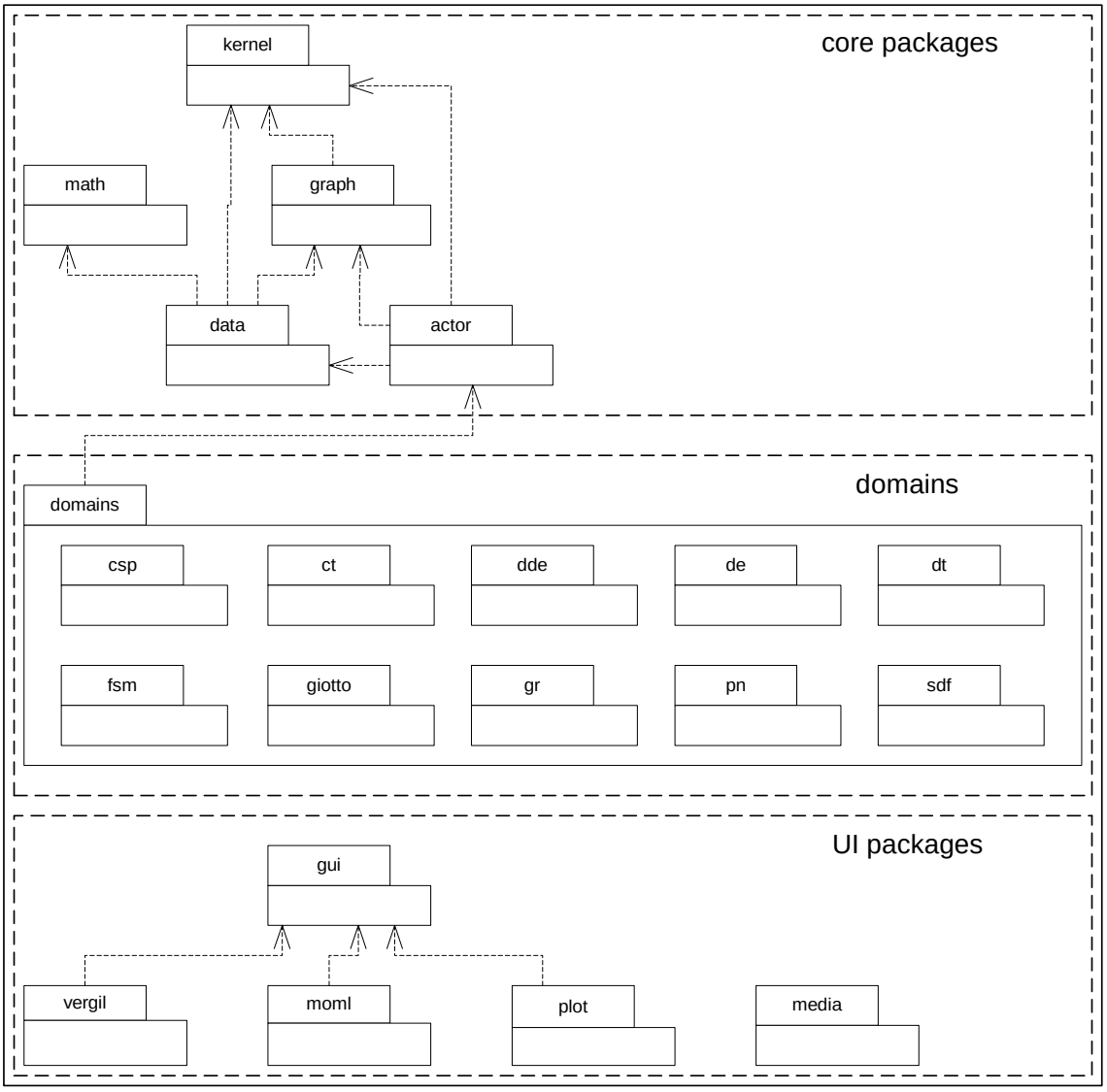


Figure 5.1 The Java packages in Ptolemy II.

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 multi-way associations. Hierarchical graphs can be constructed by encapsulating one graph inside the composite entity of another graph. This encapsulation can be nested arbitrarily.

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 types. The types 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 100 polymorphic actors.

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 matrix 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.

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.

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, 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 be used in different applications.

In [46], Gamma, *et al.* distinguished three classes of software: application, toolkit, and framework. The role of toolkit and framework is reversed. In framework, the main body of the code is reused, but the code it calls must be written. In this view, the kernel and the actor packages are a general framework for building executable graphs, the domain packages are more specialized frameworks that implement models of computations, and the rest of the packages, including the actor libraries, are toolkits.

5.2 CPO and Constraint Solving Infrastructure

As mentioned earlier, CPO and constraint solving support are implemented as generic tools in the graph package. Figure 5.2 shows the UML diagram for the classes in this package[1]. The classes `Graph`, `DirectedGraph` and `DirectedAcyclicGraph` support graph construction and provide graph algorithms. Currently, only topological sort and transitive closure are implemented; other algorithms will be added as needed. The `CPO` interface defines the basic CPO operations, and the class `DirectedAcyclicGraph` implements this interface. An instance of `DirectedAcyclicGraph` is also a finite CPO where all the elements and order relations are explicitly specified. Defining the CPO operations in an interface allows expansion to support infinite CPOs and finite CPOs where the elements are not explicitly enumerated. The `InequalityTerm` interface and the `Inequality` class model inequality constraints over the CPO. The `InequalitySolver` class implements the constraint solving algorithm described in section 3.2.3. The usage of these classes can be found in the Ptolemy II design document [33].

One of the fundamental operations on a CPO is to find the least element of a subset, if it exists. This operation is the basis for many other important operations, such as com-

1. Currently, the graph package is being updated by Shuvra Bhattacharyya to support annotation on edges. Figure 5.2 reflects the state of this package before this change.

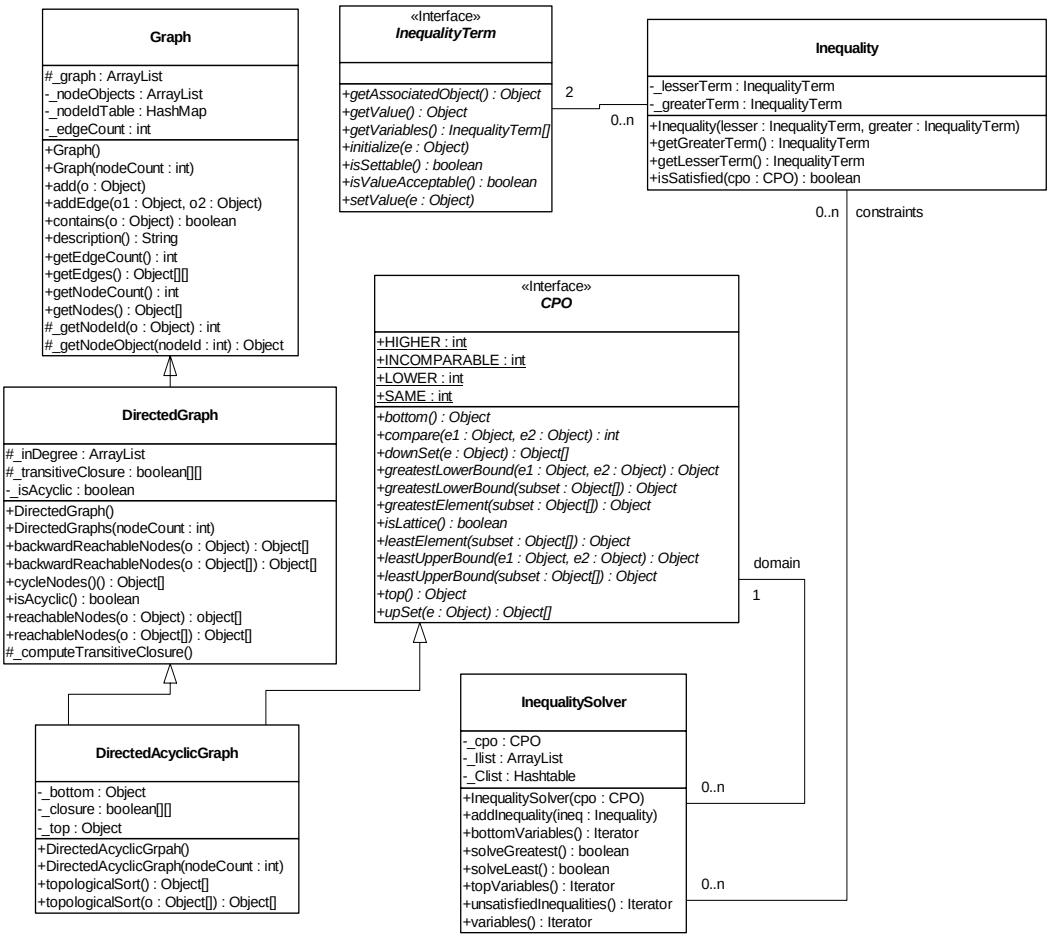


Figure 5.2 Classes in the graph package.

puting the least upper bound. On a totally ordered set, the least element can be found by simply scanning all the elements. The complexity of a linear scan is $O(n)$, where n is the number of elements. On a poset, where elements may be incomparable, the least element cannot be found by a linear scan. One straightforward algorithm is to compare each element in the poset with every other element. This algorithm has a complexity of $O(n^2)$, which is both the average and the worst case complexity.

We use an algorithm that improves the average complexity. The worst case complexity of our algorithm is still $O(n^2)$, but on many posets, the complexity can be significantly reduced. The idea is the following. Suppose the poset in which we want to find the least element is called *inputSet*. We do a linear scan on all the elements in *inputSet*. If we

encounter elements that are incomparable, we temporarily store these elements in a set called *incomparableSet*. Since the least element must be less than all the other elements in *inputSet*, we know that none of the elements in *incomparableSet* can be the least element, but the least element, if it exists, must be less than every elements in *incomparableSet*. As we continue scanning the elements in *inputSet*, we first compare the current element with the elements in *incomparableSet*. If the current element is greater than any element in *incomparableSet*, we ignore the current element and continue the scan. If the current element is less than some of the elements in the *incomparableSet*, but incomparable with the other elements, we remove the elements that are greater than the current element from *incomparableSet*, and add the current element to the *incomparableSet*. If the current element is less than all the elements in *incomparableSet*, we discard all the elements in *incomparableSet*, and the current element becomes a *candidate* for the least element of *inputSet*. The following is a more precise description of this algorithm:

- Initialize: let *inputSet* = {e_i} ($1 \le i \le n$);
  candidate = `null`; *incomparableSet* = `empty set`;
- Scanning:
  For `i` = `1` to `n`
    If *candidate* = `null` and *incomparableSet* = `empty set`
      candidate = e_i;
    Else if *candidate* ≠ `null` and *incomparableSet* = `empty set`
      If $e_i \le$ *candidate*
        candidate = e_i;
      Else if e_i and *candidate* are incomparable
        Add both e_i and *candidate* to *incomparableSet*;
        candidate = `null`;
    Else if *candidate* = `null` and *incomparableSet* ≠ `empty set`
      For each element `c` in `incomparableSet`
        If $e_i \le$ `c`
          Remove `c` from *incomparableSet*;
        Else if `c` $\le e_i$
          ignore e_i and continue scanning the next element in *inputSet*
      If *incomparableSet* = `empty set`

// e_i is less than all elements in *incomparableSet*
candidate = e_i;
Else
Add e_i to *incomparableSet*;
Else if *candidate* ≠ `null` and *incomparableSet* ≠ `empty set`
// This case cannot happen

- If *candidate* ≠ `null`
  candidate is the least element of *inputSet*;
  Else
  The least element does not exist;

The worst case for this algorithm happens when *inputSet* is an anti-chain (a partial order in which every element is incomparable with every other element). The best case is that *inputSet* is a chain (a totally ordered set). In this case, no element will be added to *incomparableSet* during the execution of the algorithm and the complexity reduces to $O(n)$. This algorithm is implemented in a private method of the `DirectedAcyclicGraph` class.

5.3 Data Types

5.3.1 Data Encapsulation and Type Representation

The data package contains a set of token classes that encapsulate data. The UML diagram for these classes is shown in figure 5.3. 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.

Type conversion is done by the 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 `con-`

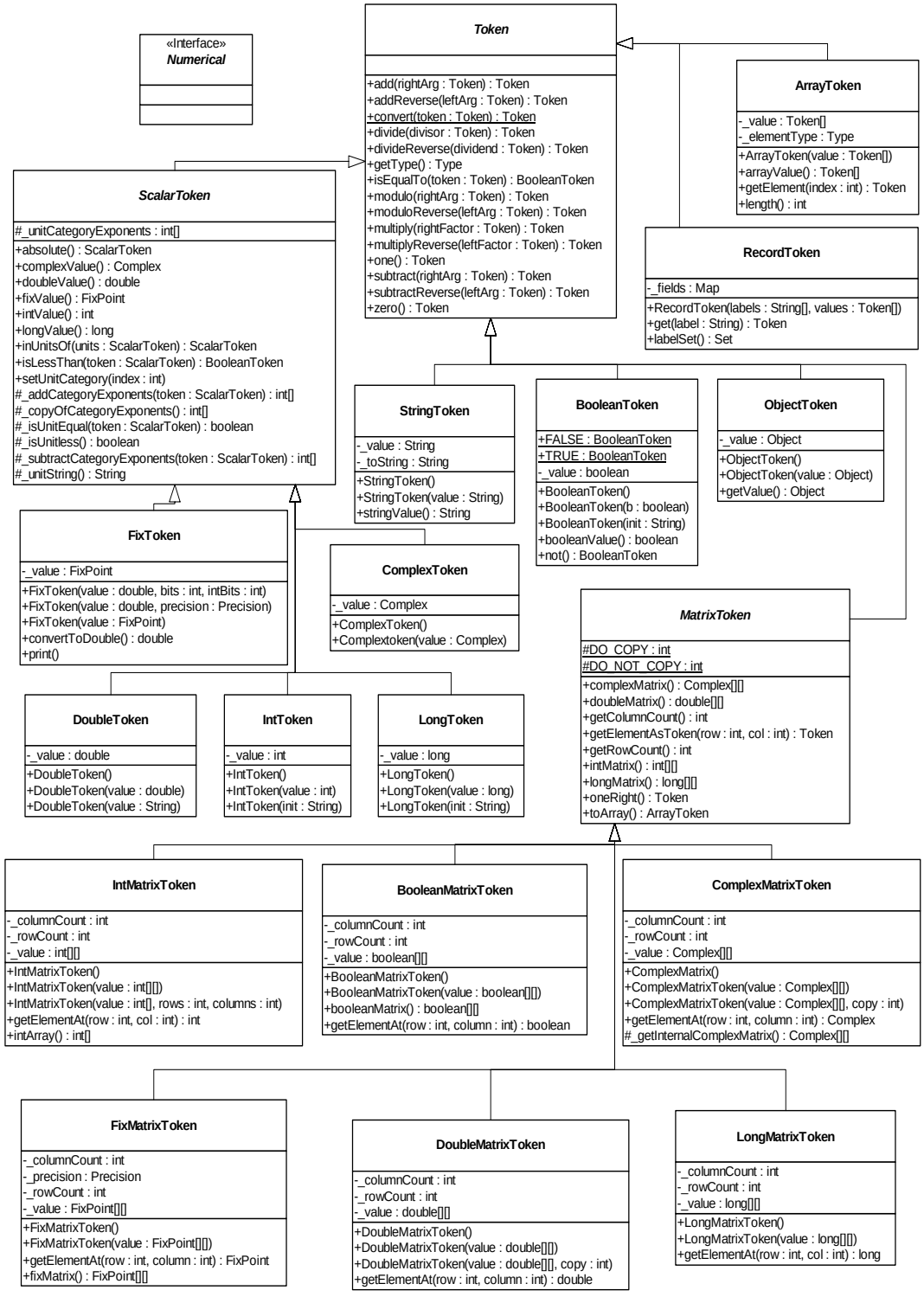


Figure 5.3 Classes in the data package.

`vert()` method one level below to do the conversion. If the argument is higher in the 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.

All the classes for representing the types and the type lattice are under a subpackage of data, `data.type`. Figure 5.4 shows the UML diagram for this package. The `Type` interface defines the basic operations on a type. `BaseType` contains a type-safe enumeration of all the primitive types. `ArrayType` and `RecordType` are derived from

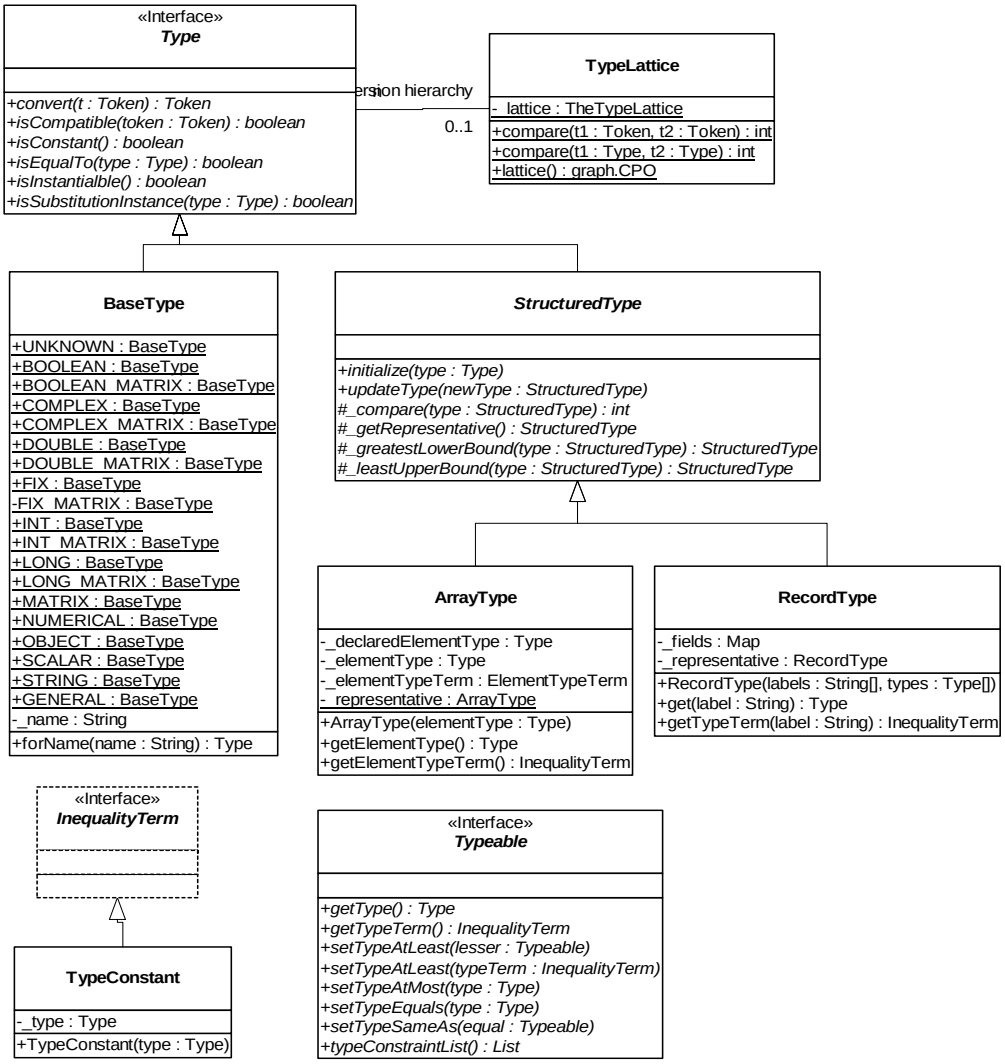


Figure 5.4 Classes in the data.type package.

an abstract class `StructuredType`. The class `TypeLattice` contains the lattice shown in figure 3.2 in chapter 3. This lattice is constructed using the CPO infrastructure in the graph package. Each type has a `convert()` method to convert a token lower in the type lattice to one of its type. For base types, this method just calls the same method in the corresponding tokens. For structured types, the conversion is done within the concrete structured type classes.

The `Typeable` interface defines a set of methods to set type constraints between typed objects. It is implemented by the `Variable` class in the `data.expr` package and the `TypedIOPort` class in the `actor` package. Details of these two classes can be found in [33]. `TypeConstant` encapsulates a constant type. It implements the `InequalityTerm` interface and can be used to set up type constraints between a typed object and a constant type.

5.3.2 Type Checking and Type Conversion

Type checking and type conversion are implemented in the actor package. The detailed information about this package can be found in [33]. The classes and interfaces related to type handling are `TypedActor`, `TypedAtomicActor`, `TypedCompositeActor`, `TypedIOPort`, and `TypedIORelation`. They extend the untyped version of the corresponding classes and interfaces, as shown in figure 5.5. `TypedIOPort` has a declared type and a resolved type. The undeclared type is represented by `BaseType.UNKNOWN`. If a port has a declared type that is not `BaseType.UNKNOWN`, the resolved type will be the same as the declared type.

Static type checking is done in the `checkTypes()` method of `TypedCompositeActor`. This method finds all the connections within the composite by first finding the output ports on deep contained entities, and then finding the deeply connected input ports for those output ports. For each connection, if the types on both ends are declared, static type checking is performed using the type compatibility rule described in section 3.2.2. If the composite contains other instances of `TypedCompositeActor`, this method recursively calls the `checkTypes()` method of the contained actors to perform type

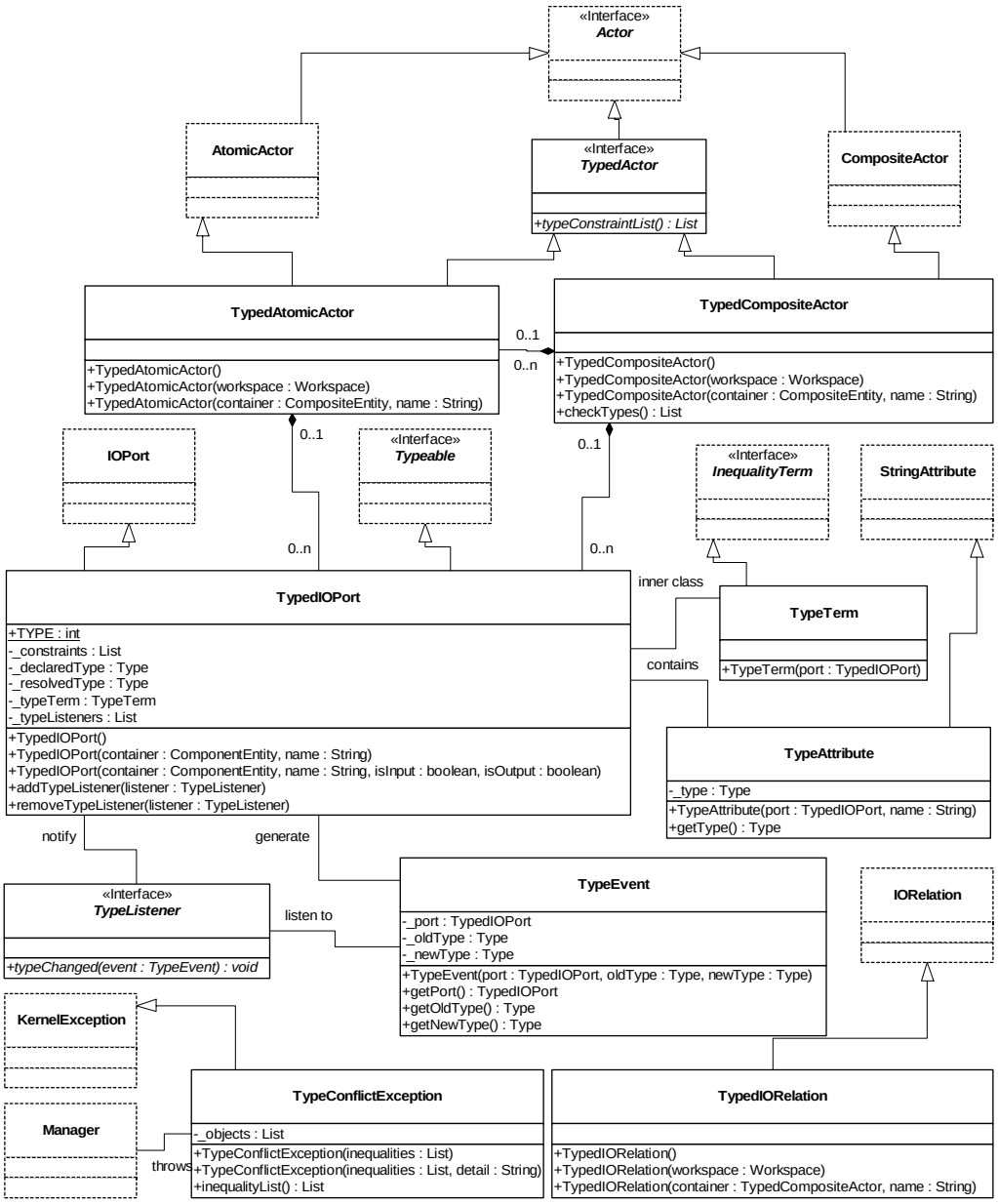


Figure 5.5 Classes in the actor package that support type checking.

checking down the hierarchy. Hence, if this method is called on the top level `TypedCompositeActor`, type checking is performed through out the hierarchy.

If a type conflict is detected, i.e., if the declared type at the source end of a connection is greater than or incomparable with the type at the destination end of the connection, the ports at both ends of the connection are recorded and will be returned in a list at the end of

type checking. Note that type checking does not stop after detecting the first type conflict, so the returned list contains all the ports that have type conflicts. This behavior is similar to a regular compiler, where compilation will generally continue after detecting errors in the source code.

The `TypedActor` interface has a `typeConstraintList()` method, which returns the type constraints of this actor. For atomic actors, the type constraints are different in different actors, but the `TypedAtomicActor` class provides a default implementation, which is that the type of any input port with undeclared type must be less than or equal to the type of any undeclared output port. Ports with declared types are not included in the default constraints. If all the ports have declared type, no constraints are generated. This default works for most of the control actors such as commutator and multiplexer. So by providing the default, we make it easier to write such actors. In addition, the `typeConstraintList()` method also collects all the constraints from the contained `Typeable` objects, which are instances of `TypedIOPort` and `Variable`.

The `typeConstraintList()` method in `TypedCompositeActor` collects all the constraints within the composite. It works in a similar fashion as the `checkTypes()` method, where it recursively goes down the containment hierarchy to collect type constraints of the contained actors. It also scans all the connections and forms type constraints on connections involving undeclared types. As with `checkTypes()`, if this method is called on the top level container, all the type constraints within the composite are returned.

The `Manager` class has a `resolveTypes()` method that invokes type checking and resolution. It uses the `InequalitySolver` class in the graph package to solve the constraints. If type conflicts are detected during type checking or after type resolution, this method throws `TypeConflictException`. This exception contains a list of inequalities where type conflicts occurred.

Run-time type checking is done in the `send()` method of `TypedIOPort`. This method puts a token into the destination receiver. The checking is simply a comparison of the type of the token being sent with the resolved type of the port. If the type of the token is less

than or equal to the resolved type, type checking passes, otherwise, an exception is thrown.

Type conversion, if needed, is also done in the `send()` method. The type of the destination port is the resolved type of the port containing the receivers that the token is sent to. If the token does not have that type, the `convert()` method on that type is called to perform the conversion.

5.3.3 Structured Types Implementation

The implementation of the structured types is more involved than the base types. The complexity is due to the following:

- Type constraints may involve the element type of a structured type.
- As discussed in section 3.3.2.1, when the right side of an inequality term is a variable structured type, the update step in the type resolution algorithm involves a unification of the right side variable structured type with the least upper bound of both sides.
- While base types are atomic entities that will not change, variable structured types are mutable and may change. For example, using the syntax of chapter 3, a type {*UNKNOWN*} may be updated to {*Double*} during type resolution. Because of the mutation, two instances of typed objects, such as two instances of `TypedIOPort`, cannot share the same instance of the variable `StructuredType` as their type.

To understand these issues better, let's first take a look at how type resolution is conducted with base types.

In Ptolemy II, the type of a `TypedIOPort` is stored in a local variable. This variable is a reference to an instance of `Type`, and it is encapsulated in an inner class of `TypedIOPort` that implements the `InequalityTerm` interface. For example, figure 5.6 shows the run-time object structure of a `TypedIOPort` p1 with type *Int*. Here, a solid line box represent an instance of a class or interface, a dashed line box represents a reference, and an arrow from the dashed line box points to the object that the reference is referring to. If a solid line box is contained within another box, it is an instance of an inner class. For example, e1 is an instance of an inner class that implements the `InequalityTerm` interface.

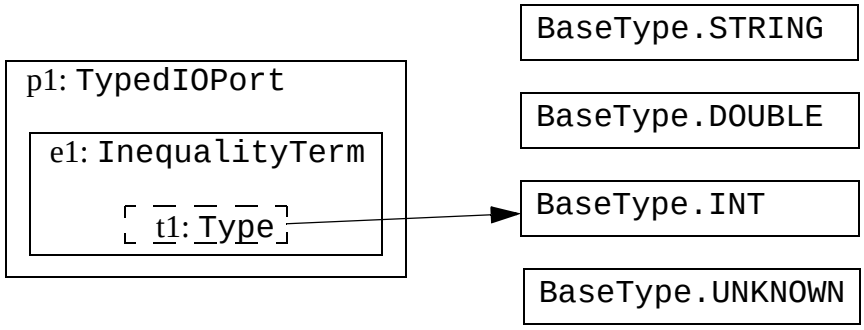


Figure 5.6 Run-time object structure of a `TypedOIPort` with type *Int*.

The inequality type constraints are implemented by the `Inequality` class, which contains two references for the lesser and greater inequality terms. Figure 5.7 shows the run-time object structure of two ports and a constraint that the type of the first port is less than or equal to the type of the second. At the moment depicted by figure 5.7, the type of the first port is *Int*, and the type of the second port is *UNKNOWN*, so the type constraint is *Int* ≤ α, and the current value of α is *UNKNOWN*. During type resolution,

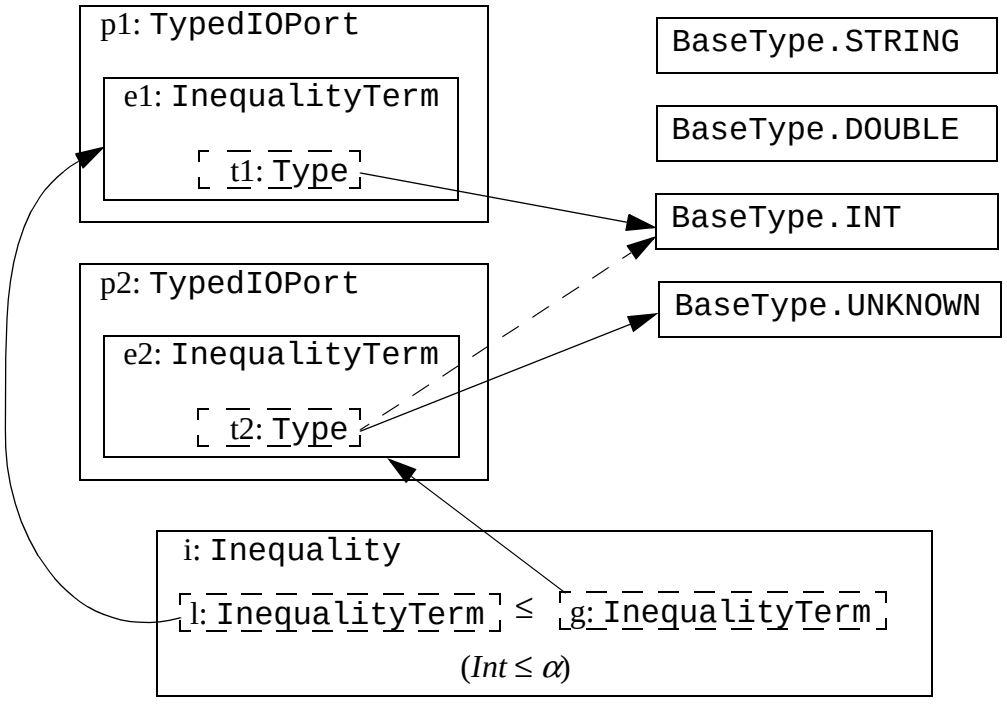


Figure 5.7 Run-time object structure during type resolution for the base type system.

α will be updated to the least upper bound of *Int* and *UNKNOWN*, which is *Int*. This update is done by directly changing the reference t2 (the type of the port p2) to point to the object `BaseType.INT`, as shown by the dashed arrow.

This implementation works well for base types, but adding structured types requires some non-trivial extensions. Since we allow type constraints to involve the element type of structured types, the element type must also be wrapped in an instance of `InequalityTerm`. For example, figure 5.8 shows two instances of `ArrayType`. Here, at1 and at2 are the element types, and the two array types are {*Int*} and {*UNKNOWN*} respectively. Now, assuming the object a2 in figure 5.8 represents the type of a port, and during type resolution, we want to update the type of this port to {*Int*}. This update cannot be done simply by moving the type reference of the port to point to the object a1, because the element type of a2 may be part of another type constraint. Figure 5.9 shows an example of this situation. Here, we have three ports p1, p2 and p3, with types {*Int*}, {*UNKNOWN*}, and *UNKNOWN*, respectively. We also have two type constraints. The first one is that the type of p1 is less than or equal to that of p2. If we use α to denote the element type of the array type a2, this constraint is {*Int*} $\leq$ {α}. The second constraint is that the type of p3 is less than or equal to the element type of a2. If we use β to denote the type of p3, this constraint is $\beta \leq \alpha$. During type resolution, when processing the inequality {*Int*} $\leq$ {α}, we compute the least upper bound of both sides,

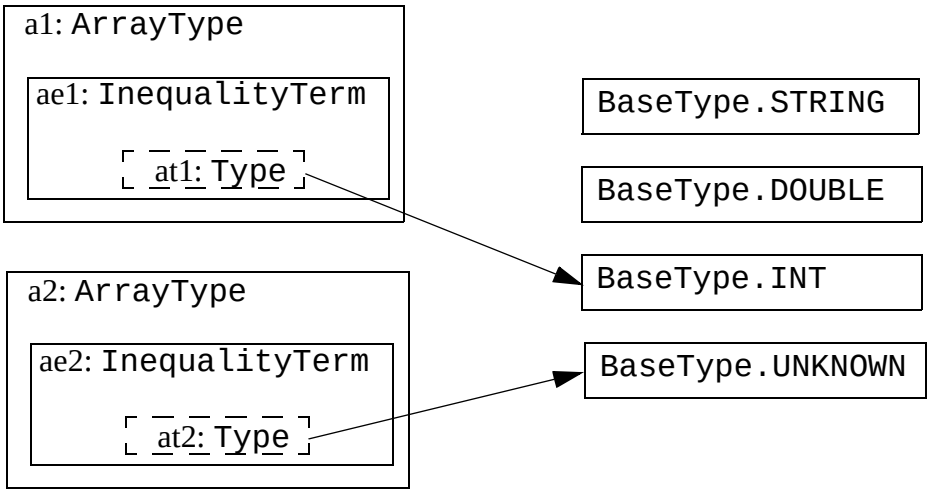


Figure 5.8 Run-time object structure for two instances of array type.

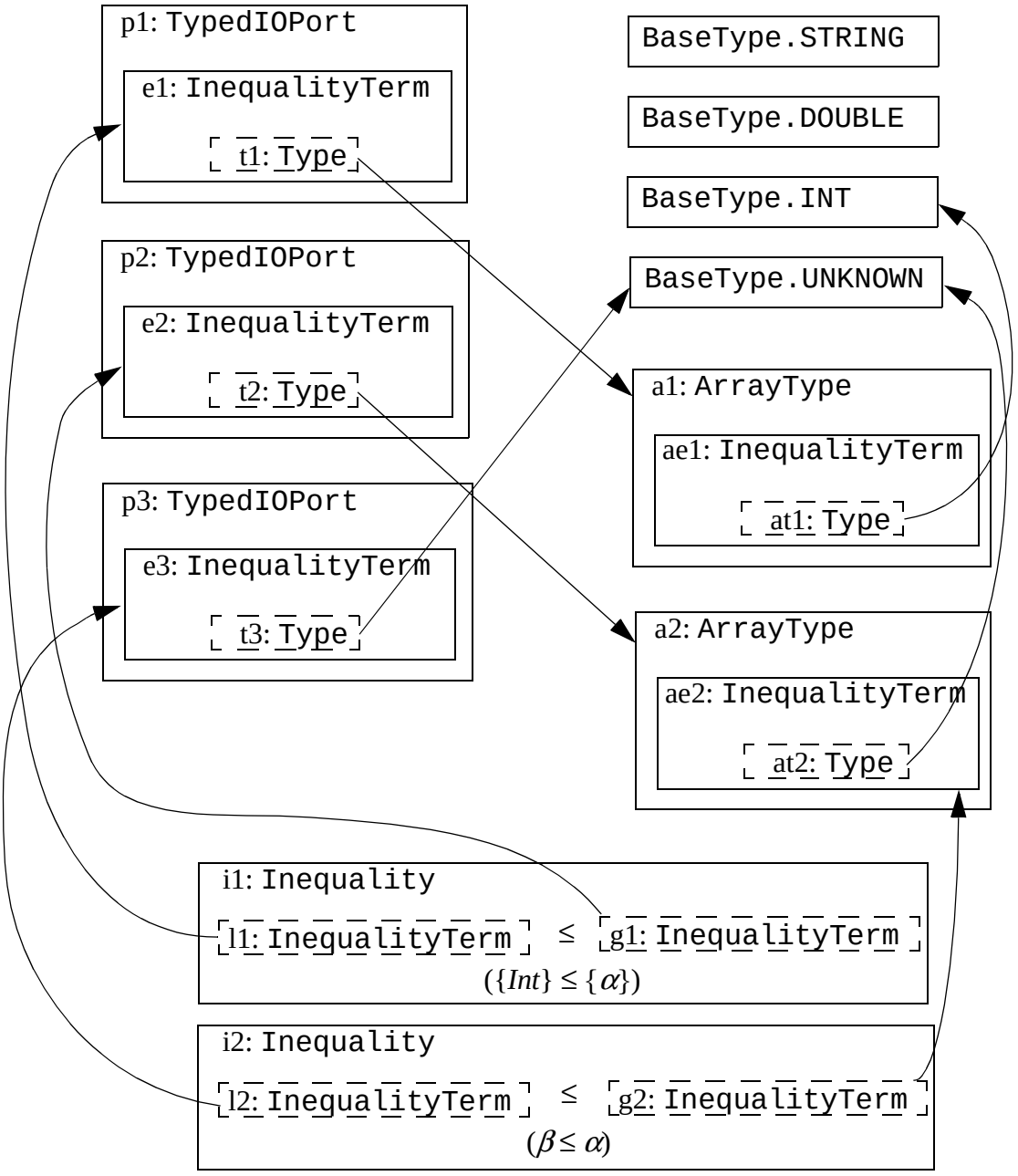


Figure 5.9 Run-time object structure during type resolution for structured types.

which is {*Int*}. Now, we cannot simply update the type of p2 to {*Int*} by moving the reference t2 to point to the array type a1. If we do so, the greater term reference of the second inequality would be pointing to the wrong inequality term. This means that the update for the type of p2 must be done *in place*. Instead of changing the reference t2,

we should update the element type reference of a2. That is, we move at2 to point to `BaseType.INT`. This step is implemented by the `updateType()` method in the `ArrayType` class.

For this update to succeed, the least upper bound of both sides of the inequality must be a substitution instance of the right hand side. In the case of figure 5.9, {*Int*} is indeed a substitution instance of {α}. The method `isSubstitutionInstance()` defined in the `Type` interface performs this check. This method is called from `updateType()`. If the check fails, an exception is thrown to indicate a type error in the model.

Another issue in the implementation of structured type is that an instance of variable `StructuredType` cannot be shared by multiple typed objects. This is different from base types. In Ptolemy II, base types are implemented as a type-safe enumeration, and there is only one instance of the `BaseType` class for each base type. For example, `BaseType.INT` is the only instance of `BaseType` that represents the *Int* type. If there are multiple ports that all have the *Int* type, their type references all point to the same instance `BaseType.INT`. For structured types, since there are an infinite number of them, and they are mutable, they cannot be implemented by a type-safe enumeration. Hence, multiple instances of the same structured type, such as {*Int*}, can be created. However, the same instance of variable structured type can not be shared by multiple ports. If so, changing the element type for one port will affect other ports. To see this difference from the developer's point of view, let's look at a program example.

Assuming there are two ports in an actor, p1 and p2. If a developer wants to declare that these two ports can work with any type, he or she can write the following code:

```
Type declaredType = BaseType.UNKNOWN;
p1.setTypeEquals(declaredType);
p2.setTypeEquals(declaredType);
```

After type resolution, p1 and p2 may be resolved to different types. Note that this code is usually not necessary since the default type of ports is `BaseType.UNKNOWN`. Nevertheless, this code is perfectly legal.

On the other hand, if the developer wants to declare that these two ports can work with any array type, and their element types are not necessarily the same. An obvious way of coding seems to be:

```
// An array of anything
Type declaredType = new ArrayType(BaseType.UNKNOWN);
p1.setTypeEquals(declaredType);
p2.setTypeEquals(declaredType);
```

This code will result in a run-time object structure shown in figure 5.10. Here, p1 and p2 share the same instance of `ArrayType`. As discussed earlier, type update for structured types are done in place, so these two ports will always have the same type. In another word, a type constraint that requires the types of the two ports to be the same is added implicitly, without the awareness of the developer.

One way to resolve this issue is to ask the developer to write code in another style:

```
p1.setTypeEquals(new ArrayType(BaseType.UNKNOWN));
p2.setTypeEquals(new ArrayType(BaseType.UNKNOWN));
```

This way, the two ports will use different instances of `ArrayType`, and they may resolve to different types after type resolution. However, this solution is not very satisfactory because the developer needs to understand and remember the difference between the implementation of base types and structured types, and the implicit type constraint may cause unexpected errors.

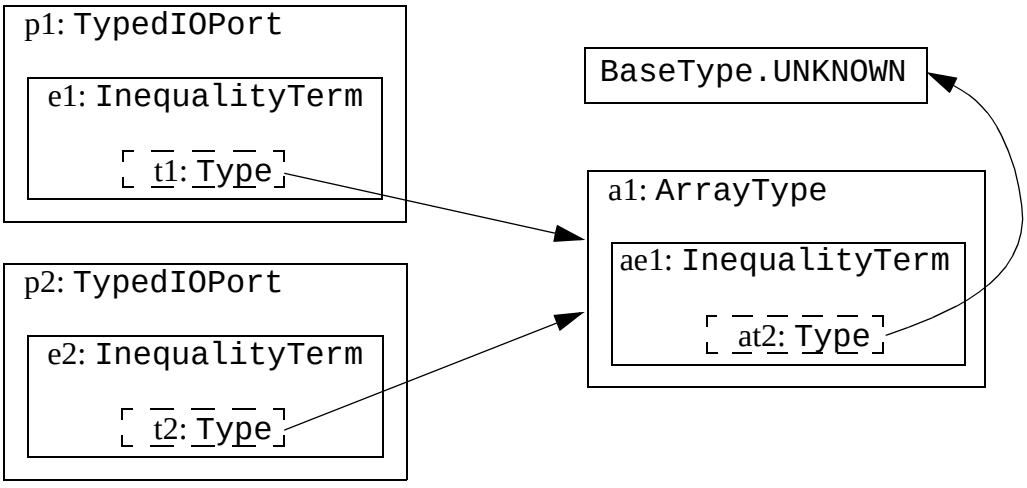


Figure 5.10 Two ports share the same instance of structured type.

To make structured types behave like base types, we adopt another solution, which is to not allow the same instance of variable structured types to be used by two typed objects, such as two ports. To achieve this, we make a clone of the structured type object if it is passed to the `setTypeEquals()` method for the second time. More specifically, each instance of structured type is given a "user". Constant structured types can have many users, variable structured types can only have one user. In the first section of the code above where the developer reuses `declaredType` on the port p2, the second call to `setTypeEquals()` would cause the `ArrayType` instance to be cloned because it is already used by a port. This way, the two code sections above will have the same effect.

5.3.4 Fork Connection and Transparent Port

In a Ptolemy II model, a port can be connected to multiple other ports at the same time. Also, a port can be *transparent*, in which case it does not play a role during message passing. Our type system can handle these special cases easily. We illustrate this through some simple examples.

Fork Connection

Consider two simple topologies in figure 5.11, where a single output is connected to two inputs in 5.11(a) and two outputs are connected to a single input in 5.11(b). Denote the types of the ports by α_1, α_2, α_3, β_1, β_2, β_3, as indicated in the figure. The type constrains in these models are:

$$\alpha_1 \leq \alpha_2$$
$$\alpha_1 \leq \alpha_3$$
$$\beta_1 < \beta_3$$
$$\beta_2 \leq \beta_3$$

Some possibilities of legal and illegal type assignments are:

In 5.11(a), if $\alpha 1$ = *Int*, α_2 = *Double*, α_3 = *Complex*. The topology is well typed. At run-time, the `IntToken` sent out from actor `A1` will be converted to `DoubleToken` before being transferred to `A2`, and converted to `ComplexToken` before being trans-

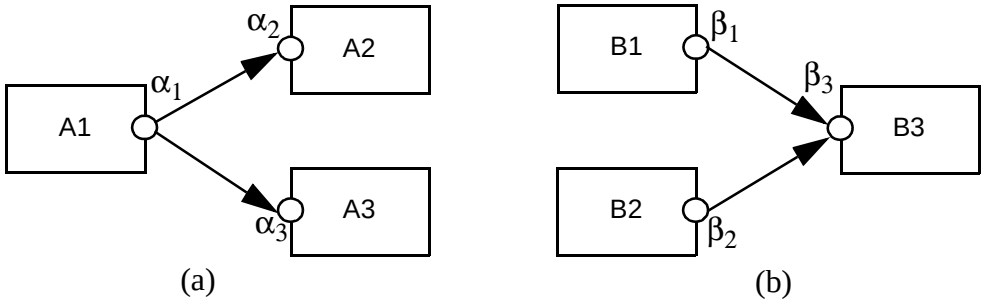


Figure 5.11 Two topologies in which one port is connected to two other ports.

ferred to A3. This shows that multiple ports with different types can be interconnected as long as the type compatibility rule is obeyed.

In 5.11(b), if β_1 = *Int*, β_2 = *Double*, and β_3 is undeclared. The resolved type for β_3 will be *Double*. If β_1 = *Int* and β_2 = *Boolean*, the resolved type for β_3 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 β_3 to be less than *String*, then type resolution is not possible, and a type conflict will be signaled.

Transparent Ports

In Ptolemy II, there are two kinds of composite actors: *opaque* and *transparent*. Opaque composite actors contain a local *director* that manages the execution of the actors inside the composite. On the other hand, transparent composite actors do not contain a local director and the actors inside the composite are managed by an outside director. Ports on a transparent actor are *transparent ports*. These definitions are explained in detail in [33].

Transparent ports are not involved in token passing. For example, in figure 5.12, if the actor A2 is transparent, the token sent out by A1 will be put into the input ports of B1 and B2 without being temporarily stored in the port p1. Similarly, tokens sent out by B1 or B2 will be put into the input of A3 directly. Since the transparent ports p1 and p2 do not play a role in token passing, we do not include them in type checking and type resolution. However, in the user interface of Ptolemy II, we want to assign types to these ports so their information can be displayed in a way consistent with opaque ports. Obviously, the types we assign should meet all the type constraints should these

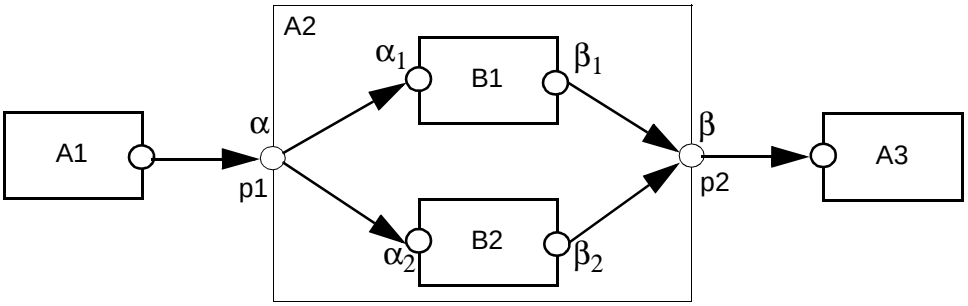


Figure 5.12 A model with a transparent composite actor.

ports be opaque. In particular, the type of `p1` should be less than or equal to the types of the input ports of `B1` and `B2`, and the type of `p2` should be greater than or equal to the types of the output ports of `B1` and `B2`. This is very similar to the fork connection cases discussed above. To meet such constraints, we define the type of a transparent input port to be the greatest lower bound of the types of the input ports connected on the inside, and the type of a transparent output port to be the least upper bound of the types of the output ports connected on the inside. In figure 5.12, this means that $\alpha = \alpha_1 \wedge \alpha_2$, and $\beta = \beta_1 \vee \beta_2$.

An interesting special case is empty transparent composite actors. The type of an input port on such an actor is the greatest lower bound of an empty set, and the type of an output port on such an actor is the least upper bound of an empty set. As discussed in section 2.4.1.1, the greatest lower bound of an empty set is the top element of the CPO, and the least upper bound is the bottom element. This means that the type of the input port is *General*, and the type of the output port is *UNKNOWN*. Figure 5.13 shows two screen shots of a transparent actor in Ptolemy II with its input and output types displayed in tooltips. This type assignment makes perfect sense. Since an empty compos-

Figure 5.13 An empty transparent composite actor in Ptolemy II with input and output types displayed.

ite actor does not do anything with the input token, it can accept tokens with any type. On the other hand, since this actor does not send out any token, its output type should be the least element in the type lattice so that it can be connected with ports of any type. From the typing point of view, this actor is maximally reusable.

5.4 Interface Automata

5.4.1 Implementation Classes

Interface automata are implemented in the FSM domain in Ptolemy II. This implementation leverages heavily the finite state machine infrastructure, so only two new classes are needed. Figure 5.14 shows the UML diagram. The class `InterfaceAutomaton` models an interface automaton. It contains instances of `State` and `InterfaceAutomatonTransition`. This class also includes some tools that manipulate interface automata or find information about them. The `combineInternalTransitions()` method searches for chains of internal transitions and combines each chain into a single internal transition. The resulting automaton may be smaller, but does not make an observable difference in behavior since multiple internal transitions in a row cannot be distinguished from outside the automaton. The `compose()` method com-

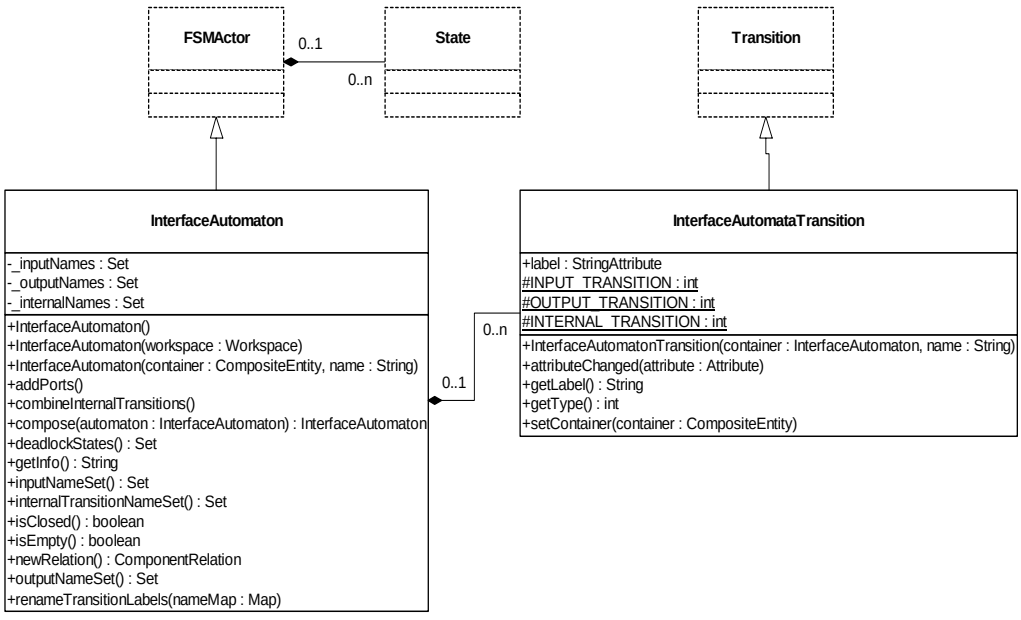


Figure 5.14 Classes implementing interface automata.

putes the composition of two automata. The composition algorithm will be described later. The method `deadlockStates()` searches for states that do not have any outgoing transitions. The `getInfo()` method returns some high level information about the automaton, such as the number of states, the number of transitions, and the names of the transitions. The `isClosed()` method checks whether the automaton is closed. That is, whether it only has internal transitions. The `isEmpty()` method checks whether the automaton is empty.

The `InterfaceAutomatonTransition` class implements the transition of interface automata. This class ensures that the label of the transition ends with an appropriate character (?, !, or ;) and determines the type of the transition from this ending character.

5.4.2 Composition Algorithm

As mentioned above, the `compose()` method in `InterfaceAutomaton` computes the composition of two interface automata. There are four major steps in this method:

1. Check composability
2. Construct the product automaton
3. Prune illegal states
4. Remove unreachable states

In step 1, we check that the transitions of the two automata are disjoint, except that an input transition of one may coincide with an output transitions of the other, in which case, the transitions will become a shared transition in the composition. If this condition is not met, an exception is thrown. If this condition is met, we proceed to construct the product automaton in step 2. The product automaton is constructed progressively from the initial state, so only states that are reachable from the initial states are explored. In addition, when we encounter illegal states in the product, we stop further exploration from those states. Hence, this procedure does not actually construct the full product automaton, but only the portion that may survive in the final composition. Since the composition of two interface automata may be very small, this procedure may result in significant savings than the construction of the full product automaton. In step 3, we

prune out the illegal states in the product, and all the states that can reach those illegal states through output and internal transitions. After this pruning, the resulting automaton may be a disconnected graph, so we remove the states that are not reachable from the initial state in step 4.

In step 2, 3 and 4 above, we use the standard frontier exploration algorithm, which is similar to breadth first search of graphs. We outline the procedure in step 2 here. We start from the initial state of the product automaton, and explore other states along the legal transitions. During the exploration, the whole state space of the product automaton is divided into three parts, as shown in figure 5.15. The frontier is the set of states that are currently being explored. Let the two automata to be composed be `P` and `Q`, the algorithm can be described as follows:

- Initialize:
  `product` = `frontier` = (`initialState_of_P`, `initialState_of_Q`);
- Iterate:
  Pick (remove) a state (`p`, `q`) from `frontier;`
  Exploring from `p`: pick a transition `pTr` (`p`, `r` are the source and destination states of transition `T` in `P`);

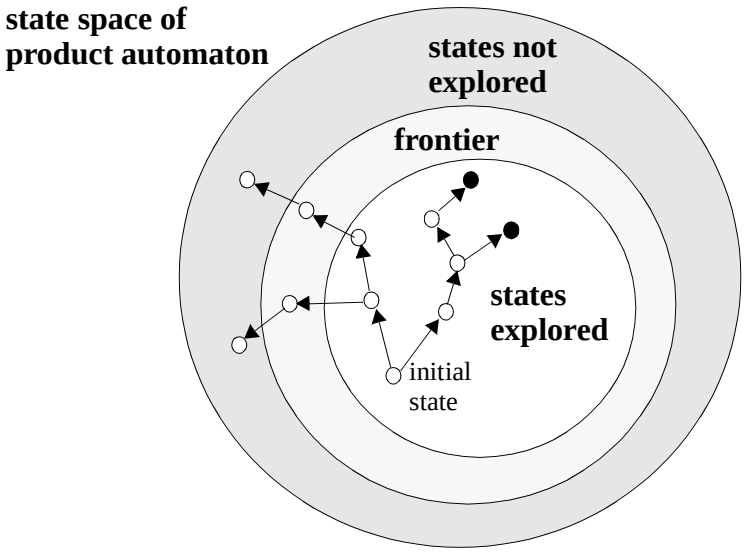


Figure 5.15 Frontier exploration in the product automaton. The black dots represent illegal states.

case 1: T is input transition in P

case 1A: T is input transition in P x Q

Add state (r, q) to product;

Add transition (p, q)T(r, q) to product;

case 1B: T is shared transition in P x Q

case 1Ba: state q in Q has output transition qTs

Add state (r, s) to product;

Add internal transition (p, q)T(r, s) to product;

case 1Bb: state q in Q does not have output transition T

Transition T cannot happen in (p, q), ignore;

case 2: T is output transition in P

case 2A: T is output transition in P x Q

Add state (r, q) to product;

Add transition (p, q)T(r, q) to product;

case 2B: T is shared transition in P x Q

case 2Ba: state q in Q has input transition qTs

Add state (r, s) to product;

Add internal transition (p, q)T(r, s) to product;

case 2Bb: state q in Q does not have input transition T

Mark (p, q) as illegal state;

Stop exploring from (p, q);

case 3: T is internal transition in P

Add state (r, q) to product;

Add transition (p, q)T(r, q) to product;

Exploring from q: (the procedure is symmetrical with exploring from p, but shared transitions are not added twice. Details omitted here.)

- Stop when frontier is empty;

When pruning illegal states, frontier exploration starts from the set of illegal states marked in the above procedure. In the last step of the compose() method, the exploration starts from the initial state.

The composition of interface automata is also supported through the vergil interface of Ptolemy II.

6 Conclusions and Future Work

6.1 Summary

In this thesis, we have presented an extensible type system for component-based design. Fundamentally, a type system detects incompatibilities at component interfaces. Incompatibility may happen at two different levels: data types and dynamic behavior. Accordingly, the type system presented in this thesis also has two parts. For data types, our 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 subtyping relation among types, and use inequalities defined over the type lattice to specify type constraints in components and across components. Compared with the subtyping hierarchy in many conventional languages, our lattice formulation is more restrictive in that we require the subtyping relation to be antisymmetric and the least upper bound of any two types to exist. This restriction enables us to use a very efficient algorithm to solve the inequality type constraints, 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. It addition, it can be extended in two ways: by adding more types to the lattice, or by using different lattices to model different system properties.

We have also presented our approach for supporting structured types in this system. The addition of structured types requires extensions on both the theoretical formulation and the implementation of the system. In particular, we extend the format of inequality constraint to admit variable structured types, and add a unification step in the constraint solving algo-

rithm to handle these types. We have also analyzed the issue of convergence on an infinite type lattice with structured types.

The data-level type system has been implemented in the Ptolemy II environment. Our implementation is modular. The CPO and lattice support, including the algorithm for solving inequality constraints, are implemented in the graph package of Ptolemy II. This infrastructure is generic in that it is not bound to one particular type lattice. Data encapsulation and type definition were implemented in the data package, and type checking and type conversion are implemented in the actor package.

To describe the dynamic behavior of components and perform compatibility check, we extend the concepts of conventional type system to behavioral level and capture the dynamic interaction between components, such as the communication protocols the components use to pass messages. In our system, the interaction types and the dynamic behavior of components are defined using interface automata. To check whether a component is compatible with a certain interaction type, we can simply compose the automata models of the component and the interaction type, and check whether the result is empty. This yields a straightforward algorithm for type checking at the behavioral level. Our system is polymorphic in that a component may be compatible with more than one interaction types. We have shown that the alternating simulation relation of interface automata can be used for defining subtyping, and it induces a partial order for the interaction types. This behavioral type order provides significant insight into the relation among various interaction types. It can be used to facilitate the design of polymorphic components and simplify type checking. In addition to static type checking, we also propose to extend the use of interface automata to the on-line reflection of component states and to run-time type checking. We have also discussed the trade-offs in the design of behavioral type systems.

We have implemented interface automata in the FSM domain of Ptolemy II. All the automata in this thesis are built in Ptolemy II and their compositions are computed in software, except that some manual layout is applied for better readability of the diagrams.

6.2 Future Work

6.2.1 Data Types

Our data-level type system can be improved in several ways:

Type Resolution for Open Systems

When a typed component with inputs and outputs is used in an untyped environment, the environment does not provide type constraints for the inputs and outputs. In this case, type resolution is done on an open system. This is analogous to compiling an individual library module in text-based languages. To maximize the utility of the typed component, we want to resolve the input type to the most general while keeping the type everywhere else to the most specific. To achieve this, we plan to use a two-pass algorithm[1]. In the first pass, we find the most general types in the system. In the second pass, we fix the types of the input, and search for the most specific types for everywhere else.

Deriving Type Constraints for Expressions

As discussed in chapter 3, we assume that the detailed operation of the components are not exposed to the type system and we only check type consistency at component interface. However, in some cases, it is possible to examine the operation of the components and extend the reach of type checking. For example, the `Expression` actor in Ptolemy II computes an expression specified by the user. It should be possible for the type system to analyze the expression and generate type constraints that link the type of the expression with the types of the inputs and outputs of the actor. As another example, in the FSM domain in Ptolemy II, the guard and action on the transitions can be specified using expressions. These expressions can also be used to generate type constraints for the inputs and outputs of the finite state machine. By doing this, we can reduce run-time type errors and the need for the user to specify the types at the inputs and outputs of the components.

1. This algorithm came up during a discussion with Jörn Janneck.

Adding More Structured Types

Currently, Ptolemy II only supports two structured types: array and record. More types, such as *union* and *tuple* [21], can be added. Union type may be useful, for example, in a communication system where a received message are drawn from a predefined set. Tuple can be used as a generalized array, where the types of the elements do not have to be the same, or as a simplified record, where the record labels are reduced to numerical indices. We believe these types can be added without much difficulty, but the question is whether they are useful enough to warrant their presence in the type system.

6.2.2 Behavioral Types

The behavioral types framework presented in this thesis can be extended in a number of ways. Most of these extensions are speculative, but they may lead to some interesting opportunities.

Running the Reflection Automata

Currently, the interface automata implementation in Ptolemy II does not support execution. By leveraging the execution framework of the FSM domain, we can make interface automata executable. We have mentioned the possibility of using automata to do on-line reflection of component states. One immediate application of the reflection automata is to provide debug information. In a model, the collective states of the reflection automata provide a snapshot of the model behavior. This information may be more intuitive than the conventional debugging trace. In addition, reflection automata can also be used to control configuration changes. For example, in a reconfigurable architecture or distributed system, the state of the reflection automata can provide information on when it is safe to perform mutation.

Automata Generation

In conventional type systems, data types may be declared by the programmer or inferred by the system. So far in our behavioral types framework, the interaction types and the reflection automata are designed manually based on the source code of Ptolemy II. This is analogous to type declaration. An alternative is to automatically generate these automata from the source code, which can be viewed as inferring the behavior types from the imple-

mentation of components. The Bandera project [31] may provide a starting point for this effort.

Using the Reflection Automata to Aid Actor Development

In Ptolemy II, many actors have a similar behavioral pattern. For example, the reflection automaton in figure 4.3 of chapter 4 describes the common behavior of most of the polymorphic consumer actors. We might be able to use these common patterns to simplify actor development. Instead of asking the user to write a complete Java class for each new actor, we can provide a library of actor templates in the form of interface automata. The library may include such templates as Polymorphic Consumer, Polymorphic Transformer, SDF Consumer, SDF Transformer. When the user needs to develop a new actor, he or she can pick a template, and just write the code that processes the input token, which corresponds to the operation in a particular state in the automaton. The rest of the code in the actor, including the code for execution control and communication, can be provided by the design environment. This approach will have a similar benefit as the Caltrop language [42] for actor development. Note that this approach puts the creation of the reflection automata before the development of actor code, so the behavior types do not need to be inferred from the source code.

Capturing Timing Properties

The interface automata we have used in this thesis are untimed. Using a timed automata model, we may be able to model real-time requirements and constraints. Such automata may be used to analyze the schedule for real-time systems. This is potentially a very rich research area.

Support User Customization

One of the challenges in type systems research is to make type systems amenable to user definition and customization [100]. This is important for behavioral types due to the highly diverse information they may capture. It might be possible to design behavioral types as a metalanguage that can be specified by the user. Eventually, such an approach might lead to flexible design tools that allow the user to specify the relevant properties to check.

Bibliography

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

[2] S. Adolph, "Whatever Happened to Reuse?" *Software Development*, Nov. 1999.

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

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

[5] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1988.

[6] A. Aiken, "Set Constraints: Results, Applications and Future Directions," *Proc. of the Second Workshop on the Principles and Practice of Constraint Programming*, Orcas Island, Washington, May 1994.

[7] A. Aiken, *Lecture Notes for CS263: Design and Analysis of Programming Languages*, Univ. of California at Berkeley, Spring, 1998. (http://www.cs.berkeley.edu/~aiken/cs263/lectures/index.html)

[8] F. Baader and W. Snyder, "Unification Theory," *Handbook of Automated Reasoning*, Elsevier Science Publishers, 2001.

[9] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, 21(8), Aug. 1978.

[10] J. Backus, "The History of Fortran I, II, and III," *IEEE Annals of the History of Computing*, 20(4), 1998.

[11] H. P. Barendregt, "The Lambda Calculus, Its Syntax and Semantics," Revised Edition, *Studies in Logic and the Foundations of Mathematics*, Vol. 103, North-Holland, 1984.

[12] A. Benveniste and G. Berry, “The Synchronous Approach to Reactive and Real-Time Systems,” *Proceedings of the IEEE*, Vol.79, No.9, Sept. 1991.

[13] P. A. Bernstein, T. Bergstraesser, J. Carlson, S. Pal, P. Sanders and D. Shutt, “Microsoft Repository Version 2 and the Open Information Model,” Microsoft Corporation, May 3, 1999.

[14] G. Borriello, L. Lavagno, and R. B. Ortega, “Interface Synthesis: a Vertical Slice from Digital Logic to Software Components,” *Proc. of International Conference on Computer Aided Design (ICCAD)*, San Jose, CA, USA, 8-12 Nov. 1998.

[15] R. L. Brumfield, “Type Systems in Visual Languages,” *Project Report*, Department of Computer Science, University of Colorado, Dec. 1995.

[16] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” *Int. Journal of Computer Simulation*, special issue on ``Simulation Software Development," vol. 4, pp. 155-182, April, 1994.

[17] J. T. Buck and E. A. Lee, “Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model,” *Proc. of ICASSP*, Minneapolis, April, 1993.

[18] M. M. Burnett, “Types and Type Inference in a Visual Programming Language,” *Proc. 1993 IEEE Symposium on Visual Languages*, Bergen, Norway, Aug. 24-27, 1993.

[19] J. Byous, “Java Technology: An Early History,” http://java.sun.com/features/1998/05/birthday.html.

[20] L. Cardelli, “Basic Polymorphic Typechecking,” *Science of Computer Programming* 8, pp. 147-172, North-Holland, 1987.

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

[22] L. Cardelli and P. Wegner, “On Understanding Types, Data Abstraction, and Polymorphism,” *Computing Surveys*, 17(4), Dec. 1985.

[23] W.-T. Chang, S. Ha, and E. A. Lee, “Heterogeneous Simulation - Mixing Discrete-Event Models with Dataflow,” Invited paper for *RASSP special issue of the Journal on VLSI Signal Processing*, 1996.

[24] W.-T. Chang, A. Kalavade, and E. A. Lee, “Effective Heterogenous Design and Co-Simulation,” *Nato Advanced Study Institute Workshop on Hardware/Software Codesign*, Lake Como, Italy, June 18-30, 1995.

[25] P. Chou, R. B. Ortega and G. Borriello, “Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems,” *Proc. ICCAD*, pp488-495, Nov. 1992.

[26] P. Chou, R. B. Ortega and G. Borriello, “Interface Co-Synthesis Techniques for Embedded Systems,” *Proc. of the Int. Conf. on Computer Aided Design*, Nov. 1995.

[27] P. Ciancarini, “Coordination Models and Languages as Software Integrators,” *ACM Computing Surveys*, 28(2), June, 1996.

[28] E. M. Clarke, J. M. Wing, *et al.*, “Formal Methods: State of the Art and Future Directions,” *ACM Computing Surveys*, 28(4), Dec. 1996.

[29] J-L. Colaco, M. Pantel and P. Salle, “A Set-Constraint-Based Analysis of Actors,” *International Workshop on Formal Methods for Open Object-Based Distributed Systems*, Canterbur, UK, July 21-23, 1997.

[30] D. Compare, P. Inverardi, A. L. Wolf, “Uncovering Architectural Mismatch in Component Behavior”, *Science of Computer Programming* 33 (1999) 101-131.

[31] J. C. Corbett, M. D. Dwyer, J. Hatcliff and S. Laubach, “Bandera: Extracting Finite State Models from Java Source Code,” *Proc. of the 2000 International Conf. on Software Engineering*, Limerick, Ireland, June, 2000.

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

[33] J. Davis II, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel and Y. Xiong, “Heterogeneous Concurrent Modeling and Design in Java,” *Technical Memorandum UCB/ERL M01/12*, EECS, University of California, Berkeley, March 15, 2000. (http:// ptolemy.eecs.berkeley.edu/publications/papers/01/HMAD)

[34] L. de Alfaro and T. A. Henzinger, “Interface Automata,” *Proc. of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/ FSE 01)*, Austria, 2001.

[35] J. Dean, D. Grove and C. Chambers, “Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis,” *Proc. of the 9th European Conference on Object-Oriented Programming (ECOOP’95)*, Aarhus, Denmark, Aug., 1995.

[36] A. Diwan, K. S. McKinley and J. E. B. Moss, “Type-Based Alias Analysis,” *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada, 1998.

[37] B. P. Douglass, “Components, States and Interfaces, Oh My!” *Software Development*, April 2000.

[38] B. P. Douglass, “The Evolution of Computing,” *Software Development*, Jan. 2001.

[39] S. A. Edwards, *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*, Ph.D. thesis, University of California, Berkeley, May 1997.

[40] M. Eisenring and M. Platzner, “Synthesis of Interfaces and Communication in Reconfigurable Embedded Systems,” *IEE Proc. Comput. Digit. Tech*, 147(3), May 2000.

[41] M. Eisenring, J. Teich and L. Thiele, “Rapid Prototyping of Dataflow Programs on Hardware/Software Architectures,” *Proc. 31st Annual Hawaii International Conference on System Sciences*, 1998.

[42] J. Eker and J. Janneck, *An Introduction to the Caltrop Actor Language*, Dept. of Electrical Engineering and Computer Sciences, Univ. of California at Berkeley, 2001.

[43] M. Fahndrich, “Effect Systems,” *CS263 Guest Lecture*, Univ. of California at Berkeley, 1996. (http://citeseer.nj.nec.com/249349.html)

[44] E. Freeman, S. Hupfer and K. Arnold, *JavaSpaces Principles, Patterns, and Practice*, *Addison-Wesley Pub. Co.*, 1999.

[45] Y-C. Fuh and P. Mishra, “Type Inference with Subtypes,” *Second European Symposium on Programming*, Nancy, France, 1988.

[46] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[47] J. D. Gannon, “Verification and Validation,” *The Computer Science and Engineering Handbook*, CRC Press, 1997.

[48] A. Girault, B. Lee, and E. A. Lee, “Hierarchical Finite State Machines with Multiple Concurrency Models,” *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 18(6), June 1999.

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

[50] 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.

[51] M. R. Henzinger, T. A. Henzinger and P. W. Kopke, “Computing Simulations on Finite and Infinite Graphs,” *Proc. of the 36th Annual IEEE Symp. on Foundations of Computer Science (FOCS ‘95)*, pp. 453-462, Oct. 1995.

[52] J. B. Hext, “Compile-Time Type-Matching,” *Computer Journal*, 9, 1967.

[53] C. A. R. Hoare, “Hints on Programming Language Design,” In C. Bunyan (ed.), *Computer Systems Reliability: State of the Art Report*, Vol. 20, Pergamon/Infotech, 1974. pp. 505-34.

[54] C. A. R. Hoare, “Communicating Sequential Processes,” *Communications of the ACM*, 28(8), August 1978.

[55] D. Howe (editor), “The Free On-line Dictionary of Computing,” http:// www.foldoc.org.

[56] P. Hudak, “Conception, Evolution, and Application of Functional Programming Languages,” *ACM Computing Survey*, 21(3), Sept., 1989.

[57] V. Illingworth, *Dictionary of Computing*, 3rd ed., Oxford University Press, 1990.

[58] G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” *Proc. of the IFIP Congress 74*, Amsterdam, The Netherlands, North-Holland, 1974.

[59] G. Kahn and D. B. MacQueen, “Coroutines and Networks of Parallel Processes,” *Information Processing 77*, B. Gilchrist (ed.) North-Holland Publishing Co., 1977.

[60] E. A. Lee, “Modeling Concurrent Real-Time Processes Using Discrete Events,” *Annals of Software Engineering*, Special Volume on Real-Time Software Engineering, Volume 7, 1999.

[61] E. A. Lee, “Embedded Software - An Agenda for Research,” *Memorandum UCB/ERL M99/63*, EECS Dept., UC Berkeley, 2001. (http:// ptolemy.eecs.berkeley.edu/publications/papers/99/embedded/)

[62] E. A. Lee, “Embedded Software,” *Memorandum UCB/ERL M01/26*, EECS Dept., UC Berkeley, 2001. (http://ptolemy.eecs.berkeley.edu/publications/ papers/01/embsystems/)

[63] E. A. Lee, “Soft Walls - Modifying Flight Control Systems to Limit the Flight Space of Commercial Aircraft,” *Memorandum UCB/ERL M01/31*, EECS Dept., UC Berkeley, 2001. (http://ptolemy.eecs.berkeley.edu/publications/papers/01/ softwalls/)

[64] E. A. Lee and D. G. Messerschmitt, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing,” *IEEE Transactions on Computers*, January, 1987.

[65] E. A. Lee and D. G. Messerschmitt, “Synchronous Data Flow,” *Proc. of the IEEE*, 75(9), Sept. 1987.

[66] E. A. Lee and T. M. Parks, “Dataflow Process Networks,”, *Proc. of the IEEE*, 83(5), pp. 773-801, May, 1995. (http://ptolemy.eecs.berkeley.edu/publications/papers/95/processNets)

[67] E. A. Lee and A. Sangiovanni-Vincentelli, “A Framework for Comparing Models of Computation,” *IEEE Transactions on CAD*, 17(12), Dec., 1998.

[68] E. A. Lee and P. Varaiya, *Structure and Interpretation of Signals and Systems*, EECS20 Class Reader, EECS Dept., UC Berkeley, 2001.

[69] E. A. Lee and Yuhong Xiong, “System-Level Types for Component-Based Design,” *Technical Memorandum UCB/ERL M00/8*, EECS, University of California, Berkeley, Feb. 29, 2000. (http://ptolemy.eecs.berkeley.edu/publications/papers/00/systemLevel/)

[70] E. A. Lee and Yuhong Xiong, “System-Level Types for Component-Based Design,” *First International Workshop on Embedded Software (EMSOFT2001)*, Tahoe City, CA, USA, Oct. 8-10, 2001. Lecture Notes in Computer Science (LNCS) 2211, Springer.

[71] C. K. Lennard, “VSIA Develops System-Level Modeling Standards,” *EE Times*, June 9, 2000.

[72] P. Lieverse, P. van der Wolf, E. Deprettere and K. Vissers, “A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems,” *IEEE Workshop on Signal Processing Systems (SiPS99)*, Taipei, Taiwan, Oct. 20-22, 1999.

[73] C. H. Lindsey and S. G. van der Meulen, *Informal Introduction to ALGOL 68*, North-Holland Publishing Company, 1971.

[74] B. H. Liskov and J. M. Wing, “A Behavioral Notion of Subtyping,” *ACM Transactions on Programming Languages and Systems*, 16(6), Nov. 1994.

[75] J. Liu, B. Wu, X. Liu and E. A. Lee, “Interoperation of Heterogeneous CAD Tools in Ptolemy II,” *Symp. on Design, Test, and Microfabrication of MEMS/MOEMS*, Paris, France, Mar., 1999.

[76] J. M. Lucassen, *Types and Effects - Towards the Integration of Functional and Imperative Programming*, Ph.D. Thesis, MIT Laboratory for Computer Science LCS TR-408, Aug., 1987.

[77] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems," *Proc. of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, San Diego, California, Jan. 1998.

[78] N. Lynch and M. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," *Proc. 6th ACM Symp. Principles of Distributed Computing*, pp 137-151, 1981.

[79] The MathWorks, Inc., *Using Simulink: Model-Based and System-Based Design*, June, 2001. (http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf)

[80] M. D. McIlroy, "Mass Produced Software Components," In *P. Naur and B. Randell, Software Engineering, Report on a Conference Sponsored by the NATO Science Committee*, Garmisch, Germany, Oct. 7-11, 1968. Scientific Affaires Division, NATO, Brussels, 1969.

[81] B. Meyer, "Every Little Bit Counts: Toward More Reliable Software," *IEEE Computer*, Nov. 1999.

[82] B. Meyer, "The Significance of Components," *Software Development*, Nov. 1999.

[83] B. Meyer, "What to Compose," *Software Development*, Mar. 2000.

[84] B. Meyer, "Contracts for Components," *Software Development*, July, 2000.

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

[86] R. Milner, J. Parrow, D. Walker, "A Calculus of Mobile Processes (Port I and Part II)," *Information and Computation*, 100:1-77, 1992.

[87] J. C. Mitchell, "Coercion and Type Inference," *Proc. of 11th Annual ACM Symp. on Principles of Programming Languages*, pp. 175-185, 1984.

[88] J. C. Mitchell, *Foundations for Programming Languages*, The MIT Press, 1998.

[89] M. Mitchell, "Type-Based Alias Analysis: Optimization that Makes C++ Faster Than C," *Dr. Dobb's Journal*, Oct., 2000.

[90] A. Mycroft and R. A. O'Keefe, "A Polymorphic Type System for Prolog," *Artificial Intelligence*, 23(3), North-Holland, Amsterdam, 1984.

[91] M. A. Najork and E. Golin, “Enhancing Show-and-Tell with a Polymorphic Type System and Higher-Order Functions,” *1990 IEEE Workshop on Visual Languages*, Skokie, IL, Oct., 1990.

[92] M. A. Najork and S. M. Kaplan, “The CUBE Language,” *1991 IEEE Workshop on Visual Languages*, Kobe, Japan, Oct. 1991.

[93] E. Najm, A. Nimour, “Explicit Behavioral Typing for Object Interface,” *Semantics of Objects as Processes, ECOOP’99 Workshop*, Lisboa, Portugal, June, 1999.

[94] E. Najm, A. Nimour and J.-B. Stefani, “Infinite Types for Distributed Object Interfaces,” *Third IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS’99)*, Firenze, Italy, Feb., 1999.

[95] G. Necula and P. Lee, “Safe Kernel Extension Without Run-Time Checking,” *Second Symp. on Operating Systems Design and Implementation (OSDI ‘96)*, Oct. 28-31, 1996.

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

[97] G. Necula, *Lecture Notes for CS263: Design and Analysis of Programming Languages*, Univ. of California at Berkeley, Spring, 1999. (http://www.cs.berkeley.edu/~necula/cs263/lectures.html)

[98] F. Nielson, “Annotated Type and Effect Systems,” *ACM Computing Surveys*, 28(2), June, 1996.

[99] H. R. Nielson and F. Nielson, “Higher-Order Concurrent Programs with Finite Communication Topology,” *ACM Symp. on Principles of Programming Languages*, Jan., 1994.

[100] M. Odersky, “Challenges in Type Systems Research,” *ACM Computing Surveys*, 28(4), 1996.

[101] Object Management Group, *OMG Unified Modeling Language Specification*, version 1.3, June 1999.

[102] R. B. Ortega and G. Borriello, “Communication Synthesis for Embedded Systems with Global Considerations,” *Proc. of the 5th International Workshop on Hardware/Software Co-Design (Codes/CASHE’97)*, March 1997.

[103] J. K. Ousterhout, “Scripting: Higher Level Programming for the 21 Century,” *IEEE Computer Magazine*, March 1998.

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

[105] J. Palsberg, "Type-Based Analysis and Applications," *ACM SIGPLAN/ SIGSOFT Workshop on Program Analysis for Software Tools (PASTE'01)*, Utah, USA, June, 2001.

[106] B. C. Pierce, "Foundational Calculi for Programming Languages," *The Computer Science and Engineering Handbook*, CRC Press, 1997.

[107] J. Plevyak and A. A. Chien, "Precise Concrete Type Inference for Object-Oriented Languages," *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, Oregon, USA, 1994.

[108] F. Puntigam, "Types for Active Objects Based on Trace Semantics," *Proc. of the Workshop on Formal Methods for Open Object-Oriented Distributed Systems (FMOODS'96)*, Paris, France, March, 1996.

[109] A. Ralston, E. D. Reilly and D. Hemmendinger (editors), *Encyclopedia of Computer Science*, 4th ed., Nature Publishing Group, 2000.

[110] H. J. Reekie, *Realtime Signal Processing, Dataflow, Visual, and Functional Programming*, Ph.D. Thesis, University of Technology at Sydney, Sept., 1995.

[111] J. Reekie, S. Neuendorffer, C. Hylands and E. A. Lee, "Software Practice in the Ptolemy Project," *Technical Report Series, GSRC-TR-1999-01*, Gigascale Semiconductor Research Center, University of California, Berkeley, CA 94720, April 1999.

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

[113] J. H. Reppy, "CML: A Higher-Order Concurrent Language," *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Ont., Canada, June, 1991.

[114] D. Ritchie, "The Development of the C Language," *SIGPLAN Notices*, 28(3), *ACM SIGPLAN HOPL-II. 2nd ACM SIGPLAN History of Programming Languages Conference*, Cambridge, MA, USA, 20-23 April 1993.

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

[116] J. A. Rowson, A. Sangiovanni-Vincentelli, "Interface-Based Design," *Proc. of the 34th Design Automation Conference (DAC-97)*. pp. 178-183, Las Vegas, June 1997.

[117] D. Scott, “Outline Of a Mathematical Theory of Computation,” *Proc. of the 4th Annual Princeton Conf. on Information Sciences and Systems*, pp. 169-176, 1970.

[118] Synopsys Inc., *CoCentric System Studio*, 2000. (http://www.synopsys.com/products/cocentric_studio/cocentric_studio_ds.html)

[119] C. Szyperski, *Component Softare, Beyond Object-Oriented Programming*, Addison-Wesley, 1999.

[120] C. Szypersky, “Point, Counterpoint,” *Software Development*, Feb. 2000.

[121] C. Szypersky, “Components and Contracts,” *Software Development*, May 2000.

[122] C. Szyperski and C. Pfister, “Workshop on Component-Oriented Programming, Summary,” In M. Muhlhauser (ed.), *Special Issues in Object-Oriented Programming - European Conference on Object-Oriented Programming (ECOOP96) Workshop Reader*, dpunkt Verlag, Heidelber, 1997.

[123] Carolyn L. Talcott, “Composable Semantics Models for Actor Theories,” *Higher-Order and Symbolic Computation*, 11(3), pp. 281-343, Kluwer Academic Publishers, 1998.

[124] J. D. Ullman, *Elements of ML Programming*, Prentice Hall, 1998.

[125] J. M. Wing and J. Ockerbloom, “Respectful Type Converters,” *IEEE Transactions on Software Engineering*, Nov. 1998.

[126] G. Winskel, *The Formal Semantics of Programming Languages*, The MIT Press, 1993.

[127] Hongwei Xi and Frank Pfenning, “Eliminating Array Bound Checking Through Dependent Types,” *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI ‘98)*, pp. 249-257, Montreal, June, 1998.

[128] Y. Xiong and E. A. Lee, “An Extensible Type System for Component-Based Design,” *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, March/April 2000. Lecture Notes in Computer Science (LNCS) 1785, Springer.