

State Information in Statically Checked Interfaces

Franz Puntigam

Technische Universität Wien, Institut für Computersprachen

Argentinierstraße 8, A-1040 Vienna, Austria

E-mail: franz@complang.tuwien.ac.at

Abstract

Reliable contracts are absolutely necessary in systems containing substitutable and especially hot-swappable software components. In the ideal case, automatic tools like type checkers ensure that all parties of a contract actually behave as formally stated in the contract. However, current static type systems ensure consistency only for quite simple information about available services and cannot deal with more advanced contract specifications. This work aims at improving the expressiveness of type systems for contract information while ensuring type consistency statically and separately for each component. The proposed concepts allow the availability of services to change dynamically.

1 Introduction

In the object-oriented paradigm, types or interfaces specify contracts between objects or software components and their users [6]. An interface usually consists of the component's signature and a name as informal description of the behavior. In languages like Eiffel, programmers express expectations on input and result parameters in preconditions and postconditions. Users can safely invoke routines whenever the preconditions are satisfied; then they shall get the promised results.

Quite often a routine can only be invoked under some conditions depending on the component's state. For example, “iconify” (that replaces a window on a screen with an icon) can be invoked only while the window is displayed. It is easy to specify this constraint in a precondition. In general, however, preconditions cannot be statically checked. Furthermore, at the presence of aliasing or concurrency, routines may be executed in contexts different from those expected by clients even if all preconditions are satisfied. This may lead to unintended program behavior.

Process types [7, 8, 9] solve this problem. They specify constraints on acceptable routine invocations. For example, a process type specifies that “iconify” and “uniconify” shall be invoked only in alternation. A compiler statically ensures proper routine invocations satisfying the constraints.

So far process types are mainly based on active objects in concurrent systems. They have not yet been explored sufficiently as tools to specify expressive interfaces between software components written in more conventional programming languages like C++ and Java although some properties of process types would be ideal for such uses. Especially

the possibility to express dynamic changes of availability in statically checkable types and the support of conventional techniques like separate compilation, genericity, and inheritance are worth to be mentioned. However, simple concepts like member variables in conventional object-oriented languages have not yet been considered. First experience with variables under a process type regime show that the types are often too restrictive in practice. In the present paper we propose conditional process types as a solution of this problem. We also propose some techniques for specifying interfaces to further increase the dynamic adaptability of components. Moreover, we present a readable representation of process types appropriate for interface specification.

All examples used in this text are written in a new programming language currently under development by the author. This language supports process types. Essential parts of the static type checker are already implemented. Although there is not enough space to explain language details, the examples should be clear enough to readers familiar with languages like Eiffel and Java.

2 Interface Specification

The following interface specification represents forks as needed in a Dining Philosophers example usually used to demonstrate synchronization in concurrent systems:

```
type Fork is
  type var up, down;
  method take when down->up;
  method giveBack when up->down;
  method newFork: Fork[down];
```

Each fork is either down on a table or up in a philosopher's hand. The type variables `up` and `down` represent these two states, and invocations of `take` and `giveBack` switch between them. New forks returned by invocations of `newFork` are in the state `down`. Type variables (declared by `type var`) are variables holding non-negative numbers and belonging to interfaces. These variables can be modified only by `when`-clauses that specify conditions under which methods and variables are accessible. The expressions `down` and `up` in `when`-clauses are shortcuts of `down[1]` and `up[1]`, respectively, this is the type variables have a value of at least one. Invocations of `take` require the value of `down` to be at least one, decrement the value by one, and increment the value of `up` by one. Invocations of `giveBack` require the value of `up` to be at least one, decrement the value by one, and increment the value of `down` by one.

Invocations of `take` shall occur only while the fork is down, and invocations of `giveBack` only while the fork is up. Since compilers shall be able to ensure that all invocations occur in expected states, `take` can be invoked only in instances of `Fork[down]` and `giveBack` only in instances of `Fork[up]`, these are forks with type variables `down` and `up`, respectively, known to have a value of at least one. The caller has to know that the fork is up or down. These types swap with each invocation. To be precise, if `x` is an instance of `Fork[down]`, then an application of `x.take` first changes the type of `x` to `Fork` (without any type variable known to be different from zero), and on return changes the type of `x` to `Fork[up]`.

In each fork either `down` or `up` has a value of one and the other a value of zero. Values of these type variables different from zero represent a limited resource. In the whole system there shall be at most one reference to each fork either with a type `Fork[down]` or `Fork[up]`. The methods `take` and `giveBack` can be invoked only through this reference. Any number of references to forks can be of type `Fork` without knowledge about type variable values. However, no methods can be invoked through them. There cannot be several simultaneous invocations in the same fork because each invocation takes away the only value of a type variable different from zero. Another type variable different from zero becomes available only on return. Such uses of type variables as limited resources serve for synchronization.

Values of type variables shall remain limited resources when introducing aliases. For example, when invoking `something(x)` as defined by

```
method something1 (f: Fork[down]);
```

with `x` of type `Fork[down]` as actual parameter, the type of `x` changes first to `Fork` and on return from `something1` again to `Fork[down]`. This is, the values of the type variables specified in the formal parameter's type are subtracted from those in the actual parameter's type and added again on return. Values of type variables can change while executing methods. Using the following declarations,

```
method something2 (f: Fork[down -> up]);
method something3 (f: Fork[down ->]);
```

an invocation of `something2(x)` changes the type `Fork[down]` of `x` first to `Fork` and then to `Fork[up]`, and an invocation of `something3(x)` changes the type just to `Fork`.

Type variables correspond essentially to “tokens” in process types as proposed in earlier work with some syntactical improvements. Extensions given below have not been proposed so far.

Sometimes it is useful to associate several alternative `when`-clauses with methods and especially variables:

```
type Philosopher is
  type var sleeping, thinking, eating, nice;
  method newPhilosopher: Philosopher[sleeping];
  method wakeUp when sleeping->thinking;
  var fork: Fork when thinking, when eating, when nice;
  ...
```

Each instance of `Philosopher` is always either sleeping, thinking, eating, or being nice (this is, allowing another philosopher to use his fork). The variable `fork` shall be accessible when the philosopher is thinking, eating or nice, but not while sleeping. Possibly `fork` does not get initialized before invoking `wakeUp`. Several `when`-clauses represent alternatives. It shall be determined which `when`-clauses to use in invocations if the uses may modify type variables.

The variable `fork` does not specify whether the fork is up or down. A more elaborate declaration allows us to use this state information:

```
var fork: Fork[down when thinking; up when eating]
    when thinking, when eating, when nice;
```

With this declaration, `fork` is known to be down when the philosopher is thinking and up when the philosopher is eating. A nice philosopher has no information about the state of his fork.

Such conditional state information on variables solves a problem with non-local variables: If each user of an instance of `Philosopher` sees `fork` known to be of type `Fork[down]`, then several users could invoke `fork.take` simultaneously. That shall not be allowed. Because of the condition, only users of instances of `Philosopher[thinking]` etc. can access `fork`. Unfortunately, they cannot simply invoke `fork.take` because then the type of `fork` would become `Fork[up]` although it should be `Fork[down]` in all instances of `Philosopher[thinking]`. However, within closed environments it is possible to invoke `fork.take` and then `fork.giveBack` as long as between these two calls there is no access to the instance of `Philosopher[thinking]` requiring the value of `thinking` to be at least one (except for accessing `fork` for the invocation of `giveBack`). Between the invocations the known value of `thinking` is assumed to be exclusively usable by the client who accessed `fork`. The exclusive use is required until the value of `down` of `fork` becomes at least one again. While the value of a type variable is exclusively used, this type variable value shall not be used in parameter passing or method invocations or accesses to other variables. This is, the value of `thinking` is assumed to be decremented by one until the value of `down` becomes at least one again, except for accesses of the variable `fork`. While the value of `thinking` is exclusively used, the type of `fork` depends on the previous uses of `fork`. After invoking `fork.take`, the type of `fork` is `Fork[up]` so that an invocation of `fork.giveBack` is possible.

Within a method of the philosopher it is possible to invoke just `fork.take` if the `when`-clause of the method causes the value of `thinking` to be decremented and the value of `eating` to be incremented (for example, `... when thinking->eating`).

Interface specifications with type variables can be used to determine a large class of communication protocols. As interfaces of software components the protocols specify all communication of systems with their components to be supported by the components. Such interfaces contain enough information for a compiler to statically ensure that (1) a component can deal with each method invocation and variable access and (2) clients access only methods and variables as specified by the protocol. Each component can be checked separately. Furthermore, such interfaces support inheritance as shown by the next (rather retrieved) example:

```
type Fork2: embed Fork is
    method newFork: Fork2[down[2]];
```

The method `newFork` is overridden to return a fork with `down` having a value of at least two. Hence, `take` can be invoked twice, then `giveBack` twice, and so on. As required by this kind of inheritance, all invocations specified by `Fork[down]` are also supported by `Fork[down[2]]`, this is `take` and `giveBack` in alternation. Components implementing the interface `Fork2` can be used where a component of instance `Fork` was expected.

3 Dynamic Changes of Components and Behavior

Type variables as explained above are quite useful to specify simple communication protocols in interfaces. However, we need more. Especially we want to be able to dynamically replace one component with another. For example, we want to replace the `fork` component of a philosopher with another component, let us say an instance of `Fork2`. At a first glance this seems easily to be done by assigning a new instance of `Fork2` to `fork`. As long as the philosopher is sleeping this is actually just a simple assignment. Unfortunately, the situation becomes much more complicated after an execution of `wakeUp`: We have to ensure that the assigned new fork is in the expected state although there need not be any single point in a program where complete information about the fork's expected state is available.

To be precise, it is not necessary that a replacing component is in exactly the same state as a replaced component. Only the protocol specified in the interface shall be in the same state or in a state satisfying the substitution principle. Using type variables, the substitution principle is satisfied if the interfaces of the components are related by subtyping (possibly based on inheritance) and all type variables of the replacing interface have at least the same values as corresponding type variables of the replaced interface. Statically checking this condition is much simpler than statically ensuring that component states match. The only issue remaining to be discussed is how to find sufficient information about current values of type variables in a running system since this information can be distributed over the whole system.

This issue can be formulated more directly: Under which conditions can we assign new values to non-local variables? The rule is simple: Assignments can occur if there is enough knowledge of type variables as required in `when`-clauses. In our example, `sleeping`, `thinking`, `eating`, and `nice` are exclusive, this is, at most one of these type variables can have a value of one and the other are zero. Compilers can always determine exclusivity. In this case it is sufficient to know that `thinking` has a value of at least one to be able to assign another instance of type `Fork[down]` to `fork`. If `eating` is known to be one, another instance of type `Fork[up]` can be assigned to `fork`, and if `nice` or `sleeping` are known to be one, any instance of `Fork` can be assigned to `fork`. Only if neither of these type variables is known to be at least one, no value can be assigned to `fork`.

Let us assume that `thinking` and `eating` are not exclusive, this is, both `thinking` and `eating` can have a value of at least one at the same time. The assignments to `fork` require both `thinking` and `eating` to be known as different from zero, and assigned values have to be instances of `Fork[down,up]` where both `down` and `up` are known to be at least one. Since `down` and `up` are exclusive in `Fork`, only instances of `Fork2[down,up]` are appropriate in this case.

Assignments to variables can occur even if a variable is exclusively used by a client. For example, if a client invokes `fork.take` in an instance of `Philosopher[thinking]`, this client can then assign an instance of `Fork[down]` to `fork`. This will cause the exclusive use to come to an end. The client can also first invoke `fork.take`, then assign an instance of `Fork[up]` to `fork`, and finally invoke `fork.giveBack`.

After assigning a new value to a variable, each read access of the variable will return the new value. However, there can still exist other references to the value replaced by

the new value in the system. For example, a `nice` philosopher lets another philosopher use his fork. The other philosopher probably has also a reference to this fork. An assignment to `fork` replaces just one of the references. The other philosopher can still invoke `take` and `giveBack` in the replaced fork (while the `nice` philosopher cannot do so). When using type variables, the compiler can ensure that all details of protocols specified in interfaces will be observed. The compiler does not guarantee that a single assignment replaces all references. There may be several components in parallel, each fulfilling a different part of the protocol at the server side. All references to a component have to be replaced before a component can be regarded as replaced. Everything else can lead to errors. For example, if there are two references to a component functioning as a buffer where elements are written to the buffer through one reference and read through the other reference, then replacing just one reference will cause that nothing written to a buffer can be read back. Hence, it is a good idea in systems design to have essentially only one reference to each hot-swappable component. It is probably not feasible to completely enforce such design rule. Usually it will be possible to allow several simultaneous references to components (or parts of components) with the restriction that all but one of these references become obsolete after a short time period (after communicating with the component according to a finite part of the protocol). In each case it is necessary to be aware of possible lost information like the contents of the buffer in the replaced component. There may be a need to copy the contents of the old component to the new component before the last reference is lost, or to require that replacements occur only in safe states. These considerations require knowledge about the semantics of components far beyond what can be usefully expressed in an interface protocol. We do not deal with this difficult topic in the present paper.

In the rest of this section we briefly describe how components can spontaneously change behavior in a way that is not predefined in the interface. For example, a philosopher can dynamically decide to change its behavior from `thinking` to `eating` based on criteria not visible to users. A precondition for doing so is that the philosopher knows that `thinking` has a value different from one. Here is an example:

```
method decide: (p:?Philosopher[thinking]; q:?Philosopher[eating])
    when thinking-> do
        if ..., ->eating then do return (null, this);
        else if ..., ->thinking then do return (this, null);
        else do return (null, null);
end decide;
```

An invocation of `decide` first decrements the value of `thinking`. This method returns two values. Question marks in front of the return types specify that `null` can be returned instead of instances of the types. In fact at most one of the result values will be different from `null`. Depending on some hidden conditions either a reference to an instance of `Philosopher[thinking]` or one to an instance of `Philosopher[eating]` or nothing is returned. The expressions `->eating` and `->thinking` in the conditions of `if`-statements increment the specified type variables associated with the type of `this` by one. Therefore, the returned values `this` are of the appropriate type. In earlier work this kind of adding values to type variables of self references was called “type renewal”.

In general, values of type variables belonging to self references can arbitrarily be modified without violating type consistency. This property is the only source of spontaneous state change as far as states are specified in interfaces. Thus spontaneous state changes can occur only within components. Users can request possible state changes only by invoking methods of components that return new references.

Methods performing spontaneous state changes can delete all available values of type variables different from zero and return nothing. Such methods cause all services of a component depending on type variables to become unavailable, this is, they withdraw components. It is usually not very difficult to model interfaces so that a specific type variable (or a small set thereof) has a value different from zero as long as the component is available and becomes zero on withdrawal. All method invocations of components depend on this type variable. This design rule ensures that a single method invocation destroys all useful references to a component. However, there is a drawback: Since this type variable is a limited resource, all users of a component have to synchronize themselves to get access to the limited resource before they can access the component.

4 Related and Future Work

Quite well known is the work of Kobayashi, Pierce and Turner on linearity [5] which allows us to statically ensure that all sent messages are acceptable, although the acceptability changes. However, this concept is quite restrictive. Process types [7, 8, 9] consider constraints on the ordering of messages, support subtyping and ensure that only acceptable messages are sent. The present article improves previous work mainly by considering aspects relevant to hot-swapping of components.

Much work was done on type concepts that consider object states [11, 3, 1, 2, 4, 10, 12]. In principle most of this work can be regarded as a basis for interface specifications appropriate for components. However, a large amount of this work would imply an extensive analysis of components together with the environment where they are supposed to be used. It is hard to imagine how such approaches could be used in dynamically configurable systems.

Much work remains to be done. The concepts proposed in this paper need support from the programming language. So far no practically usable programming language supports these or similar concepts. An appropriate language is currently under development. However, it will take some time before first results can be shown. Organizational aspects of component interfaces based on these techniques that are needed to develop hot-swappable components are a completely open future work.

5 Conclusions

The presented work shows that it is possible to develop component interfaces specifying non-trivial protocols for the communication between components and the rest of a system. With appropriate language support compilers can statically ensure the correspondence between specification and actual communication. Strictly specified interfaces help to develop components exchangeable at run time while static type checking ensures that new components continue to communicate with the rest of the system in

the same way as withdrawn components did before (as far as interfaces determine the communication).

References

- [1] S. Abramsky, S. Gay, and R. Nagarajan. A type-theoretic approach to detect deadlock-freedom of asynchronous systems. In *Theoretical Aspects of Computer Software*, number 1281 in Lecture Notes in Computer Science, pages 295–320. Springer-Verlag, 1997.
- [2] F. Flanagan and M. Abadi. Types for safe locking. In *Proceedings ESOP'99*, Amsterdam, The Netherlands, March 1999.
- [3] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings HLCL'98*, Nice, France, September 1998. Elsevier ENTCS.
- [4] Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):434–482, March 1998.
- [5] Naoki Kobayashi, Benjamin Pierce, and David Turner. Linearity and the pi-calculus. In *Proceedings POPL'96*, New York, 1996. ACM Press.
- [6] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice-Hall, Englewood Cliffs, NJ, 1994.
- [7] Franz Puntigam. Types for active objects based on trace semantics. In Elie Najm and Jean-Bernard Stefani, editors, *Proceedings FMOODS'96*, pages 4–19, Paris, France, March 1996. Chapman & Hall.
- [8] Franz Puntigam. Coordination requirements expressed in types for active objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, number 1241 in Lecture Notes in Computer Science, pages 367–388, Jyväskylä, Finland, June 1997. Springer-Verlag.
- [9] Franz Puntigam. *Concurrent Object-Oriented Programming with Process Types*. Der Andere Verlag, Osnabrück, Germany, 2000.
- [10] P. Quaglia and D. Walker. On encoding $p\pi$ in $m\pi$. Technical Report BRICS Report RS-98-26, Aarhus University, Denmark, 1998.
- [11] Vasco T. Vasconcelos. Processes, functions, datatypes. *Theory and Practice of Object Systems*, 5(2):97–110, 1999.
- [12] Nobuko Yoshida. Graph types for monadic mobile processes. In *Foundations of Software Technology and Theoretical Computer Science*, number 1180 in Lecture Notes in Computer Science, pages 371–386, Hyderabad, India, 1996. Springer-Verlag.