

Negotiable Interfaces for Components

Simon D. Kent, Chris Ho–Stuart, Paul Roe,
Centre for Information Technology Innovation,
Queensland University of Technology, Australia

Component specifications are vital for communicating a component's requirements, as components are subject to third-party composition. Most modern programming languages lack sufficient features to express the full requirements of a component, however, much less enforce them. Pre- and post-conditions can capture functional aspects of a component's requirements, but are unable to express many temporal constraints such as re-entrance restrictions or changing availability of services over the lifetime of a component instance (object).

The approach described within forms a basis for extending the specification of components at the programming language level, thus making such specifications enforceable. Interfaces are extended with a factorable, abstract state, and methods of interfaces are extended with state transformation behaviours. A new programming language command, the **USE** statement, allows clients to negotiate for those services provided by an object. A mixture of static and dynamic checking ensures the consistency of an object's state according to the specification of the object's interfaces.

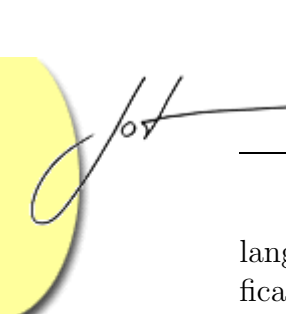
The mechanism proposed allows a clearer expression of re-entrance conditions and dynamic service availability, and a greater level of checking that allows erroneous cases to be prevented or detected during development time. The mechanism also acts as a self-documenting feature for interfaces.

1 INTRODUCTION

The move towards software components represents an attempt to bring software engineering in line with more mature engineering disciplines, in which tried and tested parts are reused to speed production and improve reliability. Szyperski [18] gives three characteristic properties of software components:

1. A component is a unit of independent deployment.
2. A component is a unit of third-party composition.
3. A component has no persistent state.

In particular, property two implies that components must come with a clear specification as to how the component may (and must) be used. Many programming



languages lack support for expressing such specifications, however, leading to specifications being expressed informally in accompanying documentation. Programming languages such as Eiffel [7] and Sather [17] have support for programming by contract, but pre- and post-conditions can only capture strictly functional properties and cannot express temporal constraints and behaviours such as the presence or absence of outcalls, and re-entrance conditions. The situation is complicated further by the inherently concurrent nature of component software (take for example, Windows programs). Concurrency and re-entrance are a major source of bugs, some of which only appear when systems are stressed.

This paper presents the integration of a set of programming language features called *negotiable interfaces* into a Component Pascal like language. Negotiable interfaces are a variation on state abstractions and transitions, and are targetted at specification and checking of temporal behaviours of objects such as re-entrance. Language level specification is desirable as it promotes self-documentation of code, and enables formal checking (static and/or dynamic) to improve reliability and aid debugging.

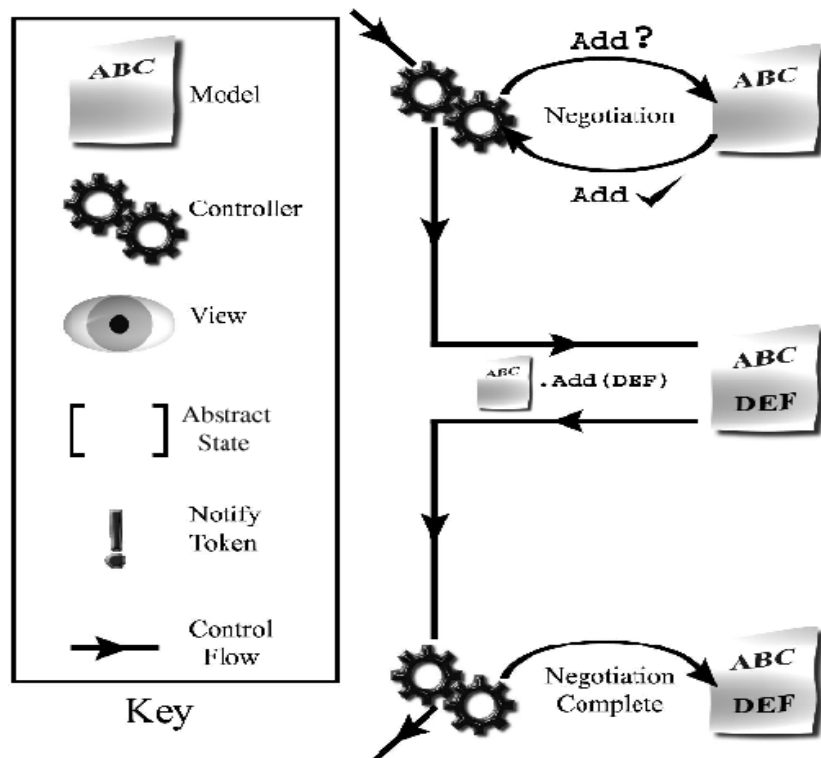
Negotiable interfaces are ordinary interfaces, with behavioural information added to effectively extend the type of a component. Negotiable interfaces provide a mechanism for both expressing the availability of methods as a result of the object's past and current actions, and for providing guarantees about future method invocations by clients. Negotiable interfaces also serve as the groundwork for expressing more complex inter-object behaviours such as method availability as the result of the interaction of a neighbourhood of objects. As mentioned above, we have implemented negotiable interfaces in a variant of Component Pascal, although any language that supports interfaces, such as Java or C#, would be suitable.

In the next section we introduce the concepts behind negotiable interfaces in broad terms, Section 3 goes into language specifics, Section 4 discusses related work, Section 5 gives future directions, and Section 6 presents our conclusions.

2 NEGOTIABLE INTERFACES

In this section we delve into the ideas behind negotiable interfaces. Although negotiable interfaces rely on a mixture of static and dynamic checking, we defer discussion of static checking until the end of this section as the dynamic aspects provide a more intuitive introduction.

As the name implies, the concept of a negotiation is central to our proposal. In a negotiation, two parties negotiate over a subject and the outcome of the negotiation may depend on preceding or pending negotiations. If negotiation fails, the status quo is unchanged but the parties are aware that the negotiation failed. If negotiation succeeds, the parties are aware of the success and must meet their obligations within the scope of the negotiation.

Figure 1: Negotiation for method `Add`

In our proposal, the parties are a client and an object, and the subject is the ability to invoke some constrained sequence of services. If the negotiation fails, the client must continue without using those services, but if it succeeds then the client gains the ability (and in some cases, obligation) to invoke those services, while the object is obliged to handle them.

We demonstrate this process through a simple Model-View-Controller example. The examples given are motivated by those given by Szyperski in [18] on pages 57–66, in which he describes how object re-entrance can break pre- and post-conditions. In particular, we focus on how our system can express and enforce re-entrance constraints, thus capturing some of the temporal properties that pre- and post-conditions cannot.

In our example, we have a Controller, a Model, and several Views on the Model. We note that any time a client wishes to Add data to our Model, the Model will need to notify its Views. Notification requires a callout to each of the Views on the Model and, in this implementation, is non re-entrant. We enforce this condition by requiring that clients negotiate for the ability Add data to the Model, in order to prevent re-entrance to the notification code.

Figure 1 demonstrates a simple negotiation. A client, the Controller, requests the ability to Add some data to the Model object. The Model object agrees and the Controller Adds some data to the Model during the scope of the negotiation.

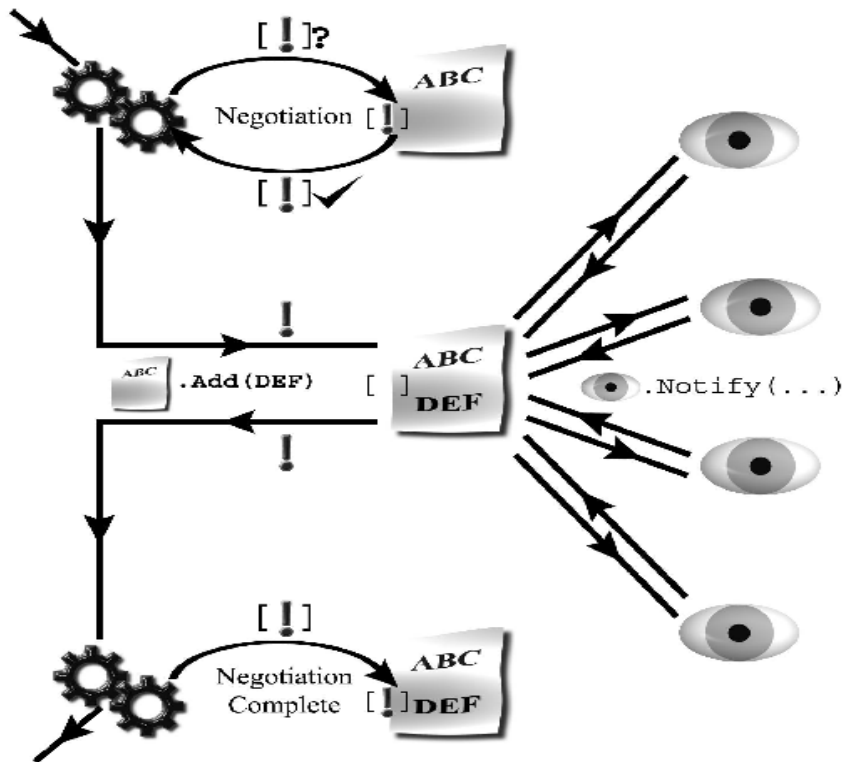


Figure 2: Negotiation with tokens

More concretely, we represent the “ability” to invoke some set of methods as a bag of keys or *tokens*. In order to invoke a certain method, the client must hold the correct bag of tokens, which is passed to the object and transformed during the method call. On completion of the method call, the transformed bag of tokens is returned to the client. At any particular time, an object holds a bag of tokens that represent those services of the object that are available for use - the bag of tokens represents the object’s temporal state in an abstract and factorable manner.

Figure 2 demonstrates the same negotiation, with tokens made explicit. A client that wishes to call an operation requiring notification must hold a bag containing a Notify token; a client obtains this token by negotiating with the Model.

The negotiation process starts with a request by the Controller for a bag of tokens, in this case, a bag containing single Notify token. We assume that the Model is not currently notifying its Views and hence has a Notify token available. The Model therefore grants the Controller’s request. The Controller begins its operations, possibly performing other tasks now that it is sure of being able to Add to the Model, and eventually uses the Notify token to Add the required data. The Model consumes the Notify token, adds the data, notifies its views, and then returns the new tokens produced as a result of the Add operation - again a Notify token since the Add operation has finished, and the Model will be in a state in which it can perform notifications. Control returns to the Controller and at the end of the negotiation

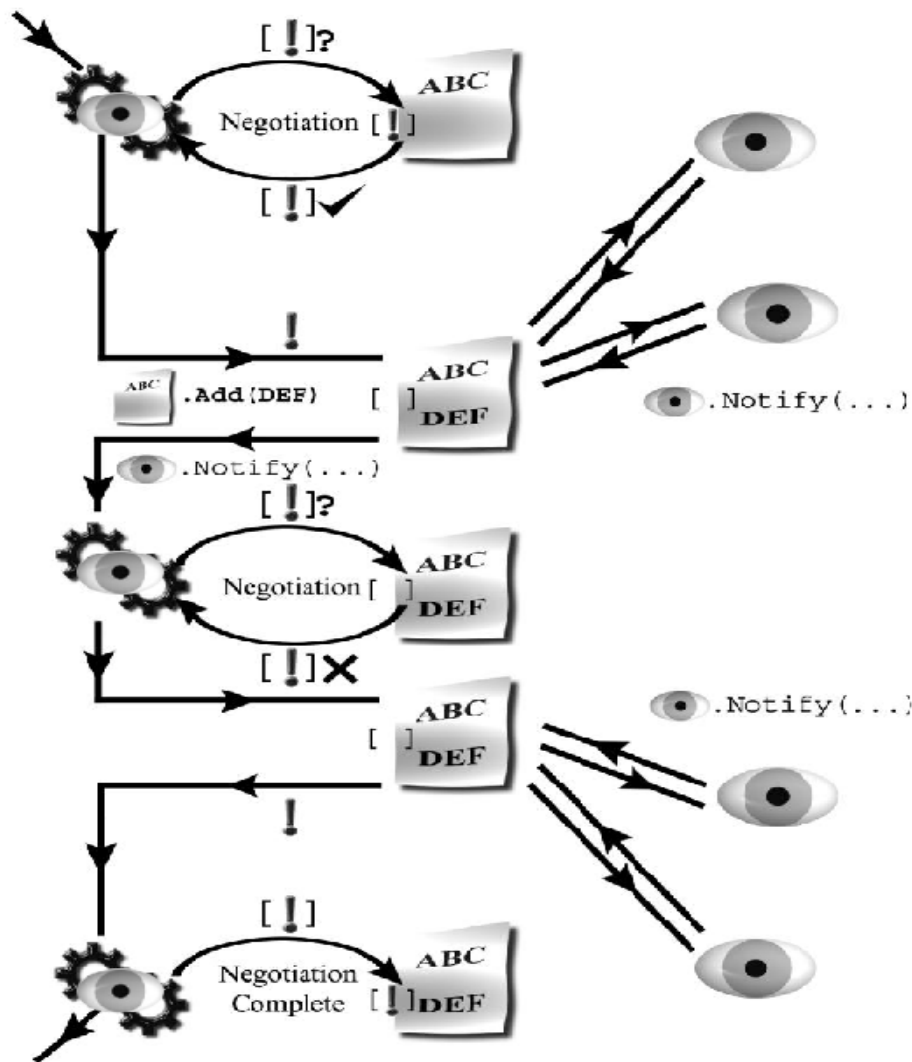


Figure 3: Re-entrance detection

the Controller returns the left over tokens to the Model.

Figure 3 demonstrates re-entrance detection. Here we have a Controller which is also a View upon the document. The example runs similarly to that in Figure 2 up until the Model notifies its Views, at which point the Controller/View is notified. During the notification, the Controller/View wishes to Add or Delete data from the Model (in order to flush a message queue say), but first must negotiate for a Notify token. At this point, the Model is still in the middle of an Add operation which the Controller invoked by using the Model's Notify token. Hence, no Notify tokens are available and re-entrance is detected. The negotiation fails and the Controller/View must defer flushing its buffers until later. Control returns to the Model, which notifies the rest of its Views without interruption.

Concurrency

Thus far, we have only examined simple examples and not yet fully explained when a negotiation fails or succeeds. In the sequential case, the semantics of negotiation is trivial, as the presence or absence of requested tokens will determine success or failure. As mentioned earlier, however, we aim to handle concurrency as well, and although the simple presence or absence of tokens could be used to determine the result of a negotiation, another option is available: a negotiation can block on the result of pending negotiations.

This is possible as negotiations consist not only of a request for tokens, but a promise to return certain tokens. The future state of an object can be determined therefore, and a client may wait if the object may be in a suitable state in the future.

The negotiation mechanism can prevent deadlock due to circular blocking patterns as well. By recording the causality (logical thread identification) of negotiations, we can ensure that nested negotiations never directly or indirectly block on the result of outer negotiations: ensuring that causalities never block on themselves. Negotiations that would lead to deadlock if blocked can therefore be made to fail allowing a programmer to either rollback changes or throw an error and use the negotiation chains that caused the error (potentially available from the runtime system) to isolate the problem.

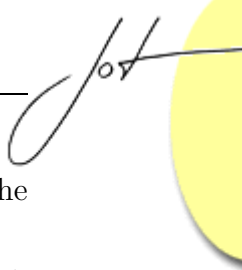
It is worth noting that in the concurrent case, detection of re-entrance reduces to the special case of deadlock detection in which a nested negotiation attempts to block on the result of an outer negotiation.

Static Checking

The reader will have noticed a vital omission — how are we sure the client will keep its promises? If a negotiation consists of a request for tokens and a promise to return certain tokens, how are we sure the client will use only the tokens requested, and return those tokens specified in its promise? Since method invocations are the manner by which tokens are consumed and produced, we must ensure that any sequences of methods that are called:

1. Are allowable according to the tokens the client is holding at the time.
2. Produce the correct bag of tokens when the negotiation finishes.

Our proposal does this via static checking (the type system). By tracking the negotiable state of an object over the scope of a negotiation, we can statically detect if any illegal method invocations are made, and if the client will hold the promised set of tokens at the end of the negotiation. Our approach associates a “negotiable” type with any references that are the subject of negotiation and maintains a set of possible states that the subject may be in. A method can only be called if it is legal



in all the possible states of the subject, and a negotiation is only well-typed if the final states of the negotiation agree with the promise made by the client.

Such a system grants other benefits as well. When a negotiation has succeeded, the client may pass obligations to perform certain token transformations/method calls to other objects via negotiable typed parameters. The bodies of methods with such parameters effectively become part of the negotiation and can be checked in a similar fashion to the scope of negotiations. As such, methods can promise to fulfill part of a client's obligation.

3 LANGUAGE SPECIFICS

Component Pascal

This work has been implemented in an experimental version of gpcp [4], a Component Pascal [10] compiler that generates Java bytecode or .NET intermediate language. We describe negotiable interfaces in a Component Pascal like language. Interface types are declared using the `INTERFACE` record attribute:

TYPE

```
Interface* = POINTER TO INTERFACE RECORD
            END;
```

A record type may inherit from one extensible base class and implement multiple interfaces. A record is declared to implement interfaces as follows:

```
(* Declaration of extensible class *)
```

```
Base* = POINTER TO EXTENSIBLE RECORD
      ... (* attributes *)
      END;
```

```
(* Declaration of sealed class *
 * inheriting from Base          *
 * and implementing Interface. *)
```

```
Sub* = POINTER TO RECORD (Base +
                        Interface)
      ... (* attributes *)
      END;
```

Methods of interfaces are declared similarly to methods of normal classes (but must be abstract):

```
PROCEDURE (this:Interface) Foo*(),  
          NEW, ABSTRACT;
```

`Sub` would implement method `Foo` through a method with a signature matching that of `Foo`:

```
PROCEDURE (this:Sub) Foo*(), NEW;  
BEGIN  
... (* implementation *)  
END Foo;
```

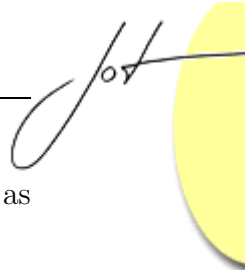
As an alternative, `Sub` may “explicitly” implement method `Foo` by using the following syntax:

```
PROCEDURE (this:Sub.Interface) Foo*(),  
          NEW;  
BEGIN  
... (* implementation *)  
END Foo;
```

This concept of “explicit” method implementation is the same as that used in C# [8]. Explicit implementation removes the method from the namespace of the class, thus preventing the method from being called except through an interface reference. This allows for the implementation of several versions of the same interface in one class.

Negotiable interface types

Our approach uses bags of tokens to represent the abstract state of an object. Such an approach allows the state to be factored between clients (between references in fact). The type of an interface includes a declaration of the tokens that may represent the abstract state of an object over its lifetime, accompanied by a declaration of the initial bag of tokens. We demonstrate the syntax giving simple examples from our Model-View-Controller scenario.



A declaration for a hypothetical Model interface from the example above is as follows:

```
TYPE
  IModel* = POINTER TO INTERFACE RECORD
    TOKEN Initial,Notify;
    INIT [Initial];
  END;
```

Following the **TOKEN** keyword are the potential tokens that may represent the abstract state of a Model object and following the **INIT** keyword is a bag of tokens representing the initial state of a Model object when it is created.

Negotiable method types

Methods of each interface are given a behavioural type consisting of a pre-state and a post-state. The pre-state and post-state are again bags of tokens representing the abstract state required by the method before execution, and the abstract state returned after the method has executed respectively. These abstract method behaviours serve a similar purpose to pre- and post-conditions, but with respect to availability of the interface's methods over time; something that pre- and post-conditions generally have difficulty expressing. Thus for initialisation and addition methods we have:

```
PROC (this:IModel) Initialise(...) ::
  [Initial]->[Notify];
```

```
PROC (this:IModel) Add(...) ::
  [Notify]->[Notify];
```

(Note that for brevity we have omitted **NEW** and **ABSTRACT** keywords and used **PROC** in place of **PROCEDURE**.) So for **IModel.Add**, the pre-state is **[Notify]** and the post-state is **[Notify]**; when invoking **IModel.Add** a client must first be certain that the Model object referred to is able to notify its views (a client achieves this through negotiation using the **USE** statement - see Section 3) and after the method has completed the object will once again be able to notify. Behavioural types for methods of interfaces are optional, but the type system requires that any classes implementing the interface “explicitly” implement methods that have a behavioural type.

Negotiation

In order to allow clients to negotiate with objects, we introduce a new programming language statement called the **USE** statement. A **USE** statement can be thought of as a type guard to determine an object's behavioural type (abstract state). In a similar fashion to the Component Pascal type case statement, the **USE** statement performs a test on an object rather than simply acting as an assertion.

Returning to our simple Model-View-Controller example we have the Controller creating a new document and adding data from a template:

```
PROC (c:Controller) CreateTemplate
    (i:IModel);
BEGIN
    ...
    USE i :: [Initial] -> [Notify] DO
        (* USE command *)
        i.Initialise(...);
        i.Add(...);
    ELSE
        (* ELSE command *)
    END;
    ...
END CreateTemplate;
```

In the above example, *i* is the subject of the negotiation, **[Initial]** is the requested state, and **[Notify]** is the state that the client promises to return once the negotiation is finished. When executed, the abstract state of the object that *i* refers to is dynamically and atomically inspected. If the object has the requested state, **[Initial]** in this case, the negotiation succeeds, the client is granted requested state, and the **USE** command may be executed; otherwise, the state remains unchanged and the **ELSE** command is executed.

Restrictions and Considerations

Currently we restrict the subject of **USE** statements to interface reference identifiers only. That is, one may only negotiate with an object through an interface reference. This helps simplify the syntax of the **USE** statement so clients do not have to state which interface of the object they wish to negotiate with. This is also the reason for enforcing explicit implementation of behaviourally typed methods.

Identifiers that are the subject of negotiations are also necessarily read-only for the duration of the negotiation. Clearly this must be the case in order to ensure the object referred to by the identifier is in a state consistent with the tokens held locally.



Negotiable type system and re-USE

The request and promise of a negotiation provides the basis for our type system to check that behaviourally typed methods are used safely. Our type system statically checks that behaviourally typed methods are only called when the object is in the correct state; i.e. within the scope of a negotiation. As with most type systems, our type system is conservative and will reject some correct programs.

During the scope of a negotiation, the type system maintains a set of possible states (bags of tokens) for subject of the negotiation. Modeling the negotiable type of an identifier by a set of states allows modeling the non-determinacy introduced by control-flow statements and makes the type system less restrictive. For example, if `i` holds an `IModel` reference in an `{[Initial]}` state then the following code will leave the reference in an `{[Initial], [Notify]}` state.

```
IF bool THEN
  i.Initialise(...);
END;
```

In order to reconcile this non-determinacy, we allow for nested `USE` statements within the scope of the negotiation to query the local state in order to disambiguate non-determinate states introduced by control flow statements.

On typing loop statements, we adopt a conservative approach and require that commands enclosed by a loop have a constant negotiable type: that is, the set of states passed into the loop be the same as the set of states returned by the loop. Other more flexible approaches (based on domains) are under consideration.

Finally, our type system includes support for typing parameters as stated in Section 2. Parameters with negotiable types transform the body of a method into part of a negotiation. For example, it would be more useful to ensure the `Model` passed to the `CreateTemplate` method above is able to receive template data before `CreateTemplate` is invoked. The method signature below would require the client to do exactly that.

```
PROC CreateTemplate(i:IModel::
  [Initial]->[Notify]);
```

4 RELATED WORK

The concept of associating an abstract state to objects in order to describe changing method availability is not a new one. In particular, many concurrent object-oriented

languages, and type systems for active objects use a concept of object state to help determine method availability. More recently, this concept has also been applied in a similar fashion to interfaces for component programming.

Many concurrent O-O languages that use abstract state to determine method availability have “active” objects such as those in Actor languages [1]. Each object has its own thread of execution, sends messages to other objects, and has its own message queue. Different schemes for using abstract state for concurrency control, such as path expressions in PROCOL [19] and BCOOPL [3], behaviour sets in ACT++ [6] and guards in Guide [2], result in differing levels of expressiveness for determining method availability. Most are generally not factorable, however. As these languages deal mainly with active objects the focus is placed upon the ability of objects to handle specific orderings of messages. Specification and prevention of re-entrance is not a major concern of such languages so expressing intermediate state during method invocation is given little focus.

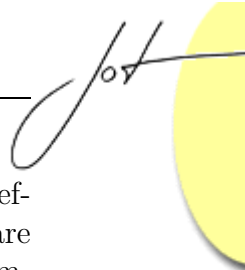
The focus on active objects also means that such languages are usually only concerned with guarding a single method at a time as opposed to blocks of code presented in our proposal. The authors are also unaware of any concurrent O-O language that incorporates a type system that incorporates behavioural obligations and promises in a similar fashion to negotiable interfaces.

Type systems for active objects such as in [9] [20] [11], [12], and [13], annotate the methods of interfaces or classes using some process formalism to describe the changing availability of services during execution. As the focus of such type systems is to type behaviour patterns in languages with active objects, describing re-entrance and intermediate state are again not a primary focus.

The type systems of [9] and [20] utilise behaviour descriptions of both client and server objects to examine the combined behaviour of an object and its clients at composition time. Such approaches are not suitable for handling dynamically changing sets of clients or situations in which aliasing occurs as the identity of objects needs to be known statically. Our approach uses dynamic checking in the form of negotiation to handle such behaviour, while statically checking that clients attempt to use services correctly i.e. within the scope of negotiations.

Furthermore, although composition level checking is desirable, it does not help during the construction process. Our type system enables static checking during construction of a component thus preventing certain errors before composition (which may occur after the component has been deployed). The nature of our type system also allows location of specific points at which a component’s behavioural specification has been violated, whereas such information may be difficult to extract from a composition level type system using a process formalism to check the joint behaviour of several components.

The type system presented and developed primarily by Puntigam in [11, 12] and Puntigam and Peter in [13] has a similar scheme for decorating interfaces with (extended) bags of tokens and assigning behaviours to methods. Our proposal was



directly influenced by this work as bags are factorable and may be split over references. The object model and objectives of the type system proposed here are different from that of Puntigam and Peter. They consider active objects that communicate via asynchronous message passing with unbounded buffers - once again re-entrance is not a concern.

Puntigam and Peter seek to ensure that the sequence of messages received by an object conforms to the (dynamic) type of the object and achieve this by typing individual references to an object. Whenever a reference is passed to another client, the dynamic type of the reference is split between the old and new references. This allows the clients to access services of a server object without querying the object as to its current state; however, if the client wishes to access more services than its reference allows, it must somehow obtain a reference from another client or the server to access those services. This may actually lead to clients becoming coupled, which is an unnatural behaviour for component systems.

Although Puntigam states that his proposal may be used for component interfaces, we believe our proposal is more natural as we aim at a programming language and object model more natural to contemporary component programming. Our concept of negotiation also has semantic application for synchronous method invocation, and our type system checks that negotiations leave the object in a predetermined state.

Finally, the type system developed by Reussner in [14, 15], which based on earlier work by Reussner and Heuzeroth in [5, 16], extends Java interfaces with augmented finite state automata. Reussner's approach decorates each interface with a call automata that specifies legal orderings of calls, and each method of a component is with an automaton that describes which other services the method uses. Combining the call and method (or function) automata gives the complete behaviour of a component (called the EC-Automaton). The combined automata can be checked with behaviour of other components at composition time to accept or reject legal compositions. Reussner's type system focuses upon adaptation of components and composition level checking, whereas our type system is aimed at component construction and is accompanied with the runtime mechanism of negotiation.

5 FUTURE DIRECTIONS

Our implementation of negotiable interfaces constitutes the groundwork for future research in language level component specification and as such there is still work to be done before negotiable interfaces become a truly usable language feature.

One current focus in expanding our research is increasing the expressiveness of behaviour specification. Part of this entails increasing the flexibility of interface descriptions and behaviours of methods and negotiations. Another avenue of investigation is different types of tokens and their negotiation semantics. Finally, further expressiveness is required to fully capture object-wide and inter-object behaviours.

Flexibility of abstract state

- **Multiple behaviours per method:**

In the model proposed, methods may only have one behaviour but in order to express certain behaviour protocols (such as finite state automata), multiple behaviours per method are required.

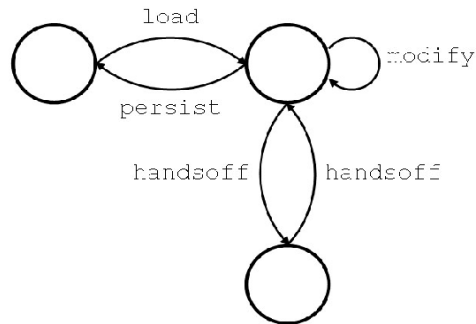


Figure 4: `IPersist` interface

Figure 4 demonstrates the desired behaviour of an interface, `IPersist`, that provides persistence functionality. Of particular interest is the `handsoff` method, which prevents all other methods from being called until a second `handsoff` method is called. This cannot be expressed in a single behaviour per method scheme. We might represent `IPersist` as follows:

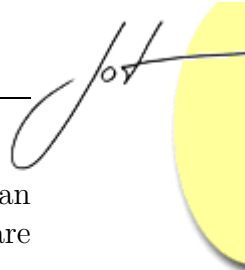
```

IPersist = POINTER TO INTERFACE RECORD
    TOKEN Stored,Loaded,Handsoff;
    INIT [Stored];
    END;

PROC (i:IPersist) Load(...) ::
    [Stored]->[Loaded];
PROC (i:IPersist) Persist(...) ::
    [Loaded]->[Stored];
PROC (i:IPersist) Modify(...) ::
    [Loaded]->[Loaded];
PROC (i:IPersist) Handsoff(...) ::
    [Loaded]->[Handsoff] |
    [Handsoff]->[Loaded];
  
```

- **Unbounded behaviour:**

Currently, we only use bags (vectors of non-negative integers) to describe the



state of an object. To express unbounded behaviours, such as allowing an unbounded number of clients to access a service in particular state, we are investigating the use of infinite numbers of tokens or negative tokens.

- **Non-determinacy:**

The abstract state of an object should reflect the concrete state of the object. It is clearly conceivable that a method may have several possible outcomes and hence modify the state of an object in several possible ways. This may complicate the semantics considerably. Negotiations would block on potential outcomes of other negotiations but then fail if the desired outcome does not eventuate.

Different types of tokens

- **Causality/Thread specific tokens:**

Such tokens are only visible or usable by one causality or thread. Methods guarded by such tokens become usable by only by the nominated thread. Such a mechanism may be useful for representing thread affinity, or recursive (or re-entrant) but mutually exclusive behaviours.

- **Secure tokens:**

Secure tokens require that negotiating clients be trusted to guard privileged operations. Secure tokens would add an additional check to the semantics of negotiation, causing failure for non-secure requests.

- **Transactional tokens:**

Negotiation points may serve as the boundaries for lightweight transactions. Transactional tokens may help in rolling back changes when erroneous cases such as deadlock are detected.

Inter-object behaviour

- **Shared tokens:**

The interfaces of a component are often not orthogonal. Currently, method invocations can only affect the state associated with one interface of an object. Sharing or equating tokens between the objects of an interface may allow methods to effect object wide (over all interfaces) state changes. This same principle may be extended to sharing tokens between objects. Such a scheme will allow specification of some inter-object behaviours and relationships not currently expressible.

Shared tokens may also help model transactional behaviour, by propagating transactional tokens to nested negotiations.

- **Event handlers and state splitting:**

Puntigam's approach of associating state with individual object references as

opposed to the object itself may be useful for modeling registration of event handlers.

6 CONCLUSIONS

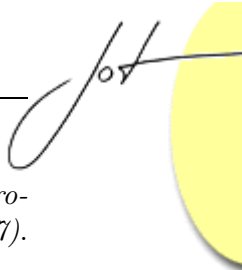
Negotiable interfaces allow inclusion of behavioural information into the type system of a programming language and provide runtime checking to ensure consistency of an object's state. The mechanism provides a mechanism for expressing method availability, re-entrance constraints, and synchronization conditions and can be used to prevent re-entrance and deadlock due to circular blocking. Combined with pre- and post-conditions, negotiable interfaces may allow more complete specification of components.

We are currently extending the type system and semantics to include more expressive behaviours in order to make the proposal a useful programming language feature.

We have implemented negotiable interfaces in an experimental version of our Component Pascal compiler, `gpcp`, and expect to release a public, open source version in the near future.

REFERENCES

- [1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Series in Artificial Intelligence. MIT Press, Cambridge, MA, 1986.
- [2] R. Balter, S. Lacourte, and M. Riveill. The Guide language. *The Computer Journal*, 37(6):519–530, 1994.
- [3] H. de Bruin. BCOOPL: Basic concurrent object-oriented programming language. *Software — Practice and Experience (SPE)*, 30:849–894, 2000.
- [4] `gpcp`. Gardens point component pascal. <http://www.fit.qut.edu.au/CompSci/PLAS/ComponentPascal/>, 2001.
- [5] D. Heuzeroth and R. Reussner. Dynamic coupling of binary components and its technical support. In *First Workshop on Generative and Component based Software Engineering – Young Researchers Workshop*, Erfurt, 1999.
- [6] D. G. Kafura and K. H. Lee. ACT++: Building a concurrent c++ with actors. *Journal of Object-Oriented Programming*, 3(1), 1990.
- [7] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
- [8] Microsoft. C# language specification. available from <http://msdn.microsoft.com/vstudio/nextgen/technology/csharpdownload.asp>, 2000.
- [9] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.
- [10] Oberon Microsystems Inc. Component pascal language report. available from http://www.oberon.ch/docu/language_report.html, 1997.
- [11] F. Puntigam. Types for active objects based on trace semantics. In *1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS'96)*. Paris, France, 1996.



- [12] F. Puntigam. Coordination requirements expressed in types for active objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*. Springer-Verlag, Jyväskylä, Finland, 1997.
- [13] F. Puntigam and C. Peter. Changeable interfaces and promised messages for concurrent components. In *ACM Symposium on Applied Computing (SAC'99)*. San Antonio, Texas, 1999.
- [14] R. Reussner. Formal foundations of dynamic types for software components. Technical Report 08/2000, Department of Informatics, Universität Karlsruhe, Department of Informatics, 2000.
- [15] R. Reussner. Enhanced component interfaces to support dynamic adaptation and extension. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, 2001.
- [16] R. Reussner and D. Heuzeroth. A meta-protocol and type system for the dynamic coupling of binary components. In *Proceedings of the OOPSLA'99 Workshop on Object Oriented Reflection and Software Engineering*, Denver, 1999.
- [17] D. Stoutamire and S. Omohundro. Sather1.1. available online at <http://www.icsi.berkeley.edu/~sather/Specification/Sather1.1/index.html>, 1996.
- [18] C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, Harlow, Essex, 1 edition, 1998.
- [19] J. van den Bos and C. Laffra. PROCOL: A concurrent object-language with protocols, delegation and persistence. *Acta Informatica*, 28:511–538, September 1991.
- [20] H. Wehrheim. Subtyping patterns for active objects. In *Proceedings 8th Workshop des GI Arbeitskreises GROOM (Grundlagen objekt-orientierter Modellierung): Visuelle Verhaltensmodellierung verteilter und nebenläufiger Software-Systeme*, Muenster, 2000.

ABOUT THE AUTHORS

Simon D. Kent is a PhD student within the Programming Languages and Systems research group of the Centre for Information Technology Innovation at QUT. His main interests lie in programming languages, compiler design and implementation, and component-based programming. His email is s.kent@qut.edu.au.

Chris Ho-Stuart is a lecturer in the School of Software Engineering and Data Communications at QUT. His interests include real time process algebra and various kinds of esoteric finite automata.

Paul Roe is an Associate Professor in the School of Software Engineering and Data Communications at QUT and heads the Programming Language and Systems research group of the Centre for Information Technology and Innovation.