Towards a Type Theory for Active Objects[1]

Oscar Nierstrasz
Michael Papathomas[2]

Abstract

Currently popular notions of types, such as signature compatibility, fail to express essential properties of concurrent active objects that are necessary for their correct use in new contexts. We propose and explore a new notion of compatibility called *interaction conformance* defined in terms of the possible interactions between an object and its clients. We relate interaction conformance to known equivalence relations between communicating concurrent agents, and we show that, by viewing types as certain kinds of indeterminate agents, interaction conformance gives us a subtype relationship. We briefly explore the potential for applying these ideas to concurrent object-oriented languages.

1 Introduction

A key property of object-oriented languages is that they promote software reuse through interchangeability of software components that conform to the same message-passing interface. A characterization of such an interface is a *type*, and can be viewed as a constraint on the behaviour of an object. An object that conforms to the type meets the constraint. A subtype, in this view, is simply a stronger constraint: all objects that conform to a subtype also conform to the supertype. The nature of these constraints may vary, however, as may the rules for determining when one type is a subtype of another. The choice of characterization will depend on the computational model of a particular language and the way in which objects interact. We seek to develop a notion of type that will serve to characterize concurrent, active objects whose behaviour may not conform to a strict client/server model of interaction and communication.

Types are useful in programming languages in three significant ways: (1) as a means to express the abstract behaviour (interface) of software components, (2) to support error-detection by either static or run-time type-checking, and (3) in the case that static analysis can be performed, as a means to improve execution efficiency by taking advantage of type information when generating code. We are primarily interested in the first so that we can reason about the compositionality of active objects, but we hope that this work will lead to benefits of the second and third kind.

In object-oriented languages that support type-checking, types are typically expressed as *signatures*. Signature-compatibility is fairly well-understood, especially for functional objects [2]: a signature is a set of operation names, together with their argument and return types; a subtype is a signature with at least the same operations, such that the type of each subtype operation conforms to the supertype operation.

1. A version of this paper has appeared in: ACM OOPS Messenger, Proceedings OOPSLA/ECOOP 90 workshop on Object-Based Concurrent Systems, vol. 2, no. 2, April 1991, pp. 89-93.

2. Authors' address: Centre Universitaire d'Informatique, 12 rue du Lac, CH-1207 Geneva, Switzerland. E-mail: {oscar,michael}@cui.unige.ch. Tel: +41 (22) 787.65.80. Fax: +41 (22) 735.39.05.

What is at work here is the *principle of substitutability* [13]: we wish to ensure that a client of an object receive no surprises when communicating with a subtype instance. This means that all expected operations must be supported, the return types conform, and all arguments passed conform to the object's expectations. Note that there is a two-way contract between the object and its client: it is the arguments provided by the client that must conform to the expectations of the object, not vice versa. (It is for this reason that subtypes are not permitted to refine the types of operation arguments.)

Type systems based on signature compatibility may be viewed as an approximation to an underlying notion of behavioral compatibility which gives rise to practical type checking algorithms. However, there is no well-understood notion of behavior compatibility for active objects, and several limitations to signatures that make them inappropriate to characterize concurrently executing objects[10]:

- ***Non-uniform service availability:*** one cannot express the fact that requests to perform certain operations may be delayed or rejected depending on the state of the object.
- ***Multiple concurrent clients:*** signatures do not capture the interleaving interactions of an object with multiple concurrent clients.
- ***Non-client/server protocols:*** alternative message-passing protocols such as early reply [6] and send/receive/reply [3] are not expressible with signatures.
- ***Re-use criteria:*** signatures do not provide enough information about the behaviour of a concurrent object to determine whether it is safe to use in a new context.

Our approach to defining types of active objects is motivated by substitutability, and is based on a notion of conformance we develop for systems of communicating agents [4][8]. This approach to types may be applied to actual concurrent object-oriented languages by mapping objects to agents and carrying over substitutability from the domain of agents to that of objects. Thus, if *M* is a mapping from objects of a concurrent object-oriented language to agents we can discuss whether two objects O1 and O2 are mutually substitutable by considering the observable behaviour of the corresponding agents M(O1) and M(O2). We then would like to have an equivalence relation over agents that captures our intuitions about substitutability at the level of the *objects* O1 and O2. For example, *strong equivalence* [8], by being a *congruence* relation guarantees the substitutability of M(O1) and M(O2). This is, however, far too strong for capturing our intuitive understanding of substitutability of two objects O1 and O2 in concurrent object-oriented languages: it would not, in general, be possible to express the substitutability of two objects with the same functionality but with different implementations, or of two objects with the same visible interface to a client, but providing slightly different functionality.

We take as a starting point various previously defined and explored forms of equivalence between communicating concurrent agents [4][8] and we define a weaker equivalence called *interaction equivalence* that expresses when two agents are equivalent with respect to a given set of observers. We show that for the set of all possible observers, this equivalence is identical to *failures equivalence*. We then define an asymmetric variant of this equivalence called *interaction conformance*. We then explore some of the properties of interaction conformance, and we show how certain kinds of indeterminate agents can be viewed as types and interaction conformant types as subtypes.

The two-way contract between an object and its clients can be expressed in terms of *liveness* and *safety* conditions [5][7]. Safety conditions express what is *allowed* to happen, and liveness conditions express what is *guaranteed* to happen. Much previous work in temporal specifications has concentrated on specification and formal verification of concurrent programs. Our goal is less ambitious: we seek to approximate the *possible interactions* of an object with its environment as restricted liveness and safety conditions, and thus obtain a notion of types for active objects. Since we can view such conditions as constraints on the behaviour of an object, we obtain a notion of substitutability, and thus of subtyping.

In order to be effective and generally applicable this approach requires the ability to model object-oriented language features by agents in such a way that one will easily be able to devise mappings *M* that accommodate the object models and constructs supported by various classes [11] of concurrent object-oriented programming languages. Apart from being useful for determining substitutability of objects in a single language, this approach is also interesting because it provides a common framework for representing and comparing the approaches taken by concurrent object-oriented languages. In §4, we succinctly describe CCS agents that are used for modelling objects of two rather different classes of concurrent object-oriented programming languages and we also discuss some object-oriented features that appear difficult to model by CCS agents. These issues are treated more extensively in [12].

We conclude by summarizing some open problems.

2 Interaction Equivalence

We shall view computations as systems of communicating concurrent agents. An *agent* is an entity that may communicate with other agents or invisibly change state due to internal communication. A “system” is just another word for a complex agent made up of other agents. We can view such systems as labelled transition systems [8] where transitions represent both offers to communicate and invisible state changes.

A *labelled transition system* $\langle S, T, \{ \xrightarrow{t} | t \in T \} \rangle$ consists of a set S of states, a set T of transitions and a set of next-state relations $\xrightarrow{t}$ for each $t \in T$. In particular, we shall suppose that $T = L \cup \{\tau\}$ where τ represents an *invisible transition* (i.e., an internal communication), and $L = C \cup \bar{C}$ is the set of *visible transitions* (i.e., offers to communicate). The set C represents *input offers* and $\bar{C}$ represents *output offers*. Furthermore, we adopt the convention that for $e \in L$, $\bar{\bar{e}} = e$. If $a = \bar{b}$ then we say that offers *a* and *b match*.

Every element of *S* represents (the state of) an agent or a system. If two agents *p* and *q* are concurrently composed (written $p|q$), then they may communicate with each other if their input and output offers match. Furthermore, if there are other agents in the environment then either *p* or *q* is free to communicate independently with those agents. We can express these rules as follows:

1. $p \xrightarrow{e} p' \Rightarrow p|q \xrightarrow{e} p'|q$,
2. $q \xrightarrow{e} q' \Rightarrow p|q \xrightarrow{e} p|q'$, and
3. $p \xrightarrow{e} p'$, $q \xrightarrow{\bar{e}} q' \Rightarrow p|q \xrightarrow{\tau} p'|q'$.

Note that if p and q succeed in communicating with each other, this results in an invisible transition (i.e., invisible to other agents).

The *initials* of an agent is the set of transitions it is initially willing to undergo: $\text{initials}(p) \equiv \{e \in T \mid \exists p', p \xrightarrow{e} p'\}$ [1]. An agent p is *stable*, written stable(p), if it cannot initially make an invisible transition, i.e., if $\tau \notin \text{initials}(p)$. A stable agent must communicate with another agent in order to change state.

A system consisting of a pair of agents u and p is *deadlocked*, written $dl(u,p)$, if both u and p are stable and $\overline{\text{initials}(u)} \cap \text{initials}(p) = \varnothing$. Neither u nor p can make an invisible transition, nor can they communicate with each other. We assume, of course, that the system is *closed*. Otherwise either agent could progress by communicating with external agents and so eventually reach a state where it could communicate with the other.

Definition 1 There is an *interaction path* $s \in L^*$ from p to p' if $p \overset{s}{\Rightarrow} p'$, where:

1. $p \overset{\varepsilon}{\Rightarrow} p$ (ε is the empty sequence),
2. if $p \overset{s}{\Rightarrow} p'$ and $p' \xrightarrow{e} p''$ then $p \overset{se}{\Rightarrow} p''$, and
3. if $p \overset{s}{\Rightarrow} p'$ and $p' \xrightarrow{\tau} p''$ then $p \overset{s}{\Rightarrow} p''$.

That is, s is a trace of the communications that p is willing to exchange with other agents, possibly interleaved with invisible transitions, and leading eventually to state p'. We can now define the interactions between an agent and its environment.

Definition 2 The set of *interactions up to deadlock* of an agent p with an observer u is $\omega_u(p) = \{s | p \overset{s}{\Rightarrow} p', u \overset{\bar{s}}{\Rightarrow} u', dl(u', p')\}$.

Definition 3 Two agents p and q are *interaction equivalent* with respect to an observer u, written $p \sim_u q$ if $\omega_u(p) = \omega_u(q)$.

What we wish to capture with this definition is the idea that, starting from some initial state, a client (or observer) cannot distinguish between two agents if they support exactly the same set of interactions. Note that interaction equivalence is not recursively defined, so states reachable from p and q by the same sequence of transitions need not themselves be interaction equivalent. Interaction equivalence is therefore a much weaker notion than *observation equivalence* [8]. If we consider certain kinds of observers, however, we obtain other known equivalences.

Recall that p and q are *trace equivalent* if they have the same set of observable actions, i.e., if traces(p) = traces(q), where $\text{traces}(p) \equiv \{s \mid \exists p', p \overset{s}{\Rightarrow} p'\}$. Now let us consider the agent *STAR* that repeatedly accepts all offers or invisibly changes state to a dead agent, i.e., $\forall e \in L$, $STAR \xrightarrow{e} STAR$ and $STAR \xrightarrow{\tau} nil$, where $\text{initials}(nil) = \varnothing$. We can observe the following simple result:

Theorem 4 p and q are trace equivalent iff $p \sim_{STAR} q$.

The proof is straightforward, as $\omega_{STAR}(p)$ is equal to the traces of p.

We now obtain the main result of this section. We shall extend interaction equivalence to *sets* of observers in the obvious way: $p \sim_U q$ iff $\forall u \in U,\ \ p \sim_u q$.

Recall that p and q are *failures equivalent*[1], written $p \sim_f q$ if failures(p) = failures(q), where:

$$\text{failures}(p) \equiv \{ \langle s, X \rangle \mid p \overset{s}{\Rightarrow} p',\ \text{stable}(p'),\ \ X \text{finite},\ X \cap \text{initials}(p') = \emptyset \}$$

This expresses the idea that two agents are equivalent if they fail (i.e., cannot proceed) under the same circumstances.

Theorem 5 p and q are failures equivalent iff they are interaction equivalent with respect to the set of all possible observers, i.e., if $p \sim_S q$.

Proof: (I) We shall first show that $p \sim_S q \Rightarrow p \sim_f q$.

Suppose that $p \sim_S q$. If failures(p) $= \emptyset$ then p never deadlocks with respect to any observer, and so neither does q. In this case p and q are clearly failures equivalent. Now let us suppose that failures$(p) \neq \emptyset$.

Choose an arbitrary $\langle s, X \rangle \in$ failures(p). Now consider an observer $u \in S$ that accepts exactly the complement of s, leading to u' where $u' \overset{\bar{e}}{\rightarrow}\ \ nil, \forall e \in X$.

In this case, $u \overset{\bar{s}}{\Rightarrow}\ u'$, initials($u'$) $= \bar{X}$ and so $dl(u',p)$. As a consequence, $s \in \omega_u(p)$, but since p and q are interaction equivalent for all observers, s is also in $\omega_u(q)$. Hence $\langle s, X \rangle \in$ failures(q) and so by symmetry we conclude that p and q are failures equivalent.

(II) Let us now prove the converse. Suppose that $p \sim_f q$ and that $u \in S$ is an arbitrary observer. If $\omega_u(p) = \emptyset$ then $\langle \varepsilon, \text{initials}(u) \rangle \in$ failures(p), hence also in failures(q). We can then conclude that $\omega_u(q) = \emptyset$ and that $p \sim_u q$.

Now we suppose that $\omega_u(p) \neq \emptyset$ and select some $s \in \omega_u(p)$. By Definition 2 we deduce that there is some u' such that $u \overset{\bar{s}}{\Rightarrow} u'$, $p \overset{s}{\Rightarrow} p'$ and $dl(u', p')$. Hence, for every finite $X \subseteq \text{initials}(u')$ we have $\langle s, \bar{X} \rangle \in$ failures(p). But by failures equivalence of p and q we also have $\langle s, \bar{X} \rangle \in$ failures(q), hence $s \in \omega_u(q)$. By symmetry we conclude that $\omega_u(p) = \omega_u(q)$, and by ranging over all observers u, we conclude that $p \sim_S q$ $\square$.

3 Interaction Conformance

We shall now define an asymmetric variant of interaction equivalence that will give us a form of “type compatibility” for agents.

Definition 6 An agent p is *interaction conformant* to an agent q with respect to an observer u, written $p \ll_u q$, if:

1. $\omega_u(p) \subseteq \omega_u(q)$,
2. $\omega_u(q) \neq \emptyset \Rightarrow \omega_u(p) \neq \emptyset$, and
3. $s \in \omega_u(p),\ ss' \in \omega_u(q) \Rightarrow \exists s'',\ \ ss'' \in \omega_u(p)$.

By condition 1 it is obvious that if $p \ll_u q \ll_u p$, then $p \sim_u q$.

The other two conditions are there to provide additional liveness and safety conditions. Condition 2 says that if *q* can interact at all with *u*, then *p* must offer at least some interactions. Condition 3 says that if *p* (and *q*) can communicate with *u* up to the interaction path *s*, and if *q* can possibly continue after this point, then *p* must also offer to continue (though it may restrict the choices offered by *q*).

At this point we introduce the idea that $\omega_u(p)$ somehow expresses *u*'s *expectations* of *p*. Any of the interactions in $\omega_u(p)$ are acceptable, with the choice up to *p*. By the same token, if *u* expects any of the interactions in $\omega_u(q)$, and $p \lll_u q$, then *u* will be satisfied by the refined choices offered by *p*.

Ultimately a practical notion of type must translate into some notation for specifying the expectations of an observer. Without considering yet what such a notation might look like, let us consider some of the properties it should have. First of all we would like to be able to determine whether an agent *p* satisfies a type constraint without explicit consideration of the behaviour of the observer. One way to do this is to let the type expression itself be an agent that characterizes $\omega_u(p)$.

Definition 7 We define the *runs* of an agent *t* to be the set of its *potential* interactions up to deadlock, $\text{runs}(t) \equiv \omega_{RUN}(t)$, where *RUN* accepts all offers *ad infinitum*: $\forall e \in L$, $RUN \xrightarrow{e} RUN$.

If an observer *u* declares its interest in an agent of type *t*, that is taken to mean that $\omega_u(t) = \text{runs}(t)$, and that *u* will be satisfied by any agent *p* such that $p \lll_u t$. In order to determine whether we have such a *p*, we should be able to test whether "*p* is of type *t*," which we write as *p*:*t*, without having to explicitly consider the relation $\lll_u$.

Since *t* is independent of *u* it must somehow express both the possibility of deadlock (termination of interaction) at certain points as well as the possibility of continuing. Such an agent is *indeterminate*, since its behaviour may differ in two identical situations after the same sequence of communications. Agents representing programmed objects, however, (normally!) should be determinate[1].

A plausible definition of *p*:*t* would be:

1. *p* is determinate,
2. $\text{traces}(p) \subseteq \text{traces}(t)$, and
3. $s \in \text{traces}(p), ss' \in \text{traces}(t) \Rightarrow \exists s'', ss'' \in \text{traces}(p)$.

Because of condition 1, *p* cannot choose to deadlock unless a deadlock was in any case inevitable, i.e., $s \in \text{runs}(p)$ iff $s \in \text{runs}(t)$ and $\neg\exists ss' \in \text{runs}(t)$. It is now easy to see from the definition of $\lll_u$ that $\omega_u(t) = \text{runs}(t)$ and *p*:*t* would guarantee that $p \lll_u t$.

A plausible subtype relation, *t´* :< *t* might obtained by simply dropping condition 1 above, however this would permit *t´* to introduce new deadlocks. Somewhat more satisfactory would be the following definition:

1. For a precise definition of determinacy, see [8].

1. $\text{runs}(t') \subseteq \text{runs}(t)$, and
2. $s \in \text{traces}(t'), ss' \in \text{traces}(t) \Rightarrow \exists s'', ss'' \in \text{traces}(t')$.

This immediately gives us the desirable property that $t'{:<}t, t{:<}t' \Rightarrow \text{runs}(t') = \text{runs}(t) \Rightarrow t' \sim_u t$. Our subtype relationship is defective, however, in that it only allows us to refine the choices offered by *t*, not to provide new alternatives ignored by *u*. This is important if we are to capture properties of inheritance in object-oriented languages where subclass instances that understand additional messages may be used without danger in contexts where superclass instances are expected.

To handle this, we must suppose that $\text{runs}(u) = \overline{\text{runs}(t)}$, that is, *u* is capable of *only* the communications specified by *t*. (This may seem rather extreme, but remember that *u* is the rest of the environment, so we are only concerned with communications between *u* and *t*.) Then we can allow *t´* to add new interaction paths outside those offered by *t*, that is condition 1 would become: $\text{runs}(t') \cap \text{traces}(t) \subseteq \text{runs}(t)$. A subtype must provide at least some of the interactions of a supertype, but otherwise is free to offer any other communications, provided they will be ignored (!). Unfortunately this definition invalidates the property that t´:<*t*:<*t´* makes *t* and *t´* equivalent as types.

In fact, there is something deeper at issue here, which is that we have only taken into account the expectations of a *single* concrete observer *u*. So far we have not expressed the idea that the observer might also make choices, that is, that *u*'s behaviour might not be completely determined. For this, we must extend interaction conformance to sets of observers just as we did for interaction equivalence: $p \ll_U q$ iff $\forall u \in U,\ p \ll_u q$. For *p* to conform to *q*, it must do so for *all* possible elements of *U*, representing the possible choices of behaviour for the observer. The choice, in this case, is up to the observer, not *p*. What we need is to specify the expectations of an abstract observer as a *conjunction* of the expectations of a set of concrete observers. An agent satisfies the specification if it satisfies the expectations of all possible instantiations of the observer. We shall now briefly consider some possible mappings of objects to communicating agents using CCS as a formalism for describing the behaviour of agents.

4 Modelling Objects by CCS Agents

The detailed description of a mapping of concurrent object-oriented programming languages to CCS agents is treated elsewhere [12]. Here, we provide a brief indication of how the semantics of various objects models ranging from passive data objects to active objects can be accommodated within a common framework. We will also briefly discuss some shortcomings of CCS for expressing the semantics of active objects and discuss possible extensions. The presentation of objects as CCS agents also illustrates some difficulties concerning the notion of types expressed on agents, as objects that service requests concurrently should be equated with objects that process requests one after the other.

The approach we take for expressing objects as agents follows the definition of a programming language in terms of CCS given in [8]. We omit the details of representing statements and expressions as agents which are described in [12] and is essentially the same as in [8].

Objects may be represented by the composition of three kinds of agents: V_i representing the object's instance variables, M_{i} representing the object's methods, and *OB* being used to describe the message passing and method activation behavior of a particular model of objects.

An object with instance variables V1,...,Vn and methods M1,…,Mk is represented by:

$$O = ((V_1|V_2\ldots|V_n|M_1|M_2\ldots|M_k)\backslash ACC(V_1)\backslash\ldots\backslash ACC(V_n)|OB)$$
$$\backslash ACC(M_1)\ldots\backslash ACC(M_k)$$

The agents V_i, representing the instance variables, are composed[1] with the methods M_i, and the access sort $ACC(V_i)$ of the variables is hidden so that they may only be accessed by the agents that represent the object's methods.

An instance variable *V* is represented by an agent $V = \mathrm{put}_V x.Loc_V(x)$ where:

$$Loc_V(y) = \mathrm{put}_V x.Loc_V(x) + \overline{\mathrm{get}_V} y.Loc_V(y)$$

Loc(y) represents a location that stores the value *Y*. *V* models a variable that first has to be initialized and then acts as *Loc*.

Methods are mapped to agents of the form *M* given below:

$$M \equiv call_{Mj}.arg_{Mj1}x_1.\cdots.arg_{Mjn}x_n.Body_{Mj}(x_1, \ldots, x_n)|M$$

A sequence of events $arg_{Mi}x$ is used for getting the method arguments. *Body* is an agent, not described here, that represents the actual execution of the method's statements. This agent generates the events $reply_{M,j}x$ and $done_{M,j}$ used for supplying the result, *x*, computed by the method, and for indicating that the method execution is finished. The use of separate events for returning the result and for indicating completion of method execution is necessary for modelling languages in which a result may be returned before the end of the execution of a method. The index *j* is used for telling apart the arguments and return values of multiple activation of the same method. It is necessary for modeling languages in which methods may execute concurrently.

The agent *OB* is used for describing various message acceptance and method activation disciplines. It accepts method invocation requests $call_{C,\,O,\,M,\,j}a_1\ldots a_n$ from other agents and invokes the object's methods according to the object model of the language. *C* is used for identifying the caller, *O* the object called, *M* the method to be invoked and *j* is used to distinguish among several calls made by the same caller.

Objects of languages in which method execution is mutually exclusive may be modelled by the agent *OB*, shown below, which accepts a new message and invokes the requested method only after the previous method execution is completed.

$$OB_O \equiv \sum_{C,\,M,\,j} (call_{C,\,O,\,M,\,j}a_1\cdots a_n.\overline{call_{M,\,1}}.\overline{arg_{M,\,1}}a_1\cdots\overline{arg_{M,\,1}}a_n.$$
$$\overline{reply}_{M,\,1}x.reply_{C,\,O,\,M,\,j}.x.\overline{done}_{M,\,1}.OB_O)$$

1. The operator | mentioned earlier is used to compose concurrent agents; \ is the restriction operator of CCS, which is used to hide a list of event labels; + is the exclusive choice, or summation operator; and . is the prefixing operator of CCS that specifies the replacement behaviour following a communication event. See [8] for a complete description of CCS, or [9] in this volume for the description of a similar notation.

The sum is taken with *C* ranging over the set of callers, *M* ranges over the set of the object's methods and *j* over natural numbers.

Objects that behave like passive abstract data types whose methods may be executed concurrently may be modelled by an agent *OB*´. This agent accepts method invocation requests, invokes the requested method and creates a new instance of itself that may handle further invocations concurrently.

$$\begin{aligned} OB'_O \equiv \sum_{C,M} [& call_{C,O,M,1} a_1 \ldots a_n . \overline{call_{M,1}} . \overline{arg_{M,1}} a_1 \ldots \overline{arg_{M,1}} a_n . \\ & \overline{reply_{M,1}} x . reply_{C,O,M,1} . x . \overline{done_{M,1}} . OB'_O] \\ & | OB'_O [f_{O,M}] \end{aligned}$$

$f_{O,M}$ is a relabelling used to distinguish among multiple concurrent activations of a method handled by different instantiations of *OB*'.

$$\begin{aligned} f_{O,M} = \{ & call_{C,O,M,i+1} / call_{C,O,M,i}, call_{M,i+1} / call_{M,i}, arg_{M,i+1} a_k / arg_{M,i} a_k, \\ & done_{M,i+1} / done_{M,i}, reply_{C,M,i+1} x / reply_{C,M,i} x \} \end{aligned}$$

5 Conclusions

We have argued that a new notion of type based on substitutability of observable behaviour is needed for concurrent object-oriented languages, and that formalisms based on process calculus and labelled transition systems are appropriate for developing and reasoning about object types.

We have introduced *interaction equivalence*, a weak form of equivalence that equates agents that are indistinguishable for a given set of observers, and we have demonstrated its relationship to trace equivalence and failures equivalences. We have also introduced an asymmetric variant called *interaction conformance* that permits us to compare non-equivalent agents, and we have shown how indeterminate agents can be viewed as "types" that constrain the behaviour of interaction conformant, determinate agents. When we restrict our attention to a single observer at a time, we can obtain a form of subtype relation over types.

Our notions of conformance apply to notations such as CCS [8] or Abacus [9], which can be used as semantic targets for the definition of object models. We illustrate how CCS can be used to capture two rather different models of objects used in concurrent object-oriented languages. There are, however, object-oriented features that are difficult to represent in CCS. In particular, systems with dynamic intercommunication structure or dynamic linkage are not directly representable [8]. This is problematic as concurrent object-oriented languages support systems with dynamic intercommunication structure: object identifiers may be passed around in messages allowing any object to communicate with another provided it knows its identifier. It is easy to extend the CCS language so that dynamic linkage is directly representable. See, for example, the specification of an actor based-language using Abacus [9]. However, we have to consider more carefully the effects such extensions may have on the algebraic laws that apply to the operators used for modelling object-oriented features, as this may affect our ability to reason about the properties of objects.

We can summarize the open problems as follows:

1. How should we characterize substitutability for a *set* of possible observers? (Can the notion of a type as an indeterminate agent be usefully extended?)

2. Does interaction conformance lead to a useful notion of substitutability at the level of objects? (Are there other, more appropriate conformance relations?)

3. What mappings of objects to agents are useful for comparing various object models?

4. Are there object models and conformance relations that can lead to practical (static or run-time) type-checking algorithms for concurrent object-oriented programming languages?

References

[1] S.D. Brookes, C.A.R. Hoare and A.W. Roscoe, "A Theory of Communicating Sequential Processes," Journal of the ACM, vol. 31, no. 3, pp. 560-599, July 1984.

[2] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," ACM Computing Surveys, vol. 17, no. 4, pp. 471-522, Dec 1985.

[3] W.M. Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept," Software – Practice and Experience, vol. 11, pp. 435-466, 1981.

[4] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[5] L. Lamport, "Specifying Concurrent Program Modules," ACM TOPLAS, vol. 5, no. 2, pp. 190-222, April 1983.

[6] B. Liskov, M. Herlihy and L. Gilbert, "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing," 13th Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, Jan 13-15, 1986.

[7] Z. Manna and A. Pneuli, "Verification of Concurrent Programs: the Temporal Framework," in *The Correctness Problem in Computer Science*, ed. R.S. Boyer and J.S. Moore, pp. 215-273, Academic Press.

[8] R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

[9] O.M. Nierstrasz, "A Guide to Specifying Concurrent Behaviour with Abacus," in *Object Management*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, July 1990, (to be submitted for publication).

[10] O.M. Nierstrasz and M. Papathomas, "Viewing Objects as Patterns of Communicating Agents," Proceedings OOPSLA '90, 1990, (to appear).

[11] M. Papathomas, "Concurrency Issues in Object-Oriented Programming Languages," in *Object Oriented Development*, ed. D.C. Tsichritzis, pp. 207-245, Centre Universitaire d'Informatique, University of Geneva, July 1989.

[12] M. Papathomas, "Using Process Algebra for the Description and Comparison of Concurrent Object-Oriented Languages," in preparation

[13] P. Wegner and S. B. Zdonik, "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like," in *Proceedings of the European Conference on Object-oriented Programming*, ed. S. Gjessing and K. Nygaard, Lecture Notes in Computer Science 322, pp. 55-77, Springer Verlag, Oslo, August 15-17, 1988.