Chapter 4

Regular Types for Active Objects*

Oscar Nierstrasz

Abstract Previous work on type-theoretic foundations for object-oriented programming languages has mostly focused on applying or extending functional type theory to functional "objects." This approach, while benefiting from a vast body of existing literature, has the disadvantage of dealing with state change either in a roundabout way or not at all, and completely side-stepping issues of concurrency. In particular, dynamic issues of non-uniform service availability and conformance to protocols are not addressed by functional types. We propose a new type framework that characterizes objects as regular (finite state) processes that provide guarantees of service along public channels. We also propose a new notion of subtyping for active objects, based on Brinksma's notion of *extension*, that extends Wegner and Zdonik's "principle of substitutability" to non-uniform service availability. Finally, we formalize what it means to "satisfy a client's expectations," and we show how regular types can be used to tell when sequential or concurrent clients are satisfied.

4.1 Introduction

Much of the work on developing type-theoretic foundations for object-oriented programming languages has its roots in typed lambda calculus. In such approaches, an object is viewed as a record of functions together with a hidden representation type [10]. While this view has the advantage of benefiting from a well-developed body of literature that has a great deal to say of relevance to OOP about polymorphism and subtyping — see, for ex-

* This chapter is a revised and corrected version of a previously published paper. © ACM. *Proceedings OOPSLA '93,* Washington DC, Sept. 26 – Oct. 1, 1993, pp. 1–15. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Oscar Nierstrasz, "Regular Types for Active Objects," *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Eds.), pp. 99-121, Prentice Hall, 1995.

ample, chapter 6 of this book — the fact that objects in real object-oriented languages change state is typically dealt with in an indirect way.

The mismatch is even more acute in concurrent object-oriented languages. In such languages, "active objects" may have their own thread of control and may delay the servicing of certain requests according to synchronization constraints [20]. Such objects may furthermore require a particular protocol to be obeyed (such as an initialization protocol) for them to behave properly. Chapter 2 of this book presents a survey of such languages and a thorough discussion of issues. See also chapter 12 for an example of an object-oriented framework in which "gluons" encapsulate protocols to facilitate dynamic interconnection of components. Existing notions of object types coming from a functional setting do not address the issues of non-uniform service availability or conformance to a service protocol. (Although these issues are also relevant for passive objects and sequential OOPLs, we draw our main motivation from object-based concurrency, and so we will refer in a general way to "active" objects.)

We argue that, in order to address these issues, it is essential to start by viewing an object as a *process*, not a function. (See [26] for other reasons.) By "process" we mean an abstract machine that communicates by passing messages along named channels, as in Milner's CCS [24] or the polyadic π-calculus [25]. Processes naturally model objects since they represent pure behaviour (i.e. by message passing). Behaviour and "state" are indistinguishable in such an approach, since the current state of a process is just its current behaviour. Unfortunately there has been considerably less research done on type models for processes than for functions, and the work that has been done focuses primarily on typing *channels*, not processes (see, for example [25] [33]).

Although processes in general may exhibit arbitrary behaviour, we can (normally) expect objects to conform to fairly regular patterns of behaviour. In fact, we propose on the one hand to characterize the *service types* associated with an object in terms of types of request and reply messages, and on the other hand to characterize the *availability* of these services by *regular types* that express the abstract states in which services are available and when transitions between abstract states may take place. Services represent contracts or "promises" over the message-passing behaviour of the object: in a given state the object will accept certain types of requests over its public channels, and promises to (eventually) send a reply along a private channel (supplied as part of the request message). When providing a particular service, an object may (non-deterministically) change its abstract state to alter the availability of selected services.

Subtyping in our framework is based on a generalization of Wegner and Zdonik's "principle of substitutability" [34]: services may be refined as long as the original promises are still upheld (by means of a novel application of intersection types [5] [31]), and regular types may be refined according to a subtype relation — based on Brinksma's *extension* relation for LOTOS processes [7] — that we call "request substitutability."

In section 4.2 we shall briefly review what we mean by "type" and "subtype," and how we may understand the notion of *substitutability* in the context of active objects. In section 4.3 we introduce *service types* as a means to characterize the types of request messages understood by an object and their associated replies, and we show how *intersection* over service types provides us with a means to refine these specifications.

In section 4.4 we define *request substitutability* for transition systems and we demonstrate its relationship to failures equivalence. In section 4.5 we introduce *regular types* as a means to specify the protocols of active objects. In section 4.6 we propose to use request substitutability as a subtype relationship for regular types, and we demonstrate a simple algorithm for checking that one regular type is request substitutable for another. Next, we formalize a client's expectations in terms of *request satisfiability*, and we show how regular types relate to this notion.

In section 4.8 we summarize a number of open issues to be resolved on the way to practically applying our type framework to real object-oriented languages. We conclude with some remarks on unexplored directions.

4.2 Types, Substitutability and Active Objects

Before we embark on a discussion of what types should do for active objects, we should be careful to state as precisely as possible (albeit informally) what we believe types are and what they are for. Historically, types have meant many things from templates for data structures and interface descriptions, to algebraic theories and retracts over Scott's semantic domains. We are interested in viewing types as *partial specifications of behaviour* of values in some domain of discourse. Furthermore, types should express things about these values that tell us how we may use them safely. Naturally, we would also like these specifications to (normally) be statically checkable.

Subtyping is a particular kind of type refinement. The *interpretation* of a type for some value space determines which values satisfy the type. A subtype, then, is simply a stronger specification and guarantees that the set of values satisfying the subtype is a *subset* of those that satisfy the supertype. If T is a type (expression) and U is some universal value space of interest, then we shall write $\mathsf{x}{:}\mathsf{T}$ to mean x satisfies T, and $[\![\mathsf{T}]\!]$ to mean $\{\,\mathsf{x} \mid \mathsf{x}{:}\mathsf{T}\,\}$ (i.e. where U is understood). Another type S is a subtype of T, written $\mathsf{S}{\le}\mathsf{T}$, if $\mathsf{x}{:}\mathsf{S} \Rightarrow \mathsf{x}{:}\mathsf{T}$, i.e. $[\![\mathsf{S}]\!]{\subseteq}[\![\mathsf{T}]\!]$.

But specifically what *kinds* of properties should types specify? It is worthwhile to recall Wegner and Zdonik's principle of substitutability:

> An instance of a subtype can always be used in any context in which an instance of a supertype was expected. [34]

It is important to recognize that "can be used" implies *only* "safely," and nothing more. It does not imply, for instance, that an application in which a type has been replaced by some subtype will exhibit the same behaviour. We are not concerned with full behavioural compatibility, but only with safe usage.

What does type safety mean in an object-oriented setting? First of all, that objects should only be sent messages that they "understand." We must therefore be able to specify the types of request and reply messages exchanged by objects. If we think of objects as

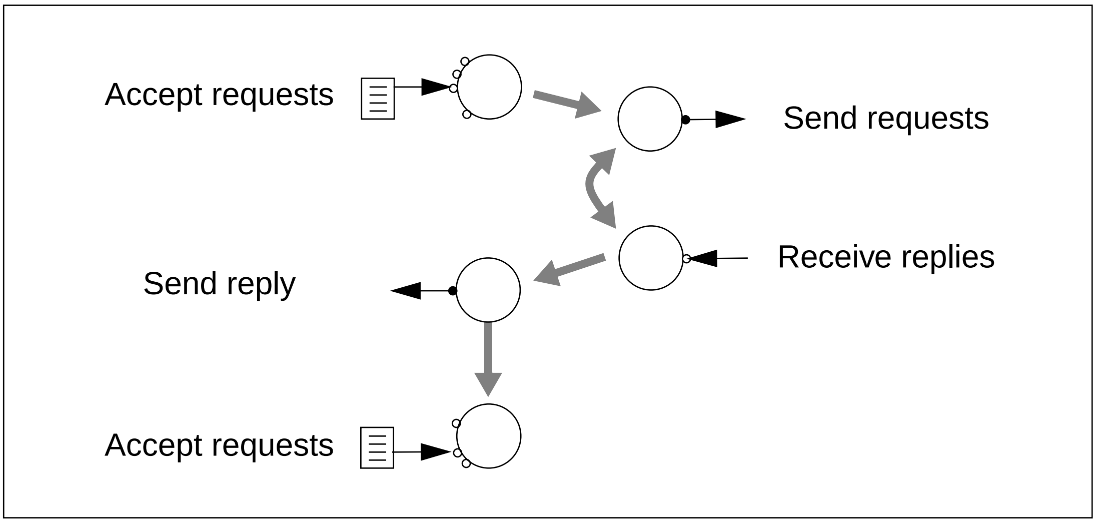


Figure 4.1 *Non-uniform service availability.*

"servers," then the services they provide are promises that they understand certain types of requests, and that, in response to a particular request, they will eventually send a certain type of reply. Subtyping of services can then be defined in a fairly conventional way, in that a subtype at least guarantees the promises of the supertype: at least the same requests are understood (possibly more) and consequent replies to those requests are guaranteed to be of the right type.

Services may not always be available, however. If requests must be sent in a certain order, or if certain services may be temporarily unavailable, then, we argue, the object's type should describe this. Type safety, in this case, means that clients (or, more generally, environments) that interact with such objects do not deadlock because of protocol errors. Type substitutability is correspondingly defined so that sequences of interactions that are valid for a supertype are also valid for a subtype. A client will never be unexpectedly starved of service because a subtype instance has been substituted.

In order to explain our type approach, we will adopt an object model that views objects as certain kinds of communicating processes [4][8][17][24]. (Although we could formalize our model in process-theoretic terms, as in, for example, [30], for the purposes of this presentation we will attempt to be rigorous and precise without being excessively formal.)

Figure 4.1 depicts an object's behaviour in an idealized fashion. The large circles represent the object in its various states and the small circles represent its communication channels, white for input and black for output. The input channels on the left side are for receiving requests. Note that the set of "enabled" input requests changes over time.

In our object model, every object receives requests along uniquely identified channels, one per request name. Each request consists of a message containing a number of arguments and a unique reply address (also a channel name). The arguments must be of the correct type. (We will not be concerned with what kinds of values may be passed, but the reader may assume that any reasonable value — objects, object identifiers, channel names, etc. — is fair game.)

An object, then, accepts requests addressed to it through its (public) request channels, and it may issue requests to other objects it is acquainted with via *their* request channels. All replies, however, are communicated along *private* channels that are temporarily established by clients of requests. When an object accepts a request, it implicitly *guarantees* to (eventually) send a reply (of the correct type) to the client. This reply may be delivered by a third party to which the reply address has been forwarded. Furthermore, the object may vary the requests accepted over time by selectively listening only to certain request channels. When an object is ready to accept a message addressed to one of its request channels, we say that the request is *enabled*, and that the corresponding service is *available*. We assume that the complete set of public request channels is finite and fixed in advance for any object.

We will now separately discuss the issues of specifying types of services associated with an object (section 4.3), and specifying when those services are available (section 4.4).

4.3 Intersecting Service Types

We will start by introducing the following syntax for service types:

$S ::= \mathsf{all} \mid \mathsf{none} \mid M(V) \rightarrow V \mid S\hat{}S$

$V ::= \mathsf{all} \mid \mathsf{none} \mid (V, ...) \mid ...$

where M is a request name and V is a value type (i.e. types for argument and return values). "→" binds more tightly than "^". We assume that V includes some base types, the types all and none, and tuples over value types.

We will write x : m(A)→R to mean that object x may receive a value a of type A together with a reply address along a request channel x_m and will consequently promise to return a value r of type R. We may also write x.m(a) : R to say that x understands the message m(a) and returns a value of type R. We call the type expression m(A)→R a *service* of x, and we say that x *offers* this service. Note that this does not imply anything about other services that x may or may not offer.

We may refine these expressions by the *intersection* operator for types (^). Intersection types have been studied extensively in functional settings (see [31] for a bibliography). Here we propose to assign an interpretation to them for objects in a process setting. If we write x:S1^S2, we wish that to mean precisely that x:S1 *and* x:S2. In set-theoretic terms, then:

$$[\![S1\hat{}S2]\!] = [\![S1]\!] \cap [\![S2]\!]$$

As specifications, we mean that both S1 and S2 are true statements about x. As we shall see, this device allows us not only to attribute sets of services to objects, but also permits us to refine their types in interesting ways.

The expressions all and none represent, respectively, the set of all objects and the empty set. That is, all tells us nothing about the services of an object, and none demands so much that no object can possibly satisfy it. (all and none are the “top” and “bottom” of our type hierarchy.)

Let us now briefly look at the subtyping properties of service types. Some facts are clear:

1. T ≤ all (i.e. for any value or service type T)
2. none ≤ T
3. m(none)→T = all (since no such request can ever be received)
4. R1 ≤ R2 ⇒ m(A)→R1 ≤ m(A)→R2
5. A2 ≤ A1 ⇒ m(A1)→R ≤ m(A2)→R (i.e. a contravariant rule)

Now, considering intersections, the following are straightforward:

6. S1^S2 ≤ S1 and S1^S2 ≤ S2
7. S ≤ S1 and S ≤ S2 ⇒ S ≤ S1^S2
8. S1 ≤ S2 ⇒ (S1^S2) = S1 (follows from (6) and (7))

Now consider:

9. m(A1)→R1 ^ m(A2)→R2 ≤ m(A1^A2)→(R1^R2)

Normally we may expect to write type expressions like:

put(all)→(Ok) ^ get()→(all)

but nothing prevents us from writing:

inc(Int)→Int ^ inc(Real)→Real

or even:

update(Point)→Point ^ update(Colour)→Colour

If an incoming message satisfies more than one request type in the intersection, then the result must satisfy *each* of the result types. Our (informal) semantics of intersection types requires that *all* applicable service guarantees must hold. In this case, if:

cp:ColouredPoint,

where ColouredPoint = Point^Colour
then x.update(cp):Point *and* x.update(cp):Colour. The result, therefore, must have type ColouredPoint.

Notice that as a corollary of (9), via (6) , (4) and (7), we also have:

10. m(A)→(R1^R2) = m(A)→R1 ^ m(A)→R2

This also means, however, that we must take care not to intersect services with abandon. For example, suppose Int and Real are disjoint types. Then:

```
size(Point)→Int ^ size(Colour)→Real
        ≤ size(ColouredPoint) → (Int^Real)
        = size(ColouredPoint) → none
```

Since the two size services have contradictory result types, their intersection yields the result type none.

As a final remark, notice that type-safe covariance is naturally expressed:

$$\text{update(Point)} \rightarrow \text{Point} \wedge \text{update(ColouredPoint)} \rightarrow \text{ColouredPoint}$$

is a subtype of both update(Point)→Point and update(ColouredPoint)→ColouredPoint. A client supplying an instance of ColouredPoint as an argument can be sure of getting a ColouredPoint back as a result, whereas clients that supply Point arguments will only be able to infer that the result is of the more general type Point.

4.4 Request Substitutability

Service types tell us what types of requests are understood by an object and what types of reply values it promises to return, but they do not tell us *when* those services are available. In particular, we are interested in specifying when an object's request channels are enabled. The sequences of requests that an object is capable of servicing constitute the object's *protocol*. An object that *conforms* to the protocol of another object is safely substitutable for that second object, in the sense that clients expecting that protocol to be supported will receive no "unpleasant surprises."

Before tackling the issue of how to specify protocols, let us first try to formalize the appropriate substitutability relation.

According to our abstract object model, objects can do four things: accept requests, issue requests, receive replies and send replies. Since the behaviour of objects should be properly encapsulated, clients should only need to know about the first and the last of these, i.e. the requests accepted and the replies sent. If we can safely assume that an object that accepts requests promises to deliver replies according to service type specifications, then the only additional thing a client needs to know about an object's protocol is when it will accept requests. We therefore adopt an abstract view of an object's protocol that only considers *requests* received along its request channels, and *ignores all other messages*. (Later, in section 4.7, we will model clients' protocols by considering only requests issued.)

In this view we model an object as a transition system where each state of interest represents a *stable* state of the object, in which it blocks for acceptance of some set of requests. A transition takes places upon the receipt of some request and leads to a new stable state. If an object in state x can accept a request r leading to a new state x', we would write:

$$x \xrightarrow{r} x'$$

Note that we ignore all intervening communications leading to the new state. If these communications are purely internal to the object, we can view it as a closed system, but if some of these communications are with external acquaintances, then an element of nondeterminism is introduced, since the transitions to new stable states may depend upon the current state of the environment. In cases like this, we feel it is correct to view the object's protocol as inherently non-deterministic, since it would be unreasonable to expect clients to monitor the environment to know the state of an object's protocol.

Clients are typically interested not just in issuing a single request, but in issuing series of related requests. Suppose s is such a sequence r1,r2,... of requests. If an object in state x can accept such a sequence, leading to state x′, then we write:

$$x \overset{s}{\Rightarrow} x'$$

An important part of the protocol of an object is the set of sequences of requests that it may accept. This is conventionally captured by the notion of set of *traces* [8] of a transition system:

Definition 1 $\textit{traces}(x) \equiv \{ s \mid \exists x', x \overset{s}{\Rightarrow} x' \}$.

Suppose we wish to express that an object in state x is *request substitutable* for an object in state y, which we will write x:<y. Then clearly we must have $\textit{traces}(y) \subseteq \textit{traces}(x)$, for if a client of y expects y to accept a sequence of requests s, and we substitute x for y, then x must accept the same sequence s. x may accept additional sequences, but since the client does not expect* them, they are of no concern to us.

But the inclusion of traces is not enough to guarantee request substitutability, for suppose that after a sequence of requests s, y will move to state y′, but x will move to either state x′ or x″. Furthermore, suppose that state x′ is identical to y′ — i.e. behaviour from that point on is identical — and x′ permits a request r to be accepted, but x″ denies it. Then it is possible that $\textit{traces}(y) \subseteq \textit{traces}(x)$, but nevertheless the client may receive a nasty surprise if x is substituted for y and the request r is refused after the sequence s. Traces tell us what sequences are acceptable, but they do not tell us if they are *necessarily* acceptable! For this, we need the help of a finer notion of *failures* [8].

First, we need to define the *initials* of an object — the requests which are initially enabled:

Definition 2 $\textit{init}(x) \equiv \{ r \mid \exists x', x \xrightarrow{r} x' \}$.

Definition 3 The set of *failures* of an object x is

$$\textit{failures}(x) \equiv \{ (s,R) \mid \exists x', x \overset{s}{\Rightarrow} x', R \text{ is finite}, R \cap \textit{init}(x') = \emptyset \}.$$

That is, (s,R) is a failure of x if x may simultaneously refuse all of the requests in the set R after accepting the sequence s. It may be the case that x will reach a state in which some or all of the requests in R will be accepted, but we know that it is *possible* that they will all be refused. (NB: It is also important that the state x′ be stable for the set R to be well-defined, but we have already assumed that.)

Now, suppose that we want x:<y and we know that (s,R) is a failure of x. Furthermore, suppose that s is a sequence of requests in *traces*(y). Then a client will be satisfied *only* if it expected that (s,R) was also a failure of y. Note that if s is *not* a sequence in the protocol of y, then the client is unconcerned whether (s,R) is a failure of x or not, since it is in any

* Although we have not yet formalized clients' expectations, we are implicitly assuming here that clients are *sequential*, i.e. they only issue a single request at a time. Later, when we define *request satisfiability*, we will see how request substitutability relates to concurrent clients.

case not expected to be handled. To express this notion of *relative failures*, we need the following definition:

Definition 4 The set of *relative failures* of an object in state x with respect to an object in state y is: $failures_y(x) \equiv \{ (s,R) \in failures(x) \mid s \in traces(y) \}$.

Now we come to the definition of request substitutability:

Definition 5 An object in state x is *request substitutable* for an object in state y, written x:<y iff:

(i) $traces(y) \subseteq traces(x)$
(ii) $failures_y(x) \subseteq failures(y)$.

(This turns out to be identical to the *extension* relation introduced by Brinksma [7]. See also Cusack [13] for a discussion of various conformance relations, including extension, in the context of CSP [8].)

That is, a client expecting x to follow the protocol of y will expect that all sequences of requests supported by y will also be accepted by x, and that any requests refused by x after accepting one of those sequences might also have been refused by y. Note that x may (1) accept additional sequences of requests that the client does not expect and therefore will not use, and (2) may eliminate some non-determinism in y by providing *fewer* possible transitions between states. On the other hand, x may introduce new transitions and states as long as they can be explained from the viewpoint of y. In general, either x or y may have more or less states or transitions.

Note also that the set of failures of an object tells us all we need to know in order to determine request substitutability, since the traces can be derived from the failures set by projections, and relative failures can be determined from the failures of one object and the traces of another.

Proposition 1 Request substitutability is a pre-order.

Proof

(i) :< *is reflexive*: $\forall x$, x:<x — immediate, since $failures_x(x) = failures(x)$.

(ii) :< *is transitive*: Suppose x:<y and y:<z. Then $traces(z) \subseteq traces(y) \subseteq traces(x)$.
Next, suppose $(s,R) \in failures_z(x)$. Then $s \in traces(z) \subseteq traces(y)$,
so $(s,R) \in failures_y(x) \subseteq failures(y)$. But then $(s,R) \in failures_z(y) \subseteq failures(z)$,
so we conclude x:<z. ❑

There exists a vast literature on process equivalences and pre-orders (see, for example, [1][14] for some interesting comparisons). Interestingly, the equivalence induces by request substitutability is the same as failures equivalence [7][8].

Definition 6 Objects in states x and y are *failures equivalent* iff $failures(x) = failures(y)$.

Proposition 2 x and y are failures equivalent iff x:<y and y:<x.

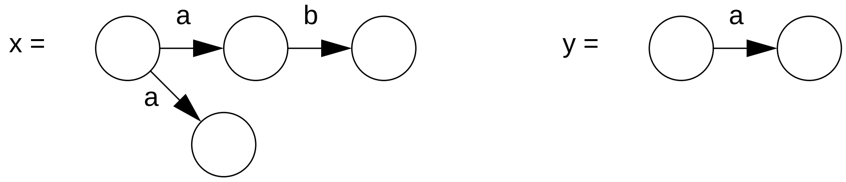
Proof

$\Rightarrow$) $failures(x) = failures(y) \Rightarrow traces(x) = traces(y)$
$\Rightarrow failures(x) = failures_y(x) = failures_x(y) = failures(y) \Rightarrow$ x:<y and y:<x.

$\Leftarrow$) x:<y and y:<x $\Rightarrow traces(x) = traces(y)$.

Hence $failures_y(x) = failures(x) \subseteq failures(y)$.
By symmetry, $failures(x) = failures(y)$. ❑

Although failures equivalence is exactly request equivalence, the inclusion of failures sets does not imply request substitutability, nor vice versa. It suffices to consider:



It is easy to see that x:<y (but not the reverse, since y does not permit a.b) and $failures(y) \subseteq failures(x)$ (but not the reverse, since (a.b,{a,b}) is a failure of x but not of y). See also Brinskma [7] for a detailed discussion.

4.5 Viewing Objects as Regular Processes

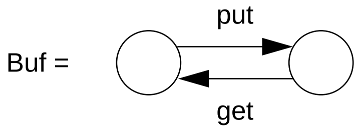
We now have a plausible definition of protocol conformance in terms of request substitutability — what we still need is a way to specify protocols, and a way to check that an object conforms to a protocol, or that one protocol conforms to another. In the most general case, unfortunately, request substitutability will be undecidable since failures equivalence is undecidable in general [18]. (If request substitutability were decidable, we could use its decision procedure to check if two processes were failures equivalent according to proposition 2.)

We therefore propose to specify protocols as *regular processes*, i.e. processes with a finite number of "states" or behaviours [6][11][15][23]. A regular process is essentially a finite state machine (hence the adjective "regular"), where transitions take place upon communications with other processes. We will call the specification of such a process a *regular type*, since we intend to use it to specify object protocols. It turns out that by restricting ourselves to finite state protocols, request substitutability is decidable by a simple procedure.

Furthermore, although we cannot specify all protocols exactly with a finite number of states, we can *approximate* infinite state protocols by non-deterministic regular processes. These approximations can then be used in many cases to check request substitutability.

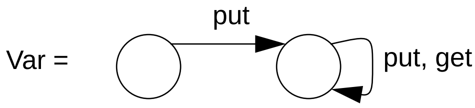
Let us consider a few canonical examples using various kinds of "container" objects (bounded buffers, stacks, variables) each supporting (at least) put and get requests. We can associate with these objects a number of abstract states, each corresponding to a set of currently enabled requests. Since we assume that the total set of possible services is finite, a finite number of abstract states suffices to characterize all the possible combinations of enabled requests (and normally only a few of these combinations should be needed). From the client's point of view, transitions may take place when services are provided (since this is all the client may observe).

First, consider a one-slot bounded buffer.



It has two abstract states: one in which only a put is accepted, and one in which only a get is allowed. Upon accepting a put or a get request, the object changes state. We express this by the protocol (regular type) Buf.

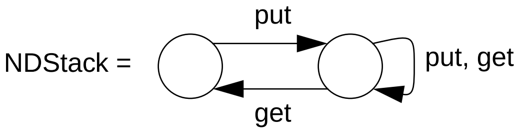
Now consider an uninitialized variable with the protocol Var.



Its protocol requires that a put must first be requested, but then put and get requests may be interleaved arbitrarily. In this case, we see that Var:<Buf since a client that expects an object to obey the Buf protocol will never be "disappointed" if an object obeying Var is substituted. The reverse does not hold, because Buf will refuse the sequence put.get.get, whereas Var will not.

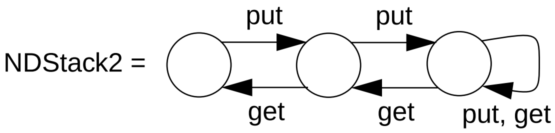
In these two cases, the transitions are deterministic, since Buf and Var are really finite state protocols.

Now consider a stack (with put and get instead of push and pop). Initially only a put is possible. Then both put and get are enabled. Further put requests will not change this, but a get may bring us back to the initial state. The corresponding regular type is specified below as NDStack.



It resembles Var except that after a get, we do not necessarily know what state we are in. Clearly, such a description is an approximation because we are attempting to express the service availability of a deterministic process (the object) by means of a non-deterministic one (the regular type).

We can try to add another intermediate state, as in NDStack2:



but after two put requests and a get we again do not know what state we are in. In fact, we would need an infinite number of states to describe completely the Stack protocol.

As we argued before, however, non-determinism is inherent in some protocols, because objects are not, in general, closed systems. Furthermore, the non-deterministic regular types are still useful to us. We can determine, for example, that an object conforming to the NDStack regular type also conforms to Buf since NDStack:<Buf.

Choosing a simple *and* readable syntax for specifying regular types is somewhat problematic. For the purpose of this chapter we will opt for simplicity. We specify a regular type by a pair, (x_1,E) consisting of a finite system of equations E of the form:

$E = \{ x=t, ... \}$

where x_1 is a distinguished start state, and the t are regular type expressions of the form:

$t ::= r.x \mid t + t$

r is a request name and x is a state name. Every x used in E must have exactly one defining equation in E (except for nil, which stands for a dead state with no transitions). Regular types have the following interpretation as transition systems:

1. $init(\mathrm{nil}) = \varnothing$
2. $\mathrm{r.nil} \xrightarrow{\mathrm{r}} \mathrm{nil}$
3. $\mathrm{x{=}t} \in E \Rightarrow \mathrm{r.x} \xrightarrow{\mathrm{r}} \mathrm{t}$
4. $\mathrm{t1} \xrightarrow{\mathrm{r1}} \mathrm{t1'} \Rightarrow \mathrm{t1{+}t2} \xrightarrow{\mathrm{r1}} \mathrm{t1'}$
5. $\mathrm{t2} \xrightarrow{\mathrm{r2}} \mathrm{t2'} \Rightarrow \mathrm{t1{+}t2} \xrightarrow{\mathrm{r2}} \mathrm{t2'}$

With this simple syntax, then, we could specify the various regular types we have seen as follows:

```
Buf = (b1, { b1=put.b2, b2=get.b1 })
Var = (v1, { v1=put.v2, v2=put.v2+get.v2 })
NDStack = (s1, { s1=put.s2, s2=put.s2+get.s2+get.s1 })
NDStack2 = (s1, { s1=put.s2, s2=put.s3+get.s1,
                  s3=put.s3+get.s2+get.s3 })
```

At this point the reader may wonder why we cannot simply use regular expressions to specify regular types. The reason is that regular expressions stand for regular *languages*, i.e. sets of strings, not regular processes. Regular expressions can consequently tell us about the traces of a transition system but not its failures. Consider, for example, the regular types Var and NDStack. If we consider any state to be a valid final state, then they recognize exactly the same regular language, namely:

$\varepsilon + \mathrm{put.(put{+}get)}^*$

But this does not tell us that after accepting a put followed by a get, NDStack may *refuse* another get, whereas Var never will. (A similar argument is elaborated in [16] to introduce the difference between language and process equivalence.) For precisely the same reason, it is *not* generally possible to convert a non-deterministic regular process into a deterministic one without losing information.

4.6 Subtyping Regular Types

We now propose to use request substitutability as a *subtyping* relationship over regular types. We are justified in this since we have shown that request substitutability is a preorder, so if Var:<NDStack and NDStack:<Buf, then we can conclude that Var:<Buf.

The fact that regular types have finite states means that a simple algorithm exists for checking the subtype relationship (not surprisingly, the algorithm is similar to that for checking equivalence of finite state automata [2]). To derive the algorithm, we must introduce a multi-state variant of request substitutability. First let us extend *init*() and $\rightarrow$ to work with sets of states:

Definition 7 $init(X) \equiv \{ r \mid \exists x \in X, x', x \xrightarrow{r} x' \}$.

Definition 8 $X \xrightarrow{r} X'$ *iff* $X' = \{ x' \mid \exists x \in X, x \xrightarrow{r} x' \}$.

Note in particular that $\rightarrow$ for sets of states is a *function*, not just a relation. In effect, we are turning a non-deterministic transition system into a deterministic one in the traditional way by expanding single states into sets of reachable states [2].

Now let us consider the following definition:

Definition 9 A set of object states X is *multi-state request substitutable* for a set of states Y, written $X{:}{<<}Y$, iff:

(i) $init(Y) \subseteq init(X)$

(ii) $\forall x \in X, \exists y \in Y, init(y) \subseteq init(x)$

(iii) $\forall r \in init(Y)$, if $X \xrightarrow{r} X'$ and $Y \xrightarrow{r} Y'$, then $X'{:}{<<}Y'$.

Condition (i) guarantees that all transitions possible from some state of Y are also possible from some state of X. Condition (ii) says that any failure possible in some state of X can be explained by a failure of some corresponding state of Y (some y has the same or fewer initial transitions possible). Condition (iii) is simply the recursive case.

Proposition 3 $\{ x \}{:}{<<}\{ y \} \Leftrightarrow x{:}{<}y$.

Proof

$\Rightarrow$ **)** Suppose that $\{ x \}{:}{<<}\{ y \}$, then $traces(y) \subseteq traces(x)$ by 9(i) and 9(iii).

Next, suppose $(s,R) \in failures_y(x)$. Then $\exists x', x \overset{s}{\Rightarrow} x', init(x') \cap R = \emptyset$ and $\exists y', y \overset{s}{\Rightarrow} y'$, $init(y') \subseteq init(x')$ by 9.ii and 9.iii so $(s,R) \in failures(y)$ and $failures_y(x) \subseteq failures(y)$ hence $x{:}{<}y$.

$\Leftarrow$ **)** Similar argument in reverse. ❑

Note that this result is independent of whether we restrict our attention to finite state transition systems or not. If the sets of reachable states are finite, however, i.e. if x and y are regular types, then proposition 3 provides us with a simple procedure to check whether $x{:}{<}y$ by simply generating all the sets of states reachable from $\{x\}$ and $\{y\}$ by transitions in *traces*(y) and checking conditions 9(i) and 9(ii) for all the comparable sets. Since the state space is finite, the set of reachable state sets must also be finite, and so the comparison must terminate in finite time.

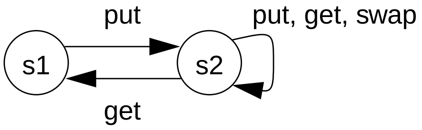
The following iterative algorithm suggests itself: we maintain a LIST of comparable sets of states and possible transitions, of the form (X,Y,R), where X and Y are the sets of states of x and y reachable from some common trace s of y, and R is the set of possible transitions (requests) from Y that the algorithm must traverse. We follow each possible request to new comparable state sets until we have exhausted all transitions and checked all comparable state sets, or until we fail to satisfy one of the conditions in definition 9.

1. Verify that $\mathit{init}(\mathrm{y}) \subseteq \mathit{init}(\mathrm{x})$, else FAIL
2. Add ({x},{y},*init*(y)) to LIST
3. If possible, select some (X,Y,R) from LIST where R is not empty, else SUCCEED
4. Select some r in R and replace (X,Y,R) by $(X,Y,R\backslash\{\mathrm{r}\})$ in LIST
5. Compute X' and Y', where $X \xrightarrow{\mathrm{r}} X'$ and $Y \xrightarrow{\mathrm{r}} Y'$
6. If (X',Y',R') for some R' is already in LIST, then go to step 3, else continue
7. If $\mathit{init}(Y') \subseteq \mathit{init}(X')$, then continue, else FAIL
8. If for each $\mathrm{x_i} \in X'$ there exists some $\mathrm{y_j} \in Y'$ such that $\mathit{init}(\mathrm{y_j}) \subseteq \mathit{init}(\mathrm{x_i})$, then continue, else FAIL
9. Add $(X',Y',\mathit{init}(Y'))$ to LIST and go to step 3.

Note that steps 2 and 7 guarantee that X' generated in step 5 will never be empty.

Since there is a finite number of reachable sets X and Y to compare, the algorithm clearly terminates. In the worst case, there will be $(2^n-1)\times(2^m-1)$ comparisons (i.e. the size of LIST), where n and m are the number of states reachable from x and y respectively, but normally there will be far fewer, since not all subsets of states will be generated, and not all possible combinations will need to be compared. In the special case that one compares two deterministic regular types, the maximum number of comparisons is just $n\times m$, but may be even as little as m (in case of success, that is).

Let us briefly look at an example that compares Buf to the regular type of a stack that supports an additional swap operation:



```
NewNDStack = (s1, { s1=put.s2,
                    s2=put.s2+get.s2+get.s1+swap.s2 })
```

We wish to check whether NewNDStack:<Buf. We start with: ({s1},{b1},{put}). Both s1 and b1 permit a put, and they have the same requests enabled, so we can add this to our list:

({s1},{b1},{put})

The only possible transition is put, so we remove it from LIST and generate: ({s2},{b2},{get}). s2 enables at least the requests that b2 enables, so we add this to our list:

({s1},{b1},{~~put~~})
({s2},{b2},{get})

Now only a get is possible, so we generate: ({s1,s2},{b1},{put}). We verify that s1 and s2 each enable at least the requests of b1 and add this to our list:

({s1},{b1},{~~put~~})
({s2},{b2},{~~get~~})
({s1,s2},{b1},{put})

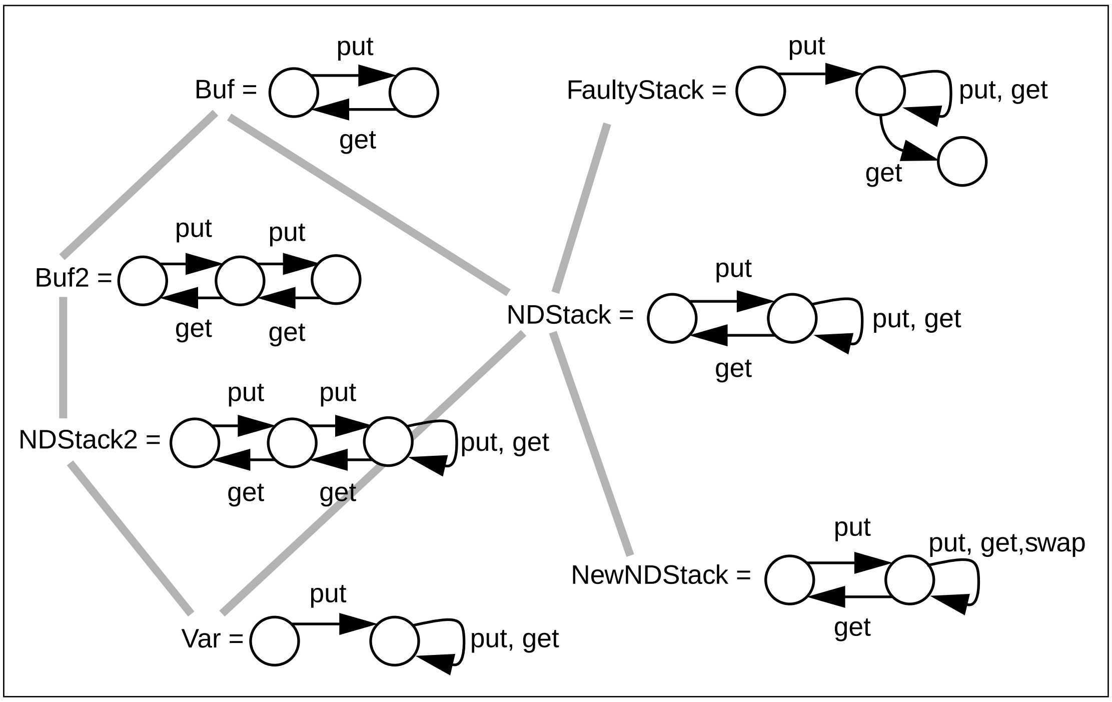


Figure 4.2 *Some subtype relationships between regular types.*

Now we can perform a put, but this just generates ({s2},{b2},{get}), which is already represented in the list. There is nothing left to check, so we SUCCEED. (In the reverse direction we would quickly FAIL in step 7 after a single put because b2 enables neither put nor swap.) Note that the total number of comparisons (3) is far less than the worst case possible (9).

Note that NewNDStack is request substitutable for Buf even though it is, in a sense, *less* deterministic than Buf. The key point is that it is safe to use wherever we are expecting Buf-like behaviour.

Figure 4.2 shows the subtype relationships between a few of the regular types we have seen. Curiously, NDStack and NDStack2 are not related (to see why, consider the sequence put.get.get, which is in *traces*(NDStack) but not in *traces*(NDStack2), and the failure (put.put.get,{get}), which is in *failures*(NDStack), but not in *failures*(NDStack2)).

4.7 Request Satisfiability

Up to now our discussion has focused on the protocols of service providers. Request substitutability tells us when an object obeying some protocol can be safely substituted by some second object, assuming that the first object satisfies the client's expectations. But we have not yet formalized what it means to satisfy a client. It turns out that we need to define a new relation, called *request satisfiability*, which expresses this idea.

If the protocol of a service provider expresses when its services are available, then the protocol of its client expresses when those services are requested. We propose that a client is *satisfied* if its requests are always honoured. Up to now we have implicitly assumed that clients issue at most one request at a time. In general, however, a client may issue multiple requests simultaneously (particularly if the “client” is actually an environment consisting of multiple concurrent clients) — in such cases, we do not ask that all of the requests be honoured together, just that the client be guaranteed to make progress, i.e. at least one request must always be accepted. Since the current state of the client may not necessarily be deterministic, the object must be prepared for the client to be in any one of its reachable states. The object is allowed to terminate (i.e. refuse all further requests) only if it can be sure that the client will issue no more requests. In short, we must ensure that an object can only *fail* if the client makes no more *offers*.

We can formalize this as follows:

Definition 10 The set of *offers* of a transition system c is:

$$\mathit{offers}(c) \equiv \{\, (s,R) \mid \exists c',\ c \overset{s}{\Rightarrow} c',\ R = \mathit{init}\,(c') \,\}.$$

So, if (s,R) is an offer of c, then we know that c may issue the sequence of requests s and then may issue the set of requests R. It is also possible that c may issue some other set of requests R', if (s,R') is also an offer of c.

Definition 11 An object x is *request satisfiable* for a client c, written $x \models c$, iff:

$$(s,R) \in \mathit{failures}(x) \cap \mathit{offers}(c) \Rightarrow R = \emptyset$$

If both client and server protocols are specified as regular types, then request satisfiability can be determined by an algorithm along the lines of the one we demonstrated for checking request substitutability.

4.7.1 Sequential Clients

How does request substitutability relate to request satisfiability? Clearly, we would expect that if $x{:<}y$ and $y \models c$, then $x \models c$. It turns out that if c is sequential, then this is in fact the case.

Definition 12 A client c is *sequential* if $(s,R) \in \mathit{offers}(c) \Rightarrow |R| \leq 1$.

Lemma 4 If c is sequential, then $y \models c \Rightarrow \mathit{traces}(c) \subseteq \mathit{traces}(y)$.

Proof By induction on the length of traces of c. ❑

Proposition 5 If c is sequential, then $x{:<}y$ and $y \models c \Rightarrow x \models c$.

Proof $(s,R) \in \mathit{failures}(x) \cap \mathit{offers}(c) \Rightarrow s \in \mathit{traces}(c) \subseteq \mathit{traces}(y)$

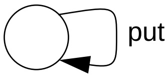
$$\Rightarrow (s,R) \in \mathit{failures}_y(x) \subseteq \mathit{failures}(y) \Rightarrow R = \emptyset.\ ❑$$

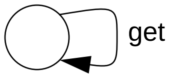
We are taking advantage of the fact that c is sequential to conclude that y *completely* satisfies the expectations of c. (Note that it also suffices to require that $\mathit{traces}(c) \subseteq \mathit{traces}(y)$ for the same result to go through.) But if there are different ways of satisfying a client (particularly a concurrent one), then it is no longer true that the client will necessarily be satisfied by a request substitutable service provider. Some additional preconditions must be imposed.

4.7.2 Concurrent Clients

Let us consider a simple example of a concurrent client consisting of a producer and a consumer connected by a bounded buffer. The producer and the consumer each have their own view of the buffer, but we are interested in the requirements posed by their concurrent composition.

Presently we might separately specify expectations of the producer and consumer respectively as:

Prod = 

Cons = 

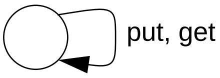
We might write their concurrent composition as Prod&Cons, where:

$$\mathrm{c1} \xrightarrow{r} \mathrm{c1}' \Rightarrow \mathrm{c1\&c2} \xrightarrow{r} \mathrm{c1}'\mathrm{\&c2}$$

and

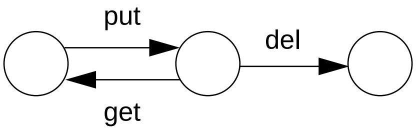
$$\mathrm{c2} \xrightarrow{r} \mathrm{c2}' \Rightarrow \mathrm{c1\&c2} \xrightarrow{r} \mathrm{c1\&c2}'$$

So we can conclude:

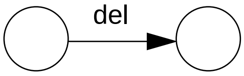
Prod&Cons = 

Note that Prod&Cons is *not* sequential according to definition 12.

It is easy to check that Buf ⊨ Prod&Cons, since Buf never refuses both put and get. But what is the role of request substitutability now? Since we know that Var:<Buf can we necessarily conclude also that Var ⊨ Prod&Cons? Unfortunately this is not quite right. The reason is that a regular subtype may introduce additional behaviour that can perturb the client's expectations. Consider, for example, a deletable buffer:

DelBuf = 

It is clear that DelBuf:<Buf. But suppose that we now compose the producer and consumer with a malevolent object whose only goal is to try to delete the buffer:

Del = 

Now Buf ⊨ Prod&Cons&Del but it is not the case that DelBuf ⊨ Prod&Cons&Del. In the first case only Del will be starved out because Buf provides no delete operation, but the client as whole will still be satisfied since Prod&Cons continues to make progress.

In the second case, however, the delete operation may succeed, then causing the client as a whole to deadlock, and thus remain unsatisfied.

What we need to do in order to be sure that DelBuf can be safely substituted for Buf is to *restrict* its behaviour to that allowed by Buf:

Definition 13 $x/Y \xrightarrow{r} x'/Y'$ iff $x \xrightarrow{r} x'$ and $Y \xrightarrow{r} Y'$.

What we mean to capture by x/Y is that some object in state x is restricted to accept only the requests allowed by a second object whose state is some $y \in Y$. We do not know precisely which state the second object is in, so we keep track of the set of possible states.

Usually the initial state of the second object is known, so we will simply write x/y instead of $x/\{y\}$.

Proposition 6 $x{:}{<}y \Rightarrow x/y :< y$.

Proof

(i) $\mathit{traces}(x/y) = \mathit{traces}(x) \cap \mathit{traces}(y)$. But $x{:}{<}y \Rightarrow \mathit{traces}(y) \subseteq \mathit{traces}(x)$,
so $\mathit{traces}(x/y) = \mathit{traces}(y)$.

(ii) $(s,R) \in \mathit{failures}(x/y) \Rightarrow \exists x', x \overset{s}{\Rightarrow} x', \{y\} \overset{s}{\Rightarrow} Y'$, such that $R \cap \mathit{init}(x') \cap \mathit{init}(Y') = \varnothing$
$\Rightarrow R \cap \mathit{init}(x') \cap \cup\{ \mathit{init}(y') \mid y' \in Y'\} = \cup\{ R \cap \mathit{init}(x') \cap \mathit{init}(y') \mid y' \in Y'\} = \varnothing$.
But $x{:}{<}y \Rightarrow \{x\} :<< \{y\} \Rightarrow \exists y' \in Y', \mathit{init}(y') \subseteq \mathit{init}(x')$
$\Rightarrow \exists y' \in Y', R \cap \mathit{init}(y') = \varnothing \Rightarrow (s,R) \in \mathit{failures}(y) \Rightarrow \mathit{failures}(x/y) \subseteq \mathit{failures}(y)$.
But $\mathit{failures}_y(x/y) = \mathit{failures}(x/y)$, so $x/y :< y$. ❑

Finally, the result we want:

Proposition 7 $x{:}{<}y$ and $y \models c \Rightarrow x/y \models c$.

Proof $x{:}{<}y \Rightarrow x/y :< y$ (by proposition 6), so $\mathit{failures}(x/y) \subseteq \mathit{failures}(y)$.
Now $(s,R) \in \mathit{failures}(x/y) \cap \mathit{offers}(c) \Rightarrow (s,R) \in \mathit{failures}(y) \cap \mathit{offers}(c) \Rightarrow R = \varnothing$.
Hence $x/y \models c$. ❑

So, for example, we can conclude that:

DelBuf/Buf ⊨ Prod&Cons&Del

since we effectively *hide* the additional behaviour introduced by DelBuf from the client.

This is not as strong a result as we might have hoped for, but it is a natural consequence of the fact that multiple concurrent clients may interfere with one another if their expectations are not consistent. This is essentially the observation of Liskov and Wing [22] who propose a new definition of subtyping that requires view consistency. Briefly, the idea is that a type that extends the behaviour of another type may only be considered a subtype of the second if the additional behaviour can always be explained in terms of behaviour that was *already* there in the supertype.

In some cases we may get this consistency for free. Note, for example, that if the subtype's behaviour is properly included in the supertype's, in the sense that $\mathit{failures}(x) = \mathit{failures}(x/y)$, then the subtype will be request substitutable for the supertype. We must be sure, though, that the subtype behaviour is consistent with the restriction imposed by the supertype. This leads to the following result:

Proposition 8 If *traces*(x) = *traces*(y) and *failures*(x) $\subseteq$ *failures*(y), then $y \models c \Rightarrow x \models c$.

Proof Follows from proposition 7 since *failures*(x) = *failures*(x/y). ❑

It may still be the case that a subtype provides additional behaviour that does *not* perturb the client. But to be sure that the subtype is truly substitutable, it is necessary to know more about the client's expectations. We have previously explored *interaction equivalence* with respect to the expectations of particular sets of observers, and found that equivalence with respect to all possible observers (also) reduces to failures equivalence [27]. We expect that *relativizing* request substitutability with respect to the expectations of specific classes of clients will lead to more general and more useful results for the case of multiple concurrent clients.

4.8 Open Problems

We have proposed service types as a means of characterizing the services an object provides, and regular types as a means to express non-uniform service availability. In both cases we have presented an approach to subtyping. Furthermore, we have formalized what it means to satisfy a client's expectations, and we have shown the role that subtyping plays in determining substitutability.

Although regular types appear to be a novel and promising approach for reasoning about some of the dynamic (type) properties of concurrent object-oriented programs, there remains much to be studied before we can claim to have a pragmatically acceptable approach for type-checking object-oriented languages. Let us briefly summarize some of these considerations.

4.8.1 Regular Service Types

So far we have treated the typing of services and their availability as orthogonal issues. Service types express types of requests and replies, and regular types tell us when requests are enabled. There is nothing to prevent us from proposing a syntax for regular service types that simply expands request names in regular types to the complete service type specification corresponding to that request. For example, an integer variable could be assigned the regular service type:

```
IntVar = (v1, {   v1 = put(Int)⟶Ok.v2,
                  v2 = put(Int)⟶Ok.v2 + get⟶Int.v2     })
```

Since this is somewhat verbose (the type of the put service must be given twice), it seems more desirable to keep the type specifications of services and their protocols separate.

It is conceivable, however, that the type of a service may itself change with time. In particular, the result types associated with certain requests may depend on the argument types of earlier requests (as is the case with all of the container objects we have seen). To handle this case, it would seem necessary to introduce term variables into regular types to express the dependencies between services in the protocol (i.e. à la “dependent types” [32]). It is not clear, however, what effect this would have on the determination of request substitutability.

It may also be interesting to consider bounded polymorphism in our framework, since the integration of intersection types and bounded polymorphism has been previously studied [31], but only in a functional setting. Finally, we have not considered the issue of recursively defined types, in which the regular type of an object may contain services whose argument and return types refer to the object’s own type. Previous work on “F-bounded” quantification [9] addresses subtyping for such types [3], and is likely to be relevant to our framework.

4.8.2 Applying Regular Types to Object-Oriented Languages

We have presented our type model without giving any concrete interpretation for types. The objects to which we wish to assign types have been described only informally by means of a very general model of objects as transition systems. The next step would be to provide a concrete syntax for objects, either in terms of a programming language or a process calculus that can model objects in a straightforward way.

We have been working towards an *object calculus* that incorporates those features of process calculi that are most needed for expressing the semantics of concurrent object-oriented languages [28]. We intend to use the object calculus as an (executable) abstract machine for a *pattern language for (typed) active objects* [29], and assign regular types to the expressions of this language.

Since the type expressions we are dealing with can become rather unwieldy, it is especially important that we be able to do as much type *inference* as possible. In languages that directly represent abstract states of objects (such as ACT++ [20]) this job will be easier. The main difficulty will be in determining what transitions between the abstract states are possible.

We have already pointed out that objects may satisfy many different regular types, and, since regular types are only approximations, in some cases they may be refined *ad nauseam*. In order to assign regular types automatically to objects, it is necessary to generate some type assignment which is perhaps not the finest possible but which assigns at least one abstract state to every reachable subset of available services. (Recall that our first NDStack was such a minimal representation, whereas NDStack2 had two distinct states with the same services available.)

Another consideration, however, is whether a deterministic regular type can be assigned to an object. If such a type specification exists (e.g. Var and Buf), then this is in any case to be preferred to a non-deterministic regular type that may have less states. Such types not only completely describe service availability for an object, but are well-behaved during type-checking since the sets of reachable nodes for a given trace are always singletons. (So LIST stays small.)

4.9 Concluding Remarks

We have proposed a type framework for object-oriented languages that expresses the *services* of an object as an intersection of *service types* characterizing request and reply messages, and *non-uniform service availability* in terms of *regular types* over a finite number of abstract states associated with subsets of services. Subtyping of regular types is defined by introducing *request substitutability*, a novel pre-order over processes that has special interest for object-oriented applications. Subtyping is easy to determine for regular types, and a simple algorithm is presented. Satisfaction of client's expectations is formalized as *request satisfiability*, and we show how request substitutability relates to it.

A number of technical issues must first be resolved before the framework can be practically applied to real object-oriented languages. In particular, we seek some results that will simplify reasoning about substitutability with respect to multiple concurrent clients.

We expect that it will be easier to reason about regular types in the presence of concurrency if we interpret them either using a temporal logic or a modal process logic (such as Hennessy–Milner logic with recursion [21]). A logical characterization of the concepts we have presented will be the topic of further research.

Despite a number of open research problems, the approach seems to hold a great deal of promise, since numerous tools and algorithms exist not only for analysing properties of finite state processes [11][15][23] but also for reasoning about processes in general [12][19]. This suggests that regular types may be more generally useful for reasoning about temporal properties of concurrent objects.

We have concentrated on client–server-based protocols in which requests eventually entail replies. Can we accommodate other kinds of communication protocols (to support, for example, transactions)? If so, must we modify our model of regular types to incorporate bidirectional communications (instead of just enabling of request channels)? Can we easily accommodate *exceptions* in our framework by, for example, allowing replies to be union types?

Finally, our approach considers only objects with fixed sets of known services. Can we accommodate *reflective* objects that acquire new services with time? In such a setting, would we have to consider not only services, but also types as first-class values?

Acknowledgements

Many thanks are due to José Rolim, Didier Buchs, Laurent Dami, Michael Papathomas, and to the anonymous referees for their help and suggestions in the preparation of the original version of this work published in *OOPSLA '93*. I would also like to thank Benjamin

Pierce, Egil Andersen, William Ferreira and Patrick Varone for their careful reading of the manuscript, and for uncovering various technical errors and deficiencies.

References

[1] Samson Abramsky, “Observation Equivalence as a Testing Equivalence,” *Theoretical Computer Science*, vol. 53, North-Holland, 1987, pp. 225–241.

[2] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[3] Roberto M. Amadio and Luca Cardelli, ‘‘Subtyping Recursive Types,’’ *Proceedings POPL ’91*, pp. 104–118.

[4] Jos C.M. Baeten and Peter Weijland, *Process Algebra,* Cambridge University Press, Cambridge, 1990.

[5] Franco Barbanera and Mariangiola Dezani-Ciancaglini, “Intersection and Union Types,” in *Proceedings Theoretical Aspects of Computer Software (TACS ’91)*, ed. T. Ito and A.R. Meyer, *Lecture Notes in Computer Science* 526, Springer-Verlag, Sendai, Japan, Sept. 1991, pp. 651–674.

[6] Jan A. Bergstra and J.W. Klop, “The Algebra of Recursively Defined Processes and the Algebra of Regular Processes,” in *Proceedings ICALP ’84*, ed. J. Paredaens, *Lecture Notes in Computer Science* 172, Springer-Verlag, Antwerp, pp. 82–95.

[7] Ed Brinksma, Giuseppe Scollo and Chris Steenbergen, ‘‘LOTOS Specifications, Their Implementations and Their Tests,’’ *Protocol Specification, Testing and Verification VI,* ed. G. Bochmann and B. Sarikaya, North Holland, 1987, pp. 349–360.

[8] Stephen D. Brookes, C.A.R. Hoare and Andrew W. Roscoe, “A Theory of Communicating Sequential Processes,” *Journal of the ACM*, vol. 31, no. 3, July 1984, pp. 560–599.

[9] Peter S. Canning, William Cook, Walter L. Hill, John C. Mitchell and Walter G. Olthoff, “F-Bounded Quantification for Object-Oriented Programming,” *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, 11–13 Sept. 1989, pp. 273–280.

[10] Luca Cardelli and Peter Wegner, “On Understanding Types, Data Abstraction, and Polymorphism,” *ACM Computing Surveys*, vol. 17, no. 4, Dec. 1985, pp. 471–522.

[11] Edmund M. Clarke, E. A. Emerson and A. P. Sistla, “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications,” *ACM TOPLAS*, vol. 8, no. 2, April 1986, pp. 244–263.

[12] Rance Cleaveland, Joachim Parrow and Bernhard Steffen, “The Concurrency Workbench,” in *Automatic Verification Methods for Finite State Systems: Proceedings*, ed. Joseph Sifakis, *Lecture Notes in Computer Science* 407, Springer-Verlag, 1989, pp. 24–37.

[13] Elspeth Cusack, ‘‘Refinement, Conformance and Inheritance,’’ *Formal Aspects of Computing*, vol. 3, 1991, pp. 129–141.

[14] Rocco De Nicola, “Extensional Equivalences for Transition Systems,” *Acta Informatica*, vol. 24, 1987, pp. 211–237.

[15] Suzanne Graf and Joseph Sifakis, “A Logic for the Specification and Proof of Regular Controllable Processes of CCS,” *Acta Informatica*, vol. 23, no. 5, 1986, pp. 507–528.

[16] Matthew Hennessy, “Acceptance Trees,” *Journal of the ACM*, vol. 32, no. 4, Jan. 1985, pp. 896–928.

[17] Matthew Hennessy, *Algebraic Theory of Processes*, MIT Press, Cambridge, Mass., 1988.

[18] Hans Hüttel, “Decidability, Behavioural Equivalences and Infinite Transition Graphs,” Ph.D. thesis, ECS-LFCS-91-181, Computer Science Dept., University of Edinburgh, Dec. 1991.

[19] Paola Inverardi and Corrado Priami, “Evaluation of Tools for the Analysis of Communicating Systems,” *Bulletin of EATCS*, vol. 45, Oct. 1991, pp. 158–185.

[20] Dennis G. Kafura and Keung Hae Lee, “Inheritance in Actor Based Concurrent Object-Oriented Languages,” in *Proceedings ECOOP ’89*, ed. S. Cook, Cambridge University Press, Nottingham, 10–14 July, 1989, pp. 131–145.

[21] Kim G. Larsen, “Proof Systems for Hennessy-Milner Logic with Recursion,” in *Proceedings CAAP ’88*, ed. M. Dauchet and M. Nivat, *Lecture Notes in Computer Science* 299, Springer-Verlag, Nancy, March 1988, pp. 215–230.

[22] Barbara Liskov and Jeannette Wing, “A New Definition of the Subtype Relation,” in *Proceedings ECOOP ’93*, ed. O. Nierstrasz, *Lecture Notes in Computer Science* 707, Springer-Verlag, Kaiserslautern, Germany, July 1993, pp. 118–141.

[23] Robin Milner, “A Complete Inference System for a Class of Regular Behaviours,” *Journal of Computer and System Sciences*, vol. 28, Academic Press, 1984, pp. 439–466.

[24] Robin Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

[25] Robin Milner, “The Polyadic π Calculus,” ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, Oct. 1991.

[26] Oscar Nierstrasz and Michael Papathomas, “Viewing Objects as Patterns of Communicating Agents,” *Proceedings OOPSLA/ECOOP ’90, ACM SIGPLAN Notices*, vol. 25, no. 10, Oct. 1990, pp. 38–43.

[27] Oscar Nierstrasz and Michael Papathomas, “Towards a Type Theory for Active Objects,” ACM OOPS Messenger, Proceedings OOPSLA/ECOOP ’90 Workshop on Object-Based Concurrent Systems, vol. 2, no. 2, April 1991, pp. 89–93.

[28] Oscar Nierstrasz, “Towards an Object Calculus,” in *Proceedings of the ECOOP ’91 Workshop on Object-Based Concurrent Computing*, ed. M. Tokoro, O. Nierstrasz and P. Wegner, *Lecture Notes in Computer Science* 612, Springer-Verlag, 1992, pp. 1–20.

[29] Oscar Nierstrasz, ‘‘Composing Active Objects,’’ *Research Directions in Concurrent Object-Oriented Programming*, ed. G. Agha, P. Wegner and A. Yonezawa, MIT Press, Cambridge, Mass., 1993, pp. 151–171.

[30] Michael Papathomas, “A Unifying Framework for Process Calculus Semantics of Concurrent Object-Oriented Languages,” in *Proceedings of the ECOOP ’91 Workshop on Object-Based Concurrent Computing*, ed. M. Tokoro, O. Nierstrasz and P. Wegner, *Lecture Notes in Computer Science* 612, Springer-Verlag, 1992, pp. 53–79.

[31] Benjamin C. Pierce, ‘‘Intersection Types and Bounded Polymorphism,’’ *Conference on Typed Lambda Calculi and Applications*, March 1993, pp. 346–360.

[32] Simon Thompson, *Type Theory and Functional Programming*, International Computer Science Series, Addison-Wesley, 1991.

[33] Vasco T. Vasconcelos and Mario Tokoro, ‘‘A Typing System for a Calculus of Objects,’’ *Object Technologies for Advanced Software, First JSSST International Symposium*, *Lecture Notes in Computer Science*, vol. 742, Springer-Verlag, Nov. 1993, pp. 460–474.

[34] Peter Wegner and Stanley B. Zdonik, “Inheritance as an Incremental Modification Mechanism or What Like Is and Isn’t Like,” in *Proceedings ECOOP ’88*, ed. S. Gjessing and K. Nygaard, *Lecture Notes in Computer Science* 322, Springer-Verlag, Oslo, Aug. 15–17, 1988, pp. 55–77.