PROTOCOL ASSURING UNIVERSAL LANGUAGE

Rick van Rein
Maarten M. Fokkinga

University of Twente, Dept INF
PO Box 217, NL 7500 AE Enschede, the Netherlands
{vanrein,fokkinga}@cs.utwente.nl

Abstract: Conventionally, interfaces of objects export a set of messages with their types, and suggest nothing about the order in which these services may be accessed. This leaves room for a large number of run-time errors or misbehaviours in type correct designs. To mend this, we introduce the notion of *protocol*, expressing offered and expected orderings of messages, along with a notion of protocol correctness. We do this by defining the Protocol Assuring Universal Language Paul, which describes protocol aspects of classes, and a semantics of in terms of CSP.

Keywords. object orientation, interface, behaviour, role, protocol, process, type checking, CSP, Paul.

1 INTRODUCTION

Interfaces exported by objects are usually unsorted collections of messages, annotated with types. The types help in assuring type correctness, either at run-time or compile-time, depending on the language. Type correctness is considered important because it eliminates a large amount of conceptual errors from designs. This is also why dynamically binding backbone architectures, such as CORBA [20], offer typed interfaces (IDL in the case of CORBA).

Type disciplines enable generic reasoning on data aspects of designs. The care with which type checking establishes integrity of data manipulations is in sharp contrast with the ad hoc approaches with which process aspects are verified in designs. This paper is a (admittedly small) step on the road of integrating process aspects into object oriented designs more systematically. The specific aspect of processes dealt with in

this paper is the order in which invocations over object interfaces take place. In this paper we refer to such an ordering aspect with the term *protocol*.
We will demonstrate the failure of type checking to detect certain conceptual errors that *can* be detected with protocol checking. We will introduce a formalism for protocol correctness inside a single class as well as between classes, together resulting in protocol correctness of an entire application.

> **Example:** Throughout this paper, we use the example of a bank account, which is accessed by an automated teller machine, or ATM, given in Figure 1. The

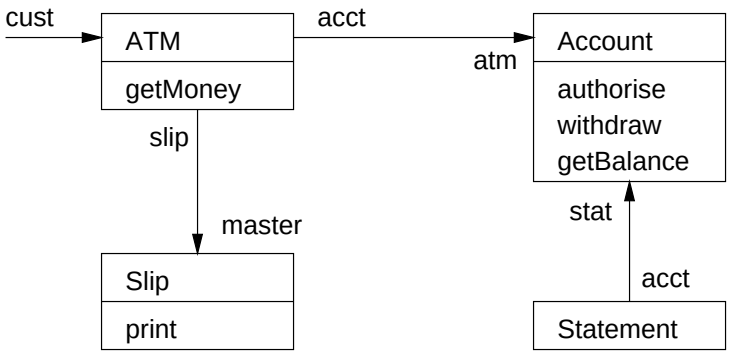

Figure 1 Class diagram for the banking example

> application is an ATM that accesses an Account to withdraw money from it. When this is done, a withdrawal slip is printed by the Slip class. Independent of this, the Account may be accessed by the Statement class in an attempt to retrieve the current balance for printing a monthly statement.

Normally, associations between classes represent a data relation, which is a thought that originates from the entity-relationship background of object orientation. In this paper however, we will interpret an association as a process, that consists of messages passing over the association, and is described by the protocols at both sides of the association. We limit ourselves to directed 1:1 associations (*from* the invoking client *to* the invoked server, though with a provision for *feedback* from the server to the client); in the diagrams they are drawn as arrows.
Lacking a specification of message order on the interfaces, no algorithm can ever decide whether classes can be associated in such a way that expected and offered orders match. We therefore believe we introduce a fundamental improvement of interface descriptions by extending them with *protocols*, or *order descriptions of messages*.

> **Example:** Consider the messages supported by the Account class. These include authorise and withdraw. The conventional interface towards the environment does not reveal *any* required or allowed order. Only by applying real world knowledge is it obvious that an authorise must be invoked before a withdraw.

But even knowing this, after an authorise and a subsequent withdraw, it is undefined whether another withdraw is possible without a preceding authorise. This varies in the real world, thereby introducing a possible problem when the Account and ATM are not implemented at the same time by the same person.

A single class may offer different behaviours to different clients; it plays, so to say, different *roles* for different clients. Therefore, a class in general has several roles and each role is an interface (including a protocol). The idea of multiple roles on classes is not new; see [11] [14] [15].

Example: Class Statement should be allowed to retrieve balance information from Account. Some banks do not allow an ATM to retrieve such information, for reasons of privacy and security. We model this with role Account:atm which is specifically meant for the association to the ATM. Since a role describes which messages are acceptable at what time, there must be a separate protocol for each role.

Protocols have long since been used in the OO world informally, hardly supported by language, theory and tools. Life cycles, for instance, are widely used in OO developments; our protocols *are* life cycles per role. As another example, consider interaction diagrams in the design patterns world. For many of the patterns, Gamma et al. [7] give type specifications with comments, but because that is not sufficient to understand the class, they give an interaction diagram. If an interaction diagram is so important and useful, why is not something like it part of the language? Our protocols will do.
Note that, in general, not only the method names in the history of method calls, but also the parameter values determine the state of an object, and therefore what next messages are expected and allowed. Our approach covers this situation, since each element (formally called 'message name') in a protocol may be interpreted as a method name *plus* its parameter values.

Before delving into the details, let us first describe informally the main concepts, their names, and their interrelations. We call the interfaces that offer services to client classes the *export roles* of the server class. At the other end of the association, the client side, there is an *import role*. In Paul, each role has its own protocol. In addition, each class has a class protocol, describing the life cycle of the class' objects.
Seen from within an association, the protocol I of an import role expresses the behaviour that is expected from the client; the protocol E of an export role expresses behaviour that is offered by the server. An important check performed on Paul designs is whether I *correctly uses* E, formally $I \leq E$. All such checks together assure *association protocol correctness* of a design.
Seen from within a class, a protocol E of an export role is a perspective on the class protocol C; and C in turn describes in what order the class' methods will be called, and thus how use is made of an import protocol I. A check done on Paul designs is whether E is a correct use of C, and whether C via its methods correctly uses I, both being formalised by means of $\leq$ again. All such checks together assure *class protocol correctness*.

The combination of association and class correctness leads to *overall protocol correctness* of a design.
By deliberate choice we want to do all checking locally. Consequently, when an object has several export roles, there is no check whether the messages over the associations arrive at the object arbitrarily interleaved or in a specific order determined by the environment. In order to relieve the designer from the obligation to choose each class protocol as the interleaving of its export protocols, we will assume that each object has a *scheduler*: it lets the messages over the (protocol correct!) links pass into the object in a way that is acceptable to the class protocol. The scheduler is to be generated *automatically*.
Finally, in our setting, messages have no return values. In order to express *feedback*, we will interpret a nondeterministic choice $a.p + a.q$ as a single message a with two return values; one leading to continuation p and the other to q. This interpretation is nicely formalised with CSP [12], and fits well with conventional use of state transition diagrams, as explained in the sequel.

Our work has currently several limitations that need be lifted in order to make our results more practically useful. First note that from protocol correctness of a design it follows that anything that does happen, is protocol correct, but there is no guarantee of proper termination or freedom of deadlock. These latter aspects are related to protocol checking, but require a global analysis of the design; our current checking (implemented by the tool `paul`) is done locally. To relieve the object oriented designer of this global perspective, we intend to address this issue in our future work. Another future topic is to release the current restriction that associations may only be 1:1 relations, and that the configuration is static.

The remainder of this paper is structured as follows. Section 2 introduces the syntax and intuitive meaning of Paul, including the formulation of part of the ATM example. Section 3 gives a formal semantics to Paul, by translating protocols to CSP. Then Section 4 defines protocol checking. The final sections discuss related and future research, and draw conclusions.

This work has been performed in the scope of the Quantum project, in which Compuware's UNIFACE lab and the University of Twente cooperate. UNIFACE is a leading component-based development tool for mission-critical applications. The goal of our research is to add the protocol concept to UNIFACE's component model.

2 DEFINITION OF PAUL

"Programs" in Paul express protocol aspects of designs (so we will speak of *design* rather than program). That means Paul contains no things like int variables or addition operators, but everything that plays a role in protocols is expressed in Paul. There are looping and choice constructs (abstracting from the actual conditions in those constructs), there are message sends, there are (behavioral) links between objects, and there are import and export roles.
A design in Paul contains several protocols: one for each class, one for each import role and one for each export role, and the main task of the Paul protocol checker is

to ensure the absence of conflicts between these protocols. The checking rules are discussed in Section 4; this section introduces only the syntax of Paul.

Syntax. In Paul's grammar we use C for class names, R for role names (both imported and exported), M for message names, and X for protocol variables; their syntax is not elaborated. Symbol ? denotes a test whose outcome is determined by the method (client) itself, whereas the outcome of a test $R.M()$ is determined by the server. We use '. . . ' to denote zero or more occurrences of the preceding nonterminal. The grammar is given in Figure 2.

design	::=	*def* . . .	
def	::=	*class* | *assoc*	
assoc	::=	$C : R$ — $C : R$	// association between roles
class	::=	class C : *proto* is *decl* . . . end	
decl	::=	exports R : *proto*	// export of a role, as a server
	|	imports R : *proto*	// import of a role, as a client
	|	method $M()$ is *stat* end	// method definition
stat	::=	while *test* do *stat* end	// while loop
	|	if *test* then *stat* else *stat* end	// if statement
	|	*stat*; *stat*	// sequential composition
	|	invoke $R.M()$	// method invocation
	|	ε	// (no text) no action
test	::=	$R.M()$ | ?	// decision information retrieval
proto	::=	(*proto**)	// arbitrary repetition
	|	(*proto* . *proto*)	// sequence
	|	(*proto* + *proto*)	// choice, made by client when deterministic
	|	letrec *pdef* . . . in *proto* end	// scoping of definitions
	|	var X	// protocol variable use
	|	M	// message send
pdef	::=	X = *proto*;	// recursive protocol definition

It is required that recursion in protocols (via letrec) is tail recursion only.

Figure 2 Paul's grammar

In the current version of Paul, methods are parameterless and resultless. The conditions that control the loops and conditional statements are abstracted out of Paul when they are based upon data. Thus we cannot distinguish between 'while (3>2) *s*' and 'while (3<2) *s*': both are represented in Paul by 'while ? do *s* end'. Method invocations as tests are not abstracted away; here it is the server who decides the outcome.
General recursion is not possible for method definitions *within* a class, since each method invocation 'invoke $R.M()$' in a method body refers to method M via role R.
We shall leave out some parentheses in protocols under the convention that the binding strength of the operators is given in decreasing order in Figure 2; so . binds stronger than +.

Example: Figure 3 expresses the classes ATM and Account in Paul.
Consider class ATM. In its first line the class protocol is declared; it describes the possible orders in which its own method getMoney is called. (Here the class protocol getMoney* is rather trivial, but in class Account it is not.) The protocols of the import roles slip and acct are declared with an imports clause. The "typing" acct:(authorise.withdraw*)*, for instance, expresses the possible orders of invocations of the imported methods acct.authorise and acct.withdraw. We shall later see that this acct protocol does not correctly use the protocol of Account's export role atm. The protocol for the export role cust is declared with an exports clause; it constrains the order in which a client may invoke getMoney on the ATM.
The bottom line defines the association between the two classes.

```
class ATM:getMoney* is
   imports slip:print*
   exports cust:getMoney*
   imports acct:(authorise.withdraw*)*
   method getMoney () is
      invoke acct.authorise ();
      while ? do
         invoke acct.withdraw ();
         invoke slip.print ();
      end
   end
end

class Account:(authorise.withdraw + getBalance)* is
   exports atm:(authorise.withdraw)*
   exports stat:getBalance*
   method . . .
end

ATM:acct — Account:atm
```

Figure 3 Definition of the ATM and Account class, and their association

Given the class definitions only, a Paul protocol checker can verify for *class* protocol correctness. *Association* protocol correctness can be checked when the associations are known too, as in the bottom line of Figure 3. A complete Paul *design* is a combination of class and assoc definitions.

Example: The check performed for association ATM:acct — Account:atm compares protocol (authorise.withdraw*)* of the import role acct of class ATM with protocol (authorise.withdraw)* of the export role atm of class Account. We shall see that the check yields the result 'incorrect.'

The check performed within class ATM between its export protocol and the class protocol, compares getMoney* with getMoney*. The outcome will be 'correct'.

The check performed within class ATM between its class protocol *together with* the method body at one hand and the import role acct at the other hand, compares the protocol $(\mathsf{authorise}.(\mathsf{withdraw}.\mathsf{print})^*)^*$ —with concealment of print— with the protocol $(\mathsf{authorise}.\mathsf{withdraw}^*)^*$. The outcome will be 'correct'.
The same check for the import role slip compares $(\mathsf{authorise}.(\mathsf{withdraw}.\mathsf{print})^*)^*$ —with concealment of authorise and withdraw— with print*. The outcome will be 'correct'.

3 TRANSLATING PAUL TO CSP

In the next section protocol checking is defined in terms of CSP processes. CSP is the well-known theory of Communicating Sequential Processes [12, 25]. In this section we define the translation of Paul protocols to CSP processes. We start with a very brief exposition of CSP (just enough to follow the main line of our exposition), and an explanation of how feedback from server to client is realised in Paul.

CSP. CSP is an elaborate theory about possibly nondeterministic processes, with a precisely defined notion of equality. Processes P, Q, R are built, at least syntactically, from *event*s a, b, c by means of the *then* operator $\rightarrow$ (having the highest priority in parsing), and two "branching" constructs $\Box$ and $\sqcap$. An important operator is the *concurrency* (or in-parallel-communicating) operator $\|$.
An event denotes "something that may happen," and a communication between two processes is defined as the occurrence of the same event in both processes. In the process $P = a{\rightarrow}Q \,\Box\, b{\rightarrow}R$ it is the environment (the communicating counter-part of P) which influences the branch that P will take; thus $\Box$ denotes *external choice*. In the process $P = a{\rightarrow}Q \sqcap b{\rightarrow}R$ it is P itself that makes the choice; thus $\sqcap$ is called *internal choice*. There are several laws describing the semantics more precisely; in particular:

$$(a{\rightarrow}P) \,\Box\, (a{\rightarrow}Q) \;=\; a{\rightarrow}(P \sqcap Q) \tag{1.1}$$

$$(a{\rightarrow}P) \sqcap (a{\rightarrow}Q) \;=\; a{\rightarrow}(P \sqcap Q) \tag{1.2}$$

Each process P has an alphabet, denoted αP, consisting of all the events in which it possibly can participate. Concealment, or hiding, of a set A of events is denoted $P\backslash A$, and means that the events A within P occur autonomously without an opportunity for the environment to participate.
Succesful termination of a process is represented by the event $\surd$. Because of its interpretation, the event often gets special treatment. The process that just terminates successfully is denoted Skip; so $\mathsf{Skip} = \surd{\rightarrow}\ldots$, where the remainder process on the ellipses is irrelevant.
A recursively defined process like $X = \cdots X \cdots$ is written $\mu X \bullet \cdots X \cdots$. For example, $\mu X \bullet a{\rightarrow}X$ is the process that is only and forever able to participate in event a.
The mathematical semantics of the constructs and notions above is described in terms of *traces* and *failures*. A trace of a process is a sequence of events in which the process

may possibly participate, in succession, during a run. A failure A of a process P after a trace s, denoted $A \in \mathit{Failures}(P, s)$, is a set of events that P may refuse to participate in after trace s. Of particular importance are the facts (definitions!) that for the empty trace ε:

$$\begin{aligned} \mathit{Failures}(P \sqcap Q, \varepsilon) &= \mathit{Failures}(P, \varepsilon) \cup \mathit{Failures}(Q, \varepsilon) \\ \mathit{Failures}(P \,\Box\, Q, \varepsilon) &= \mathit{Failures}(P, \varepsilon) \cap \mathit{Failures}(Q, \varepsilon) \end{aligned}$$

Feedback. Recall that in the version of Paul considered in this paper all associations are directed, from client to server, meaning that the messages over the associations are sent by the client and received by the server. Moreover, in our formalism messages do not return resulting values. Together with the approach to perform checks locally, this implies that a dialogue between two objects will be hard to model in a way that passes the protocol checks.
Our solution to this weakness is to model the process aspects of *feedback* from server to client. This is achieved by our interpretation of nondeterministic choice in a protocol (typical example: a sub-expression $a.p + a.q$) as a server choice, and of other choices (such as $a.p + b.q$ with distinct a and b) as client choices. This interpretation fits well with conventional use of state transition diagrams, as shown by Figure 4 below. Thus the designer, the Paul user, is freed from explicitly indicating at various places (export, class, and import protocol) whether the choice is a client choice or a server choice.

Let us show this feedback by two typical examples (in anticipation of the formal definitions below). Consider the first state transition diagram in Figure 4. It expresses the protocol for opening a file and reading to end-of-file: the two eof-transitions indicate a single invocation of the eof-method with a negative and a positive return value, respectively. Here the decision whether the end of file has been reached (the negative or positive return value) is be made by the server, the object that exports the methods to access the file. After our translation into CSP, this is reflected by an internal choice $\sqcap$ in the server side protocol.
Now consider the second state transition diagram in Figure 4. It expresses the protocol for writing a series of data onto a file. There is no nondeterminism: the decision whether to continue writing is made by the client, the object that imports the write method. After our translation into CSP, this is reflected by an external choice $\Box$ at the *server* side of the association.

About the translations. The translation from Paul to CSP is in principle a straightforward replacement of Paul symbols by CSP symbols, and is easily definable by induction on the structure of protocols. There are, however, a few points that need special attention, and that complicate the definition:

- Paul's operator . expects a protocol as left and right operand, whereas the CSP operator $\rightarrow$ has an event at the left and a process at the right. The technique of continuations is used here to express the translation: the translation function gets an additional parameter, called the continuation, denoting "what comes after the protocol to be translated." In this way, protocol $m.n$ can be translated to

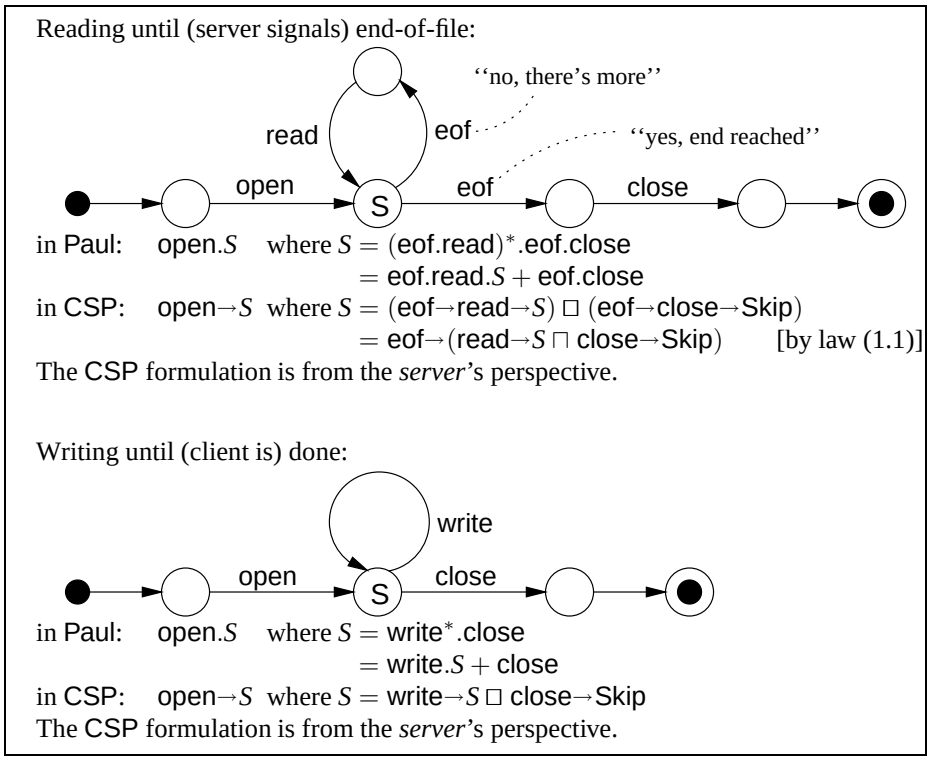

Figure 4 Protocols with/without feedback

$m{\rightarrow}(n{\rightarrow}P)$, given that P is the translation of what comes after the protocol $m.n$. At the very end of the protocol, we take Skip as continuation.

- In CSP there is a syntax constraint that Skip must not appear unguarded as an operand of $\Box$ when this occurs to the left of the sequential composition. Thanks to the technique of continuations there appears no sequential composition in the translation, and such unwanted Skips are avoided.

- The nondeterminism in an protocol like $a.p + a.q$ is meant to denote a choice made by the server, irrespective of whether the protocol occurs in the server or in client. Function $\mathcal{T}$ translates the protocol into a *server* process, and operation $\overline{\ }$ (as in $\overline{P}$, pronounced *switch P*) changes the viewpoint from server to client, or vice versa.

- We have already said that in a protocol some choices are client choices whereas other choices are server choices. This holds for protocol checks between an import and export role, and between an export and a class protocol. But when a class protocol is related to the import roles, then *all* its choices are client choices (with the class being the "client," and the import role being the "server"); here the server choices arise *only* from within the method bodies. This translation

is realised by means of a function $\mathcal{M}$. At the same time, $\mathcal{M}$ takes care of incorporating the method bodies (hence the name $\mathcal{M}$).

We shall now define the functions $\overline{\ }$, $\mathcal{T}$, $\mathcal{S}$ (for statements), and $\mathcal{M}$, in that order. The latter three translate Paul into CSP, the former works within CSP.

Switching of perspective. To change the perspective from server to client or vice versa we now define overbar operation $\overline{\ }$ (pronounced *switch*). Thanks to this operation, one translation $\mathcal{T}$ from protocols to processes suffices, both for 'server side' protocols and for 'client side' protocols. In principle, operation $\overline{\ }$ simply interchanges the choices $\Box$ and $\sqcap$ in a CSP process. The exchange is defined as a purely syntactic operation, and must only be applied when the process has the so-called *determined choice form*, that is, no sub-expression has the form of the left hand sides of CSP laws (1.1) or (1.2) (repeated below), nor can it be brought in this form using associativity and commutativity of $\Box$ and of $\sqcap$:

$$(a{\rightarrow}P) \,\Box\, (a{\rightarrow}Q) \quad = \quad a{\rightarrow}(P \sqcap Q) \tag{1.1}$$

$$(a{\rightarrow}P) \sqcap (a{\rightarrow}Q) \quad = \quad a{\rightarrow}(P \sqcap Q) \tag{1.2}$$

There are two reasons for the condition on the form on which the replacement takes place. First, without the condition, the replacement would sometimes not have the intended effect; for example, the replacement in the left hand side of (1.2) has no semantic effect due to law (1.1). Second, without the condition on the form, semantically equal expressions would result in semantically distinct expressions; for example, this would happen when the replacement takes place throughout law (1.1), or (1.2).
Thus the definition of operation $\overline{\ }$ on CSP processes reads:

> repeatedly apply laws (1.1) and (1.2) from left to right
> (until no subexpression has the form of the lhs of these laws
> modulo associativity and commutativity of $\sqcap$ and $\Box$);
> replace each $\Box$ by $\sqcap$, and each $\sqcap$ by $\Box$.

Clearly, the operation satisfies the property that $\overline{\overline{P}} = P$. A particularly noteworthy idiom that we shall use is:

$$\overline{\overline{P} \sqcap \overline{Q}}$$

In this way we express a "really external choice between P and Q," even though P and Q may start with the same event. For instance, take $P = a{\rightarrow}P'$ and $Q = a{\rightarrow}Q'$. Then $P \,\Box\, Q$ equals $a{\rightarrow}(P' \sqcap Q')$, due to law (1.1), thus failing to express an external choice. However, assuming that P' and Q' do not start with the same event, we have:

$$\begin{array}{lcll}
\overline{\overline{a{\rightarrow}P'} \sqcap \overline{a{\rightarrow}Q'}} & = & \overline{a{\rightarrow}\overline{P'} \sqcap a{\rightarrow}\overline{Q'}} & \\
& = & \overline{a{\rightarrow}(\overline{P'} \sqcap \overline{Q'})} & \text{[by law (1.2)]} \\
& = & a{\rightarrow}(\overline{\overline{P'} \sqcap \overline{Q'}}) & \\
& = & a{\rightarrow}(\overline{\overline{P'}} \,\Box\, \overline{\overline{Q'}}) & \\
& = & a{\rightarrow}(P' \,\Box\, Q') &
\end{array}$$

So the choice is really external.

Paul protocol to CSP process. $\mathcal{T}$ is the function that translates a Paul protocol (*proto* in Figure 2) to a CSP process with CSP's $\Box$ for Paul's $+$. As explained above, it is defined by induction on the structure of the protocol, using a continuation parameter P for "what comes after the protocol":

$$\begin{aligned}
\mathcal{T}(m,P) &= m{\rightarrow}P \\
\mathcal{T}(p.q,P) &= \mathcal{T}(p,\mathcal{T}(q,P)) \\
\mathcal{T}(p+q,P) &= \mathcal{T}(p,P)\,\Box\,\mathcal{T}(q,P) \\
\mathcal{T}(p^*,P) &= \mu X \bullet \mathcal{T}(p,X)\,\Box\,P \\
\mathcal{T}(\textsf{letrec } X{=}p \textsf{ in } q \textsf{ end},P) &= \mathcal{T}(q[\textsf{var } X/(\mu X \bullet p)],P) \\
\mathcal{T}((\mu X \bullet p),P) &= \mu X \bullet \mathcal{T}(p,P) \\
\mathcal{T}(\textsf{var } X,P) &= X
\end{aligned}$$

In the letrec clause, each occurrence of var X within q is replaced by the recursive protocol $\mu X \bullet p$ (which requires an extension of the syntax with μ expressions). In the last clause, we see that a recursive call *recurs* and therefore discards the continuation P.
Observe that $\mathcal{T}(a.p + a.q, \textsf{Skip}) = (a{\rightarrow}P)\Box(a{\rightarrow}Q) = a{\rightarrow}(P{\sqcap}Q)$, where $P{=}\mathcal{T}(p,\textsf{Skip})$ and $Q = \mathcal{T}(q,\textsf{Skip})$, as required.
A Paul protocol p in isolation is then translated to CSP as: $\mathcal{T}(p,\textsf{Skip})$.

Paul statement to CSP process. $\mathcal{S}$ is the function that translates a Paul statement (*stat* in Figure 2) to a CSP process (where 'internal' is 'client side'). Here we have to bear in mind that a test ? denotes a client choice ($\sqcap$), whereas a test $R.m()$ denotes a server choice. The latter is expressed by means of the idiom $\overline{\overline{P} \sqcap \overline{Q}}$ explained above. For simplicity we assume that the names of all imported methods are distinct, so that the role name R in $R.m()$ is superfluous. With these remarks in mind, the definition suggests itself:

$$\begin{aligned}
\mathcal{S}(\varepsilon,P) &= P \\
\mathcal{S}(R.m(),P) &= m{\rightarrow}P \\
\mathcal{S}(s;\ t,P) &= \mathcal{S}(s,\mathcal{S}(t,P)) \\
\mathcal{S}(\textsf{if ? then } s \textsf{ else } t \textsf{ end},P) &= \mathcal{S}(s,P)\ \sqcap\ \mathcal{S}(t,P) \\
\mathcal{S}(\textsf{if } R.m() \textsf{ then } s \textsf{ else } t \textsf{ end},P) &= m{\rightarrow}\ \overline{\overline{\mathcal{S}(s,P)}\ \sqcap\ \overline{\mathcal{S}(t,P)}} \\
\mathcal{S}(\textsf{while ? do } s \textsf{ end},P) &= \mu X \bullet\ \ \mathcal{S}(s,X) \sqcap P \\
\mathcal{S}(\textsf{while } R.m() \textsf{ do } s \textsf{ end},P) &= \mu X \bullet\ \ m{\rightarrow}\ \overline{\overline{\mathcal{S}(s,X)} \sqcap \overline{P}}
\end{aligned}$$

The translation of a statement s in isolation is defined to be: $\mathcal{S}(s,\textsf{Skip})$.

Paul class protocol plus methods to CSP process. In order to compare a class protocol with the import protocols, the method definitions have to be taken into account. Function $\mathcal{M}$ translates a class protocol *together with the method definitions* to a CSP process. We assume that for each method name m the body is given by *body m*.

The definition differs from $\mathcal{T}$'s definition only in the clause for a method name *m* and in the use of $\sqcap$ instead of $\Box$, since, as explained earlier, when a class protocol is related to its import roles, the class is considered the client and the import role is considered the server:

$$\begin{aligned}
\mathcal{M}(m,P) &= \mathcal{S}(body\ m,P)\\
\mathcal{M}(p.q,P) &= \mathcal{M}(p,\mathcal{M}(q,P))\\
\mathcal{M}(p+q,P) &= \mathcal{M}(p,P)\sqcap\mathcal{M}(q,P)\\
\mathcal{M}(p^*,P) &= \mu X\bullet\mathcal{M}(p,X)\sqcap P\\
\mathcal{M}(\mathsf{letrec}\ X{=}p\ \mathsf{in}\ q\ \mathsf{end},P) &= \mathcal{M}(q[\mathsf{var\ X}/(\mu X\bullet p)],P)\\
\mathcal{M}((\mu X\bullet p),P) &= \mu X\bullet\mathcal{M}(p,P)\\
\mathcal{M}(\mathsf{var\ X},P) &= X
\end{aligned}$$

The translation of a class protocol *p* in isolation is defined to be: $\mathcal{M}(p,\mathsf{Skip})$.

4 PROTOCOL CHECKING

Within the framework of CSP, a relation *correctly uses*, denoted $\leq$, is defined between processes. The checks performed for each association, and for each class, are based on the relation $\leq$.
In order to avoid complications in the formulas, we assume that within each class the alphabets of the export protocols are disjoint and contained in the alphabet of the class protocol, and similarly that the alphabets of the import protocols are disjoint and jointly contain all method names in the method bodies of the class. Relaxing the disjointness assumption would introduce some explicit renaming in the formulas and thus obscure the essentials.
Recall further that another, serious, restriction is that associations are 1:1 only. Relaxation of this restriction will be addressed in our future work.

Association correctness. Consider an association with protocols *C* and *S* at the end points. Both *C* and *S* have been obtained by translating the Paul protocols into CSP expressions by function $\mathcal{T}$. We shall assume $\overline{C}$ as a correct description of the client's behaviour over the association, and *S* as that of the server: $\overline{C}$ and *S* correctly describe independently what messages occur and who takes decisions. (This assumption is checked for in the paragraphs on Class correctness below.)
The question is now: when does the client correctly use the server? The answer is: when during each possible execution of $\overline{C}\parallel S$ (in which $\overline{C}$ and *S* proceed in a lock-step synchronised way, making the decisions as indicated by $\Box$ and $\sqcap$) it never happens that a party insists on participating in another action, yet the other party can refuse the actions offered. So, as long as there is no successful termination, the concurrent composition makes progress, and semantically there is no deadlock in this composition (though deadlock may still arise in the entire system). In CSP terminology, given that *C* and *S* have the same alphabet, *A* say, we define that *C correctly uses S* precisely when $\overline{C}\parallel S$ does not have a pair (s,A) in its failures whenever trace *s* has not signalled

successful termination:

$$C \leq S \quad \equiv \quad \overline{C} \leftrightarrow S \tag{1.3}$$

$$P \leftrightarrow Q \quad \equiv \quad \forall s \bullet \quad \langle\surd\rangle \text{ in } s \quad \vee \quad (s, A) \notin \mathit{Failures}\,(P \parallel Q) \tag{1.4}$$

$$\text{where } A = \alpha P = \alpha Q$$

Relation $\leftrightarrow$ is pronounced 'co-operate well'. So, leaving the translation $\mathcal{T}$ from Paul to CSP implicit, we define that the check performed for each association is:

$$C \quad \leq \quad S$$

where C is the client's import protocol, and S is the server's export protocol.

Example: To illustrate the relation $\leq$, here are some examples:

$$\begin{array}{rcl}
\mathsf{authorise} & \leq & \mathsf{authorise} + \mathsf{getBalance} \\
\mathsf{authorise} + \mathsf{getBalance} & \not\leq & \mathsf{authorise} \\
\mathsf{authorise} + \mathsf{getBalance} & \leq & \mathsf{authorise} + \mathsf{getBalance} \\
\mathsf{authorise.}(\mathsf{withdraw} + \mathsf{getShot}) & \not\leq & \mathsf{authorise.getShot} \\
\mathsf{authorise.withdraw} + \mathsf{authorise.getShot} & \leq & \mathsf{authorise.getShot} \\
(\mathsf{authorise.withdraw}^*)^* & \not\leq & (\mathsf{authorise.withdraw})^*
\end{array}$$

The latter example is the faulty association ATM:acct — Account:ATM in the ATM example. When translated to CSP its client and server side protocols $\overline{C}$ and S become (with a for authorise, and w for withdraw):

$$\begin{array}{rcl}
\overline{C} & = & (a \rightarrow P) \sqcap \mathsf{Skip} \quad \text{where} \quad P = (a \rightarrow P) \sqcap (w \rightarrow P) \sqcap \mathsf{Skip} \\
S & = & (a \rightarrow w \rightarrow S) \,\Box\, \mathsf{Skip}
\end{array}$$

Taking trace s to be $\langle a \rangle$, we see that $\neg(\langle\surd\rangle \text{ in } s)$ and $(s, \{a, w, \surd\}) \in \mathit{Failures}\,(\overline{C} \parallel S)$. This proves $C \not\leq S$, meaning that the ATM example contains a protocol error.

Class correctness I. Here we consider what relation should hold between the export protocols and class protocol (in order to declare the class protocol correct), and in the next paragraph the relation between the class protocol and import protocols.
Suppose a class with class protocol $a.b$ has two export roles with protocols a and b, respectively. See the left side of Figure 5. Clearly, serving b first followed by a is in conflict with the class protocol. Even though it might follow from the actual environment that a will occur before b, it seems that the class is *not* protocol correct since, by deliberate choice, protocol correctness is a local property.
Our solution to this weakness is to assume an implicit *scheduler* in a class. A scheduler of protocols P and Q is an object that is able to serve *all* interleavings of P and Q, and that only requests *some* interleaving of these from its server. (It does so, apparently, by alternately holding messages of some roles while propagating messages of other roles.) Formally, a scheduler of P and Q has export protocols P and Q, class protocol $P \parallel\!\parallel Q$ (P *interleaved* with Q), and an import protocol R that satisfies $P = R \backslash \alpha Q$ and $Q = R \backslash \alpha P$;

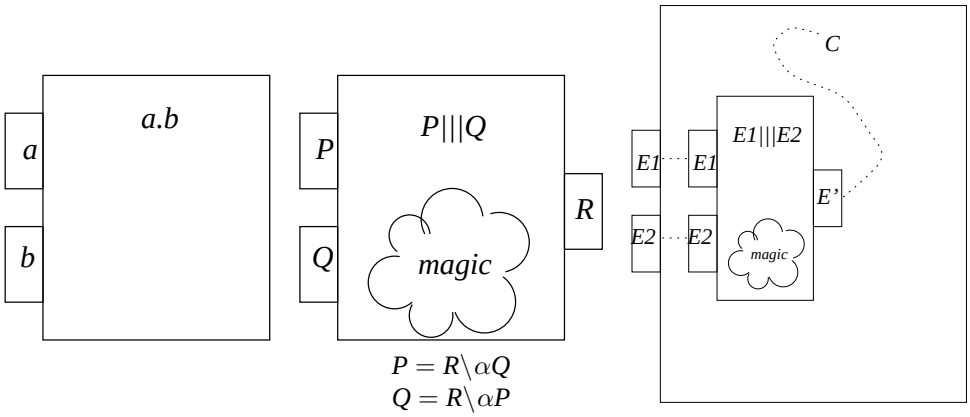

Figure 5 Left: a class; mid: a scheduler; right: class with scheduler

see Figure 5. The internals of the scheduler are just *magic* and of no concern to the designer: it is to be generated *automatically*. For example, the tuple $(P, Q;\ R)$ with P=*a*, Q=*b* and R=*a*.*b* specifies a scheduler: assuming that it is implicitly placed into the class as in the left of Figure 5, the class is protocol correct regarding the export roles and class protocol. Thus, once more, the schedulers must occur in an actual object system, and so our object systems differ from what is conventional.
The formal correctness requirement between the export protocols $\vec{E}$ and the class protocol C reads now as follows: there exists some scheduler $(\vec{E};\ E')$ that serves all interleaving of the class' export protocols and whose request correctly uses C:

$$\exists E' \bullet (\forall i \bullet E_i = E' \restriction \alpha E_i) \wedge E' \leq C \tag{1.5}$$

Here we use $P \restriction X$ as an abbreviation for $P \backslash (\alpha P - X)$. Again we have left the translation $\mathcal{T}$ from Paul to CSP implicit.

Class correctness II. Now we consider the protocol correctness requirement regarding the class protocol C and the import roles $\vec{I}$. This will again be expressed in terms of correct use of protocols derived from C and $\vec{I}$.
Recall that all choices expressed within C are decided by either the clients of this class or by the class' internal workings; only choices within the method bodies can possibly be decided through the import roles. This interpretation is given by translation $\mathcal{M}$.
Now, observe that the actual use of the class' methods is described by the class protocol C. The import roles $\vec{I}$ say what services have been imported, for use by the method bodies. So, process $\mathcal{M}(C)$ should co-operate well with the processes of the import roles. Given that the import roles have disjoint alphabets, this requirement reads:

$$\overline{\mathcal{M}(C) \restriction \alpha I_i} \quad \leq \quad I_i \qquad\qquad \text{for all } i \tag{1.6}$$

Note the use of $\overline{\ }$ to change the class' protocol from client side to server side protocol, which is expected by the $\leq$ relation.

Implementation. An implementation of Paul exists. The syntax from Figure 2 (with some trivial extensions) is supported by Paul 1.0. The protocol checker can be obtained from ftp://ftp.cs.utwente.nl/pub/doc/Quantum/Paul.
The tool prints a counter-example in case of detected protocol errors. For the faulty associations under paragraph "Association correctness," the counter-examples generated are $\langle$getBalance . . .$\rangle$, $\langle$authorise.withdraw . . .$\rangle$ and $\langle$authorise$\rangle$, respectively.

5 RELATED WORK

Protocols. Our work on protocols in object designs is related to work on protocols in the networking sense in LOTOS [4] [6]. The work on LOTOS however, does not apply directly since it was designed from a networking perspective, rather than an object oriented perspective.
In general, process algebras [2] have been an important source of inspiration to our work. Specifically, the work of Basten et al [3] [1] has been a great source of inspiration. Process algebras often use bisimulation semantics, which is too strong for our purposes; we like the intuitive meaning of failure semantics for non-deterministic choice better. This is one reason for our choice of Hoare's CSP [12]. Another reason leading to CSP was its direct support of distributed choice-making with the operators $\Box$ and $\sqcap$.
An alternative theory to CSP, also with failure semantics, is CCS, but comparison [8] shows that CCS is based on 'internal actions' τ where CSP uses two kinds of choice. The CSP approach is closer to our role model of objects, specifically because the concealment operator in CSP treats choices made by *other* roles the same as choices made internally in a class. The τ actions in CCS model sudden internal changes, which easily degrades guarantees; our roles guarantee their protocol regardless of interactions over other roles, and our checks are designed to sustain these guarantees.

Objects and roles. Roles are end-points of associations, whose intent is to describe objects from a specific perspective. Several authors have attacked, and sometimes attempted to solve, the lack of semantics related with associations in major object methods: [9] [24] [17] [16]. Other authors adopted roles as a solution to this problem, and have worked specifically on this area: [14] [11] [15].

Objects and processes. Work which is similar in spirit to our own has been conducted by Nierstrasz [18] [19]. This work focuses on substitutability of protocol aspects of objects, while our work focuses on protocol aspects over associations and internally in a single class; our work further distinguishes itself by contributing a model for roles, for feedback from server to client, and by the availability of an implementation.
Many object methods (such as Catalysis [5] and UML [21]) describe object life cycles in terms of state charts, or a flavour of Harel's state charts [10]. Therefore, we have striven for a strong relation with that 'formalism,' but with some second thoughts [22] about them.
The work on the object oriented method KISS [13] contains an interesting process notion *between* objects, which resembles our notion of the communicated process over an association. It does not distinguish client and server protocols.

The ROOM method [23] for real-time object design supports an idea similar to our roles, each having a state diagram protocol description. Its angle is that of real-time system design.
We expect that class protocols can also be formulated in terms of PSL [15] instead of CSP. In fact, we consider any construct that limits the possible orders of execution of implementation fragments as a candidate formalism for class protocols.

6 FUTURE RESEARCH

The protocol checks performed by Paul ensure protocol correctness for applications, provided they terminate. Such a check is sufficient for batch programs, but modern programs are increasingly interactive, and therefore demand stronger proofs. What we want to add to Paul to solve this, is an additional check if the application may deadlock. The complete protocol check then becomes the current check conjugated with deadlock freedom.
Several practical aspects of object programs still remain unsolved: Multiple instances of the same role should be allowed, as opposed to exactly one. Methods should support parameters, as far as these are of influence on protocols. Assignment to role references and inheritance between Paul classes are also absent. These topics are all on our research agenda.
Protocols are one aspect of a more complicated notion of process that we wish to integrate in the object paradigm. Additional aspects to cover with these processes are (automatic) synchronisation and transactional/workflow aspects to deal with failing executions.
Finally, a more theoretical point. We want to define an operational semantics for Paul, and prove the protocol correctness as defined in this paper to be sufficient for absence of protocol errors in the operational semantics.

7 CONCLUSIONS

This paper studied Paul, a language for assuring protocol correctness in object designs. Protocol-checking is complementary to type-checking and catches a different class of conceptual errors.
To introduce protocols in an object model which features roles, it is necessary to define a class protocol, import role protocols and export role protocols. None of these can be omitted without sacrificing features of Paul.
Paul's protocol checking is based on a 'correctly uses' relation. This relation was defined in terms of CSP semantics in this paper. Paul can be extracted from class diagrams with sufficient semantics, enabling protocol checks on these software representations. Paul can express decisions based on return values from a server.
Concluding, Paul offers flexible, precise and practical protocol checking facilities for use in object languages with roles.

Acknowledgments

We would like to thank Compuware for organising the interesting Quantum project, and the related budget, without which none of this work would have existed. Specifically, we wish to thank Wim Bast for sharing his vision on the object future.
We would also like to thank Twan Basten, whose deep insight in process algebra has been a great help in establishing this work, and Rik Eshuis for several significant comments.

References

[1] W.M.P. van der Aalst and T. Basten. Life-cycle inheritance: A Petri-net-based approach. In P. Azæcutema and G. Balbo, editors, *Application and Theory of Petri Nets*, volume 18, pages 62–81. Springer-Verlag, June 1997.

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[3] T. Basten and W.M.P. van der Aalst. A process-algebraic approach to life-cycle inheritance: Inheritance = encapsulation + abstraction. Computing Science Report 96/05, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, March 1996.

[4] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, 1988.

[5] D. D'Souza and A. Wills. *Catalysis: Practical Rigor and Refinement*. Addison-Wesley, 1998.

[6] P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors. *The formal description technique LOTOS*. Elsevier Science Publishers B.V., 1989.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading, 1996.

[8] R.J. van Glabbeek. Notes on the methodology of CCS and CSP. Technical report, Centre for Mathematics and Computer Science, PO Box 4079, 1009 AB Amsterdam, the Netherlands, Aug 1986.

[9] I. Graham, J. Bischof, and B. Henderson-Sellers. Associations considered a bad thing. *Journal of Object-Oriented Programming*, pages 41–48, Feb 1997.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[11] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). *OOPSLA*, pages 411–428, 1993.

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

[13] G. Kristen. *Object Orientation: The KISS Method: From Information Architecture to Information System*. Addison-Wesley, 1994.

[14] B.B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.

[15] D. Lea and J. Marlowe. Interface-based protocol specification of open systems using PSL. *Lecture Notes in Computer Science*, 952:374–398, 1995.

[16] K.J. Lieberherr and I.M. Holland. Assuring good style for object-oriented programs. *IEEE software*, pages 38–48, Sep 1989.

[17] B. McCarthy. Associaten inheritance and composition. *Journal of Object-Oriented Programming*, pages 69–72,74–77,80–81, Jul/Aug 1997.

[18] O. Nierstrasz. Regular types for active objects. In *Proceedings OOPSLA'93*, volume 28 of *ACM SIGPLAN Notices*, pages 1–15. ACM Press, October 1993.

[19] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995.

[20] OMG, editor. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 1995.

[21] Rational Software Corporation. *UML Semantics*. Rational Software Corporation, 1997.

[22] R. van Rein. Life cycles in Quantum. Technical report, University of Twente, Faculty of Computer Science, 1997. Internal report.

[23] B. Selic, G. Geullekson, and P.T. Ward. *Real-time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.

[24] C. Tanzer. Remarks on object-oriented modeling of associations. *Journal of Object-Oriented Programming*, pages 43–46, Feb 1995.

[25] WWW. The CSP archive. The WWW page on the Internet: http://www.comlab.ox.ac.uk/archive/csp.