BRICS

Basic Research in Computer Science

BRICS NS-99-2 Hüttel et al. (eds.): SOAP '99 Proceedings

Proceedings of the Workshop on

Semantics of Objects As Processes SOAP '99

Lisbon, Portugal, June 15, 1999

Hans Hüttel
Josva Kleist
Uwe Nestmann
António Ravara (editors)

BRICS Notes Series NS-99-2

ISSN 0909-3206 May 1999

Copyright © 1999, BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

See back inner page for a list of recent BRICS Notes Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone:	+45 8942 3360
Telefax:	+45 8942 3255
Internet:	BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/99/2/`

SOAP — Semantics of Objects As Processes

Purpose One of the most widespread programming paradigms today is that of object-oriented programming. With the growing popularity of the language C++ and the advent of Java as the language of choice for the World Wide Web, object-oriented programs have taken centre stage. Consequently, the past decade has seen a flurry of interest within the programming language research community for providing a firm semantic basis for object-oriented constructs.

Recently, there has been growing interest in studying the behavioural properties of object-oriented programs using concepts and ideas from the world of concurrent process calculi, in particular calculi with some notion of mobility. Not only do such calculi, as the well-known π-calculus by Milner and others, have features like references and scoping in common with object-oriented languages; they also provide one with a rich vocabulary of reasoning techniques firmly grounded in structural operational semantics.

The process calculus view has therefore proven to be advantageous in many ways for semantics and verification issues. On the one hand, the use of encodings of object-oriented languages into existing typed mobile process calculi enables formal reasoning about the correctness of programs; on the other hand, using standard techniques from concurrency theory in the setting of calculi for objects may help in reasoning about objects, e.g. by finding appropriate and mathematically tractable notions of behavioural equivalences. Encodings may also help clarify the overlap and differences of objects and processes, and suggest how to integrate them best in languages with both.

The aim of the one-day SOAP workshop is to bring together researchers working mainly in this area, but in related fields as well, where other process models or calculi are used as a basis for the semantics of objects.

Contents The '99 edition of SOAP, taking place as a satellite workshop of ECOOP '99, is composed of two complementary thematic building blocks.

The first is addressing the motto 'Semantics of Objects *As* Processes' literally in that objects are represented as a derived concept within a framework of processes; we are happy to welcome Oscar Nierstrasz, Markus Lumpe, and Jean-Guy Schneider as invited speakers to present the work they have been accomplishing in this area—starting out from a mobile process calculus—and to let us learn about their conclusions. This session is rounded up by a verification approach using a temporal logic as a target setting for, in this case, UML-style objects.

The second building block, divided into a session on behavioral subtyping and another one on behavioral typing, is more to be seen as an adaptation of the process-theoretic viewpoint to some object-oriented framework. While the typed λ-calculus is a firm ground to study typing for object-oriented languages, the typing of concurrent objects poses particular problems due to synchronization constraints. A static notion of typing is not powerful enough to capture dynamic properties of objects' behavior, like non-uniform service availability. Concurrency theory inspires dynamic notions of typing and subtyping, and the works that constitute this block of SOAP '99 exemplify the research currently being done in the field.

By means of a formal refereeing process, among the nine submitted abstracts five were recommended by the programme committee (Hans Hüttel, Josva Kleist, Uwe Nestmann, and António Ravara) and are presented here. (It is expected that the contributions will also appear elsewhere at other conferences or in journals.)

We would like to thank the organizers of ECOOP '99, in particular Ana Maria Moreira, for helping us logistically to set up the SOAP workshop, we thank BRICS, in particular Uffe H. Engberg, for the publication of these proceedings, and we thank Massimo Merro and Silvano Dal-Zilio for their assistance in the refereeing process.

Hans Hüttel Josva Kleist Uwe Nestmann António Ravara

Table of Contents

The workshop will be held at the Faculty of Sciences, University of Lisbon on June 15, 1999, in the order appearing in these proceedings.

Piccola – A Small Composition Language

Oscar Nierstrasz

Software Composition Group, University of Berne,
Institute for Computer Science and Applied Mathematics (IAM),
Neubrückstrasse 10, CH-3012 Bern, Switzerland.
oscar@iam.unibe.ch
http://www.iam.unibe.ch/∼oscar

Abstract

Although object-oriented languages are well-suited to implementing software components, they fail to shine in the construction of component-based applications, largely because object-oriented design tends to obscure a component-based architecture. We propose to tackle this problem by clearly separating component implementation and composition. Piccola is a small "composition language" that embodies the paradigm of "applications = components + scripts." Piccola models components and composition abstractions by means of a unifying foundation of communicating concurrent agents. Flexibility and extensibility are obtained by modelling both interfaces to components and the contexts in which they live by extensible records, or "forms". We illustrate the realization of an architectural style in Piccola and show how external components may be adapted and composed according to the style. We show how separating components from their composition can improve maintainability.

2

The $\pi\mathcal{L}$-Calculus - A Formal Foundation for Software Composition

Markus Lumpe

Software Composition Group, University of Berne,
Institute for Computer Science and Applied Mathematics (IAM),
Neubrückstrasse 10, CH-3012 Bern, Switzerland.
lumpe@iam.unibe.ch
http://www.iam.unibe.ch/∼lumpe

Abstract

In this talk, we present a formal language for software composition that is based on the π-calculus. More precisely, we present the $\pi\mathcal{L}$-calculus, a variant of the π-calculus, in which agents communicate by passing extensible, labeled records, or so-called "forms", rather than tuples. This approach makes it much easier to model compositional abstractions than it is possible in the plain π-calculus, since the contents of communications are now independent of positions, agents are more naturally polymorphic since communication forms can be easily extended, and environmental arguments can be passed implicitly. The $\pi\mathcal{L}$-calculus is developed in three stages: (i) we analyse whether the π-calculus is suitable to model composition abstractions, (ii) driven by the insights we got using the π-calculus, we define a new calculus that has better support for software composition (e.g., provides support for inherently extensible software construction), and (iii), we define a first-order type system with subtype polymorphism that allows us to statically check an agent system in order to prevent the occurrences of runtime errors.

4

Object Models in the $\pi\mathcal{L}$-Calculus

Jean-Guy Schneider

Software Composition Group, University of Berne,
Institute for Computer Science and Applied Mathematics (IAM),
Neubrückstrasse 10, CH-3012 Bern, Switzerland.
`schneidr@iam.unibe.ch`
`http://www.iam.unibe.ch/~schneidr`

Abstract

The development of concurrent object-based programming languages has suffered from the lack of any generally accepted formal foundation for defining their semantics, although several formal models have been proposed. Most of these models define objects and object-oriented abstractions as primitives, but they either do not incorporate important features found in object-based programming languages (e.g., they lack inheritance), hard-wire the underlying inheritance model, or integrate concepts in a non-orthogonal way. As an approach to overcome the problems of existing models, we present a (meta-level) framework for object models in the $\pi\mathcal{L}$-calculus. We show that common object-oriented programming abstractions such as instance variables and methods, different method dispatch strategies as well as class features are most easily modelled when class metaobjects are explicitly reified as first class entities. We illustrate that various concepts which are typically merged (or confused) in object-oriented programming languages can be expressed in a more natural way by making a clear separation between functional elements (i.e. methods) and their compositions (i.e. inheritance). Furthermore, we show that the same concepts can also be applied for modelling mixins, mixin application, and mixin composition.

Composing object-oriented specifications and verifications with cTLA

Günter Graw, Peter Herrmann, Heiko Krumm
Dept. of Computer Science, Dortmund University, D-44221 Dortmund, Germany
Internet: {graw|herrmann|krumm}@ls4.cs.uni-dortmund.de

Abstract

In order to support formally correctness preserving refinement steps of object-oriented system designs, we refer at one hand to the practically well-accepted Unified Modelling Language (UML) and at the other hand to L. Lamport's Temporal Logic of Actions (TLA) which supports concise and precise notions of properties of dynamic behaviours and corresponding proof techniques. We apply cTLA which is an extension of TLA and supports the modular definition of process types. Moreover, in cTLA process composition has the character of superposition which facilitates the modular transformation of UML diagrams to corresponding formal cTLA process system definitions and their structured verification. We exemplify transformation and formal verification. Moreover we outline the application of this method for the establishment of domain-specific specification frameworks which can directly support the UML-based correct design of OO-systems.

1 Introduction

Meanwhile, the practical design of object-oriented application systems is mostly based on the Unified Modelling Language UML [21]. Systems are modeled and described by a series of UML diagrams where each diagram corresponds to a partial view of a system and concentrates on certain property types and aspects. So, class diagrams describe the static class structure. Use case diagrams are devoted to specific utilizations and the objects instances which are responsible for their realization. Collaboration diagrams focus on the partners and interactions of specific cooperation relations. Statechart diagrams describe the behaviour of object instances. While several approaches exist which assign formal semantics to the different diagram types (e.g., [16]), usually UML-based designs are non-formal. Since the diagrams support intuitive interpretations, the designers easily understand their pragmatical meanings without reference to formal models. Therefore, often formal designs are not desirable, especially, since the development and analysis of formal models would introduce considerable additional costs. Furthermore, for many interesting formal design checks separate formal models of single diagrams would not suffice. Instead, very complex models of diagram combinations would be necessary which model a set of diagrams in context with each other in order to cover interrelations.

The design of critical systems, however, can essentially profit from formal verifications. We expect benefits at least from formal checks of those functions, aspects, and properties which are as well crucial as their provision depends on complex and not easy-to-understand mechanisms. In particular, aspects of the design of dynamic object system configuration at runtime, of concurrent execution threads, of combined behaviour of object instances, and of object interactions are inherently complex and difficult to master without formal support. In order to support formal modelling and analysis of partial aspects of UML-based object-oriented system designs, especially with respect to questions of concurrency, object behaviour, and interactions, we developed transformations from UML diagrams to formal cTLA specifications [9] and underlying state transition system models.

cTLA is based on L. Lamports Temporal Logic of Actions (TLA) [17] and refers to the concepts of state transition systems, refinement mappings [1], and the separate definition of both safety and liveness properties. Unlike TLA, the cTLA composition principle is oriented at CCS [19] and Lotos [14] and applies the principle of superposition like DisCo [6]. In comparison with [2], the cTLA processes do not interact via shared variables but perform joint actions. This stateless way of interaction has different benefits. Especially constraint-oriented processes can be represented (cf. [23]) which are well suited for the diagrams of the UML. Furthermore, cTLA supports decompositional proofs. A system is the logical conjunction of its processes and the style conventions assure the absence of contradictions in the system formula. Thus, process properties are directly inherited to the system. The compositionality of cTLA supports the transformation of UML-based descriptions since each UML diagram of a system description can be modelled by a single cTLA process which contributes to the system as a whole in a well-defined way. For the analysis of properties of interrelations relatively small subsystems can be used comprising only those processes which influence the properties of special interest.

This paper shortly outlines our approach as a whole and concentrates on the formal verification of refinement steps where a step is represented by two sets of UML diagrams. The first set describes the starting point of the refinement which we call the abstract model. The second set specifies the result of the refinement by means of the so-called refined model. Both models can be transformed to cTLA. Thus, there are two corresponding systems of cTLA processes, the first describing a more abstract state transition system, the second describing a refined state transition system. The formal verification shall prove that the refined system has in fact all those safety and liveness properties which are required by the specification of the abstract system. Since TLA's formal refinement relation directly corresponds to this practically relevant notion of correct refinement, verifications can be performed on the basis of TLA where a refinement step is correct, exactly if the implication '*Refined System implies Abstract System*' can proved to be a valid TLA-formula cf. [17].

Of course, the transformation of UML diagrams to cTLA processes and the TLA-based verification introduces additional efforts. Therefore methods are of high interest which support correctness relations directly applying to UML specifications of abstract and refined systems. These objectives are similar to those of the pUML group whose members investigate diagrammatical transformation rules where a rule directly applies to an abstract diagram and transforms it to the refined diagram of a correct refinement [20]. Thus, the approach of pUML transformation rules is very ambitious and shall combine the advantages of correctness-preserving source-code transformations [4] with those of graphical specification and modelling support. With respect to the preservation of behavioural properties of concurrent and distributed systems, however, we made the experience that general correctness-preserving transformation rules are very difficult to handle in the course of practical design processes. Moreover, the rules are accompanied by so-called application conditions. The correctness of a transformation is only assured if the application condition holds. Since many application conditions are relatively complex, efforts for their proofs are necessary which are comparable to that of a posterio verifications of freely designed refinements.

Under these considerations, our present work investigates another direction of direct refinement support. It follows up the framework approach of software development (cf. [15]) and translates it into the field of specification development. Consequently, we study special domains of application (e.g., protocol design [11], distributed control of chemical plants). Corresponding collections of specification modules and patterns for abstract and refined systems are under development. Moreover, the relations between those abstract and refined modules and patterns are investigated which correspond to correct refinement steps. The results are documented by a collection of theorems. The theorems are implications between refined system patterns and abstract patterns. In principle, their function is comparable to that of general correctness preserving transformation rules. Nevertheless, the theorems connect domain-specific specification patterns and therefore can provide direct application-specific design support.

In the remainder, we outline the formal specification language cTLA. Thereafter we address basic notions of dynamic behaviours of object systems and their representations in UML models. We describe the essentials of the transformation from UML diagrams to cTLA processes. From that, the TLA-based verification of refinements is discussed. Transformation and verification are exemplified by means of a small application scenario. Finally, we sketch our present work which is constructing a domain-specific specification framework for distributed control of chemical plants.

2 Compositional specification style cTLA

cTLA [10, 18] is based on Leslie Lamports Temporal Logic of Actions (TLA) [17] and supports the definition of parametrized process and system types. A specification of a simple process or a (sub)system is formed by instantiating a cTLA process type resp. system type. As in the formal description language Lotos [14], systems are composed from processes which interact by means of joint actions. Due to this method of composition, processes can model not only implementation parts but also logical system constraints (cf. [23]).

As an example of a cTLA process type we outline *Object* in Fig. 1 describing the behaviour of an UML object (cf. Sec. 4). In the process type header the name *Object* and the process parameters *cf*, *id*, and *class* are declared. The state variables (e.g., *state*, *lifecycle*, *qu*) model the process state. The set of initial states is descibed by the predicate `INIT`. State transitions are specified by means of actions. An action (e.g., *callAction*) is a predicate about action parameters (e.g., *receiver*), state variables describing the state before executing the action (e.g., *lifecycle*), and so-called primed state variables modelling the state after executing the action (e.g., *lificycle'*). Besides of state transitions specified

```
PROCESS Object (cf : ClassFrame ; id : OId ; class : ClassName)
  VARIABLES
    state : cf.State ;  ! object data, links, and control
    lifecycle : (unborn, alive, dead);  ! life cycle state
    qu : queue of Message ;     ! messages received
    awaitReturnOf : MessageId ; ! if blocked: call message id
     ...;                       ! message id management, etc.
  INIT ≙ lifecycle = unborn ∧ ... ; ! initially, object does not exist
  ACTIONS
    callAction ( receiver : OId ; objState, objNextState : cf.State ;
                 message : Message ; mode : SyncMode ) ≙ ! send Call-message
      lifecycle=alive ∧ lifecycle′=lifecycle ∧
      cf.nextState(state,state,message,receiver,mode) ∧
      awaitReturnOf′=IF mode=blocking THEN message.id ELSE nullId ∧
      qu′=qu ∧ ... ;
    receiveAction ( objState, objNextState : cf.State ;
                    message : Message ) ≙ ... ; ! receive a message
      ! if message is a return message awaited, it is inserted at the front
      ! of qu otherwise appended.
    returnAction ( receiver : OId ; objState, objNextState : cf.State ;
                   message : Message ) ≙ ... ; ! send Return-message
    createAction ( receiver : OId ; objState, objNextState : cf.State ;
                   message : Message) ≙ ... ; ! send Create-message
    ...;
   END
```

Figure 1: Process type *Object*.

by actions, a process may perform stuttering steps where it does not change its state while the process environment performs a state transition.

The cTLA process type *Object* describes safety properties. Liveness constraints (cf. [3]) are described by additional weak or strong fairness assumptions forcing the execution of an action if it would be enabled for an infinite period of time otherwise. Weak fair actions (denoted by `WF : callAction`) are only required to execute if the action would otherwise be incessantly enabled while execution of strong fair actions (denoted by `SF : callAction`) is guaranteed even if the action is sometimes disabled. Unlike the definition of [3] and TLA, cTLA provides for conditional fairness assumptions in order to keep the compositionality of systems. A fair action has to execute only if otherwise infinitely many states exist where the action is enabled as well as its execution is tolerated by the environment.

Systems and subsystems are described as compositions of concurrent processes which encapsulate their state variables and change their local states according to the process actions. The vector of the process state variables represent the state of the entire system. System state transitions are described by system actions which are logical conjuncts of process actions and process stuttering steps. Since each process contributes to each system action by exactly one action or a stuttering step, concurrency is modeled by interleaving and the coupling of processes by joint actions. The action parameters are used to describe data transfer between processes.

```
PROCESS GlobalSystem ( cfs : [class → ClassFrame]; OId : data type;
                       classOf : [OId → class] )
  PROCESSES  ! the infinite array of object processes
    ARRAY obs [OId] of Object(cfs[classOf(index)],index,classOf(index));
  ACTIONS ! system actions defining the coupling of the objects
    operationCall (caller, callee: OId ;
                   callerState, callerNextState,
                   calleeState, calleeNextState : State ;
                   message : Message ; mode : SyncMode ) ≙
      ! caller calls operation of callee
      obs[caller].callAction(callee,callerState,callerNextState,
                             message,mode) ∧
      obs[callee].receiveAction(calleeState,calleeNextState,message) ∧
      ∀ i ∈ OId \ {caller,callee} obs[i].Stutter ;
    operationReturn (...) ≙ ...; ! callee operation returns to caller
    objectCreate (...) ≙ ...; ! object sends create message
    ...;
  END
```

Figure 2: Process type *GlobalSystem*.

As an example Fig. 2 shows the system type *GlobalSystem* modelling a system of UML objects. The processes composing the system are listed in the section `PROCESSES`. For instance, *GlobalSystem* consists of *OId* many instances *obs[i]* of the process type *Object* (cf. Fig. 1). The system actions are listed in the section `ACTIONS`. In the example, the action *operationCall* models that the object *obs[caller]* calls an operation of the object *obs[callee]*. Therefore *obs[caller]* participates to *operationCall* by the process action *callAction* and *obs[callee]* by the process action *receiveAction*. The other processes participate to *operationCall* by stuttering steps. Data between the caller and the callee are described by the system action parameter *message*. During the execution of *operationCall*, the process action parameters *message* in *obs[caller].callAction* and *obs[callee].receiveAction* have to carry identical values.

cTLA facilitates the combination of different property types like safety and liveness. Thus, in the resource oriented specification style, all relevant aspects of a component can be described by a single process type. In the constraint-oriented specification style one can specify different aspects of a component by separate constraint processes. In order to support the modularity of verifications, however, liveness properties may be combined with models of the safety behaviour of the component's environment (cf. [11]).

3 Dynamic behaviour

Since we concentrate on the issues of concurrency and concurrent object interaction we give a short outline of the according UML concepts. We view an object system as a set of objects and a set of threads of activity. An object system evolves during runtime from an initial object configuration performing steps of execution changing the system state. The relevant state of a system depends on the set of currently existing objects and their control

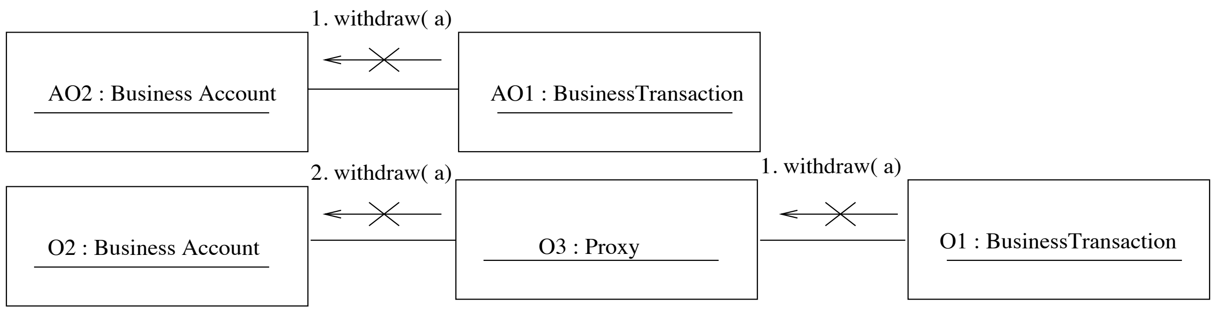


Figure 3: Collaboration diagram of the example system

and data (attribute values) states (cf. Fig. 1). The state of the object system as a whole identifies the set of currently existing objects and moreover contains the object states as components (see [9] for the according cTLA specification of the global system). The UML uses statechart, interaction (sequence and collaboration), and use case diagrams for the description of the dynamic behaviour. In the UML, an execution step with an object corresponds to an action which is modelled in a statechart. Actions may effect the local and foreign objects as well. There are several kinds of actions:

- A call action results in the invocation of an operation.
- Send actions result in the asynchronous sending of a signal.
- Create actions cause the creation of an instance of a class. They are not permitted to have a target object.
- By return actions a value or a set of values is returned to the caller.
- A terminate action results in the self-destruction of an object. It should not have parameters.
- Local invocation actions cause the local invocation of an operation without generating a call or signal event.
- Actions that are not previously defined are called uninterpreted actions.

Like the UML-metamodel [21] we assume run-to-completion semantics (RTC) for state machines which follows the idea that requests are processed in sequence one after the other. This assumption simplifies the synchronization of an object, since an incoming request is only processed, if the object has reached a stable state configuration. Communication between objects is specified by means of signal or operation requests. Objects communicate by means of operation (service provided for another object) requests if the calling object demands a service by the called object. A request is forwarded by a message instance which can carry a set of arguments. Operations can be called synchronously (sender is blocked, cross in message symbol) or asynchronously, which is modelled which is modelled in collaboration diagrams. The number which precedes the name of a message represents its order in an execution sequence.

4 Transformation

Since cTLA facilitates constraint-oriented specifications, the different diagrams of UML specifications can be modeled formally by a couple of individual cTLA-processes. Below, we will outline the transformation of UML collaboration diagrams and statecharts by means of a simple example specification. The idea of the example is that money is withdrawn from a business account in a business transaction. In the top of Fig. 3 a collaboration is shown which represents the withdrawal of money from the business account. A refined design of the example is presented in the bottom of Fig. 3. The refinement is manifested in the introduction of a new proxy object for the business account which is located in another address space. The statecharts of the according object classes are given in Fig. 4.

Collaboration diagrams are transformed to the cTLA process type *CollaborationDiagramUnit* shown in Fig. 5. We introduce a new process instance for each two objects which are relevant in the context of the according use case. The process parameters *O1* and *O2* in *CollaborationDiagramUnit* are used to address the corresponding process types. Moreover, a constraint process instance should manage the set of active use cases. This causes the introduction of a corresponding parameter *activeUseCases* to the actions. The state variable *actMessage* keeps track of call messages and *callerLocked* of blocking caused by synchronous calls (*callerLocked*). Since the process type *CollaborationDiagramUnit* shall constrain only those actions, which are related to the objects *O1*, *O2* and to the active use case *myUseCase*, each action is furnished with a term applying a stuttering step of the constraint process to those action occurrences which are irrelevant for the constraint (synchronous mode, no other message until termination.), except for disjunctive terms applying real constraints (under condition $caller = O1 \wedge callee = O2 \wedge myUseCase \in activeUseCases$).

The transformation of UML statechart diagrams to cTLA processes is performed in two steps. At first, a statechart which may contain nested states and transitions labelled by action sequences is transformed to an ordinary state transition system following the principles explained in [13, 22]. For instance, the statechart at the right side of Fig. 4 describing the BusinessTransaction *O1* is transformed to a simple state transition system

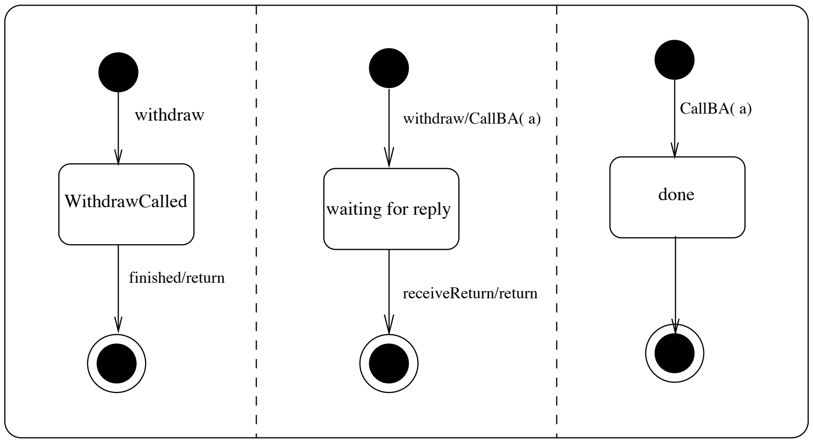


Figure 4: Statechart diagrams of the example system

```
PROCESS CollaborationDiagramUnit (O1, O2 : OId; myusecase : UseCase)
BODY
  VARIABLES
    actMessage : SUBSET(Message.Id); ! List of active messages
    callerLocked : {"yes","no"};     ! Is caller locked ?
  INIT actMessage = ∅ ∧ callerLocked = "no";
  ACTIONS
    operationCall (caller, callee : OId; message : Message;
                   mode : SyncMode; activeUseCases : SUBSET(UseCase) ) ≙
    ! If O1 is caller, O2 is callee, and myusecase is an active use case, only
    ! messages of type "Withdraw" may be send; message becomes active and
    ! caller is locked
      ( caller = O1 ∧ callee = O2 ∧ myusecase ∈ activeUseCases ∧
        callerLocked = "no" ∧
        ( ( message.operationname = "Withdraw" ∧ mode = "synchronized" ∧
            actMessage′ = actMessage ∪ {message.id} ∧
            callerLocked′ = "yes" ) ) )  ∨
    ! Otherwise process performs a stuttering step
      ( ( caller ≠ O1  ∨  callee ≠ O2  ∨  myusecase ≠ activeUseCases ) ∧
        actMessage′ = actMessage ∧ callerLocked′ = callerLocked );

    operationReturn (caller, callee : OId; message : Message;
                     activeUseCases : SUBSET(UseCase) ) ≙ ...;
    ! If O2 is caller, O1 is callee, and myusecase is an active use case, only
    ! messages of type "Withdraw" may be returned; furthermore a message must
    ! be active; message becomes passive and caller is unlocked
```

Figure 5: Process type CollaborationDiagramUnit.

listed in Fig. 6.

In the second step the transition system is transformed to a cTLA process type. Since cTLA process types model state transitions in a direct way, this step is very simple. The process type *BusinessTransaction* (Fig. 7) contains the state variable *state* modelling the three states of the state transition system. The condition `INIT` specifies that "*i*" is the initial state and the actions *callAction*, *receiveAction*, and *internalAction* model the transitions. The process parameter *id* describes the object identifier while *myusecase* is used to manage the active use cases in accordance with a further constraint process.

Relations between UML diagrams are modeled in cTLA by means of process action

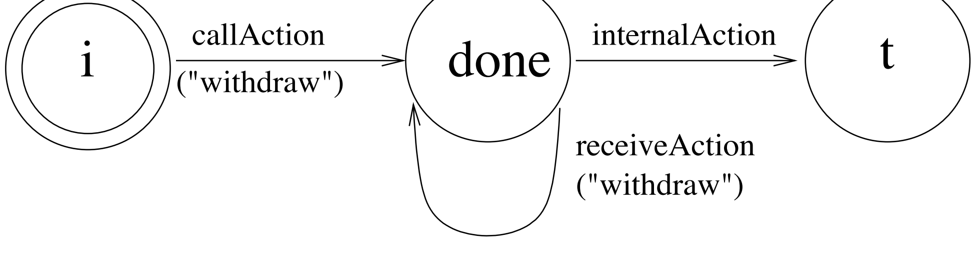


Figure 6: State transition system of the BusinessTransaction *O1*.

```
PROCESS BusinessTransaction (id : OId; myusecase : UseCase)
BODY
  VARIABLES
    state : {"i","wfr","t"}; ! actual process state
  INIT state = "i";
  ACTIONS
    callAction (caller : OId; message : Message;
                activeUseCases : SUBSET(UseCase) ) ≙
      ( state = "i" ∧ myusecase ∈ activeUseCases ∧ id = caller ∧
        message.operationname = "Withdraw" ∧ state′ = "wfr" )  ∨
      ( ( state ≠ "i"  ∨  myusecase ∉ activeUseCases  ∨
          id ≠ caller  ∨  message.operationname ≠ "Withdraw" ) ∧
          state′ = state );
    receiveAction (callee : Oid; message : Message;
                   activeUseCases : SUBSET(UseCase) ) ≙
      ( state = "wfr" ∧ myusecase ∈ activeUseCases ∧ id = callee ∧
        message.operationname = "Withdraw" ∧ state′ = "wfr" )  ∨
      ( ( state ≠ "wfr"  ∨  myusecase ∉ activeUseCases  ∨
          id ≠ callee  ∨  message.operationname ≠ "Withdraw" ) ∧
          state′ = state );
    internalAction (this : OId; activeUseCases : SUBSET(UseCase) ) ≙ ...;
END
```

Figure 7: Process Type *BusinessTransaction*.

conjunctions. Assume that *O1 : BusinessTransaction* is the cTLA process specifying the example object $O1$ and *CollO1O3 : CollaborationDiagramUnit* the cTLA process describing the operation call *withdraw* in which $O1$ is the caller and $O3$ the callee. Since the operation call is triggered by $O1$ performing a *callAction*, the process actions *callAction* of cTLA process $O1$ and *operationCall* of *CollO1O3* are conjoined. Likewise, the process actions *returnAction* in $O1$ and *operationReturn* in *CollO1O3* are coupled. The action *internalAction* of $O1$ does not correspond to any collaboration diagram transitions and therefore is linked with a stuttering step of *CollO1O3*.

5 An example proof

Below we will outline the proof that the abstract system consisting of the business transaction object $AO1$ and the business account object $AO2$ is realized by a more detailed system consisting of $O1$, $O2$, and an additional proxy $O3$. The UML collaboration diagrams and statecharts are transformed into cTLA specifications according to Sec. 4. The proof utilizes the compositionality of cTLA. It can be decomposed into three simpler proof steps. At first, we have to prove that the subsystem $SO_{1/3}$ consisting of the processes representing the statecharts of $O1$ and $O3$ (Fig. 4) composed with the process *CollO1O3* modelling the collaboration diagram unit connecting $O1$ with $O3$ (Fig. 3) fulfills the process representing the abstract business transaction object $AO1$. Secondly, we prove that the process representing $O2$ implies that implementing $AO2$. Finally, we have to verify that the process describing the collaboration between $O3$ and $O2$ realizes that representing the collaboration between $AO1$ and $AO2$.

Here, we will sketch only the first proof corresponding to the verification of the implication $SO_{1/3} \Rightarrow AO1$ which is performed as a regular TLA refinement proof (cf. [17]). In order to compare the two state spaces of $SO_{1/3}$ and $AO1$, we define a mapping between them, the so-called refinement mapping:

$$\begin{aligned} RM \mathrel{\hat=} \; & O3.state = \text{"}i\text{"} \rightarrow AO1.state = \text{"}i\text{"} \\ & O1.state = \text{"}t\text{"} \rightarrow AO1.state = \text{"}t\text{"} \\ & otherwise \rightarrow AO1.state = \text{"}withdrawCalled\text{"} \end{aligned}$$

Instead of $AO1$ we use the equivalent process $\overline{AO1}$ for the proof where the local variable $AO1.state$ is replaced by variables of $O1$ and $O3$ according to RM.

Firstly, we have to verify that all initial states of $SO_{1/3}$ are also initial states of $\overline{AO1}$. Since in the initial states of both processes the equation $O3.state = \text{"}i\text{"}$ holds, this proof is trivial. Secondly, we have to prove that each action of $SO_{1/3}$ implies either an action or a stuttering step in $\overline{AO1}$. This proof, however, cannot be performed directly. Before, we have to prove that the following formula I is an invariant of the subsystem $SO_{1/3}$:

$$\begin{aligned} I \mathrel{\hat=} \; & O1.state = \text{"}i\text{"} \Rightarrow O3.state = \text{"}i\text{"} \wedge O1.state = \text{"}t\text{"} \Rightarrow O3.state = \text{"}t\text{"} \wedge \\ & O3.state = \text{"}waitingForReply\text{"} \Rightarrow O1.state = \text{"}withdrawCalled\text{"} \wedge \\ & \text{"}withdraw\text{"} \in O3.qu \Rightarrow (O1.state = \text{"}withdrawCalled\text{"} \wedge O3.state = \text{"}i\text{"}) \wedge \\ & \text{"}withdrawReturn\text{"} \in O1.qu \Rightarrow (O1.state = \text{"}withdrawCalled\text{"} \wedge O3.state = \text{"}t\text{"}) \end{aligned}$$

The invariant proof is performed by checking that I holds initially and is preserved by all actions of $SO_{1/3}$.

Using the proven fact, that I holds before and after execution of any action in $SO_{1/3}$, we can now verify that the actions of $SO_{1/3}$ correspond to actions or stuttering steps of $\overline{AO1}$. As an example we outline that the action T changing the state *state* of $O1$ from *withdrawCalled* to t implies the action $\overline{T}$ of $\overline{AO1}$ changing the state $O1.state \neq \text{"}t\text{"} \wedge O3.state \neq \text{"}i\text{"}$ ($AO1.state = \text{"}withdrawCalled\text{"}$) to $O1.state = \text{"}t\text{"}$. T is a joint action conjoining the actions *receiveAction* of $O1$ and *operationReturn* of the collaboration between $O1$ and $O3$. It can only be executed if the message "*withdrawReturn*" is in the message queue $O1.qu$ of $O1$. This implies that due to the last conjunct of I the condition $O1.state = \text{"}withdrawCalled\text{"} \wedge O3.state = \text{"}t\text{"}$ holds before executing T. Thus, the enabling condition of T implies the enabling condition of $\overline{T}$. After the execution the condition $O1.state = \text{"}t\text{"}$ holds as well in $SO_{1/3}$ as in $\overline{AO1}$. Therefore, the effect of T implies the effect of $\overline{T}$, too, and T implies $\overline{T}$ as a whole. Likewise, all actions of $SO_{1/3}$ are proven.

6 Verification with cTLA patterns

In this section we focus on the UML descriptions of properties of a software product on different levels of abstraction and the correctness of these descriptions. Therefore we introduce two models on different levels of abstraction which stem from the software life cycle(requirements engineering, design) of a given product. These are:

- The abstract software model (ASM) serves as interface between application engineering and software development. It models the structuring of the software parts of the system into logical components. It is a result of analysis activities performed during the requirements engineering of a software product which typically bases on knowledge from previously performed domain engineering.
- The concrete software model (CSM) is a refinement of the ASM. It structures the software into implementation-oriented components. It explicitly refers to distribution and network communication, to fault- tolerant mechanisms and performance optimisation as well as to the allocation and management of resources.

Both models are described in terms of patterns which have currently a growing impact on software development. On the one hand there are the well- known design patterns (e.g. given in [8]). On the other hand, analysis patterns have to be defined for each individual domain on their own applying techniques for domain engineering. Analysis patterns have been applied by Fowler [7] who has found and applied them in several industrial projects. Analysis patterns are described by the terms and concepts of an application domain. Taking these concepts, it is obvious that analysis patterns are applied in the ASM while design patterns are used in the CSM. Furthermore, some practitioners claim that there exist relationships between collections of patterns which might be expressed in a so called system of patterns [5].

Now we concentrate on the verification of a CSM-level specification against the abstract requirements expressed by an ASM-level specification. We have to prove that the CSM specification implements the relevant properties of the ASM specification. For that purpose comprehensive and compatible formal models of the dynamic semantics of both, the ASM and the CSM model, are needed. Moreover, one needs the formal inference system TLA to perform these proofs. While cTLA is well-suited to the formal modelling of highly structured systems, there is a very high complexity when ASM and CSM specifications of practical systems are transformed to cTLA. As previously stated, the statecharts are translated to process behaviour descriptions. The interaction diagrams and the activity diagrams (used for the modelling of synchronisation aspects) are translated to configuration and process coupling descriptions. To model the dynamic creation of objects, infinite state space structures (i.e., for each object type an infinite array of object instances) are used which are accompanied with an explicit state representation of the current existence of an object. In general, the operations of objects can be executed concurrently and there is a wide spectrum of object interaction mechanisms and synchronisation methods. Since state transition systems model behaviours by series of atomic transitions, the wide spectrum of object interactions induces a very fine granularity of atomicity. Thus, models with a very complex state space and with a very fine transition structure are needed in general. We think that these models are too difficult-to-understood to form a convenient basis of manageable formal verifications in practice. On the other hand, we are aware that the complexity of the models does not result from cTLA but is a direct consequence of the modelling power of UML-descriptions of object-oriented systems. Therefore, the approach has to be enhanced by additional concepts.

In order to render possible manageable formal proofs of practical systems, we utilise the proposed application of conceptual patterns in the ASM and of software design patterns in the CSM for the verification, too. The benefits of patterns are twofold. On the one hand, patterns restrict system structures, the interactions, the concurrency, and the synchronisation of objects. The formal modelling recognises the restrictions and provides for less complex models which are more easy-to-understand since they directly correspond to application-oriented interaction schemes. On the other hand, there are logical relationships between conceptual patterns and software design patterns since a design pattern serves for the purpose of implementation of a conceptual pattern. This implementation relation between patterns of a system of patterns are formally modelled by the refinement relation of TLA, i.e., there exist valid implications from design patterns to conceptual patterns. In connection with the modularity and the genericity features of cTLA, theorems are stated which correspond directly to the logical relationships between patterns. E.g. in the example proof listed in section 5 there is a theorem which states the refinement relationship from a proxy pattern and a refined controller to the more abstract analysis pattern controller. These theorems can easily be instantiated to represent the particular refinement relations of a specific practical project.

In the domain of communication protocols comprehensive libraries of patterns and theorems are already established [11] and the experience showed that even complex practical protocols can be verified by means of theorems only, i.e., in order to verify a protocol it was not necessary to perform basic TLA deductions since all necessary implications of the proofs were instantiated from theorems [12].

7 Conclusion

We reported on present work which aims to the establishment of domain-specific specification frameworks for the object-oriented and pattern-based design of concurrent and distributed software systems. In particular, the frameworks will supply theorems which describe patterns of correct refinements and facilitate formal verification enormously since theorems can replace nearly all complex original proofs of verifications. Our report concentrated on the formal background of theorems which is given by transformations of UML diagrams to modular cTLA specifications enabling the application of TLA-based proof methods. According to this procedure the theorems of the specification frameworks under development are proven. Besides of our former work supporting the cTLA-based formal specification and verification of communication protocols, there is additional work the specification framework approach is related to. So, meanwhile extensions of cTLA exist which support the handling of real-time and continuous properties. Under application of these extensions already several hazard analysis and safety proofs for chemical plants were accomplished.

References

[1] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan. 1993.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[4] The CIP Language Group: The Munich Project CIP Volume I: The Wide Spectrum Language CIP-L. Lecture Notes in Computer Science 183 : Springer 1985

[5] F. Buschmann, R. Meunier, H. Rohnert Pattern Oriented Software Architecture : A System of Patterns. Addison-Wesley, 1996.

[6] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with a centralized control. *Distributed Computing*, (3):73-78, 1989.

[7] M. Fowler. Analysis Patterns : Reusable Object Models. Addison-Wesley, 1996.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, 1994.

[9] G. Graw, P. Herrmann, and H. Krumm. Constraint-Oriented Formal Modelling of OO-Systems. To appear in: *Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 99)*, Helsinki, June 1999. Kluwer Academic Publisher.

[10] P. Herrmann and H. Krumm. Compositional Specification and Verification of High-Speed Transfer Protocols. In S. T. Vuong and S. T. Chanson, editors, *Protocol Specification, Testing, and Verification XIV*, pages 339–346, Vancouver, B.C., Canada, 1994. IFIP, Chapman & Hall.

[11] P. Herrmann and H. Krumm. Re-Usable Verification Elements for High-Speed Transfer Protocol Configurations. In P. Dembiński and M. Średniawa, editors, *Protocol Specification, Testing, and Verification XV*, pages 171–186, Warsaw, Poland, 1995. IFIP, Chapman & Hall.

[12] P. Herrmann and H. Krumm. Modular Specification and Verification of XTP. *Telecommunication Systems* 9(2):207-221, 1998.

[13] J. Hooman, S. Ramesh, and W.-P. de Roever. A compositional axiomatization of Statecharts. *Theoretical Computer Science*, 101:289–335, 1992.

[14] ISO. *LOTOS: Language for the temporal ordering specification of observational behaviour*, International Standard ISO 8807 edition, 1989.

[15] R. Johnson and B. Foote. Designing reusable classes. *The Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[16] K.C.Lano and A.S.Evans. Rigorous Development in UML. In ETAPS'99, FASE workshop. LNCS, 1999.

[17] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[18] A. Mester and H. Krumm. Composition and Refinement Mapping based Construction of Distributed Applications. In *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Aarhus, Denmark, 1995. BRICS.

[19] R. Milner. *A Calculus for Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer, Berlin, 1980.

[20] The pUML group. http://www.cs.york.ac.uk/puml/

[21] The UML Group, Rational Software Corporation. Santa Clara, CA-95051, USA. *UML Semantics. Version 1.1*, July 1997.

[22] A. C. Uselton and S. A. Smolka. A Compositional Semantics for Statecharts using Labeled Transition Systems. In B. Johnsson and J. Parrow, editors, *CONCUR'94: Concurrency Theory*, number 836 in Lecture Notes in Computer Science, pages 2–17. Springer-Verlag, 1994.

[23] C. A. Vissers, G. Scollo, and M. van Sinderen. Architecture and specification style in formal descriptions of distributed systems. In S. Agarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification*, volume VIII, pages 189–204, Elesevier, 1988. IFIP.

A Practical Approach to Behavioural Inheritance in the Context of Coloured Petri Nets

Charles Lakos[1] and Glenn Lewis[2]

EXTENDED ABSTRACT

Introduction

As Taivalsaari observes: *Inheritance is often regarded as the feature that distinguishes object-oriented programming from other modern programming paradigms, and many of the alleged benefits of object-oriented programming, such as improved conceptual modelling and reusability are accredited to it.* [18, p438]

Inheritance means one can begin with an abstract representation of an object that is easy to understand and clutter-free, and refine that to a more concrete representation. In other words, inheritance provides support for abstraction, which is the most common and effective technique for dealing with complexity [7].

In object-oriented systems, when a more abstract class of object is incrementally changed to a more refined class (via inheritance) the relationship between the classes is often interpreted as *is-a*. For example, a *car* is a *vehicle*. By saying this, we mean that, with respect to a set of expectations about *vehicles*, anything that is a *car* also fulfils the expectations about *vehicles*. That is, the *is-a* relationship implies substitutability with respect to a given set of expectations. The *principle of substitutability* has been proposed in various forms to give the expectations that an incrementally changed component should comply with if it is to be substituted for a component.

Wegner and Zdonik enunciate a principle of substitutability as *an instance of a subtype can always be used in any context in which an instance of a supertype is expected* [21]. They propose a number of variants of this. One possibility, which is known as *weak substitutability*, relates to the compatibility of method parameter and result types — it does not require behavioural compatibility.

Many consider weak substitutability as not enough — substitution may still lead to incorrect behaviour even if the weak substitutability principle is satisfied. Liskov gives the example that a stack could be substituted for a queue without causing an error (i.e. the weak substitutability principle is satisfied), however the substitution will produce a system with incorrect behaviour [12]. Another version of the substitutability principle, referred to as *strong substitutability* is specified by Liskov as follows: *If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T* [11]. Strong substitutability requires behavioural compatibility between the type and subtype.

There has been significant work on program equivalences or preorder relations in the context of concurrent systems. Van Glabbeek examines a selection of 155 such equivalences [20]. The use of inheritance in object-oriented development together with the associated

[1] Computer Science, University of Adelaide, Charles.Lakos@adelaide.edu.au
[2] Electrical Engineering and Computer Science, University of Tasmania, Glenn.Lewis@utas.edu.au

substitutability principle has provided a fertile ground for applying such equivalences or preorders and for developing new ones.

Hence a common approach to the formalisation and characterisation of strong substitutability in concurrent systems is to use a behavioural equivalence relation (for example, [2], [3], [4], [17], and [19] all use an equivalence relation to define strong substitutability). The big question is the appropriate form of equivalence or preorder for subclasses. Some have distinguished subtyping and code reuse [1]. However, there is not a clear or obvious separation of these since subtypes are often defined by code reuse [5]. Also, relations that were originally proposed for subtypes have more recently been used to constrain code reuse instead [2], [3].

In all of this, there is the important issue of the level of practical applicability of theoretical proposals [2], [14]. In other words, are theoretical proposals helpful or too constraining for practitioners? There is another issue, namely whether common practical usage is constrained in such a way that can be exploited by the theory. The former can be illustrated by the notion of weak substitutability, as in Wegner [21]. This is essentially a matter of whether type safety can be guaranteed by compile-time analysis. It has been well recognised that contravariant redefinition of methods is (statically) type safe, while practical experience suggests that covariant redefinition is more appropriate in practice. Thus the development of the Eiffel language has supported covariant redefinition of methods in order to cater for appropriate practical application [13]. However, it has also noted that type safety can be guaranteed in most circumstances (provided there are no so-called *polymorphic cat calls*). Similarly, Palsberg and Schwartzbach have proposed inheritance mechanisms which support covariant redefinition in a type safe way because *contravariant ordering of functions ... is, however, somewhat controversial for object-oriented languages, even though there have been some attempts to justify it on purely methodological grounds* [15]. The recent work on LOOM reflects similar concerns [6].

Strong substitutability proposals

In the field of concurrent systems, full behavioural compatibility can be established by examining the presence or absence of an equivalence or preorder relation.

Clearly any subtype relation should be a *preorder* (that is, it should be *reflexive* and *transitive*) but not *symmetric*. Therefore, proposals for strong substitutability that use an equivalence relation make certain assumptions about the environment of the subtype that ensure the relationship is not symmetric.

The notion of strong substitutability as enunciated by Liskov (above) requires that an object of a type can be replaced by an object of the subtype without being visible to the (unchanged) environment. This is the motivation for the *request substitutability preorder* proposed by Nierstrasz [14]. *Lifecycle inheritance*, as proposed by van der Aalst and Basten [19] involves a combination of *protocol inheritance* and *projection inheritance*. *Protocol inheritance* allows the deletion of (new) methods, or ignoring those methods by the environment, to give the same behaviour. *Projection inheritance* allows for (new) methods which must be called by the environment or, equivalently, can be hidden in order to prove the equivalence. The assumption of a modified environment is not contemplated in Liskov's definition of strong substitutability.

The Liskov definition of strong substitutability does not cater for the situation where subclassing is used to refine the behaviour of the objects of the class. Here, further details are added which will, naturally enough, constrain the behaviour. In other words, the refinement will result in additional behaviour, but if this is abstracted then you should get behaviour that was present in the abstract version. What such refinement does *not* do is introduce additional refined behaviour which corresponds to *new* abstract behaviour. For example, Balzarotti et.al. quote an example of a Printer which is a subtype of a MonoPrinter [2]. (A MonoPrinter will print exactly one copy between the loading and discarding of a print job. On the other hand, a Printer can print multiple copies between the loading and discarding.) In our view, a Printer is not a refinement of a MonoPrinter since it introduces additional abstract behaviour — multiple print copies — which was not present in the abstract version. It may be possible to reformulate the refinement to avoid this problem, but as it stands, it would not be acceptable in our proposed framework.

The above proposals may also require difficult proofs to justify the strong substitutability — proof of bisimilarity [19], [4], proof of finite state machine equivalence [14], or proof of ST-preorder [3]. (Automatic tests for deciding equivalence are inherently complex, for example, in [8] bisimulation is shown to be DEXPTIME-complete for a class of Petri nets known as 1-Safe Petri nets).

Proposed refinements in the context of Coloured Petri Nets

Petri Nets are a well-established formalism for concurrent systems. For example, Figure 1 shows a simple Petri Net for the lifecycle of a library book. The places (shown as circles) indicate state information, while the transitions (shown as rectangles) indicate possible changes of state. The directed arcs indicate how the changes of state affect the state components. Tokens (shown as dots) indicate the current state. Thus, in Figure 1, a library book can be in one of 3 states — *Available, On loan, Overdue* — and the current or initial state shows a book in the *Available* state. In this state, the transition *Borrow* can fire and, in doing so, removes a token from the place *Available* and adds a token to the place *On loan*.

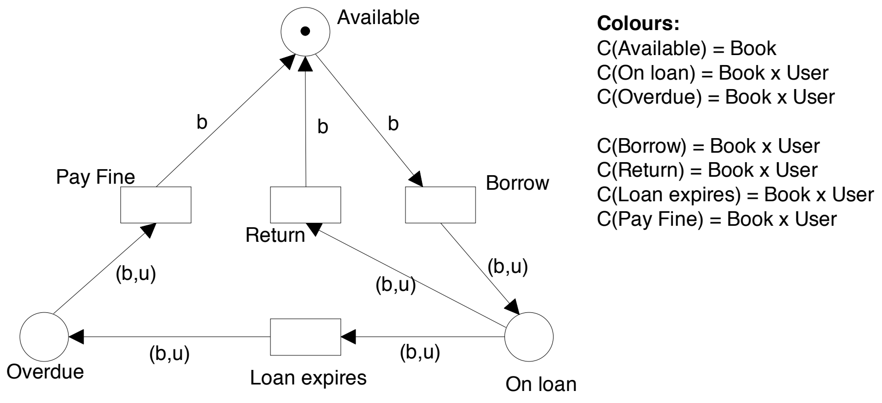


Figure 1: Petri Net for the lifecycle of library books

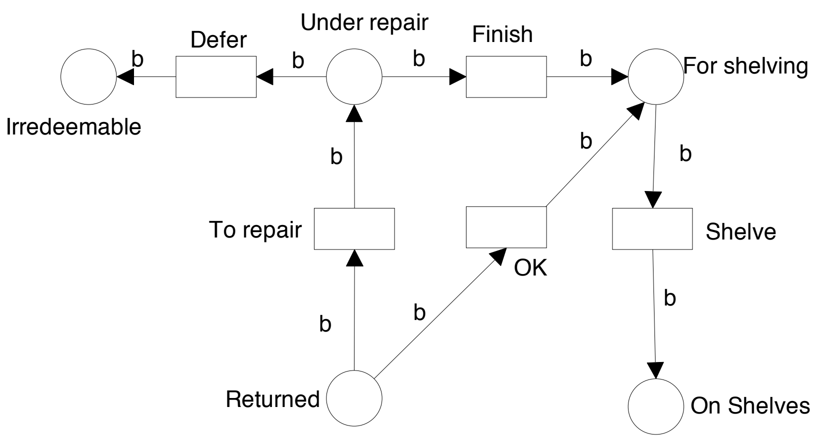
For practical application, Petri Nets are usually extended with the notion of colour. Here, the tokens can have associated data values (or colours), the arcs can indicate which colour of token is to be consumed or generated, and the transitions can have guards to ensure that certain constraints are met. In the example of Figure 1, the colour function *C* indicates how we can attach colours to places (i.e. the associated token colours) and to transitions (i.e. the associated firing modes). The colour for library book tokens would indicate the author, title, ISBN number, etc. Then the arc annotations indicate particular books (variable *b*) and particular users (variable *u*). The values of these variables are determined by the particular firing modes of the transitions.

Equivalences and preorder relations for concurrent systems can be formulated in Petri Net terms, though this tends to be in the context of elementary Petri Nets [16]. A subtype is a refinement of the (more abstract) supertype (recall our introductory discussion on the *is-a* relationship) and therefore, in the context of CPNs, we propose three forms of refinement that we consider appropriate for substitutability. Our focus on refinement, where some abstract behaviours may be lost, results in a definitions for substitutability that is more restrictive than weak substitutability, but less restrictive than strong substitutability (in the sense of full behavioural compatibility). The constraints that we propose have the advantage of being able to be checked statically. Further, if the refinement is *at least as live* as the abstraction, then strong substitutability holds. What we mean by this is as follows. If a refined behaviour has a corresponding abstract behaviour, then the liveness of the refined behaviour will imply the liveness of the abstract behaviour. The reverse does not necessarily hold, but if it does, then we say that the refinement is *at least as live* as the abstraction. (This notion is clarified below in the context of specific refinements.)

The refinements we propose are in the context of net morphisms. These are mappings between nets — from the refined to the abstract net. They specify how components from the refined net are mapped into components of the abstract net, or are simply ignored in the abstraction. They will also indicate how the refined state will map into the abstract state, and how refined behaviours map into abstract behaviours. (The latter will not necessarily be a total function.)

The first form of refinement is called *type refinement*. Here, the types of the tokens and modes are replaced by subtypes. In this context, the types simply define a set of values without any associated behaviour. Each subtype will have an associated projection function which will project subtype values onto supertype values. (Typically, a subtype will introduce additional value components which are simply ignored by the projection function.) In the library book example of Figure 1, we might modify the token type for library books by including additional information such as loan restrictions. This additional information should be propagated round the net, thus requiring corresponding subtypes for transition modes. The constraint we impose on this form of refinement is that the projection of a refined behaviour will (always) correspond to an abstract behaviour. In the example, if we have a refined behaviour in which the additional loan restrictions are ignored, then we will get a behaviour of the original (abstract) net. There is no guarantee that liveness at the abstract level implies liveness at the refined level, contrary to many of the proposals already considered. However, if this is the case (and we say that the refined version is *at least as live as* the abstract version), then strong substitutability holds.

The second form of refinement is called *node refinement*. Here, a net component — a place or a transition — is replaced by a place-bordered or a transition-bordered subnet (respectively). This reflects a situation where some state component (i.e. a place) is refined into a number of components, possibly with associated (internal) actions; or where some action component (i.e. transition) is refined into a number of actions, possibly with associated (internal) state. It is not obvious how this form of refinement is addressed in the proposals considered earlier. In the example of Figure 1, we might refine the place *Available* to indicate that the return of a library book may involve a number of subsidiary states, which take into account the delay in reshelving a book, the possibility of repairs, etc. The node refinement for this place is shown in Figure 2. Note that the border places are *Returned* (for deposit of tokens) and *On Shelves* (for removal of tokens). Thus the arcs *to* the place *Available* in Figure 1 would become arcs *to* the place *Returned* of Figure 2, and the arcs *from* the place *Available* in Figure 1 would become arcs *from* the place *On Shelves* of Figure 2. Similarly, it would be possible to refine the transition *Return book* of Figure 1 into a subnet which would identify a number of subsidiary activities, such as taking the book to the library, having the book stamped, etc.



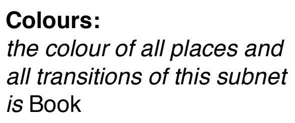

Colours:
the colour of all places and all transitions of this subnet is Book

Figure 2: Subnet indicating the processing of returned books

The constraint we apply on this form of refinement is that (in the case of place refinement) it is possible to project the refined state components into an abstract state, and (in the case of transition refinement) that it is possible to map an abstract action into a sequence of refined actions. (These mappings are constrained to maintain consistency with the abstraction [9]. Thus, for a place refinement, the abstract marking should not change without the refined subnet exchanging tokens with its environment. Similarly, for a transition refinement, the effect of firing the abstract transition should be same as the effect of firing the border transitions of the refined subnet.) In the example quoted, the book tokens in the places of the subnet of Figure 2 would correspond to the tokens in the abstract place *Available* of Figure 1. Similarly, if we refined the transition *Return book* as indicated, then the return of a book would correspond to a sequence of subsidiary actions. Again, there is no guarantee that liveness at the abstract level implies liveness at the refined level. In the example of Figure 2, the book may be become so damaged as to be irrepairable (and added to the place *Irredeemable*) and will never be placed on the shelves again. Similarly, it is not necessarily the case that every refined behaviour corresponds to an abstract behaviour since, in the case of transition refinement, a refined behaviour may correspond to part of an abstract firing. However, if liveness at the abstract level *does* imply liveness at the refined level (and we say that the refined version is *at least as live as* the abstract version), then strong substitutability holds

The third form of refinement is called *subnet refinement*. Here, a subnet is refined by the addition or modification of places, transitions and arcs. In the example of Figure 1, we might introduce extra places and transitions to deal with reservations of books, as shown in Figure 3. Here, transition *Reserve* can fire if there is no reservation on a book, in which case it will now be reserved for user *u*. Then the transition *Borrow* can only fire if the relevant book is not already reserved, or is reserved by the user in question. The constraint we impose on this kind of refinement is that a refined behaviour with the additional components ignored should (always) correspond to an abstract behaviour. This implies that a newly added (or modified) transition cannot modify (except temporarily) an existing place. Once again, liveness at the abstract level does not necessarily imply liveness at the refined level, but if it does, then strong substitutability holds.

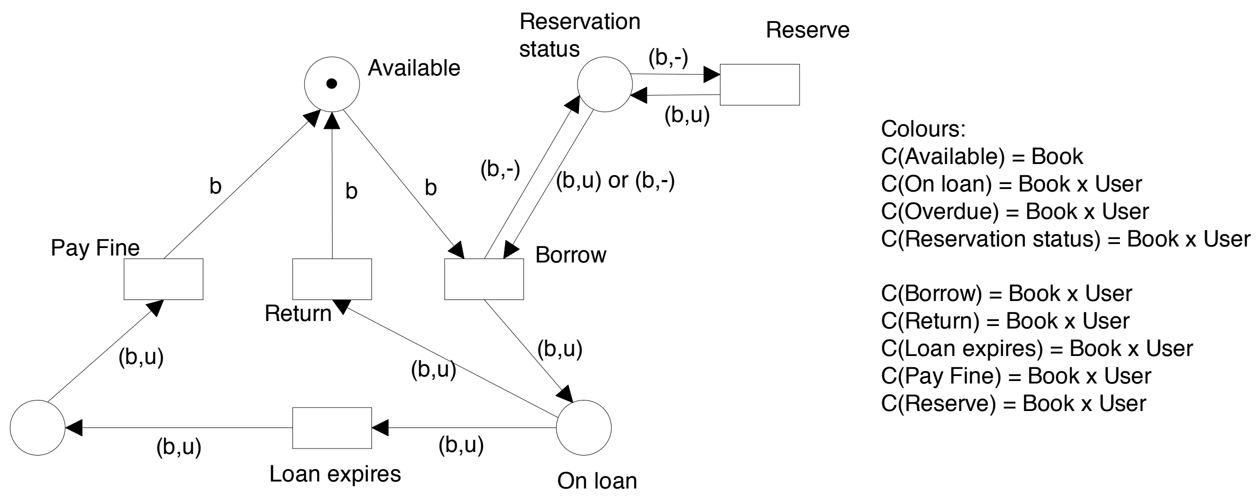


Figure 3: the net of Figure 1 refined to handle reservations

Conclusions

There are a number of proposals for substitutability in the context of concurrent object-oriented systems. It is unclear whether these proposals are overly constrained for practical application.

In the context of CPNs, we have proposed a set of three incremental modifications which lie somewhere between weak and strong substitutability. The constraints that we impose can be checked statically and they have the property that if the refinement is *at least as live as* the abstraction, then strong substitutability holds. (This property *cannot* be checked statically.) An examination of case studies in the literature suggests that the above forms of refinement are applicable in practice [9].

While the above proposals were formulated in the context of CPNs, it turns out that if the CPNs are transformed into the corresponding (elementary) Petri Nets, then the three forms of refinement correspond to recognised net morphisms. The formal definition for these morphisms can be found elsewhere [9], as can the proofs that the composition of refinements is a refinement.

Current work is investigating the extent to which analysis techniques can take advantage of the structure implicit in the above incremental modifications in producing more efficient analysis [10].

Bibliography

[1] P. America, *Designing an object-oriented language with behavioural sub-typing,* Presented at Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, LNCS 489, Springer-Verlag, 1991, pp. 60-90

[2] C. Balzarotti, F. De Cindio, and L. Pomello, *Observation Equivalences for the Semantics of Inheritance,* Presented at Proc. FMOODS 99, Firenze, to appear, Kluwer Academic Publishers, 1999

[3] E. Battiston, A. Chizzoni, and F. De Cindio, *Inheritance and Concurrency in CLOWN,* Presented at 1st Workshop on Object-Oriented Programming and Models of

Concurrency, Turin, Italy, 1995, available from http://wrcm.dsi.unimi.it/PetriLab/ws95/abstract/proc9504.html

[4] O. Biberstein, *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems, PhD Thesis*, in *Départment d'Informatique*, University of Geneva , 1997.

[5] O. Biberstein, D. Buchs, and N. Guelfi, *CO-OPN/2 Applied to the Modeling of Cooperative Structured Editors*, Swiss Federal Institute of Technology in Lausanne, Switzerland 1996, available from ftp://lglftp.epfl.ch/pub/Papers/biberTR96_184.ps.gz.

[6] K. Bruce, L. Petersen, and A. Fiech, *Subtyping is not a good "Match" for object-oriented languages,* Presented at ECOOP'97, Finland, M. Aksit and S. Matsuoka, (Eds.), LNCS1241, Springer, 1997, pp. 104-127

[7] E. W. Dijkstra, *Notes on Structured Programming*, in Structured Programming, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, (Eds.). Academic Press , 1972, pp. 1-82.

[8] J. Esparza and M. Nielsen, *Decibility Issues for Petri Nets - a survey*, Journal of Information Processing and Cybernetics, vol. 30, iss. 3, pp. 143-160, 1994.

[9] C. Lakos and G. Lewis, *A Catalogue of Incremental Changes for Coloured Petri Nets*, University of Adelaide Technical Report, TR99-02 1999

[10] G. A. Lewis and C. A. Lakos, *Incremental Reachability Algorithms TR99-01*, Department Electrical Engineering and Computer Science, University of Tasmania 1999

[11] B. Liskov, *Data abstractions and Hierarchy*, Sigplan Notices, vol. 23, iss. 5, pp. 17-34, 1987.

[12] B. Liskov and J. M. Wing, *A behavioural notion of subtyping*, ACM Transactions on Programming Langugage and Systems, vol. 16, iss. 6, pp. 1811-1841, 1994.

[13] B. Meyer, *Eiffel: The Language*. Prentice Hall , 1992.

[14] O. Nierstrasz, *Regular Types for Active Objects,* Presented at 8th annual conference on Object-Oriented Programming, Washington DC, ACM Sigplan notices, 1993

[15] J. Palsberg and M. I. Schwartzbach, *Object-Oriented Type Systems*. Wiley Professional Computing, Wiley , 1994.

[16] L. Pomello, G. Rozenberg, and C. Simone, *A Survey of Equivalence Notions for Net Based Systems*, in Advances in Petri Nets, LNCS 609, G. Rozenberg, (Ed.). Springer-Verlag (Berlin), 1992, pp. 410-472.

[17] C. Sibertin-Blanc, *Inheritance in Petri Nets and Application to Client/Server*, Toulouse University 1998

[18] A. Taivalsaari, *On the notion of inheritance*, ACM computing surveys, vol. 28, iss. 3, pp. 438-479, 1996.

[19] W. M. P. van der Aalst and T. Basten, *Life-Cycle Inheritance: A Petri-Net-Based Approach*, in *Application and Theory of Petri Nets*, LNCS 1248, Springer-Verlag (Berlin), 1997.

[20] R. van Glabbeek, *The Linear Time -- Branching Time Spectrum II,* Presented at CONCUR'93, Hildesheim, Germany, LNCS 715, Springer-Verlag, 1993, pp. 66-81

[21] P. Wegner and S. B. Zdonik, *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like,* Presented at ECOOP '88, LNCS 322, 1988, pp. 55-77

Behavioural Types in CoOperative Objects.

N. Hameurlain[1,2], C. Sibertin-Blanc[2]

[1] University of Pau,
Avenue de l'Université 64012, Pau, France.
nabil.hameurlain@univ-pau.fr

[2]University of Toulouse I,
Place Anatole France 31042, Toulouse, France.
{nabil,sibertin}@univ-tlse1.fr

Abstract. Behavioural typing and subtyping has proved to be a very useful concept for the support of incremental reuse in the area of object-oriented (O-O) languages. With the emergence of formalisms integrating the O-O approach and Petri nets, the question arises how behavioural subtyping may be supported by such formalisms. We present a formal framework for the definition of behavioural typing in CoOperative Objects, a concurrent Object Oriented language, based upon Client/Server Petri nets. This framework is based upon the preorder and equivalence relations which are considered in the study of concurrent systems, allowing to define various subtyping relations.

1. Motivation

First of all, subtyping has to be distinguished from inheritance [Cook...94]. Inheritance refers to the reuse of some (or all) elements of the definition of a class by another one. Subtyping refers to the use of components, according to the substitution principle enunciated by P. Wegner as follows [Wegner 88]: *s is a subtype of t if an instance of type s may be substituted when an instance of type t is expected.* Inheritance is a matter of structure sharing and it is mainly related to implementation issues, while subtyping is a matter of semantics and it is mainly related to behaviour analysis and validation. Since Petri nets (PN) are mainly used for analysis and validation purposes, they are highly concerned by subtyping.

Subtyping is an essential concept of the object-oriented (O-O) approach as it is both a cognitive tool to ease the understanding of complex systems, and a technical support for software reuse and change. With the emergence of formalisms integrating the O-O approach and the Petri net (PN) theory, the question arises how subtyping may be supported by such formalisms, in order that they benefit from the advantages of this concept. Subtyping has been originally introduced within the framework of data processing and sequential languages, while PN are mainly concerned with the behaviour of concurrent processes. Moreover, it has been pointed out that subtyping within concurrent O-O languages entails the occurrence of many difficult problems or anomalies [Matsuoka...93], [Aksit...94]. Thus, one may think that integrating the subtyping concept within the PN theory also raises some difficulties.

Within concurrent O-O languages, two kinds of subtyping relations are distinguished. The weak, or *interface* subtyping requires that *s is a subtype of t if an instance of type s may be substituted when an instance of type t is expected and no type error will occur* [Wegner 88].

The strong, or *behaviour* form of subtyping requires that *for a system S, s is a subtype of t if an instance of type s may be substituted when an instance of type t is expected and the behaviour of S is unchanged* [Wegner 88]. However, this definition is problematic because the subtype relation between two components depends on the considered context S; thus its use in another context S' needs a specific check, even if S' is an incremental change of S. Such a relation is of a restricted practical interest: if s is a subtype of t, we would like that this holds in any context.

In [Liskov…94], another requirement for the subtyping relation is proposed: *Let φ(x) be a property provable about objects x of a type t. Then φ(y) should be true for objects y of type s where s is a subtype of t.* Although this definition avoids the difficulty of the Wegner's one, it raises another problem entailed by the properties that one wish to be preserved by the subtype relation. Indeed, there are a great number of behavioural properties, and the properties which are expected to be preserved depend on the system under consideration. Using the type t in a context S may make desirable that a property φ be preserved, while using t in a context S' makes desirable that not(φ) be preserved. Thus, the use of Liskov's definition in any context, and therefore for any property, leads to the equivalence of s and t with regard to the properties under consideration, while we are looking for a subtype relation which is a preorder.

This paper addresses the behavioural form of subtyping. It aims to provide a general framework for behavioural subtyping relations in CoOperative Objects [Sibertin 98], a concurrent Object Oriented language based upon Petri nets. For the simplicity, the data processing aspect of CoOperative Objects is avoided and we consider only their behavioural aspect, which is based upon Client/Server nets [Sibertin 93].

In the second chapter, Client/Server nets are informally introduced. Such nets communicate in an asynchronous way by channel places according to a request/reply protocol. In the third chapter we propose two requirements that any C/S nets subtyping relation should satisfy in order to be quit easy to use in practice. The fourth chapter shows that the language semantics defines a relation that satisfies these requirements. An extended version of this paper may be found in [Sibertin...97].

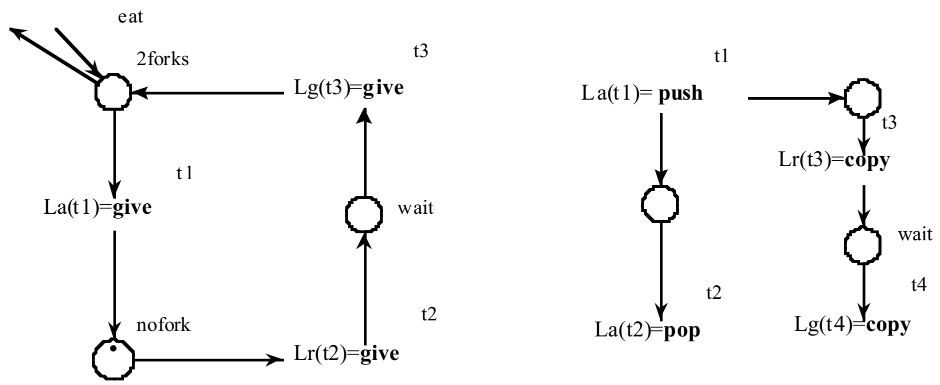
2. Client/Server Nets

A Client/Server net is a labelled Petri net which on the one hand makes some services available to other nets and is capable of rendering these services, and on the other hand can request services from other nets and needs these requests to be fulfilled. Each offered service is associated to one or several transitions which may be requested by other nets, and the service is available when one of these transitions is enabled. These transitions are referred to as *accept-transitions*, and they constitute the interface of a C/S net.

To request a service, a Client net includes two related transitions, one for issuing the request (referred to as a *request-transition)* and another for retrieving the Server's reply (referred to as a *get-transition)*. Moreover, these transitions are connected through a *waiting-place* in such a way that the occurrence of a request-transition makes its matching get-transition enabled. Thus there is a one-to-one correspondence between transitions which request and get services: a service is requested iff it can be got.

A *Server net* is a C/S net which accepts at least one service, that is SA $\neq \emptyset$, where SA is the set of Accepted Services; a *Client net* is a C/S net which requests at least one service, that is SR $\neq \emptyset$, where SR is the set of Requested Services. This distinction allows to focus either upon the server side of a C/S net or its client side. For a Server net S, the set *La(S)* $\subseteq$ SA*, called its *Supply*, is the set of service sequences that S can accept to process. Symmetrically for a Client net C, *Lr(C)* = *Lg(C)* $\subseteq$ SR*, called its *Demand,* is the set of service sequences that C can request or get. As we will see, the Supply of a C/S net is a good picture of its server side and may be considered as its *specification*, whereas its Demand is a good picture of its client side and may be considered as its *requirement*.

Figure 1 shows two simple C/S nets. The net (b) is a buffer which provides the push and pop services, and requests the copy service whenever it receives a push request (when the label of a transition is not empty, it is written inside its box). The net (a) is a dinning philosopher. When the philosopher has no fork, the t2 transition requiring the give service is enabled. Then t3 may occur to get forks. When there is a token into the 2forks place, the eat transition is enabled as well as the t1 transition which accepts the give service.



SA = {give} = SR
(a) A Dinning Philosopher C/S net

SA ={push, pop}, SR = {copy}
(b) A Buffer C/S net

Figure 1: Client/Server nets

A Client net C and a Server net S are *composable* if $SR_C \subseteq SA_S$. Then they may be *composed* into a C/S net *CP = Cp(C, S),* where *Cp: Net × Net* $\longrightarrow$ *Net* ∪ *{undefined}* is the composition function which combines a couple of C/S nets into a third one. When composing a Client net with a Server net, each accept-transition of this latter is provided with an *entry-place* for receiving the requests and a *result-place* for storing the replies. Then, the Client net is connected with the Server through these communication places by an arc from each request-transition towards the suitable entry-place and an arc from the suitable result-place towards each get-transition. Figure 2 shows the composition N = Cp(P2, P1) of two copies of the Philosopher C/S net of Figure 1 with an appropriate initial marking. In the following, we will always assume that nets which are composed are composable, without recalling the hypothesis $SR_C \subseteq SA_S$.

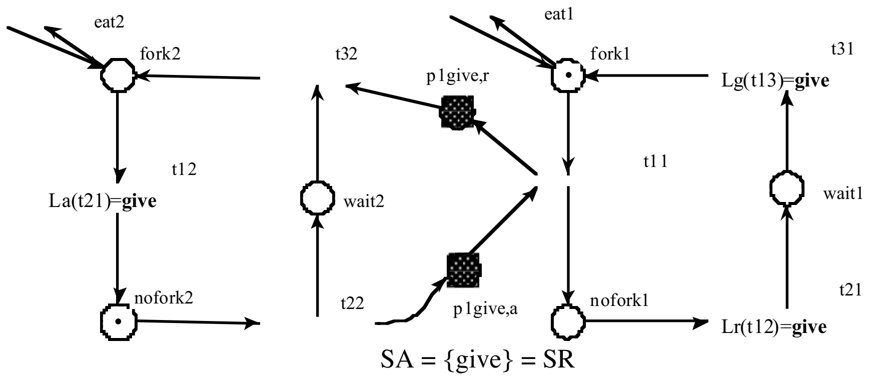


Figure 2: The composition of two Philosophers, CP(P2, P1).

C/S nets communicate in an asynchronous way by channel places, according to a request/reply protocol: when a Client net requests a service to a Server net, it puts a token into a Server's entry place for that service; then, the Server processes the request and supplies a token in the result-place of the service; the communication completes when the Client retrieves this result token. As a consequence, C/S nets obey the request/reply protocol and the encapsulation principle with regard to their behaviour, and thus they benefit from their advantages. Thanks to the PN theory, they are also provided with formal semantics of intra and inter-object concurrency. Finally, C/S nets communicate in an asynchronous mode and thus enjoy the expressiveness of Petri nets composed by *fusion of places*. (The fusion of places consists in merging a set of places into a single one surrounded by the arcs of the initial places). The request/reply protocol allows to analyse the behaviour of a system in an incremental way as Petri nets composed by fusion of transitions [Vogler 92]: the behaviour of a compound C/S net may be deduced from the behaviour of its components [Hameurlain 98].

3. Requirements for behavioural subtyping in C/S nets

Our proposal is to base subtyping relations on the preorder and equivalence relations which have been introduced to compare the behaviour of concurrent systems, such as the state, language, complete language, failure, readiness, deadlock, divergence, simulation and bisimulation preorder and/or equivalence relations (see [Pomello...92], [Glabbeek...89, 90], [Vogler 92] for a comparative study of these relations). Indeed, these relations are defined on a formal basis, without regard for the peculiarities of a given language, so that they are suitable for the study of the behavioural properties of any concurrent system. Moreover, they have been extensively studied, and a lot of results about their properties have been obtained. Summing up, such relations seem to be the best theoretical framework to compare the behaviour of concurrent objects.

In C/S nets, a behavioural subtyping relation $\prec_H$ is a preorder satisfying the following requirements, where N_1, N_2, and N are any C/S nets:

(I) $N_2 \prec_H N_1 \;\Rightarrow\; \forall N,\ Cp(N, N_2) \prec_H Cp(N, N_1)$

(II) $N_2 \prec_H N_1 \;\Rightarrow\; \forall N\ (Cp(N, N_1) \approx_{*1} N \;\Rightarrow\; Cp(N, N_2) \approx_{*2} Cp(N, N_1))$

where $\approx_{*i}$ are some behavioural equivalence relations.

Notice that N_2 subtype of N_1 is denoted $N_2 \prec_H N_1$, in accordance with the idea that N_2 extends the capabilities of N_1.

Requirement (I) asserts that $\prec_H$ is compositional for the net composition operation. This property is necessary to ensure the compatibility of the subtyping relation with the incremental design of systems; if it does not hold, there is no way to relate the behaviour of Cp(S, Cp(N, N_1)) and Cp(S, Cp(N, N_2)) when $N_2 \prec_H N_1$. The corresponding requirement for the weak subtyping is that e. g. arrays of 64-bits integers are a subtype of arrays of 32-bits integers if 64-bits integers are a subtype of 32-bits integers.

In requirement (II), the consequence ‘‘Cp(N, N_2) $\approx_{*2}$ Cp(N, N_1)’’ formalises the substitutability principle as the impossibility for N to distinguish between the behaviours of N_1 and N_2 with regard to the properties which are accounted for by the $\approx_{*2}$ relation. Indeed, the set of properties of Cp(N, N_1) which can be preserved by Cp(N, N_2) depends on the way $\prec_H$ is defined; for instance, if $\prec_H$ is defined by an interleaving semantics, it is not reasonable to require that Cp(N, N_2) satisfies the same partial order properties as Cp(N, N_1). Requirement (II) asserts that N_2 may be substituted for N_1 if $N_2 \prec_H N_1$ and in addition the net N which is considered satisfies Cp(N, N_1) $\approx_{*1}$ N. Without such a restriction, the subtyping relation would be an equivalence relation, since it certainly exists some trivial net N_0 such that Cp(N_i, N_0) $\approx_{*1}$ N_i for any net N_i. In addition, it is of interest to know that Cp(N, N_1) $\approx_{*2}$ Cp(N, N_2) only if Cp(N, N_1) behaves properly; in the opposite case, nothing is gained by knowing that Cp(N, N_1) and Cp(N, N_2) are both a wrong system! Now, the top-down approach leads to consider that the correct behaviour of $Cp(N, N_1)$ is characterised by its relation with the behaviour of N, which is formalised by the condition $Cp(N, N_1) \approx_{*1} N$.

Thus, to prove that a preorder relation is a subtyping relation for a composition operation Cp, we just have to prove that (I) it is compositional for Cp and that (II) holds for some couples of equivalence relations $\approx_{*1}$ and $\approx_{*2}$. Then, the practical use of a subtyping relation is quite easy: once $N_2 \prec_H N_1$ has been established, it is enough to check whether $Cp(N, N_1) \approx_{*1} N$ holds to be allowed to substitute N_2 for N_1 in a safe way, whatever is the net N under consideration.

The first advantage of the above requirements for the behavioural subtyping is that they have a wide range of applicability and suffer no ambiguity. They provide the concept of subtype with a formal definition, and are based on a theory (the theory of the behavioural preorder and equivalence relations) which is relevant for any concurrent language supporting a composition mechanism. They extend Liskov's approach, since requirement (II) automatically determines what are the properties of $Cp(N, N_1)$ which are preserved by $Cp(N, N_2)$ when N_2 is a subtype of N_1, so that no specific verification has to be done. Indeed, most of the $\prec$ and $\approx$ relations enjoy results of the following kind, φ being some property:

if $N_2 \prec N_1$, then $\varphi(N_1) \Rightarrow \varphi(N_2)$

if $N_2 \approx N_1$, then $\varphi(N_2) \Leftrightarrow \varphi(N_1)$

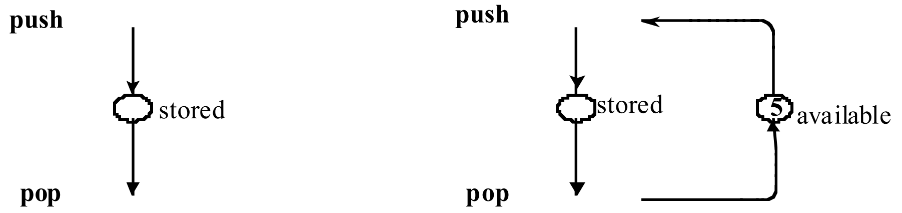
A second advantage of the above requirements is that they do not define just one subtyping relation but allow to define several such relations. Faced with a given system, the designer may use the subtyping relation preserving the properties which are of importance for that system.

4. The semantics of compound C/S nets

In [Hameurlain 98], some semantics have been considered to define subtyping relations. Namely, we have shown that the language preorder, denoted $\prec_{La}$, where $S_2 \prec_{La} S_1$ iff $La(S_1) \subseteq La(S_2)$, is a subtyping relation satisfying **requirement (I)**, i.e. for any Client net C, we have $Cp(C, S_2) \prec_{La} Cp(C, S_1)$ as soon as $S_2 \prec_{La} S_1$; extending the Supply of a Server net also extends the Supply of its composition with any Client net. According to requirement (I) and the fact that for any Client C and Server S, $C \prec_{La} Cp(C, S)$ [Hameurlain 98], the $\prec_{La}$ preorder is a subtyping relation for C/S nets, and we have the following form of **requirement (II)**:

$$S_2 \prec_{La} S_1 \text{ and } Cp(C, S_1) \approx_{La} C \;\Rightarrow\; Cp(C, S_2) \approx_{La} Cp(C, S_1).$$

As an example, consider the unbounded buffer Server net *UB* shown in Figure 3 (a) and the 5-size bounded buffer Server net *BB* shown in Figure 3 (b). It is easy to prove that $UB \prec_{La} BB$ holds. Therefore, for any Client net C, if C and Cp(C, BB) have the same behaviour when only the transitions of C are observed, then the same holds for C and Cp(C, UB), so that Cp(C, BB) and Cp(C, UB) have the same behaviour on the transitions of C. On the other hand, if the behaviour of Cp(C, BB) is more restricted than the behaviour of C (C $\prec_{La}$ Cp(C, BB)), we can only say that the behaviour of Cp(C, UB) stands in between the behaviours of Cp(C, BB) and C (C $\prec_{La}$ Cp(C, UB) $\prec_{La}$ Cp(C, BB)).



SA = {push, pop}
(a) The unbounded buffer Server net UB

SA = {push, pop}
(b) The bounded buffer Server net BB

Figure 3: UB subtype of BB.

In order to easily apply requirement (II), one needs a criterion to decide whether *Cp(C, S_1) $\approx_{La}$ C* holds where C is any C/S net. In [Sibertin97], we introduce behavioural relationship between a Client net C and a Server net S which is compatibility: S is compatible with C iff C and Cp(C, S) are trace equivalent on the transitions of C, and then *Cp(C, S_1) $\approx_{La}$ C*. In addition, we prove that S is compatible with C iff Lr (C) = La(S); therefore, the Language subtyping relation is quite easy to use, since it is enough to compute only once the Supply and the Demand of the involved C/S nets. Then, the hypothesises of requirement (II) are established by comparing the Supply of the Servers, and comparing the Demand of the Client with the Supply of the first Server.

The Language semantics is often regarded as too weak for many applications because it is not able to express branching time and deadlock behaviour. Therefore, we have addressed other semantics to define subtyping relations such as Simulation, Failure and Bisimulation semantics. In [Hameurlain 99] we show that Simulation and Failure preorders, denoted $\prec_{Fa}$ and $\prec_{sima}$, are subtyping relations for C/S nets. However, compatibility is not sufficient to decide whether *Cp(C, S_1) $\approx_H$ C* holds where H $\in$ {Fa, sima} and C is any C/S net, since it rests upon a linear time semantics. In the case of the simulation subtyping relation, we show that *Cp(C, S_1) $\approx_{sima}$ C* holds if C on its Requested Services is similar to S_1 on its Accepted Services, while for the failure subtyping relation, we have introduced behaviour relationship between the Client C and the Server S which is transparency: S is transparent for C iff C and

Cp(C, S) are bisimilar on the transitions of C, and prove that transparency constitutes a criterion to decide whether $Cp(C, S_1) \approx_{Fa} C$ holds.

The above requirements for subtyping relations give a greater place to Clients than to Servers since they compare Cp(C, S) with C and not with S, where S is a Server composable with the Client C. They are in accordance with a top-down approach where Cp(C, S) is considered as a refinement of the net C: C is a specification of (a part of) a system, while Cp(C, S) is one implementation. Since the client side of C vanishes into Cp(C, S), only the server side of C can be compared with Cp(C, S), so that the appropriate semantics are based upon the accept-transitions.

According to a bottom-up approach, Cp(C, S) would be considered as being an extension of the C/S net S, and we would be interested in comparing the behaviour of Cp(C, S) with the one of S. In fact, adapting our requirements to this approach is just achieved by focusing on the client side of C/S nets instead of their server side. For instance, consider the $\prec_{Lr}$ preorder, where $C_2 \prec_{Lr} C_1$ iff $Lr(C_1) \subseteq Lr(C_2)$. $\prec_{Lr}$ satisfies the following requirements, where C_1, C_2 and S are C/S nets:

*Requirement **(I')** : $C_2 \prec_{Lr} C_1 \Rightarrow \forall S, Cp(C_2, S) \prec_{Lr} Cp(C_1, S)$.*

*Requirement **(II')** : $C_2 \prec_{Lr} C_1 \Rightarrow \forall S\ (Cp(C_1, S) \approx_{Lr} S \Rightarrow Cp(C_1, S) \approx_{Lr} Cp(C_2, S))$.*

The La and Lr subtyping relations are not exclusive and may be used together. In this case, we can use the fact that La(S) = Lr(C) iff Cp(C, S) preserves both the server side of C ($Cp(C, S) \approx_{La} C$) and the client side of S ($Cp(C, S) \approx_{Lr} S$). As an example, consider the Philosopher C/S net introduced in chapter 2. Cp(P1, P2) is bisimilar both to P1 on the transitions of P1 and to P2 on the transitions of P2.

5. Discussion

This paper proposes a formal framework for the definition of behavioural subtyping relations in CoOperative Objects, a concurrent Object Oriented language, based upon Client/Server Petri nets. This framework consists of requirements that any subtyping relation ought to satisfy, and it is based upon the preorder and equivalence relations commonly considered in the study of concurrent systems. It is in accordance with the principle of substitutability and extends Liskov's approach.

The starting point of the presented approach are the ideas on behavioural inheritance in O-O languages based upon Petri nets. In CLOWN, inheritance is defined as preorder based upon a semantic accounting for observable places, in such a way that a derived net class "extends the parents' specification and specialises them in a restricted domain" [Battiston…95], but the consequences for net composition are not addressed. [Biberstein…95] defines an inheritance relation in COOPN/2 which ensures that the composed nets are bisimilar for a given net; this relation is based upon an Abstract Data type semantics, and it seems to be difficult to verify; our approach is based upon the preorder and equivalence relations which are considered in the study of concurrent systems, and leads to define subtyping relations in CoOperative Objects that enjoy nice properties.

References

[Aksit...94] M. Aksit, J. Bosch, W. van der Sterren, L. Bergmans
Real-time specification inheritance anomalies and real-time filter; In Proc. ECOOP' 94, M. Tokoro and R. Pareschi Eds, LNCS 821, Springer-Verlag.

[Battiston…93] E. Battiston, A. Chizzoni, F. De Cindio
Inheritance and concurrency in CLOWN; in Proc. of the first Workshop on Object-Oriented Programming and Model of Concurrency. G. Agha and F. De Cindio (eds.), Turin(I), June 27 1995.

[Biberstein…95] O. Biberstein, D. Buchs
Structured Algebraic Nets with Object-Orientation; in Proceedings of the first Workshop on Object-Oriented Programming and Models of Concurrency; Turin (I), June 27 1995.

[Cook...94] W.R. Cook, W.L. Hill, P.S. Canning
Inheritance is not Subtyping; in Theoretical Aspects of Object-Oriented Programming, C.A. Gunter & J.C. Mitchel Eds., MIT Press, 1994.

[Glabbeek...89] R. Van Glabbeek, U. Goltz
Equivalence notions for concurrent systems and refinement of actions; MFCS 89, LNCS 379, Springer-Verlag 1989.

[Hameurlain 98] N. Hameurlain
Behavioural Inheritance in Petri Nets; Ph.D Thesis, Toulouse I University (F), March 1998.

[Hameurlain 99] N. Hameurlain
Behavioural Inheritance in Client/Server Petri Nets; in proceedings of the IEEE 1999 International Conference on Systems, Man and Cybernetics (IEEE/SMC'99), 12-15 October 1999, Tokyo, Japan.

[Liskov 93] B. H. Liskov
A New Definition of the Subtype Relation; In Proc. 7th European Conf. on Object-Oriented Programming, Kaiserlautern (G), Springer-Verlag 1993.

[Liskov...94] B. H. Liskov, J. M. Wing
A Behavioral Notion of Subtyping; in ACM Trans. on Programming Languages and Systems, Vol 16, n° 6, Nov. 1994.

[Matsuoka...93] S. Matsuoka, A. Yonezawa
Inheritance anomaly in Object-Oriented concurrent Programming Languages; in Research Directions in Concurrent Object-Oriented Programming, G. Agha, P. Wegner and A. Yonezawa Eds, MIT Press, 1993.

[Pomello...92] L. Pomello, G. Rozenberg, C. Simone
A Survey of Equivalence Notions for Net Based System; Advances in Petri Nets 1992; G. Rozenberg Ed., LNCS 609, Springer-Verlag 1992.

[Sibertin 93] C. Sibertin-Blanc
A Client-Server Protocol for the Composition of Petri Nets; in Proc 14th International Conference on Applications and Theory of Petri Nets, Chicago (Il), June 1993, LNCS 691, Springer-Verlag.

[Sibertin....97] C. Sibertin-Blanc, N. Hameurlain
Behavioural Inheritance in Petri Nets and Application to Client/Server Nets; Internal Report of University Toulouse I (F), December 1997; ftp:\\www.univ-tlse1.fr/local/syroco/paper.

[Sibertin 98] C. Sibertin-Blanc
CoOperative Objects: Principles, Use and Implementation. In Object Oriented Programming and Models of Concurrency, G. Agha & F. De Cindio Eds., LNCS, Springer-Verlag, to appear 1998.

[Vogler 92] W. Vogler
Modular Construction and Partial Order Semantics of Petri Nets; LNCS 625, Springer-Verlag, 1992.

[Wegner 88] P. Wegner
Inheritance as an Incremental Modification Mechanism, or What Is and Isn't Like; in Proc ECOOP 88, Oslo (Norway), Springer-Verlag.

A Concurrent Object Calculus with Types that Express Sequences[*]

Christof Peter and Franz Puntigam
Technische Universität Wien, Institut für Computersprachen
Argentinierstraße 8, A-1040 Vienna, Austria
E-mail: {christof,franz}@complang.tuwien.ac.at

Abstract

Sequencing of messages specified by types of objects is desirable especially in concurrent systems. Types in popular concurrent object calculi cannot support sequencing of messages. We present a calculus that supports sequencing of messages and compare it to the calculus of Vasconcelos and Honda. Type safety in our calculus does not allow a certain kind of nondeterminism supported by other calculi.

1 Introduction

Objects in a concurrent system communicate by passing sequences of messages. The set of acceptable messages may depend on previously received messages. Consider, for example, a buffer object understanding "put" and "get": If empty, the object accepts only a "put" message; "get" may lead to an error. We expect that a type system ensures statically that only acceptable messages are sent. This condition must hold even though in distributed systems it is possible that new objects are added at run time. Especially in concurrent systems it is important that the set of messages accepted by an object can change dynamically, expressing synchronization conditions [8].

It is very difficult (if not impossible) to develop such type systems for process calculi like the π-calculus [7]. For example, the type system developed by Pierce and Sangiorgi [9] is quite powerful, but it cannot describe sequencing of values communicated along a channel. A possibility to express sequences of communication in the π-calculus are graph types [15]. But graph types are just an observation of the behavior of the whole system and they are not able to express nondeterminism. The use of logical languages like the modal μ-calculus to express sequences [4] results in a complex type checking system which supports finite types only.

Types in the object-based process model of Vasconcelos and Honda [12, 13] also cannot describe the sequencing of messages. The calculi proposed by Abadi and Cardelli [1] focus on sequential object-oriented systems; hence, sequencing of messages is not even a topic for these calculi. A concurrent version of one of these calculi is presented in [5], simply

[*]This work was supported by the FWF, project "Static Process Types for Active Objects" (P12703-INF).

by extending an imperative object calculus with primitives for parallel composition and for synchronization via mutexes. This concurrent calculus even supports subtyping, but acceptable message sequences cannot be expressed in types.

On the other side, process types [10] support sequencing of messages for a calculus developed just for that purpose. What is the reason why sequencing of messages works for this calculus, but not for the others? This is the question we want to answer in this paper. We argue that the major difference is a certain kind of nondeterminism supported by the other calculi, but not by our calculus: The sender of a message may not know which receiver will deal with the message; hence, the sender cannot know how the receiver will behave. We regard this kind of nondeterminism as rather unimportant for our purposes.

2 Our calculus

Our calculus is based on active objects behaving as in the actor model [2, 3]. Messages are passed asynchronously; the sender is not blocked until the message is received. Each object has a single identifier, a modifiable behavior and a single thread of execution. Different from the actor model, our communication system ensures that all messages are handled in the same (logical) ordering they were sent. A property of this communication mechanism is that when the receipt of a message changes the receiver's behavior (as specified in the receiver's type), the sender knows about the change already when sending the message. This property[1] is a precondition for process types [10].

Two infinite sets of names are considered given: $\mathbb{X}$ contains names (denoted by $a, b, c, \ldots$ and also $v, x, y \ldots$) used as object identifiers and parameters to be substituted by object identifiers. A set $\mathbb{C}$ contains labels ($l, \ldots$) with $\mathbb{X} \cap \mathbb{C} = \emptyset$. Names are underlined if they occur at positions where object identifiers shall be substituted for them. An underlined occurrence of a name binds all following free occurrences of this name.

For each meta-symbol e, a sequence $e_1, \ldots, e_n$ can be abbreviated by $\tilde{e}$ if n is arbitrary. Likewise, $\{\tilde{e}\}$ denotes the set containing $e_1, \ldots, e_n$, and $|\tilde{e}|$ the length of the sequence. For each operator $\circ$, an expression $\tilde{e} \circ \tilde{g}$ (with $|\tilde{e}| = |\tilde{g}|$) stands for $e_1 \circ g_1, \ldots, e_i \circ g_n$. All names in an underlined parameter list, for example $\underline{\tilde{x}}$, are pairwise different. The simultaneous substitution of e_i for all free occurrences of g_i (with $i = 1, \ldots, |\tilde{e}|$) in f is denoted by $f[\tilde{e}/\tilde{g}]$, where $\tilde{e}$ and $\tilde{g}$ are sequences of the same length, and $\tilde{g}$ is a list of pairwise different names. The set of all names occurring free in any expression in $\{\tilde{e}\}$ is denoted by *free*$(\tilde{e})$.

This is the syntax of objects:

$$P ::= a\langle B/\tilde{\mu}\rangle \qquad \mu ::= l(\tilde{v}) \qquad M ::= l(\underline{\tilde{x}})B$$
$$B ::= \{\tilde{M}\} \mid a \triangleleft \mu.B \mid (\underline{a})@\mu.B \mid @\mu$$

An active object (or process, denoted by $P, Q, \ldots$) of the form $a\langle B/\tilde{\mu}\rangle$ consists of an object identifier a, a current behavior B and a queue of received and not yet handled messages $\tilde{\mu}$.

A message ($\mu, \nu, \xi, \ldots$) of the form $l(\tilde{x})$ associates a message selector l with a (possibly empty) list of names $\tilde{x}$ used as actual parameters.

[1]Other communication mechanisms like synchronous message passing, rendezvous, procedure calls, etc. also have this property.

A method $(M, N, \dots)$ of the form $l(\underline{\tilde{x}})B$ specifies the behavior B preceded by a guard defining a method name (or message selector) l and formal parameters $\tilde{x}$. There are two kinds of methods: Globally visible creator methods are invoked when a new object is created or an object changes its own behavior. A message-handling method (that belongs to an object) is invokable if its guard matches the next message in the object's message queue. The invocation accepts a message sent to the object. The message used for invoking a creator message is specified in the behavior (following @). After invocation, the object behaves as B with the actual parameters substituted for the formal parameters.

A behavior (denoted by $B, C, \dots$) specifies if the object accepts a message and executes the corresponding message-handling method, sends a message, creates a new object or changes its behavior by invoking a creator method. An object behaving as $\boldsymbol{\{}\tilde{M}\boldsymbol{\}}$ selects a message-handling method in $\{\tilde{M}\}$ for invocation, depending on the next message in the queue. An object of behavior $a \triangleleft \mu.B$ sends a message μ to the object with identifier a and then behaves as B. One of behavior $(\underline{a})@\mu.B$ creates a new object (with a new object identifier substituting a) before behaving as B. The new object initially behaves as $@\mu$. The behavior $@\mu$ equals the behavior of a creator method matching μ after substituting actual parameters for formal parameters; this is, a creator method is invoked.

This is the syntax of types (denoted by $\pi, \rho, \sigma, \tau, \dots$):

$$\begin{aligned}
\tau &::= \boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{r}] \;\mid\; (\underline{t})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{r}] \;\mid\; \tau_1 \| \tau_2 \;\mid\; t \\
\alpha &::= l(\tilde{\tau})[\tilde{r}_1][\tilde{r}_2] \\
r &::= l \;\mid\; l^\infty
\end{aligned}$$

A type of the form $\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{r}]$ consists of a set of message descriptors (each denoted by $\alpha, \beta, \gamma, \dots$) and a multi-set of tokens $(r, s, \dots)$. We call $\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}$ the static part and $[\tilde{r}]$ the dynamic part of the type. An object's type changes dynamically by removing tokens from and adding tokens to the type's dynamic part when messages are accepted.

A token is either a label l or an infinite token l^∞ standing for a very large (or infinite) number of tokens l.

Message descriptors are of the form $l(\tilde{\tau})[\tilde{r}][\tilde{s}]$, where l is the message selector and the $\tilde{\tau}$ are types of parameters. Message selector and parameter types together are considered to be the signature of a message. The other parts of a message descriptor are the in-set $[\tilde{r}]$ and the out-set $[\tilde{s}]$. A message of the specified signature is acceptable (by an object of a type containing the message descriptor in its static part) if each token in the in-set occurs also in the dynamic part of the object's type. When accepting the message, the type is updated according to this message descriptor by removing all tokens contained in the in-set from the type's dynamic part and adding all tokens contained in the out-set to the dynamic part.

A type of the form $\sigma \| \tau$ denotes the parallel composition of two types; $\sigma \| \tau$ can be split into σ and τ.

A type of the form $(\underline{t})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{r}]$ is a recursive version of the type. The type parameter t can (recursively) occur in $\tilde{\alpha}$. A type parameter can occur in each position where a type can occur.

The static part of a window's type can be expressed by:

$$\mathit{Window} \overset{\text{def}}{=} \boldsymbol{\{}\text{iconify}()[\text{displ}][\text{icon}], \text{uniconify}()[\text{icon}][\text{displ}]\boldsymbol{\}}$$

When a window receives a message “iconify”, a token “displ” is removed from the type’s dynamic part, and “icon” is added. When receiving “uniconify”, “icon” is removed and “displ” added. An object of type *Window*[displ] first accepts “iconify”, then “uniconify”, then “iconify” again, and so on.

We distinguish between the type of an object and the type of a reference to an object. The type of an object describes all messages accepted by the object. The type of a reference describes the messages that can be sent to the object via this reference. It may contain only a part of the objects type information. The combination of the types of all references to an object must keep the contract that the type of the object specifies. This can be ensured with the mechanism of “type splitting”: When a new reference (alias) is created, the type information is split in an old part (associated with an already existing reference) and a new part (associated with the new reference). For example, a reference’s type *Window*[displ] can be split into *Window*[displ] and *Window*[], but not into *Window*[displ] and *Window*[displ] (because there is just a single token “displ”.

It is possible for a client to follow the changes of an object visible through a reference by keeping track of the messages that were sent via this reference. For example, when a message is passed via a reference of type *Window*[displ], the reference will be updated to *Window*[icon]. Type updating can be performed statically by means of abstract interpretation.

3 Reduction Semantics

Now we formalize the semantics of our object calculus. We introduce congruence relations on behaviors, methods, messages and objects. Each relation depends on a set of creator methods; the set represents a program. Then, we define an execution relation on system configurations; these are sets of objects reflecting snapshots of object systems.

First, we define structural equivalence of objects and their constituents: Two behaviors, methods, messages and objects are structurally equivalent if they can be made equal by renaming bound parameters (α-conversion), and swapping and duplicating methods in method collections of the form $\boldsymbol{\{}\tilde{M}\boldsymbol{\}}$.

Definition 1 *The behavior congruence $B \cong_{\{\tilde{M}\}} C$ on behaviors, method congruence $N \cong_{\{\tilde{M}\}} N'$ on methods, message congruence $\mu \cong_{\{\tilde{M}\}} \nu$ on messages and object congruence $P \cong_{\{\tilde{M}\}} Q$ on objects, each relation depending on a set $\{\tilde{M}\}$ of creator methods, are the least congruences closed under structural equivalence and the following rules:*

$$@l(\tilde{y}) \cong_{\{\tilde{M}\}} B[\tilde{y}/\tilde{x}] \quad (l(\underline{\tilde{x}})B \in \{\tilde{M}\}) \tag{call$_\cong$}$$

$$a\langle \boldsymbol{\{} l(\underline{\tilde{x}})B, \tilde{N} \boldsymbol{\}}/l(\tilde{y}), \tilde{\mu}\rangle \cong_{\{\tilde{M}\}} a\langle B[\tilde{y}/\tilde{x}]/\tilde{\mu}\rangle \tag{select$_\cong$}$$

$$a\langle a \triangleleft \xi . B/\tilde{\mu}\rangle \cong_{\{\tilde{M}\}} a\langle B/\tilde{\mu}, \xi\rangle \tag{self send$_\cong$}$$

The set $\{\tilde{M}\}$ is used by (call$_\cong$) to replace invocations of creator methods with the corresponding behaviors. The rules (select$_\cong$) and (self send$_\cong$) on objects accept messages and send messages to the objects themselves.

Sending messages to other objects and creating new objects are not dealt with by object congruence: These actions are regarded as having effects that become visible outside of objects. Object congruence relates objects that behave in the same way when viewed from outside.

A system configuration $\eta_{\{\tilde{M}\}}$ is a set of objects with pairwise different object identifiers. The creator method set $\{\tilde{M}\}$ in the subscript specifies the program executed by the object system the system configuration represents a snapshot of which.

Definition 2 *The a-execution relation* $\eta_{\{\tilde{M}\}} \overset{a}{\to} \theta_{\{\tilde{M}\}}$ *on system configurations is the least closure of these rules:*

$$\frac{\tilde{P} \cong_{\{\tilde{M}\}} \tilde{P}' \quad \eta_{\{\tilde{M}\}} \cup \{\tilde{P}'\} \overset{a}{\to} \eta_{\{\tilde{M}\}} \cup \{\tilde{Q}'\} \quad \tilde{Q}' \cong_{\{\tilde{M}\}} \tilde{Q}}{\eta_{\{\tilde{M}\}} \cup \{\tilde{P}\} \overset{a}{\to} \eta_{\{\tilde{M}\}} \cup \{\tilde{Q}\}} \qquad (\mathrm{congr}_{\to})$$

$$\eta_{\{\tilde{M}\}} \cup \{a\langle b{\lhd}\xi.B/\tilde{\mu}\rangle, b\langle C/\tilde{\nu}\rangle\} \overset{a}{\to} \eta_{\{\tilde{M}\}} \cup \{a\langle B/\tilde{\mu}\rangle, b\langle C/\tilde{\nu}, \xi\rangle\} \qquad (\mathrm{send}_{\to})$$

$$\eta_{\{\tilde{M}\}} \cup \{a\langle(\underline{x})@\xi.B/\tilde{\mu}\rangle\} \overset{a}{\to} \eta_{\{\tilde{M}\}} \cup \{a\langle B[b/x]/\tilde{\mu}\rangle, b\langle @\xi[b/x]/\rangle\} \qquad (\mathrm{new}_{\to})$$

The execution relation $\eta_{\{\tilde{M}\}} \overset{*}{\to} \theta_{\{\tilde{M}\}}$ *is the least reflexive and transitive closure of all a-execution relations* $\eta_{\{\tilde{M}\}} \overset{a}{\to} \theta_{\{\tilde{M}\}}$ *with* $a \in \mathbb{X}$.

Rule (congr$_{\to}$) states that object congruence can be applied before and after each execution step. When an object sends a message to another object (send$_{\to}$), the message is appended to the receiver's message queue. A new object created using (new$_{\to}$) gets an empty message queue and a fresh constant name as object identifier.

4 Comparing the expressiveness

Vasconcelos' calculus [12], called TyCO, describes processes and agents using this grammar:

$$P ::= a{\lhd}l{:}\tilde{v} \mid a{\rhd}[l_1{:}A_1 \& \cdots \& l_n{:}A_n] \mid P, Q \mid |x|P \mid X(\tilde{v}) \mid A(\tilde{v}) \mid \mathbf{let}\ X = A\ \mathbf{in}\ P$$
$$A ::= (\tilde{x})P \mid \mathbf{rec}\ X.A$$

The P are called processes; they are directly executable. The A are called agents; they are a kind of templates from which processes can be instantiated. These agents can be simply abstracted processes $(\tilde{x})P$ or recursively abstracted agents **rec** $X.A$.

We omit the specification of the equivalence relation $\equiv$ over processes which covers α-conversion, reordering of methods and recursion.

TyCO is based on objects of the form $a{\rhd}[l_1{:}A_1 \& \cdots \& l_n{:}A_n]$ where a is the location and $[l_1{:}A_1 \& \cdots \& l_n{:}A_n]$ an unordered collection of methods labeled by pairwise distinct labels. In a method $l{:}(\tilde{v})P$, $\tilde{v}$ represents the formal parameters and P the body of the method. A process sending a message is denoted by $a{\lhd}l{:}\tilde{v}$. The communication rule is

$$|\tilde{w}|(a{\lhd}l_i{:}\tilde{v}, a{\rhd}[l_1{:}A_1 \& \ldots \& l_n{:}A_n], \tilde{Q}) \to |\tilde{w}|([\tilde{v}/\tilde{w}_i]P_i, \tilde{Q})$$

where $A_i = (\tilde{w}_i)P_i$. The equivalence rules may be applied before and after reduction.

If a message is sent to an object, it may immediately be processed. There may be several objects with the same identifier. The sender of the message never knows which object handles the message. With this property we cannot guarantee an important precondition for the use of process types because senders cannot keep track with changes of the receivers' state.

We think that it is not necessary (for a majority of applications) to support this kind of nondeterminism. This nondeterminism may even make it difficult for programmers to understand programs. Hence, we propose to eliminate this kind of nondeterminism. In terms of Vasconcelos' calculus, we allow just one process of the form $a{\rhd}[l_1{:}A_1\&\cdots\&l_n{:}A_n]$ for each location a.

We will now sketch how TyCO programs (with the restriction that there is just one reading atomic process per location) can be mapped into our calculus. TyCO agents correspond to our creator methods. TyCO processes are always executed concurrently (denoted by $P, Q, \ldots$). To create an object corresponding to a TyCO process, we have to map its behavior to a creator method and create an object using this creator method. We simply map agent variables $X, Y, \ldots$ of TyCO to labels in our calculus.

An instance $A(\tilde{v})$ corresponds to the call of a creator method $l(\underline{\tilde{v}})\lceil Q\rceil$ corresponding to $\lceil A\rceil$, where $\lceil Q\rceil$ is the "behavior" of $\lceil A\rceil$, and $\tilde{v}$ are the parameters of $\lceil A\rceil$.

An applied agent-variable $X(\tilde{v})$ must be bound by a recursive agent or a let declaration. We create it when it is bound.

We map a process of the form $\lceil \mathbf{let} X = A \ \mathbf{in}\ P\rceil$ by creating a new object $(\underline{x})@l(\tilde{v}).\mathbf{\{\}}$ using the creator method $l(\tilde{v})\lceil Q\rceil$, where $\lceil Q\rceil$ is the behavior of A. The parameters must correspond to the parameters specified by $\lceil A\rceil$. Recursion via $\mathbf{rec}\ X.A$ is mapped to a call of a creator method $l(\tilde{v})\ldots@l(\tilde{x})$.

In general, we create a new concurrent object for each atomic process in TyCO. An atomic process $\lceil a{\rhd}[l_1{:}A_1\&\cdots\&l_n{:}A_n]\rceil$ is mapped to $a\langle\mathbf{\{}l_1(\tilde{x_1})\lceil A_1\rceil,\ldots,l_n(\tilde{x_n})\lceil A_n\rceil\mathbf{\}}/\rangle$; the message queue is empty. We provide a creator method $l()\mathbf{\{}l_1(\tilde{x_1})\lceil A_1\rceil,\ldots,l_n(\tilde{x_n})\lceil A_n\rceil\mathbf{\}}$ and create an object with $(\underline{a})@l().\mathbf{\{\}}$. For each atomic process $\lceil a{\lhd}l{:}\tilde{v}\rceil$ in TyCO, we create a new object $b\langle a{\lhd}l(\tilde{v}).\mathbf{\{\}}/\rangle$. We map a restriction $\lceil |x|P\rceil$ to an object creation $(\underline{x})@l(\tilde{y}).\lceil P'\rceil$, where $l(\underline{\tilde{v}})\mathbf{\{}l_1(\tilde{x_1})\lceil A_1\rceil,\ldots,l_n(\tilde{x_n})\lceil A_n\rceil\mathbf{\}}$ expresses the behavior of the single object reading from x, and P' corresponds to P after removing this object from P.

Now we sketch the reverse encoding from our calculus to TyCO. An active object $a\langle B/\tilde{\mu}\rangle$ corresponds to a TyCO object representing a and a further object sending messages to a if requested by a. The additional object represents the sequential message queue in each object of our calculus. Creator methods $l(\underline{\tilde{v}})B$ correspond to (recursive) agents $\lceil(\tilde{x})P\rceil$ or $\lceil\mathbf{rec}\ X.A\rceil$. Method handling methods $\mathbf{\{}l_1(\tilde{x_1})\lceil A_1\rceil,\ldots,l_n(\tilde{x_n})\lceil A_n\rceil\mathbf{\}}$ correspond to $a{\rhd}[l_1{:}A_1\&\cdots\&l_n{:}A_n]$. We translate message sending $a{\lhd}l(\tilde{v}).\lceil A\rceil$ (and the following sequential behavior) to $|b|a'{\lhd}l{:}\tilde{v}b, b{\rhd}[l'{:}A]$, where a' is the location of the message queue of object a, and l' is the label of a message sent to b by the message queue after putting the message into the queue. We translate object creation $(\underline{a})@l(\tilde{v}).\lceil B\rceil$ to $|a||b|A(\tilde{v}), A', B$, where a is a location corresponding to the new object, b a location corresponding to the object's message queue, A an agent expressing the same behavior as the creator method l, and A' the implementation of the message queue.

The calculus of Vasconcelos and Honda can be encoded into the π-calculus and vice versa [14]. Hence, all that has been said in this section also holds for the π-calculus.

5 Extensions of our Calculus

In our calculus it is also possible to introduce the kind of nondeterminism available in the other calculi for the rare cases it may be needed. However, when the receiver of a message is not known in advance, all possible receivers must have the same type. This type must not depend on changeable states. A clear distinction between object identifiers with just one reader and such with multiple readers is necessary. Such an extension might be useful in systems where one of several autonomous agents offering the same services is selected dynamically on its availability.

Types in our model support subtyping [10]. The expressiveness of types can be improved by adding higher-level constructions to the calculus [11]. For example, the use of genericity and if-then-else constructions (where the equality of object references or subtyping relationships are used as conditions) improves the flexibility while still being statically typed.

6 Conclusion

The π-calculus [7] and TyCO [12] allow that more than one agent reads from a channel. The sender never knows which agent reads a message. This kind of nondeterminism makes it impossible to use types like process types [10], where sequencing of messages can be specified. When removing this (rarely needed) kind of nondeterminism, sequencing of messages can be supported.

Unlike the untyped basic calculi, the expressiveness of statically typed calculi (using process types) can be improved by using higher-level constructions (like conditional execution). We think it is necessary to develop a process calculus together with its type system to get a statically typed setting with sufficient expressiveness. A goal of our future work is to further examine the interactions between process calculi and type systems to improve the expressiveness of statically typed concurrent programs.

References

[1] Martin Abadi and Luca Cardelli, editors. *A Theory of Objects.* Springer-Verlag, 1996.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[3] Gul Agha, Ian A. Mason, Scott Smith, and Carolyn Talcott. Towards a theory of actor computation. In *Proceedings CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, pages 565–579. Springer-Verlag, 1992.

[4] R. Amadio and M. Dam. Toward a modal theory of types for the π-calculus. In *Proc. FTRTFT'96,* Lecture Notes in Computer Science, 1135:347–365, 1996.

[5] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings HLCL '98*, September 1998.

[6] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3), 1977.

[7] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.

[8] Oscar Nierstrasz. Regular types for active objects. *ACM SIGPLAN Notices*, 28(10):1–15, October 1993. Proceedings OOPSLA'93.

[9] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings LICS'93*, 1993.

[10] Franz Puntigam. Coordination requirements expressed in types for active objects. In *ECOOP '97*, Lecture Notes in Computer Science, Jyväskylä, Finland, June 1997. Springer-Verlag.

[11] Franz Puntigam. Dynamic type information in process types. In David Pritchard and Jeff Reeve, editors, *Proceedings EuroPar '98*, number 1470 in Lecture Notes in Computer Science, Southampton, England, September 1998. Springer-Verlag.

[12] Vasco T. Vasconcelos. Typed concurrent objects. In *Proceedings ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, 1994.

[13] Vasco T. Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic π-calculus. In *Proceedings CONCUR'93*, July 1993.

[14] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *1st International Symposium on Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 460–474. Springer-Verlag, 1993.

[15] Nobuko Yosihida. Graph types for monadic mobile processes. In *16th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386, Hyderabad, India, 1996. Springer-Verlag.

Explicit behavioral typing for object interfaces

Elie Najm
École Nationale Supérieure des Télécommunications
Networks and Informatics Dept.
46, Rue Barrault
75013 Paris. France
e-mail: Elie.Najm@enst.fr

Abdelkrim Nimour
Schlumberger – Test & Transactions
Smart Cards Products, R&D Lab.
BP 620-12. 50, Av. Jean Jaurès
92542 Montrouge Cedex. France
e-mail: knimour@slb.com

Abstract

In this paper we describe an approach for typing objects with non-uniform service availability. We define behavioral types for object interfaces based on labeled transition systems that specify the succession of available methods (services) at an interface. Each transition label is a method signature. In addition, each interface has to be declared public or private. A private object interface has only one client at a time and offers non-uniform services depending on the "protocol" the client and the server have agreed on. On the other hand, a public interface can have multiple clients at the same time and is required to perform the same services for all its potential clients: the services on a public interface are uniform.

1 Introduction

Let us consider the typical example of the one place buffer. Only one element at a time can be stored in this buffer. So it offers repeatedly the method `put` then the method `get`. An implementation of this buffer in Java would be as shown in figure 1.

In this example the `Buffer` class has a boolean member: `empty`. Initially, `empty` is `true`. The service `put` is available only if `empty` is `true` and conversely the service `get` can be performed only if `empty` is `false`.

The inspection of the code of the buffer reveals that to be correctly used, first the `put` method must be called and then the `get` method can be called, but it is not possible to know this just by examining the type of the buffer. Thus, it is very difficult to detect this kind of errors (service not available), because the "synchronization constraints" are hidden in the object code.

```
public class Buffer{
  private Boolean empty;
        ...
  public Buffer(){
    empty := true;
    ...
  }
  public void put(Object o) throws ServiceNotAvailable {
    if(!empty) throw new ServiceNotAvailable();
    else{
      empty := false;
      ...
    }
  }
  public Object get() throws ServiceNotAvailable {
    if(empty) throw new ServiceNotAvailable();
    else{
      empty := true;
      ...
    }
  }
}
```

Figure 1: The one place buffer in Java

In addition, in the case of a parallel or distributed language, the buffer can have two clients (a producer and a consumer) accessing concurrently to its services. In this case, the producer and the consumer have to coordinate their actions to avoid a service call when it is not available.

Our approach consists precisely in defining a type language that solves this problem. An interface type is defined by a labeled transition systems where each transition is a method signature. We also distinguish between public interfaces that have a uniform service availability and private ones that have dynamically changing service offers. To enforce coordination between objects, we do not allow duplication of private interface references such that only one client at time can invoke services of a private interface. Multiple clients must then "cooperate" to access safely to the services.

An interface type specifies:

- the **behavior type** of the interface, i.e., the succession of the available services;
- the **mode** of the interface: public or private;
- the **role** of the interface: client or server. An object that have the client role (noted `!`) of an interface can invoke services offered at this interface. An object that have the server role (noted `?`) can offer services on the interface.

Remark: Usually, the client role of an interface is called a reference to this interface, whereas the server role of an interface is simply called the interface.

This paper is a revised and extended version of the work presented in [8].

2 Behavior Types

A behavior type is a triplet (E, x, r), noted $E \triangleright rx$, where x is the behavior type identifier, r its emitting/receiving capability ($r \subset \{!, ?\}$) and E a finite set of equations of the form $x_k = e_k$ where each x_k is a behavior type identifier that appears once and only once in the left-hand side of an equation and e_k is an expression of the form:

$$\boxed{e ::= \sum_{i=1}^{n} m_i(\tilde{\rho_i}\tilde{x_i}); x'_i} \text{ with } l \neq l' \Rightarrow m_l \neq m_{l'}$$

where:

- each m_i is a method name. We call $\mathcal{M}$ the set of method names.
- each $\tilde{\rho_i}\tilde{x_i}$ is a list of the form: $\rho_i^1 x_i^1, \cdots, \rho_i^k x_i^k$ describing the behavior types of the arguments of method m_i. Each x_i^j is a behavior type identifier and ρ_i^j a role, i.e., client (!), server (?) or both (!?).
- each x'_i is a behavior type identifier.

There exists a predefined behavior type, **nil**, that does not appear in the left-hand side of any equation.

In our model we consider only deterministic types, i.e., if there is a choice between two messages of a behavior type expression, the names of the messages must be different.

Note: In the following we will consider, without loss of generality[1] , that all our behavior types are defined in the same environment, E. We will then omit this environment when writing a behavior type, i.e., $E \triangleright rx$ will be written rx. We let X, Y, Z range over behavior types.

2.1 Type Behavior

In this section, we define the dynamic aspects of behavior types. A behavior type X that performs action a and then evolves to a new behavior type X' is denoted by the transition:

$$X \xrightarrow{a} X'$$

An action is a method signature annotated with a role: $\rho m(X_1, \ldots, X_n)$.

[1]Two behavior types defined in two distinct environments can be simply redefined in the union of these environments with an appropriate renaming of type variables.

Behavior type transitions are defined by the following rule:

$$\boxed{\frac{x = \sum_{i=1}^{n} m_i(\rho_i^1 x_i^1, \cdots, \rho_i^k x_i^k); x_i'}{r\ x \xrightarrow{\rho m_j(\rho_j^1 x_j^1, \cdots \rho_j^k x_j^k)} r\ x_j'} \quad \text{where} \quad \begin{array}{l} \rho \in \{!, ?\} \text{ and} \\ \rho \subset r \text{ and} \\ 1 \leq j \leq n \end{array}} \qquad \text{(Type-Act)}$$

Note : The behavior type, r **nil** where r is any (valid) set of capabilities, cannot evolve.

We define now the function $possible(X)$, which is the set of method names that can be handled by the behavior type X.

Definition 1 *(possible)*

$possible(X) = \{m \in \mathcal{M}, \exists X' \ \exists \tilde{Y} \mid X \xrightarrow{\rho m(\tilde{Y})} X'\}$

2.2 Behavior Type Equivalence

Bisimulation is a behavioral equality relation over behavior types. This definition is very close to the definition of process bisimulation in CCS (see [6]). The principal difference is that in our case the actions need to be bisimilar too.

Definition 2 *(Bisimulation)*

A binary relation β over behavior types is a bisimulation if $(X_1, X_2) \in \beta$ implies :

i) $X_1 \xrightarrow{\rho m(\tilde{Y}_1)} X_1' \Rightarrow X_2 \xrightarrow{\rho m(\tilde{Y}_2)} X_2'$ *and* $(X_1', X_2') \in \beta, (\tilde{Y}_1, \tilde{Y}_2) \in \beta$

ii) $X_2 \xrightarrow{\rho m(\tilde{Y}_2)} X_2' \Rightarrow X_1 \xrightarrow{\rho m(\tilde{Y}_1)} X_1'$ *and* $(X_2', X_1') \in \beta, (\tilde{Y}_2, \tilde{Y}_1) \in \beta$

Definition 3 *(Behavior type equivalence)*

Two behavior types X_1 and X_2 are equivalent, noted $X_1 \sim X_2$, if and only if, there exists a bisimulation β such that $(X_1, X_2) \in \beta$.

We define a predicate $uniform(X)$ formalizing the notion of uniform behavior type.

Definition 4 *(uniform)*

$uniform(X)$ holds if $X \xrightarrow{\rho m(\tilde{Y})} X'$ implies $X \sim X'$

2.3 Behavior Subtypes

Our subtyping relation is based on a simulation relation ([6]).

Definition 5 *(Simulation)*

A binary relation $\mathcal{S}$ over behavior types is a simulation relation if $(X_1, X_2) \in \mathcal{S}$ *implies :*

$$i)\ X_1 \xrightarrow{?m(\tilde{Y_1})} X_1' \Rightarrow X_2 \xrightarrow{?m(\tilde{Y_2})} X_2' \ \textit{and}\ (X_1', X_2') \in \mathcal{S}, (\tilde{Y_1}, \tilde{Y_2}) \in \mathcal{S}$$

$$ii)\ X_2 \xrightarrow{!m(\tilde{Y_2})} X_2' \Rightarrow X_1 \xrightarrow{!m(\tilde{Y_1})} X_1' \ \textit{and}\ (X_2', X_1') \in \mathcal{S}, (\tilde{Y_2}, \tilde{Y_1}) \in \mathcal{S}$$

Informally, an interface type X_1 is a subtype of a type X_2 iff:

client case: the processable messages of X_2 are a subset of the processable messages of X_1.

server case: the processable messages of X_1 are a superset of the processable messages of X_2.

This is to guaranty the safe substitution of a supertype by its subtype.

Definition 6 *(Subtypes)*

A behavior type X_1 is a subtype of a behavior type X_2, noted $X_1 \preceq X_2$, if there exists a simulation $\mathcal{S}$ such that $(X_1, X_2) \in \mathcal{S}$.

---

3 The Object Calculus

Our object calculus is a variant of the π-calculus ([7]) with syntactic sugar for method definition. We also restrict the choice operator to receiving actions only. Communication is asynchronous for public interfaces and by rendez-vous for private interfaces. Our calculus is in fact asynchronous. We have chosen rendez-vous communication for private interfaces for technical reasons and for simplifying the presentation. The formal syntax of the calculus is given in figure 2.

3.1 Operational Semantics

We present the operational semantics of object configurations in two steps. We first define a structural congruence relation and then we give a reduction relation that specifies how object configurations evolve.

$$
\begin{array}{rcl}
I & ::= & u\!:\!T \\
Dcl & ::= & A[\tilde{I}] = B \\
Recep & ::= & u[m_1(\tilde{I}_1) = B_1, \cdots, m_n(\tilde{I}_n) = B_n] \\
B & ::= & \mathbf{0} \\
 & | & u.m(\tilde{\rho v}) > B \\
 & | & \sum_{i=1}^{n} Recep_i \\
 & | & A[\tilde{\rho v}] \\
 & | & \textbf{create } A[\tilde{\rho v}] > B \\
 & | & \textbf{new } u\!:\!T \textbf{ in } B \\
M & ::= & [u.m(\tilde{v})] \\
C & ::= & B \\
 & | & \textbf{new } u\!:\!T \textbf{ in } C \\
 & | & M|C \\
 & | & C|C
\end{array}
$$

Symbol	**Signification**
T	interface type
A	behavior name
u, v	interface name
$m, m_1, \cdots, m_n$	method name
ρ	role
B	behavior expression
M	message
C	object configuration
Dcl	behavior definition

Figure 2: The Object Calculus Syntax

3.1.1 Structural Congruence

Bound and free variable occurrences are defined as usual.

Definition 7 *(Bound and free variables)*

*Interface names occurrences appearing in the scope of a behavior definition ($A[\tilde{I}] = B$), a reception action ($u[m_1(\tilde{I}_1) = B_1, \cdots, m_n(\tilde{I}_n) = B_n]$) or an interface creation (**new** $u\!:\!T$ **in** C) are bound, otherwise they are free.*

Definition 8 *(Substitution)*

The substitution $C[v/u]$ denotes the simultaneous replacement of all the free occurrences of u by v in C.

We can define the congruence relation, noted $\equiv$.

The first structural rules state that the choice operator between receptions is commutative and associative:

$$\mathit{Recep}_1 + \mathit{Recep}_2 \equiv \mathit{Recep}_2 + \mathit{Recep}_1$$

$$(\mathit{Recep}_1 + \mathit{Recep}_2) + \mathit{Recep}_3 \equiv \mathit{Recep}_1 + (\mathit{Recep}_2 + \mathit{Recep}_3)$$

The parallel operator is also commutative and associative and 0 is its neutral element:

$$C_1|C_2 \equiv C_2|C_1$$
$$(C_1|C_2)|C_3 \equiv C_1|(C_2|C_3)$$
$$C|0 \equiv C$$

The order of the introduction of the interfaces in meaningless:

$$\textbf{new } u_1 : T_1 \textbf{ in } (\textbf{new } u_2 : T_2 \textbf{ in } C) \equiv \textbf{new } u_2 : T_2 \textbf{ in } (\textbf{new } u_1 : T_1 \textbf{ in } C) \quad \text{for } u_1 \neq u_2$$

The two last rules are the π-calculus scope extrusion and alpha-conversion:

$$(\textbf{new } u : T \textbf{ in } C_1)|C_2 \equiv \textbf{new } u : T \textbf{ in } (C_1|C_2) \quad \text{if } u \text{ is not free } C_2$$

$$\textbf{new } u : T \textbf{ in } C \equiv \textbf{new } v : T \textbf{ in } C[v/u] \text{ if } v \text{ is not free } C$$

3.1.2 Reduction Rules

We define now the reduction rules that specify how a configuration can evolve by making a single and atomic step. The evolution of configurations may generate messages. The syntax of a message is similar to the syntax of the method invocation except that the message has no continuation. To avoid any ambiguity messages will be written between brackets: $[u.m(\tilde{v})]$.

In some cases, the reductions will be annotated by a "label": $C_1 \xrightarrow{l} C_2$. The labels are of the form $u.m$ where u is an interface name and m a method name. These labels will be used to keep track of type evolution.

To distinguish between the interactions on public and private interfaces, the emitting actions on public interfaces will be noted "$_*$" whereas the emitting actions on private interfaces will remain noted "." (dot).

The reduction relation is defined by the following rules:

$$\begin{array}{ll} & u[\cdots, m(\tilde{v}:\tilde{T}) > B, \cdots] + \Sigma \mathit{Recep}_j \\ | & \\ & u.m(\tilde{w}) > B' \end{array} \xrightarrow{u.m} B[\tilde{w}/\tilde{v}]|B' \qquad \text{(R1)}$$

The synchronization on a private interface is by *rendez-vous*.

$$u_* m(\tilde{v}) > B \longrightarrow B|[u.m(\tilde{v})] \qquad \text{(R2)}$$

The invocation of a public interface generates a message ($[u.m(\tilde{v})]$) whose behavior is to synchronize with this interface.

$$\begin{array}{l} u[\cdots, m(\tilde{v}:\tilde{T}) > B, \cdots] + \Sigma Recep_j \\ | \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow B[\tilde{w}/\tilde{v}] \\ [u.m(\tilde{w})] \end{array} \tag{R3}$$

A message is absorbed by the appropriate (public) interface and then vanishes.

$$\frac{A[\tilde{u}:\tilde{T}] \overset{Dcl}{=} B}{A[\tilde{v}] \longrightarrow B[\tilde{v}/\tilde{u}]} \tag{R4}$$

We simply replace the instantiation of the object by its corresponding behavior.

$$\frac{A[\tilde{u}:\tilde{T}] \overset{Dcl}{=} B}{\textbf{create } A[\tilde{v}] > B' \longrightarrow B[\tilde{v}/\tilde{u}]|B'} \tag{R5}$$

The created object runs in parallel with the continuation of the creating behavior.

$$\frac{C_1 \xrightarrow{a} C_1'}{C|C_1 \xrightarrow{a} C|C_1'} \tag{R6}$$

This rule states that if a sub-configuration can evolve to a new one then the whole configuration can evolve too.

$$\frac{C \xrightarrow{u.m} C' \quad T \xrightarrow{\rho\, m(\tilde{T})} T'}{\textbf{new } u{:}T \textbf{ in } C \longrightarrow \textbf{new } u{:}T' \textbf{ in } C'} \text{ with } \rho \in \{!, ?\} \tag{R7}$$

When there is an interaction on a private interface we update its type.

$$\frac{C \xrightarrow{v.m} C'}{\textbf{new } u{:}T \textbf{ in } C \xrightarrow{v,m} \textbf{new } u{:}T \textbf{ in } C'} \text{ with } v \neq u \tag{R8}$$

The type of the variable u is not changed by an interaction on an other interface v.

$$\frac{C \longrightarrow C'}{\textbf{new } u{:}T \textbf{ in } C \longrightarrow \textbf{new } u{:}T \textbf{ in } C'} \qquad \text{(R9)}$$

A non annotated reduction does not impact the **new** operator.

$$\frac{C_1' \xrightarrow{l} C_2' \quad C_1 \equiv C_1' \quad C_2 \equiv C_2'}{C_1 \xrightarrow{l} C_2} \qquad \text{(R10)}$$

This last rule states that configurations that are equivalent (according to $\equiv$) behave equally.

3.2 Static Semantics

3.2.1 Introduction

An interface type is a couple (μ, X), noted $\mu\, X$, where μ is the mode of the interface (public or private) and X its behavior type. Interface types will be ranged over by the meta-variable T.

We extend type transitions and subtyping to interface types.

Definition 9 *(Interfaces type behavior)*

An interface type can perform the same actions as its behavior type: $\mu X \xrightarrow{\rho m(\mu_1 Y_1, \cdots, \mu_n Y_n)} \mu X'$ *if* $X \xrightarrow{\rho m(Y_1, \cdots, Y_n)} X'$.

Definition 10 *(Interface subtyping)*

An interface type T_1 is a subtype of an interface type T_2 if they have the same mode μ and if the behavior type of T_1 is a subtype of the behavior type of T_2: $\mu X_1 \preceq \mu X_2$ *if* $X_1 \preceq X_2$.

Definition 11 *(Role addition)*

Role addition is defined by the following equations and is undefined otherwise:

$$\begin{array}{rcll} (\mu r\, x) \oplus ? & = & \mu(r \cup \{?\})\, x & \textit{if } ? \notin r \\ (\textit{private } r\, x) \oplus ! & = & \textit{private } (r \cup \{!\})\, x & \textit{if } ! \notin r \\ (\textit{public } r\, x) \oplus ! & = & \textit{public } (r \cup \{!\})\, x & \end{array}$$

The client role of public interfaces can be “added” without restriction. In the other cases the role addition is permitted only if the role is not present yet in the interface type.

Notation : The expression $T \oplus !?$ denotes the successive additions of the client and server roles: $(T \oplus !) \oplus ?$.

Definition 12 *(Interface type addition)*

Interface type addition is defined by the following equations and is undefined otherwise:

$$\begin{array}{lcll}
(\mu r_1\, x) \oplus (\mu r_2\, x) & = & \mu((r_1\, x) \oplus r_2) & \text{if } r_2 \neq \emptyset \\
(\mu r_1\, x) \oplus (\mu r_2\, x) & = & \mu r_1\, x & \text{if } r_2 = \emptyset
\end{array}$$

The second equation is necessary because role addition is not defined for an empty capability.

A typing context Γ is a set of bindings of the form: $x : T$ or $A : (T_1, \ldots, T_n)$. The set of interface names appearing in a context Γ is called the context domain: $Dom(\Gamma)$. The static semantics is given using the following judgments:

Judgment	Signification
$\Gamma \vdash u{:}T$	in the context Γ the interface u has type T
$\Gamma \vdash A{:}(\tilde{T})$	in the context Γ the object A has type $(\tilde{T})$
$\Gamma \vdash B$	in the context Γ the behavior B is well-typed
$\Gamma \vdash C$	in the context Γ the configuration C is well-typed

The context extension, noted $\Gamma, u : T$ or $\Gamma, A : (\tilde{T})$, is defined such that $\Gamma, u{:}T \vdash u{:}T$ and $\Gamma, A{:}(\tilde{T}) \vdash A{:}(\tilde{T})$.

We extend $\oplus$ to typing contexts as follows:

Definition 13 *(Context addition)*

The addition of two contexts is defined by the following equation and is undefined otherwise:

$(\Gamma_1, u{:}T_1) \oplus \Gamma_2, u{:}T_2 = (\Gamma_1 \oplus \Gamma_2), u{:}(T_1 \oplus T_2)$, *if* $Dom(\Gamma_1) = Dom(\Gamma_2)$ *and* $T_1 \oplus T_2$ *and* $\Gamma_1 \oplus \Gamma_2$ *are defined*

Definition 14 *(Addition of interface roles)*

The addition of an interface role to a context, noted $\Gamma \oplus \rho u$ *is defined if* $u \in Dom(\Gamma)$ *by the following equations and is undefined otherwise:*

$$\begin{array}{lcll}
(\Gamma, u{:}T) \oplus \rho u & = & \Gamma, u{:}(T \oplus \rho) & \\
(\Gamma, u{:}T) \oplus \rho v & = & (\Gamma \oplus \rho v), u{:}T & \text{if } u \neq v
\end{array}$$

3.3 Typing Rules

The basic idea underlying our typing system is to guaranty that each object uses the interfaces in a way compatible with their declared behavior type. These rules ensure also that there is no undesired duplication of interface roles.

$$\frac{\Gamma \vdash u\!:\!\mu\, r\, x \;\Rightarrow\; ((? \in r) \Rightarrow x = \mathbf{nil})}{\Gamma \vdash \mathbf{0}} \tag{T1}$$

This rule introduces the notion of receiving obligation. If the behavior type of a server can perform an action (is different from **nil**) then this server cannot stop.

$$\frac{\begin{array}{l} \Gamma, u\!:\!T_2 \vdash B \\ T_1 \xrightarrow{!m(X_1,\cdots,X_n)} T_2 \\ \Gamma \vdash v_1\!:\!\mu_1 r_1\, x_1 \cdots \Gamma \vdash v_n\!:\!\mu_2 r_n\, x_n \\ \rho_1\, x_1 \preceq X_1 \cdots \rho_n\, x_n \preceq X_n \\ \Gamma \oplus \tilde{\rho}\tilde{v}\ \textbf{defined} \end{array}}{\Gamma \oplus \tilde{\rho}\tilde{v}, u\!:\!T_1 \vdash u.m(\rho_1 v_1, \cdots, \rho_1 v_1) > B} \tag{T2}$$

This rule checks that the actual arguments of method m, v_1 of type $\mu_1 r_1\, x_1$, $\ldots$, v_n of type $\mu_n r_n\, x_n$, are subtypes of the expected argument types $X_1, \ldots, X_n$. The behavior type of u and the capabilities of the arguments $\tilde{v}$ are updated in the new context.

$$\frac{\begin{array}{l} I \subset [1..n] \\ \Gamma, u\!:\!T_i', \tilde{v_i}\!:\!\tilde{T_i} \vdash B_i \text{ for } i \in I \\ T \xrightarrow{?m_i(\tilde{T_i})} T_i' \text{ for } i \in I \\ \mathit{possible}(T) = \{m_i \mid i \in I\} \end{array}}{\Gamma, u\!:\!T \vdash u[m_1(\tilde{v_1}\!:\!\tilde{T_1}) = B_1 \cdots, m_n(\tilde{v_n}\!:\!\tilde{T_n}) = B_n]} \tag{T3}$$

This rule checks that each behavior is well-typed in a context where the method parameters have their declared type. Servers must offer all the services that are defined by their behavior type, but are free to offer additional services.

$$\frac{\Gamma \vdash \mathit{Recep}_1 \ldots \Gamma \vdash \mathit{Recep}_n}{\Gamma \vdash \sum_{i=1}^{n} \mathit{Recep}_i} \tag{T4}$$

The context is simply propagated as it is in all the branches of a choice.

$$\frac{\begin{array}{l} \Gamma, u\!:\!\text{private } X \vdash C \\ \mathit{Capabilities}(X) = \{!, ?\} \end{array}}{\Gamma \vdash \ \textbf{new}\ \ u\!:\!\text{private } X\ \textbf{in}\ C} \tag{T5.1}$$

A newly created interface has both roles [2]: client and server.

$$\frac{\Gamma, u\!:\!\text{public } X \vdash C \qquad \mathit{Capabilities}(X) = \{!, ?\} \qquad \mathit{uniform}(X)}{\Gamma \vdash \textbf{new } u\!:\!\text{public } X \textbf{ in } C} \tag{T5.2}$$

In the case of a public interface, its behavior type must be uniform.

$$\frac{\Gamma, A\!:\!(\tilde{T}), \tilde{u}\!:\!\tilde{T} \vdash B}{\Gamma, A\!:\!(\tilde{T}) \vdash A[\tilde{u}\!:\!\tilde{T}] = B} \tag{T6}$$

This rule enforces that, in the case of an object declaration, the behavior of the object is well-typed in a context where the declaration formal arguments have their declared types.

$$\frac{\Gamma \vdash B \qquad \Gamma \vdash A\!:\!(\mu_1 X_1, \cdots, \mu_n X_n) \qquad \Gamma \vdash u_1\!:\!\mu_1 r_1\ x_1 \cdots \Gamma \vdash u_n\!:\!\mu_n r_n\ x_n \qquad \rho_1\ x_1 \preceq X_1 \cdots \rho_n\ x_n \preceq X_n}{\Gamma \vdash A[\tilde{\rho}\tilde{u}]} \tag{T7}$$

An interface having a subtype of another can replace it in an object instantiation.

$$\frac{\Gamma \vdash B \qquad \Gamma \vdash A\!:\!(\mu_1 X_1, \cdots, \mu_n X_n) \qquad \Gamma \vdash u_1\!:\!\mu_1 r_1\ x_1 \cdots \Gamma \vdash u_n\!:\!\mu_n r_n\ x_n \qquad \rho_1\ x_1 \preceq X_1 \cdots \rho_n\ x_n \preceq X_n \qquad \Gamma \oplus \tilde{\rho}\tilde{u} \textbf{ defined}}{\Gamma \oplus \tilde{\rho}\tilde{u} \vdash A[\tilde{\rho}\tilde{u}] > B} \tag{T8}$$

This is the same case as the precedent except that here we must update the capabilities of the interfaces $\tilde{u}$.

[2] The function *Capabilities* is simply defined as follows: $\mathit{Capabilities}(r\ x) = r$

$$\frac{\begin{array}{l}\Gamma_1 \vdash C_1 \\ \Gamma_2 \vdash C_2 \\ \Gamma_1 \oplus \Gamma_2 \textbf{ defined}\end{array}}{\Gamma_1 \oplus \Gamma_2 \vdash C_1 | C_2} \quad \text{(T9)}$$

Finally, this last rule states that there is no duplication of the roles of a client interface when composing configurations.

4 An example

We present a small buffer example intended to show how interface types can be defined and used in a configuration of objects. We demonstrate also how coordination can be modeled using private interfaces.

Our initial step is to declare the interface type for a one place buffer that stores elements (interfaces) of type elem:

```
Buff = put(!elem) ; Full
Full = get(!r_elem) ; Buff
```

These two equations simply state that a buffer is initially waiting for the message `put` carrying an element of type `!elemt` (a reference on `elem`) and then behave according to the behavior type `Full`. When full, the buffer waits for the message `get` with an argument of type `!r_elem` which corresponds to the return address.

Interfaces of type `r_elem` return elements of type elem:

```
r_elem = ret(!elem) ; 0
```

The behavior type `0` is a predefined behavior type that cannot evolve.

4.1 A buffer and a client

An object that encapsulates a one place buffer can be now written:

```
Buffer[self: private ?Buff] =

  self[put(e: private !elem) =
        Full-Buffer[?self, !e]
     ]
```

This object starts with an initial server interface `self` which is ready to accept a `put` method. Method `put` takes an argument `e` which is the client role (the reference of the object to be stored) of an interface of type `elem`. After accepting method `put` the object becomes a `Full-Buffer` which has the continuation of the server role for `self` and a client role for interface `e`. `Full-Buffer` is ready for method `get` which takes an interface `r` that is used as a target for returning the value `e`:

```
Full-Buffer[self: private ?Full,
               e: private !elem ] =
  self[get(r: private !r_elem) =
           r.ret(!e) > Buffer[?self]
      ]
```

Note how, at the initial state of `Full-Buffer`, the behavior type of interface `self` is `?Full`. We can now define a client that uses our buffer:

```
Buffer-Client[ buf: private !Buff,
                 e: private  !elem ]=
  buf.put(!e) >
  new r: private !?r_elem in
    buf.get(!r) >
    r[ret(e: private !elem) =
         Buffer-Client[!buf, !e]
     ]
```

Object `Buffer-Client` is parameterized with the (client) interface of the buffer and the (client) interface of the element that is to be stored in the buffer. It starts by invoking method `put` and then it creates a new interface (a newly created interface always has both roles, client and server) that is used in the argument of the invocation of method `get`.

4.2 A buffer and two clients

We turn now to modeling two coordinated clients for our buffer: a producer and a consumer. We will take advantage of the private interface `buf` to synchronize these two objects: the client role for `buf` is passed between the two objects - the producer after storing an element in the buffer, and the consumer after getting this element. The `Producer` object is as follows:

```
Producer[buf: private !Buff,
           e: private  !elem,
        prod: private  ?s_empty,
        cons: private  !s_full]=
  buf.put(!e) >
  cons.token(!buf) >
  prod[token(buf: private  !Buff) =
           Producer[!buf,!e,?prod,!cons]
      ]
```

and the `Consumer` object is given hereafter:

```
Consumer[cons: private ?s_full,
         prod: private  !s_empty ]=
  cons[token(buf:private !Full) =
           new r: private  !? r_elem in
             buf.get(!r) >
             r[ret(e: private !elem) =
                  prod.token(!buf) >
```

```
                Consumer[?cons, !prod]
            ]
    ]
```

The previous objects make use of interface behavior types `s_full` and `s_empty` which are simply defined by the following equations:

```
s_full = token(!Full) ; s_full
s_empty = token(!Buff) ; s_empty
```

5 Run-time Safety

We first define run-time failures using the following rules.

$$\frac{m \neq m_i\ ,\ \forall i \in [1..n]}{\begin{array}{l} u[m_1(\cdots) > B_1, \cdots, m_n(\cdots) > B_n] + \Sigma Recep_j \\ \mid \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow error \\ u.m(\tilde{v}) > B' \end{array}} \tag{E1}$$

A configuration where the server on a private interface is not ready to accept a message m while the client of this interface is trying to invoke this service reduces to *error*.

$$\frac{k \neq l}{\begin{array}{l} u[\cdots, m(u_1, \ldots, u_k) > B, \cdots] + \Sigma Recep_j \\ \mid \qquad\qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow error \\ u.m(v_1, \ldots, v_l) > B' \end{array}} \tag{E2}$$

Similarly, a configuration where the server is ready to process the message m, but where there is a parameters number mismatch reduces to *error*.

$$\frac{m \neq m_i\ ,\ \forall i \in [1..n]}{\begin{array}{l} u[m_1(\cdots) > B_1, \cdots, m_n(\cdots) > B_n] + \Sigma Recep_j \\ \mid \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow error \\ [u.m(\tilde{v})] \end{array}} \tag{E3}$$

A configuration where a message addressed to a public interface which cannot process it reduces to *error*.

$$\frac{k \neq l}{\begin{array}{l} u[\cdots, m(u_1, \ldots, u_k) > B, \cdots] + \Sigma Recep_j \\ \mid \qquad\qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow error \\ [u.m(v_1, \ldots, v_l)] \end{array}} \tag{E4}$$

As for the private interface case, a configuration where there is a parameters number mismatch reduces to *error*.

$$\frac{C \longrightarrow error}{\mathbf{new}\ u{:}T\,\mathbf{in}\ C \longrightarrow error} \tag{E5}$$

$$\frac{C_1 \longrightarrow error}{C_1 \mid C_2 \longrightarrow error} \tag{E6}$$

If a subconfiguration reduces to *error* the entire configuration does so.

Theorem 1 *(subject reduction)*

If a configuration C_1 is well-typed in the context Γ_1 and if $C_1 \xrightarrow{l} C_2$ or $C_1 \longrightarrow C_2$ then there exists a context Γ_2 where C_2 is well-typed.

Justification of the theorem: This result is mainly due to the following properties of our calculus. These properties are enforced by the typing discipline defined and by the synchronization mechanisms of the calculus (see [11] for the proof).

- behavior types of public interfaces are uniform;
- client roles of privates interfaces cannot be duplicated;
- synchronization on private interfaces are by *rendez-vous*;
- objects use their interfaces according to their declared type.

Theorem 2 *(Run-time safety)*

A well-typed configuration C cannot reduce to error: $\Gamma \vdash C \Rightarrow C \not\xrightarrow{l} error$.

Justification of the theorem: This theorem follows (almost) immediately from the subject reduction property.

6 Discussion and future work

We presented an object calculus which combines features of the asynchronous π-calculus [7] and the actors model [1]. This calculus is endowed with typing rules that guarantee a safety result which is there is no "message not understood" errors at run-time. We extended this result to infinite types in [9] and to guaranty a liveness property in [10] with proofs in [11].

There is a lot of ongoing research in the field on behavioral typing. Among these, we can cite [12, 13, 14, 5, 3, 4, 2]. Most of these contributions are based on type inference, except [12, 13] which are based on explicit typing. The types inferred by the system described in [14] are very close to the behavior types presented in this paper. The other systems, especially [3, 2], perform a static analysis of the code based on an approach close to type inference. But what is inferred cannot really be called a type in the sense that it cannot be used, as an IDL specification can be, to know what are the services offered at an interface.

As seen in our two small examples, a well known problem with object calculi based on process calculi is the intensive use of intermediate channels used for returning results. In our case, this leads to an increase of interface type declarations since types have to be declared explicitly.

We are now exploring the possibility of using functional behavior types. For example, the behavior type of a one place buffer would be something like:

```
Buff = put(elem) ; Full
Full = get() -> elem ; Buff
```

The buffer itself could be written in pseudo-code as follows:

```
Buffer[store: elem]: Buff
begin
void put(e: elem){
        store := e;
}

elem get(){
        return store;
}
end
```

The buffer encapsulates a state which is the store in this case and possesses two methods: `put` and `get`. The declaration of `Buffer` of type `Buff` specifies the synchronization constraints on its methods. This way, we can separate the functional aspects of the objects from the synchronization aspects. In addition, the synchronization aspect are "visible" from the outside world of the object. In such a language, we will have to be careful about the semantics of a private reference and about the meaning of reference passing in a method call.

Acknowledgments: The authors would like to thank Jean-Bernard Stefani from France Telecom/CNET, for all the inspiring discussions on topics related to the present paper. The first author is partly supported by the RNRT project Marvel.

References

[1] G. A. Agha, I. A. Mason, S. F. Smith and C. L. Talcott, *A Foundation for Actor Computation*, J. Functional Programming 1 (1), 1993.

[2] Gérard Boudol. *Typing the use of resources in a concurrent calculus.* ASIAN'97, the Asian Computing Science Conference, Kathmandu, Nepal, LNCS 1345 (1997) 239-253.

[3] J-L. Colaço. *Analyses statiques par typage de langages d'Acteurs.* PhD thesis, Institut National Polytechnique de Toulouse, October 1997.

[4] J-L. Colaço, M. Pantel, and P. Sallé. *A set-constraint-based analysis of actors.* In Proc. of the 1997 IFIP International conference on Formal Methods for Open Object-based Distributed Systems, July 1997.

[5] Kohei Honda, Vasco T. Vasconcelos and Makoto Kubo. *Language primitives and type disciplines for structured communication-based programming* ESOP'98, LNCS 1381, Springer-Verlag.

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

[7] Robin Milner, Joachim Parrow, David Walker. *A calculus of mobile processes (Part I and Part II).* Information and Computation, 100:1-77, 1992.

[8] E. Najm, A. Nimour. *A Calculus of Object Bindings.* in proceedings of Second conference on Formal Methods for Object-based Open Systems, Chapman and Hall, 1997.

[9] Elie Najm, Abdelkrim Nimour, Jean-Bernard Stefani. *Infinite Types for Distributed Objects Interfaces.* in Proceedings of third IFIP conference on Formal Methods for Open Object-based Distributed Systems - FMOODS'99 - Firenze, Italy, February 99. Kluwer.

[10] Elie Najm, Abdelkrim Nimour, Jean-Bernard Stefani. *Liveness properties through behavioral Typing of Objects.* in Proceedings of Joint International Conference Formal Description Techniques For Distributed Systems and Communication Protocols AND Protocol Specification, Testing, And Verification - FORTE/PSTV'99 - Beijing, China, October 99. Kluwer.

[11] Abdelkrim Nimour. *Types non-uniformes pour les objets repartis.* PhD thesis, École Nationale Supérieure des Télécommunications, March 1999.

[12] Oscar Nierstrasz. *Regular Types for Active Objects.* Object-Oriented Software Composition. O. Oscar Nierstrasz, D.Tsichitzis (Ed.), Prentice Hall, 1993.

[13] Franz Puntigam. *Types for active objects based on Trace Semantics.* FMOODS'96, Chapmann and Hall.

[14] Antonio Ravara, Vasco T. Vasconcelos. *Behavioural Types for a Calculus of Concurrent Objects.* Euro-Par'97, LNCS. Springer-Verlag, 1997.

Recent BRICS Notes Series Publications

NS-99-2 Hans Hüttel, Josva Kleist, Uwe Nestmann, and António Ravara, editors. *Proceedings of the Workshop on Semantics of Objects As Processes, SOAP '99,* (Lisbon, Portugal, June 15, 1999), May 1999. iv+64 pp.

NS-99-1 Olivier Danvy, editor. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '99,* (San Antonio, Texas, USA, January 22–23, 1999), January 1999.

NS-98-8 Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98 Proceedings,* (Gothenburg, Sweden, May 8–9, 1998), December 1998.

NS-98-7 John Power. *2-Categories*. August 1998. 18 pp.

NS-98-6 Carsten Butz, Ulrich Kohlenbach, Søren Riis, and Glynn Winskel, editors. *Abstracts of the Workshop on Proof Theory and Complexity, PTAC '98,* (Aarhus, Denmark, August 3–7, 1998), July 1998. vi+16 pp.

NS-98-5 Hans Hüttel and Uwe Nestmann, editors. *Proceedings of the Workshop on Semantics of Objects as Processes, SOAP '98,* (Aalborg, Denmark, July 18, 1998), June 1998. 50 pp.

NS-98-4 Tiziana Margaria and Bernhard Steffen, editors. *Proceedings of the International Workshop on Software Tools for Technology Transfer, STTT '98,* (Aalborg, Denmark, July 12–13, 1998), June 1998. 86 pp.

NS-98-3 Nils Klarlund and Anders Møller. MONA *Version 1.2 — User Manual*. June 1998. 60 pp.

NS-98-2 Peter D. Mosses and Uffe H. Engberg, editors. *Proceedings of the Workshop on Applicability of Formal Methods, AFM '98,* (Aarhus, Denmark, June 2, 1998), June 1998. 94 pp.

NS-98-1 Olivier Danvy and Peter Dybjer, editors. *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98,* (Gothenburg, Sweden, May 8–9, 1998), May 1998.