Proceedings of the First Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software

Vasco T. Vasconcelos
Nobuko Yoshida
(editors)

DI–FCUL TR–08–14

May 2008

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Proceedings of the
First Workshop on

Programming Language Approaches to Concurrency and Communication-cEntric Software

June 7, 2008
Oslo, Norway
http://places08.di.fc.ul.pt/

Preface

This documents contains the proceedings of PLACES'08, the 1st Workshop on **Programming Language Approaches to Concurrency and Communication-cEntric Software**, held in Oslo, Norway, on June 7, 2008, co-located with the DisCoTec federated conferences.

PLACES aims to offer a forum where researchers from different fields exchange new ideas on one of the central challenges in programming in near future, the development of programming methodologies and infrastructures where concurrency and distribution are a norm rather than a marginal concern.

The Program Committee, after a careful and thorough reviewing process, selected for inclusion in the programme 10 papers out of 14 submissions. Each submission was evaluated by at least two referees, and the accepted papers were selected during one week electronic discussions.

The volume opens with the abstracts for the invited contributions by Jan Vitek (Purdue University) and and Alan Mycroft (University of Cambridge), and continues with the ten technical papers.

Places'08 was made possible by the contribution and dedication of many people. First of all, we would like to thank all the authors who submitted papers for consideration. Secondly we would like to thank our invited and tutorial speakers. We would also like to thank the members of the Program Committee for their careful reviews, and the balanced discussions during the selection process. Dimitris Mostrous helped to set up a web page for the PC discussion. Finally, we acknowledge the Discotec general chairs, Frank Eliassen and Einar Broch Johnsen, and the Workshop chair Amy L. Murphy.

May 2008

Vasco T. Vasconcelos
Nobuko Yoshida

Program Committee

Alastair Beresford, University of Cambridge
Manuel Fähndrich, Microsoft Research
Simon Gay, University of Glasgow
Kohei Honda, Queen Mary University of London
Andrew Myers, Cornell University
Greg Morrisett, Harvard University
Alan Mycroft, University of Cambridge
Vijay A. Saraswat, IBM Research
Vasco T. Vasconcelos (chair), University of Lisbon
Nobuko Yoshida (chair), Imperial College London

Additional Referees

Martin Berger
Marco Giunti
Olivier Tardieu

Contents

Invited Talk

Programming Models for Concurrency and Real-time

Jan Vitek

Purdue University

Modern real-time systems are increasingly large, complex and concurrent programs which must meet stringent performance and predictability requirements. Programming those systems require fundamental advances in programming languages and runtime systems. This talk presents our work on Flexotasks, a programming model for concurrent, real-time systems inspired by stream-processing and concurrent active objects. Some of the key innovations in Flexotasks are that it support both real-time garbage collection and region-based memory with an ownership type system for static safety. Communication between tasks is performed by channels with a linear type discipline to avoid copy of messages, and by a non-blocking transactional memory facility. We have evaluated our model empirically within two distinct implementations, one based on Purdue's Ovm research virtual machine framework and the other on Websphere, IBM's production real-time virtual machine. We have written a number of small programs, as well as a 30 KLOC avionics collision detector application. We show that Flexotasks are capable of executing periodic threads at 10 KHz with a standard deviation of 1.2us and have performance competitive with hand coded C programs.

Invited Tutorial

Enhancing program structures for communication or what do we want sensor-driven systems to tell us?

Alan Mycroft

University of Cambridge

Increasingly, the world of computation is no longer ”data entry, compute, output”; rather, it consists of islands of reasonably understood autonomous computation connected by communication. Examples range from interactive websites to wireless-enabled car satellite navigation units. In such systems, communication is typically understood at a very low level—a programmer works with streams, packets and re-tries, often expressed as part of a program library. We believe that if systems are to get larger and more reliable, higher-level communication specifications will become essential. By considering a range of systems, including computer architectures, sensor-based computing and client-server web applications, we identify the core issues presented by today’s standard techniques for expressing communication. We then describe some future approaches which attempt to address these deficiencies.

Encapsulation and Dynamic Modularity in the π-calculus

Daniel Hirschkoff, Aurélien Pardon, Damien Pous
ENS Lyon, Université de Lyon, CNRS, INRIA

Tom Hirschowitz
LAMA, Université de Savoie, CNRS

Samuel Hym
LIFL, Lille

Abstract

We report on work in progress in the study of high level constructs for component-oriented programming in a distributed setting. We propose an extension of the higher-order π-calculus intended to capture several important mechanisms related to component programming, such as dynamic update, reconfiguration and code migration. In order to validate the definition of our calculus, we study its implementability by describing an abstract machine for the distributed execution of processes. In doing this, we are led to define a type system to check statically some properties that are needed for the correct execution of processes. We describe current and future directions of research in our programme.

1 A Core Calculus for Dynamic Modularity

This paper describes work on component-oriented programming and the π-calculus. Our long term goal is the design and implementation of a prototype programming language meeting the following requirements.

- It should be suitable for *concurrent, distributed programming.* For instance, usual distributed, parallel algorithms should be easily implementable, as well as lower-level communication infrastructure for networks. Furthermore, it should enjoy a well-understood and tractable behavioural theory.

- It should provide constructs for *modularity*, in the standard, informal sense that programs should be built as an assembly of independent computation units (or *modules*) interacting at explicit interfaces. Moreover, modularity should come with *encapsulation* features, e.g., it should be possible to exchange two modules implementing the same interface without affecting the rest of the code.

- The modular structure of programs should be available at execution time, so as to ease standard dynamic operations such as migration, dynamic update, or passivation of modules. We call this requirement *dynamic modularity.* The notion of dynamic modularity gathers the most challenging features of component based programming we are interested in modelling and analysing.

- Finally, we are seeking a reasonably implementable language, at least permitting rapid prototyping of distributed applications.

For lack of space, we give mostly informal descriptions of our contributions. The current state of our technical definitions (in draft form), as well as the implementation described in Sect. 2, are available from [7].

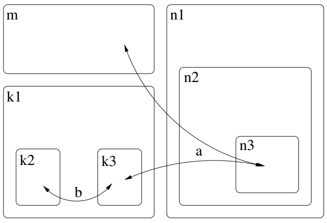
In order to provide a formal treatment of the questions described above, we study an extension of the higher-order π-calculus, called $k\pi$. We choose the π-calculus for two main reasons: first, message-based concurrency seems an appropriate choice to define a model for concurrent programming at a reasonable level of abstraction. Second, working in the setting of a process algebra like the π-calculus makes it possible to define a core formalism in which we can analyse the main questions, both theoretical and pragmatic, related to the implementation of primitives for dynamic modularity. Third, we might hope for this to benefit from the considerable amount of research that has been made on π-calculus based formalisms.

Our calculus inherits ideas from numerous previous studies, among which [6, 5, 3], and in particular the Kell calculus [2, 9]. The grammar for processes is as follows, – we suppose two sets of names (a, m, k, x) and process variables (X):

$$\begin{array}{rcl} P, Q & ::= & P|P \mid (\boldsymbol{\nu} n)\, P \mid \mathbf{0} \mid n[P] \mid X \mid M.P \mid R \triangleright P \\ M & ::= & \overline{a}\langle n\rangle \mid \overline{a}\langle P\rangle \qquad\qquad R \quad ::= \quad a(x) \mid a(X) \mid n[X] \end{array}$$

In addition to the usual π-calculus constructs, we have *modules*, $n[P]$, which can be seen as located processes (note that modules can be nested). $M.P$ is a process willing to emit (first- or higher-order) message M and then proceed as P. $R \triangleright P$ stands for a process willing to acquire a resource: this can mean either receiving a first- or higher-order message (cases $a(x)$ and $a(X)$, respectively), or passivating a module.

Due to space limitations, we only provide an informal description of the (reduction based) operational semantics. A $k\pi$ term describes a configuration consisting of a hierarchy of modules, in which processes are executed. Given some $k\pi$ processes P, P'_i, Q_i, the diagram we present depicts a process of the form



$$(\boldsymbol{\nu} a)(m[P] \mid n_1[P'_1 \mid n_2[P'_2 \mid n_3[P'_3]]] \mid k_1[(\boldsymbol{\nu} b)(Q_1 \mid k_2[Q_2] \mid k_3[Q_3])])$$

(note that the localisation of the restrictions on a and b does not appear on the picture). There are basically two forms of interaction in $k\pi$: *communication* and *passivation*. Communication involves the transmission of a name or of a process; it is *distant*, in the sense that $\overline{a}\langle b\rangle.P$ can synchronise with a receiver $a(x) \triangleright Q$ sitting in a different location, provided they share the name a. On the picture, a process running in k_3 can exchange messages with another one in n_3 (using channel a), as well as with a third process running in k_2 (using channel b). On the contrary, passivation is local: only a process running at n_1 is able to passivate module n_2. This is described by the following reduction axiom:

$$n[P] \mid n[X] \triangleright Q \quad \rightarrow \quad Q\{P/X\}$$

(up to structural congruence, the module being passivated and the process that takes control over it must be in parallel – as usual, $\{P/X\}$ denotes capture avoiding substitution). Passivation is a central construct in our formalism, and can be used to implement very different kinds of manipulations related to dynamic modularity. For instance, taking $Q = n'[X]$ in the above reduction leads to a simple operation of module renaming; with $Q = \overline{c}\langle X\rangle$, the module n will be 'frozen' and marshalled into a message to be sent on channel c; finally, taking $Q = n[X] \,|\, n'[X]$ makes it possible to duplicate a computation. From the operational point of view, the last two examples clearly involve much more costly operations than the first one.

Restricted names as localised resources. An important commitment that we make in the design of $k\pi$ is that, contrarily to several existing proposals, we do *not* allow channel names to be extruded across module boundaries: neither do we include an axiom of the form $n[(\boldsymbol{\nu} b)\, P] \;\equiv\; (\boldsymbol{\nu} b)\, n[P]$ in structural congruence, nor do we implement name extrusion across modules along reduction steps that would require it.

This design choice is related to the notion of module that we put forward in $k\pi$: indeed, if we were to allow name extrusion, the possibility to passivate a module *with* or *without* one of its restrictions (depending on whether we perform extrusion before passivation) would give rise to unpredictable behaviours (see [9]). Therefore, we interpret the names declared inside a module as private resources, that should remain local to that module. Passivating module n hence means getting hold of the local computations, as well as of the resources allocated in n. Typically, names allocated in module n can be viewed either as temporary resources allocated for the computations taking place at n, or as methods provided for submodules of n, for which n acts as a library.

As a consequence, the user is made aware of the localisation of resources; this choice also helps considerably in the implementation of $k\pi$, essentially because we always know how to route messages to channels (see below; a similar idea is present in existing implementations of π-calculus related process algebras, such as [4]). At the same time, this hinders the expressiveness of message passing: a process willing to send a name n outside the module where the restriction on n is hosted is stuck. Consequently, for two distant agents to share a common name, this name should be allocated at a place that is visible for both, i.e., above them in the hierarchy of modules. In other words, extrusion is not transparent to the user, and has to be programmed when necessary. Of course, there are situations where one would like to allocate a new name outside the current module. It turns out that a corresponding primitive for remote allocation, $\boldsymbol{\nu} n@m$, can be added at small cost to our implementation (Sect. 2).

Experiments with examples show that the idioms we would like to be able to program are compatible with the discipline we enforce in $k\pi$. Further investigations need to be made, in particular with larger examples, in order to understand the possibilities offered by programming in $k\pi$.

2 A Distributed Implementation

In this section, we describe a distributed abstract machine that implements $k\pi$. This machine abstracts from issues such as data representation, to focus on the implementation of distributed communication in the presence of passivation. The design of this machine has been tested on a prototypical distributed implementation, so as to make sure that our implementation choices are reasonable.

Computation units. The first (standard) feature of our machine is that it flattens the hierarchy of computation units: for example, each of the seven modules in the picture shown above is executed in its own asynchronous *location* by the machine. In order to retain the tree structure, each location stores and maintains the list of its children locations. As expected, when a module creates a sub-module, the latter is spawned in a fresh location. In order to make sure that locations can be implemented in an asynchronous way, we let them interact only by means of (asynchronous) messages.

Communications. The protocol for distant communication is rather standard: a process willing to send a message sends it to the location holding the *queue* that implements the channel; accordingly, a process willing to receive a message sends its location to the queue location and suspends execution, so that it can be awaken when a message is available. Using our interpretation of modules, the natural location to run the channel queue is that of the module that created the name; this is made possible by our choice to prevent the channel name from being extruded out of this module: if the module gets passivated, any process trying to communicate on the channel will get passivated as well.

Passivation. First, passivation cannot be atomic, because the hierarchy of modules has been flattened, as described above (this departs importantly from the machine in [1]). Thus, we implement it in an incremental fashion, from the passivated module down to its sub modules, transitively. Along the propagation of a passivation session, we must handle two main sources of interferences:

- First, in the case where some sub module has already started a passivation, our machine gives priority to the inner passivation session – the other option being to cancel the latter and let the dominating passivation proceed instead.

- More importantly, we need to clean up running communication sessions in the passivated sub modules. As explained above, this is not problematic for communication on names belonging to the passivated module. For communications on names that reside above the passivated module, we use simple interactions with the modules owning these names: status messages are used to query whether commitment to a communication already occurred, so that the computation can be either completed or aborted.

Distributed Implementation. We have written an OCaml implementation of this abstract machine [7]. This implementation exploits two libraries: one for high-level communications, where message passing is executed either as memory write-ups or as socket communication, depending on whether it is local or distant; and another one for communication, thunkification and spawning of OCaml threads, together with their sets of defined names (to optimise the process of passivation, the data structure implementing a module comes with a table collecting all names known by the module). Finally, each syntactic construction of $k\pi$ is compiled into a simple function that uses the previous libraries along the lines of the formal specification of the machine. In particular, we do not need to manipulate explicit abstract syntactic trees at runtime.

A type system to prevent illegal name extrusions. In order for the previous algorithms to be correct, we need to make sure that names are not extruded outside their defining module. A solution would be to inspect the content of each message at runtime, and to block illegal communications. In order to avoid the inefficiencies induced by this approach, we rely on a type system to enforce statically this confinement policy: a well-typed term will never attempt to extrude a name out of its scope.

Our type system exploits an analysis of the hierarchy of modules to detect ill-formed communications. An output $\overline{a}\langle n\rangle$ is licit only if the restriction binding

n is above the one binding a in the structure of modules (or if n is free in the process). If, instead of a or n, we have names bound to be received (as in, e.g., $c(x) \triangleright \overline{a}\langle x\rangle$), then the type information associated to the transmitting channels gives an approximation of the module where the names being communicated are allocated (intuitively, this information boils down to *"name x is allocated above module named m and under module named k"* – both informations are necessary, because the name instanciating x may then be used either as medium or as object of communication). The communication of process values follows the same ideas: in $\overline{a}\langle P\rangle$, we impose that all free names of P should be allocated above a. Consequently, in the type of module names (resp. of channels over which processes are transmitted), we provide a spatial bound of this kind on the free names of the process being executed (resp. communicated).

In addition to the standard property of subject reduction, correctness of our type system is expressed by showing that every typeable process is *well scoped*, which intuitively means that such a process does not attempt to emit a name outside its scope. Since this property is preserved by reduction, we can avoid checking for scope extrusions at run time.

Correctness of the abstract machine. To validate our implementation of $k\pi$, we should formally state and prove that the execution of the encoding of a $k\pi$ term corresponds in some sense to the original, source, process. The reductions of a $k\pi$ process and the execution of a machine state are described by two labelled transition systems. We could hope to establish a *bisimulation* result, providing evidence that the compiled version essentially exhibits the same behaviour as the source process. However, because passivation is not atomic in our setting (contrarily to [1]), this is not possible. Indeed, consider the following process: $m[\, a\langle u\rangle \mid n[b\langle v\rangle]\,] \mid m[X] \triangleright Q$. The actual execution of the passivation of m may go through a state where the emission on b is blocked while the one on a is still active; such a state has no counterpart in the original calculus. Instead, correctness of our machine is stated as a *coupled bisimulation* result [8]: although this behavioural equivalence is weaker than plain bisimulation, it entails operational equivalence (any $k\pi$ reduction step can be simulated by the machine, and any reduction step of the machine can be completed into a step of the calculus).

3 Future Work

Module Interfaces. As it is, the type system of Sect. 2 associates to a module name a rudimentary information, which is only related to the regions accessed by processes running within this module (properties like *"this module has only access to references situated above module m"*). It would be interesting to define more informative module interfaces, that in particular would describe the behaviour associated to the usage of the names hosted by the module (as well as the interfaces associated to submodules, recursively). Existing type systems for π-calculus based formalisms, as well as for object-oriented languages, should be relevant for this.

Handling (re)binding. In its current form, $k\pi$ only makes it possible to implement limited forms of dynamic modularity. When a module is passivated, it can be moved around, duplicated, and computation can be resumed, as long as the confinement constraints associated to the localisation of restrictions are respected. In writing examples in $k\pi$, it appears that it would be helpful, when passivating a module, to be able to somehow disconnect it from some of the local resources it is using. This would make it possible to send the passivated module outside the scope of the names it was using, to another site where computation can be resumed after connecting again to another bunch of resources. Extending $k\pi$ with mechanisms for dynamic (re)binding while keeping the possibility to assert statically properties of modules about their usage of resources is a difficult task. The work on Acute [10] may provide interesting inspiration for this.

Optimisations of the machine. The definition and implementation of the abstract machine plays an essential role in the design of $k\pi$, because it provides practical insight on the main design decisions behind the formal model. In addition to that, the implementation also suggests several improvements or extensions, that we would like to study further. We have already mentioned the primitive for remote name allocation $\boldsymbol{\nu} n@m$, an operation that in principle can be encoded, but comes at a very low cost as a primitive, given the current design of the machine. Another direction worth investigating is how the general behaviour of the machine can be specialised by taking into account informations such as, e.g., the fact that a whole module hierarchy runs on a single machine.

Proof of correctness. The abstract machine of Sect. 2 provides a rather low-level description of how $k\pi$ processes should be executed in a distributed setting. Proving its correctness, i.e., that the result of the compilation exhibits the same behaviour as the original source process, is a challenging task. The main difficulty is that proofs of this kind tend to be a really large piece of mathematics; appropriate techniques are necessary to render them more tractable, in order to be able to complete them. This is the case in our setting, notably because the passivation mechanism brings several technical subtleties.

Behavioural equivalences. Not only do we want to execute $k\pi$ programs, but we also would like to state and prove their properties. At a foundational level, we would be interested in analysing the notion of behavioural equivalence provided in $k\pi$, and in understanding the role of passivation in this respect.

References

[1] P. Bidinger, A. Schmitt, and J.-B. Stefani. An Abstract Machine for the Kell Calculus. In *In Proc. FMOODS '05*, volume 3535 of *LNCS*, pages 31–46. Springer, 2005.

[2] P. Bidinger and J.-B. Stefani. The Kell Calculus: Operational Semantics and Type System. In *In Proc. FMOODS '03*, volume 2884 of *LNCS*, pages 109–123. Springer, 2003.

[3] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* PhD thesis, Ecole Polytechnique, 1998.

[4] C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. JoCaml: A Language for Concurrent Distributed and Mobile Programming. In *In Proc. Advanced Functional Programming 2002*, volume 2638 of *LNCS*, pages 129–158. Springer, 2002.

[5] M. Hennessy. *A Distributed π-calculus.* Cambridge University Press, 2007.

[6] T. Hildebrandt, J. C. Godskesen, and M. Bundgaard. Bisimulation Congruences for Homer — a Calculus of Higher Order Mobile Embedded Resources. Technical Report TR-2004-52, Univ. of Copenhagen, 2004.

[7] D. Hirschkoff, T. Hirschowitz, S. Hym, A. Pardon, and D. Pous. $k\pi$: specification, working drafts and prototype implementation. available from `http://www.ens-lyon.fr/LIP/PLUME/kp`, 2008.

[8] J. Parrow and P. Sjödin. Multiway synchronizaton verified with woupled simulation. In *In Proc. CONCUR*, pages 518–533, 1992.

[9] A. Schmitt and J.-B. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *In Proc. Global Computing*, volume 3267 of *LNCS*, pages 146–178. Springer, 2005.

[10] P. Sewell, J. J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: high-level programming language design for distributed computation. In *In Proc. ICFP*, pages 15–26. ACM, 2005.

Session Types as Generic Process Types

Simon J. Gay[1] Nils Gesbert[1] António Ravara[2]

13th May 2008

1 Introduction

Session types [7, 4] are an increasingly popular technique for specifying and verifying protocols in concurrent and distributed systems. In a setting of point-to-point private-channel-based communication, the session type of a channel describes the sequence and type of messages that can be sent on it. For example

$$\& \langle \mathsf{service} : ! [\mathsf{int}] . ? [\mathsf{bool}] . \mathsf{end}, \mathsf{quit} : \mathsf{end} \rangle$$

describes the server's view of a channel on which a client can select either service or quit. In the former case, the client then sends an integer and receives a boolean; in the latter case, the protocol ends. From the client's viewpoint, the channel has a dual type in which the direction of messages is reversed:

$$\oplus \langle \mathsf{service} : ? [\mathsf{int}] . ! [\mathsf{bool}] . \mathsf{end}, \mathsf{quit} : \mathsf{end} \rangle$$

Session types provide concise specifications of protocols and allow certain properties of protocol implementations to be verified by static typechecking. Originally formulated for languages similar to pi-calculus, type systems incorporating session types have now been defined for other language paradigms including object-oriented languages [2] and service-oriented systems [1].

The theory of session types was developed in order to analyse a particular correctness criterion for concurrent systems: that every message is of the type expected by the receiver, and that whenever a client selects a service, the server offers a matching service. Igarashi and Kobayashi [5] have taken a different approach to type-theoretic specifications of concurrent systems, by developing a single generic type system for the pi-calculus from which numerous specific type systems can be obtained by varying certain parameters. Their motivation is to express the common aspects of a range of type systems, enabling much of the work of designing typing rules and proving type soundness to be packaged into a general theory instead of being worked out for each case. In the generic type system, types are abstractions of processes, so that the typing rules display a very direct correspondence between the structure of processes and the structure of types. There is also a subtyping relation, which can be modified in order to obtain specific type systems; this allows, for example, a choice of retaining

[1] Department of Computing Science, University of Glasgow

[2] SQIG at Instituto de Telecomunicações and Departamento de Matemática, IST, Technical University of Lisbon

or discarding information about the order of communications. A logic is provided in which to define an *ok* predicate that is interpreted both as a desired runtime property of processes and as a correctness condition for typings. This double interpretation allows a generic type soundness theorem to be proved, but means that type checking becomes more like model checking unless the specific subtyping relation can be exploited to yield an efficient type checking algorithm.

Kobayashi [6, Section 10] has stated that the generic type system (GTS) subsumes session types, although without presenting a specific construction. The purpose of the present paper is to clarify the relationship. This is relevant to the design of programming languages for distributed systems. For example: if we want an object-oriented language with static typing of protocols, is it better to work directly with session types or to develop an object-oriented formulation of GTS? However, the present paper considers pi-calculus so that we can study a precise question about two type systems for essentially the same language.

Kobayashi did not explain what it means for GTS to "subsume" session types. We interpret it as defining a translation $[\![\cdot]\!]$ from processes and type environments in the source language into GTS, satisfying as many of the following conditions as possible. (1) $[\![P]\!]$ should have a similar structure to P. (2) There should be a correspondence in both directions between the operational semantics, ideally $P \longrightarrow Q$ if and only if $[\![P]\!] \longrightarrow [\![Q]\!]$. (3) There should be a correspondence in both directions between typing derivations, ideally $\Gamma \vdash P$ if and only if $[\![\Gamma]\!] \triangleright [\![P]\!]$. (4) Type soundness for session types should follow from the generic type soundness theorem.

We take the source language to be the version of session types defined by Gay and Hole [3]. This version does not include the accept/request primitives [7, 4] and does not consider progress properties [2]. We remove recursive types, for simplicity, and make some changes to the structural congruence relation, to remove inessential differences compared with GTS. Three key issues remain. First, translating the polarities in the source language: x^+ and x^- refer to the two endpoints of channel x. Second, translating the labels used in branching and selection (external and internal choice). Third, obtaining a correspondence between subtyping in the source language and the subtyping relation which is always present in GTS. The present paper focuses on the first two points and does not discuss subtyping. We satisfy conditions (1–4) above in some form, although the details are more complicated.

From now on we refer to the source language as *session processes* and the target language as *generic processes*.

2 Translation

Processes and types. The languages share common process constructors (inaction, parallel composition, scope restriction[1], and replication), differing basically in two ways. In session processes, (1) channels are decorated with *polarities* (absent in generic processes), and processes only synchronise if the subjects have complementary polarities; (2) there are constructors for *branch*, an input labelled external choice, and *select*, to choose a branch of the choice. Generic processes instead have mixed guarded sums (but no labels), and input and output actions are decorated with *events* (taken from a countable set).

[1] Following a suggestion by Kobayashi, we have added a type annotation to ν in GTS.

$$
\begin{array}{rrl}
\text{Common Syntax} & C ::= & \mathbf{0} \mid (P_1 \mid P_2) \mid *P \\
\text{Source Language} & P ::= & C \mid (\nu x : S)\, P \mid x^p ?[y] \,.\, P \mid x^p ![y^q] \,.\, P \quad (x^p \neq y^q) \\
 & & \mid x^p \triangleright \{l_i : P_i\}_{i=1}^{n} \mid x^p \triangleleft l \,.\, P \\
\text{polarities} & p ::= & + \mid - \mid \varepsilon \\
\text{Session Types} & S ::= & \texttt{end} \mid ?[S_1] \,.\, S_2 \mid ![S_1] \,.\, S_2 \mid \& \langle l_i : S_i \rangle_{i=1}^{n} \mid \oplus \langle l_i : S_i \rangle_{i=1}^{n} \\
\text{Target Language} & P ::= & C \mid (\nu \tilde{x} : \tau)\, P \mid \sum_{i=1}^{n} G_i \\
\text{guarded processes} & G ::= & x ! [\tilde{y}] \,.\, P \mid x ? [\tilde{y}] \,.\, P \\
\text{Generic Types} & \Gamma ::= & \mathbf{0} \mid \sum_{i=1}^{n} \gamma_i \mid (\Gamma_1 \mid \Gamma_2) \mid (\Gamma_1 \,\&\, \Gamma_2) \\
\text{guarded types} & \gamma ::= & x ! [\tau] \,.\, \Gamma \mid x ? [\tau] \,.\, \Gamma \\
\text{tuple types} & \tau ::= & (\tilde{x})\Gamma
\end{array}
$$

Figure 1: Syntax

Since these tags are only relevant for liveness properties like deadlock-freedom, which we do not address in this work, we omit them.

Consider x, y from a countable set of *channels*, disjoint from a **finite** set of N *labels*, ranged over by l, l_i. The grammars in Figure 1 define the languages of both sessions and generic processes and types. Session processes are monadic (for simplicity), while generic processes are polyadic (as required by the encoding). A session type environment is a finite mapping from polarised channels to session types, $\Delta = x_1^{p_1} : S_1; \ldots; x_n^{p_n} : S_n$. A generic type environment is a process type Γ. We write $\prod_{i=1}^{n} P_i$ for $P_1 \mid \cdots \mid P_n$ and $\bigwedge_{i=1}^{n} \Gamma_i$ for $\Gamma_1 \,\&\, \cdots \,\&\, \Gamma_n$.

We consider the operational semantics of both languages based on a reduction relation. However, for session processes, instead of the original structural congruence relation, we take the structural preorder of the generic processes.

Encoding processes and type environments. For each input-guarded labelled sum the translation creates a new name for every possible label, sends them all to the channel subject of the input, and waits in an input-guarded sum where the subjects correspond to the labels in the original process and a fresh channel is received for the continuation of the protocol. Output selection is encoded dually.

Polarities distinguish between the two endpoints of a channel; communication only occurs between x^+ and x^-. Erasing polarities would translate processes that do not reduce into processes that reduce. However, it is possible to solve this problem by inserting some type information into the translated process. But when translating scope restriction, one cannot prefix the body of the process, otherwise there would be no meaningful operational correspondence. Therefore, for each source channel we introduce a pair of target channels, one for each polarity, and a forwarder between them, which is in parallel with the translated process.

Let σ be a numbering of the labels from 1 to N. For any channels p, m, the rules in Figure 2 inductively define the forwarder from p to m following the structure of a session type S. Consider the translation homomorphic for common processes. The rules in Figure 3 inductively define the translation $[\![P]\!]_{\varphi}^{\Gamma}$ of the session process P into a generic process, where φ is an injective

$$\begin{aligned}
\textsc{fw}\,(p, m, \mathsf{end}) &= \mathbf{0}\\
\textsc{fw}\,(p, m, ?[S_1]\,.\,S_2) &= m?[z]\,.\,p\,!\,[z]\,.\,\textsc{fw}\,(p, m, S_2)\\
\textsc{fw}\,(p, m, !\,[S_1]\,.\,S_2) &= p?[z]\,.\,m\,!\,[z]\,.\,\textsc{fw}\,(p, m, S_2)\\
\textsc{fw}\,(p, m, \&\,\langle l_i : S_i\rangle_{i=1}^{n}) &= p?[\kappa_1 \ldots \kappa_N]\,.\,(\nu\lambda_1 \ldots \lambda_N)\,m\,!\,[\tilde{\lambda}]\,.\\
&\quad \textstyle\sum_{i=1}^{n} \lambda_{\sigma(l_i)}?[m']\,.\,(\nu p' : \llparenthesis S_i \mid \overline{S_i} \rrparenthesis)\,\kappa_{\sigma(l_i)}\,!\,[p']\,.\,\textsc{fw}\,(p', m', S_i)\\
\textsc{fw}\,(p, m, \oplus\,\langle l_i : S_i\rangle_{i=1}^{n}) &= m?[\kappa_1 \ldots \kappa_N]\,.\,(\nu\lambda_1 \ldots \lambda_N)\,p\,!\,[\tilde{\lambda}]\,.\\
&\quad \textstyle\sum_{i=1}^{n} \lambda_{\sigma(l_i)}?[p']\,.\,(\nu m' : \llparenthesis S_i \mid \overline{S_i} \rrparenthesis)\,\kappa_{\sigma(l_i)}\,!\,[m']\,.\,\textsc{fw}\,(p', m', S_i)
\end{aligned}$$

Figure 2: Definition of the forwarder

$$\begin{aligned}
\llbracket x^p\,!\,[y^q]\,.\,P \rrbracket_{\varphi}^{\Gamma, x^p : ![S_1].S_2} &= \varphi(x^p)\,!\,[\varphi(y^q)]\,.\,\llbracket P \rrbracket_{\varphi}^{\Gamma, x^p : S_2}\\
\llbracket x^p?[y]\,.\,P \rrbracket_{\varphi}^{\Gamma, x^p : ?[S_1].S_2} &= \varphi(x^p)?[z]\,.\,\llbracket P \rrbracket_{\varphi + \{y^{\varepsilon} \mapsto z\}}^{\Gamma, x^p : S_2}\\
\llbracket x^p \lhd l\,.\,P \rrbracket_{\varphi}^{\Gamma, x^p : \oplus\langle l : S, \ldots\rangle} &= \varphi(x^p)?[\lambda_{1 \ldots N}]\,.\,(\nu z : \llparenthesis S \mid \overline{S} \rrparenthesis)\,\lambda_{\sigma(l)}\,!\,[z]\,.\,\llbracket P \rrbracket_{\varphi + \{x^p \mapsto z\}}^{\Gamma, x^p : S}\\
\llbracket x^p \rhd \{l_i : P_i\} \rrbracket_{\varphi}^{\Gamma, x^p : \&\langle l_i : P_i\rangle} &= (\nu\lambda_{1 \ldots N})\,\varphi(x^p)\,!\,[\tilde{\lambda}]\,.\,\textstyle\sum_{i=1}^{n} \lambda_{\sigma(l_i)}?[z]\,.\,\llbracket P_i \rrbracket_{\varphi + \{x^p \mapsto z\}}^{\Gamma, x^p : S_i}\\
\llbracket (\nu x : S)\,P \rrbracket_{\varphi}^{\Gamma} &= (\nu p, m : (p, m)(\llbracket p : S \rrbracket \mid \llbracket m : \overline{S} \rrbracket \mid \llbracket p : \overline{S} \rrbracket \mid \llbracket m : S \rrbracket))\\
&\quad (\textsc{fw}\,(p, m, S) \;\mid\; \llbracket P \rrbracket_{\varphi + \{x^+ \mapsto p; x^- \mapsto m\}}^{\Gamma, x^+ : S, x^- : \overline{S}})
\end{aligned}$$

Figure 3: Process translation

mapping from the free polarised channels in P to channels of generic processes and Γ is a session type environment such that $\Gamma \vdash P$ (we omit φ and Γ when not relevant). The rules in Figure 4 inductively define the translation of the session type environment Γ to a generic type environment, where φ is an injective mapping from $\mathrm{dom}(\Gamma)$ to the set of channels. Let $\llparenthesis S \rrparenthesis$ stand for $(z)\llbracket y : S \rrbracket_{y \mapsto z}$. The last rule uses the notion of *dual session type* (denoted $\overline{S}$), which exchanges inputs and outputs, and branch and selection [3].

3 Results

We state operational and typing correspondences.

Theorem 1. *For any well-typed closed session process P, whenever $P \longrightarrow Q$, then $\llbracket P \rrbracket \longrightarrow^{n} \llbracket Q \rrbracket$ with $n = 2$ or 4, depending on whether the reduction step is a communication or a selection.*

The extra steps are due to the forwarders and the encoding of labels. We require the source process to be well-typed to ensure that the forwarders behave correctly. There is also a reverse correspondence. It is complicated to state correctly, because a forwarder adds a one-place buffer to the synchronous communication of the pi-calculus; some deadlocked processes in the session calculus can, when translated, take one reduction step. In future, we would like to obtain a full abstraction result with respect to some behavioural equivalence.

$$\begin{aligned}
\llbracket x_1^{p_1} : S_1, \dots, x_n^{p_n} : S_n \rrbracket_\varphi &= \llbracket x_1^{p_1} : S_1 \rrbracket_\varphi \;\mid\; \cdots \;\mid\; \llbracket x_n^{p_n} : S_n \rrbracket_\varphi \\
\llbracket x^p : \mathtt{end} \rrbracket_\varphi &= \mathbf{0} \\
\llbracket x^p : ?\,[S_1]\,.\,S_2 \rrbracket_\varphi &= \varphi(x^p)\,?\,[\llparenthesis S_1 \rrparenthesis]\,.\,\llbracket x^p : S_2 \rrbracket_\varphi \\
\llbracket x^p : !\,[S_1]\,.\,S_2 \rrbracket_\varphi &= \varphi(x^p)\,!\,[\llparenthesis S_1 \rrparenthesis]\,.\,\llbracket x^p : S_2 \rrbracket_\varphi \\
\llbracket x^p : \&\,\langle l_i : S_i \rangle_{i=1}^n \rrbracket_\varphi &= \varphi(x^p)\,!\,\big[(\lambda_1, \dots, \lambda_N)\,\textstyle\bigwedge_{i=1}^n \lambda_{\sigma(l_i)}\,!\,[\llparenthesis S_i \rrparenthesis]\big] \\
\llbracket x^p : \oplus\,\langle l_i : S_i \rangle_{i=1}^n \rrbracket_\varphi &= \varphi(x^p)\,?\,\big[(\lambda_1, \dots, \lambda_N)\,\textstyle\bigwedge_{i=1}^n \lambda_{\sigma(l_i)}\,!\,[\llparenthesis \overline{S_i} \rrparenthesis]\big]
\end{aligned}$$

Figure 4: Type environment translation

Typing correspondence. Let the subtyping relation of the generic type system be such that: (1) Sequential information about the communications on different channels is removed (Igarashi and Kobayashi's Sub-Divide rule); (2) subtyping can occur beneath a prefix (input or output); (3) type $\mathbf{0}$ is a subtype of $*\,\mathbf{0}$; (4) the sum operator is idempotent. Moreover, let $ok(\Gamma)$ hold if and only if Γ is well-formed (meaning that whenever a communication is possible, the type sent is a subtype of the one expected by the receiver) and either: Γ reduces in one step to $\mathbf{0}$; or for any free variable x of Γ, there exists a session type S such that $\Gamma \downarrow \{x\} \geq \llbracket x : S \rrbracket \mid \llbracket x : \overline{S} \rrbracket$, where $\geq$ is the subtyping relation and $\Gamma \downarrow \{x\}$ is the restriction of Γ to x (that is Γ with all actions whose subject is not x removed). The first condition ensures a correct use of labels in a branching/selection, and the second one deals with regular session channels. Note that $ok(\Gamma)$ is stable by reductions, meaning that if $\Gamma \longrightarrow \Gamma'$ and $ok(\Gamma)$ hold, then $ok(\Gamma')$ holds. This makes ok a proper consistency predicate.

Lemma 1. $\llbracket p : \overline{S} \rrbracket \mid \llbracket m : S \rrbracket \triangleright \mathrm{FW}\,(p, m, S)$

Theorem 2 (Completeness)**.** *For any session process P and any corresponding φ, if $\Delta \vdash P$ then $\llbracket \Delta \rrbracket_\varphi \triangleright \llbracket P \rrbracket_\varphi^\Delta$.*

The ok predicate is checked in the typing rule for ν, so this theorem implies that bound channels in $\llbracket P \rrbracket$ are used consistently.

The reverse direction is more difficult to state. GTS types more processes, for the following reason. Sending a message of type `end` in the session calculus removes the channel from the sender's environment, but in GTS, sending a message of type $\mathbf{0}$ does not remove any capabilities.

Theorem 3 (Soundness)**.** *Let P be a closed session process. If $\llbracket P \rrbracket$ is well-typed in the generic type system and no type annotation of P is* `end`*, then P is well-typed as a session process.*

Since the conditions of the generic type soundness theorem hold in this setting, we also conclude the desired runtime safety property of session types.

4 Conclusion

We have defined a translation from a system of session types for the π-calculus into Igarashi and Kobayashi's generic type system (GTS). We have proved correspondence results between process reductions in the two systems, and between typing derivations; we can also apply the generic type soundness theorem. The translation clarifies the relationship between session types and GTS, and provides an interesting application of GTS. Because GTS can also represent more complex behavioural properties including deadlock-freedom, embedding session types into it may suggest ways of extending session types with such properties.

In our opinion, despite the translation into GTS, session types themselves remain of great interest for programming language design, for several reasons. First, session types are a high-level abstraction for structuring inter-process communication [7]; preservation of this abstraction and the corresponding programming primitives is very important for high-level programming. Second, there is now a great deal of interest in session types for languages other than the π-calculus. Applying GTS would require either translation into π-calculus, obscuring distinctive programming abstractions, or extension of GTS to other language, which might not be easy. Third, proofs of type soundness for session types are conceptually fairly straightforward, even when these are liveness properties, as is frequently the case. The amount of work saved by using the generic type soundness theorem is relatively small. Fourth, for practical languages we are very interested in typechecking algorithms for session types; GTS does not yield an algorithm automatically, so specific algorithms for session types need to be developed in any case.

Acknowledgements. António Ravara is partially supported by the Portuguese Fundação para a Ciência e a Tecnologia (via SFRH/BSAB/757/2007 and project Space-Time-Types, POSC/EIA/55582/2004), by FEDER, by the EU IST proactive initiative FET-Global Computing (project Sensoria, IST–2005–16004), and by the EPSRC grant EP/F037368/1 "Behavioural types for object-oriented languages". Nils Gesbert is supported by the EPSRC grant EP/F065708/1 "Engineering Foundations of Web Services: Theories and Tool Support". Simon Gay is partially supported by both EPSRC grants. We are grateful to Luís Caires, Kohei Honda, and Naoki Kobayashi for useful comments and discussions.

References

[1] M. Carbone, K. Honda, and N. Yoshida. Structured global programming for communication behaviour. *ESOP*, LNCS 4421:2–17, 2007.

[2] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. *ECOOP*, LNCS 4067:328–352, 2006.

[3] S. J. Gay and M. J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[4] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. *ESOP*, LNCS 1381:122–138, 1998.

[5] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1–3):121–163, 2004.

[6] N. Kobayashi. Type systems for concurrent programs. *Formal Methods at the Crossroads*, LNCS 2757:439–453, 2002. Extended version at `www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf`.

[7] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. *PARLE*, LNCS 817:398–413, 1994.

Towards trustworthy multiparty sessions[⋆] (extended abstract)

Roberto Bruni[1], Ivan Lanese[2], Hernán Melgratti[3],
Leonardo Gaetano Mezzina[4], and Emilio Tuosto[5]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Dipartimento di Scienze dell'Informazione, Università di Bologna, Italy
[3] Departamento de Computación, Universidad de Buenos Aires, Argentina
[4] IMT Lucca, Institute for Advanced Studies, Italy
[5] Department of Computer Science, University of Leicester, UK

1 Introduction

A lively research direction in concurrent and distributed computing is pushing for a synergic design of theoretical frameworks and programming mechanisms that help theoreticians and practitioners to cope with the complexity of large system development and analysis. Among others, type-based approaches look rather handy both for theoretical and applied research. Process calculi provide a paradigmatic example: they offer a convenient formal setting for modeling modern systems (e.g., Service Oriented Computing (SOC) [2, 8, 4]) and yield a natural framework for enforcing desired properties using behavioral types [5].

Recently, researchers have shown a growing interest around the notion of a multiparty session [6, 1], because typical modern distributed applications can involve the coordination of many endpoints. Remarkably, SOC adds a further difficulty as endpoints may be dynamically discovered and assembled. In fact, SOC aims to the seamless and trustworthy integration of separately developed computational entities, called *services*. To achieve this, matching criteria must be available for comparing a service description against the requirements of the invoker before binding the two.

$\mu\mathsf{se}$ (read "muse", after MUltiparty SEssions) [3], is a process calculus for expressing computations where endpoints dynamically join existing multiparty sessions. In this context, it is crucial to have type systems for the early detection of possible sources of incompatibility.

In this paper we consider $\mu\mathsf{se}$ without name passing and intra-site communication (§ 2) and sketch a type system aimed to guarantee a weak form of compatibility (§ 3), in the sense that all the interactions required by each local task can be provided either by endpoints currently participating to its session or by endpoints that can join that session later.

This is work in progress and we plan to refine the type system to have stronger guarantees. Due to space limitation we cannot give a comprehensive presentation of our ideas, which will be spelled out in the full version of this paper.

[⋆] Research supported by the FET-GC2 EU project IST-2005-16004 Sensoria by the UK project HiDeA4SOC and by the Italian FIRB project TOCAI.

$$
\begin{array}{rcll}
S,T & ::= & l :: a \Rightarrow P & \text{Service definition} \\
 & | & l :: P & \text{Located process} \\
 & | & S|T & \text{Parallel composition} \\
 & | & (\nu n)S & \text{New name} \\
 & | & r \doteq s & \text{explicit substitution}
\end{array}
\qquad
\begin{array}{rcll}
P,Q & ::= & \mathbf{0} & \text{Empty process} \\
 & | & c.P & \text{Action prefix} \\
 & | & \mathsf{install}[a \Rightarrow P].Q & \text{Service installation} \\
 & | & \mathsf{invoke}\ a.P & \text{Service invocation} \\
 & | & \mathsf{merge}^p\ e.P & \text{Entry-point} \\
 & | & r \rhd P & \text{Endpoint} \\
 & | & P|Q & \text{Parallel composition} \\
 & | & (\nu n)P & \text{New name} \\
 & | & \mathsf{rec}\ X.P & \text{Recursive process} \\
 & | & X & \text{Recursive call}
\end{array}
$$

Fig. 1. Syntax of systems and processes

$$
c.P \xrightarrow{c} P \qquad \mathsf{merge}^p\ e.P \xrightarrow{e^p} P \qquad \mathsf{invoke}\ a.P \xrightarrow{\bot a} P \qquad a \Rightarrow P \xrightarrow{r \top a} r \rhd P
$$

$$
\frac{P \xrightarrow{\alpha} Q \quad \alpha \in \{\bot a, c, e^p\}}{r \rhd P \xrightarrow{r\ \alpha} r \rhd Q} \qquad \frac{P \xrightarrow{\alpha} Q \quad \alpha \notin \{\bot a, c, e^p\}}{r \rhd P \xrightarrow{\alpha} r \rhd Q}
$$

$$
\mathsf{install}[a \Rightarrow R].P \xrightarrow{a[R]} P \qquad \frac{P \xrightarrow{a[R]} Q}{l :: P \xrightarrow{\tau} l :: Q \,|\, l :: a \Rightarrow R} \qquad \frac{P \xrightarrow{\alpha} Q \quad \alpha \notin \{a[R]\}}{l :: P \xrightarrow{\alpha} l :: Q}
$$

$$
\frac{\mathcal{A} \xrightarrow{r\ c} \mathcal{A}' \quad \mathcal{B} \xrightarrow{r\ \overline{c}} \mathcal{B}'}{\mathcal{A}|\mathcal{B} \xrightarrow{\tau} \mathcal{A}'|\mathcal{B}'} \qquad \frac{\mathcal{A} \xrightarrow{re^+} \mathcal{A}' \quad \mathcal{B} \xrightarrow{se^-} \mathcal{B}'}{\mathcal{A}|\mathcal{B} \xrightarrow{\tau} \mathcal{A}'|\mathcal{B}'|s \doteq r} \qquad \frac{S \xrightarrow{r \top a} S' \quad T \xrightarrow{r \bot a} T'}{S|T \xrightarrow{\tau} S'|T'}
$$

$$
\frac{\mathcal{A} \xrightarrow{\alpha} \mathcal{A}'}{\mathcal{A}|\mathcal{B} \xrightarrow{\alpha} \mathcal{A}'|\mathcal{B}} \qquad \frac{\mathcal{A} \xrightarrow{\alpha} \mathcal{A}' \quad n \notin \mathrm{n}(\alpha)}{(\nu n)\mathcal{A} \xrightarrow{\alpha} (\nu n)\mathcal{A}'} \qquad \frac{\mathcal{A} \equiv \mathcal{A}' \quad \mathcal{A}' \xrightarrow{\alpha} \mathcal{B}' \quad \mathcal{B}' \equiv \mathcal{B}}{\mathcal{A} \xrightarrow{\alpha} \mathcal{B}}
$$

Fig. 2. Operational semantics

2 μse Basics

The syntax of μse is in Fig. 1. We assume given sets of names for services (ranged by $a, ...$), sessions ($r, s, ...$), communication channels ($x, ...$), co-actions ($\overline{x}, ...$), entry-points ($e, ...$), and sites ($l, ...$). We let $c, ...$ range over $x, \overline{x}, ...$ and let $\overline{\overline{c}} = c$.

Processes $P, Q, ...$ comprise ordinary operators such as the nil process $\mathbf{0}$, parallel composition $P|Q$, communication prefixes $c.P$, recursion $\mathsf{rec}\ X.P$ and name restriction $(\nu n)P$, together with primitives $\mathsf{invoke}\ a.P$ for service invocation, $\mathsf{install}[a \Rightarrow P].Q$ for dynamic installation of new services, $r \rhd P$ for session endpoints and $\mathsf{merge}^p\ e.P$ for dynamic merge of sessions on suitable entry-points. Merge prefixes are polarized with $p \in \{+, -\}$. Trailing occurrences of $\mathbf{0}$ will be omitted. Systems $S, T, ...$ comprise located service definitions $l :: a \Rightarrow P$ or processes $l :: P$, parallel composition $S|T$, name restriction $(\nu n)S$, and explicit fusions of session names $r \doteq s$. We let $\mathcal{A}, \mathcal{B}, ...$ range over systems and processes.

The operational semantics of μse is in Fig. 2. It relies on the structural congruence in Fig. 3. Roughly, the following interactions are possible. If service definition $l :: a \Rightarrow P$ is available, an invocation to a within a session r creates a new endpoint $l :: r \rhd P$ for r on site l. The prefix $\mathsf{install}[a \Rightarrow R].P$ dynamically

$$\mathcal{A}|\mathcal{A}' \equiv \mathcal{A}'|\mathcal{A} \qquad \mathcal{A}|\mathbf{0} \equiv \mathcal{A} \qquad (\mathcal{A}|\mathcal{A}')|\mathcal{A}'' \equiv \mathcal{A}|(\mathcal{A}'|\mathcal{A}'')$$

$$(\nu n)(\nu m)\mathcal{A} \equiv (\nu m)(\nu n)\mathcal{A} \qquad \left.\begin{array}{c}(\nu n)\mathcal{A} \equiv \mathcal{A} \\ (\nu n)(\mathcal{A}|\mathcal{A}'') \equiv \mathcal{A}|(\nu n)\mathcal{A}''\end{array}\right\} \text{ if } n \notin \mathrm{fn}(\mathcal{A})$$

$$\mathsf{rec}\ X.P \equiv P\{\mathsf{rec}\ X.P/X\} \qquad r \rhd (\nu n)P \equiv (\nu n)(r \rhd P), \text{ if } n \neq r$$

$$l :: (\nu n)P \equiv (\nu n)(l :: P) \qquad l :: P|l :: Q \equiv l :: (P|Q)$$

$$(r \doteq r) \equiv \mathbf{0} \qquad (\nu r)(r \doteq s) \equiv \mathbf{0} \qquad r \doteq s|P \equiv r \doteq s|P\{r/s\} \qquad (r \doteq s) \equiv (s \doteq r)$$

$$r \rhd (s \doteq t|P) \equiv (s \doteq t)|r \rhd P \qquad l :: (r \doteq s|P) \equiv (r \doteq s)|l :: P$$

Fig. 3. Structural congruence rules

installs a new service definition $a \Rightarrow R$ at the top of the site where it runs. Dual action prefixes $c.P$ and $\overline{c}.Q$ can synchronize only if they run on endpoints of the same session. Complementary merges in sessions r and s on the same entry-point e (i.e., $\mathsf{merge}^+\ e$ and $\mathsf{merge}^-\ e$) can synchronize, releasing the fusion $r \doteq s$.

Example 1. Consider $T_1 = (\nu r_1, r_2, r_3)(l_1 :: P_1 \mid l_2 :: P_2 \mid l_3 :: P_3)$, where:

$$\begin{aligned} P_1 &= r_1 \rhd \mathsf{merge}^+\ e_1.x.\overline{y} \\ P_2 &= r_2 \rhd \mathsf{merge}^-\ e_1.\mathsf{merge}^+\ e_2.\overline{x} \\ P_3 &= r_3 \rhd \mathsf{merge}^-\ e_2.y \end{aligned}$$

Session r_1 and r_2 join together on the entry-point e_1 and session r_2 and r_3 join on e_2. After two steps, S reduces to $(\nu r_1)(l_1 :: r_1 \rhd x.\overline{y} \mid l_2 :: r_1 \rhd \overline{x} \mid l_3 :: r_1 \rhd y)$, where $x.\overline{y}$, $\overline{x}$ and y run in the same session.

Example 2. The two-buyers-protocol below is inspired by an example in [6]. Let $T_2 = (\nu r_1, r_2)(l_1 :: \mathit{Buy}_1 \mid l_2 :: \mathit{Buy}_2 \mid l_s :: \mathit{Sell})$, where

$$\begin{aligned} \mathit{Buy}_1 &= r_1 \rhd \mathsf{invoke}\ \mathit{sell}.\overline{\mathit{title}}.\mathit{quote}.\overline{\mathit{bid}}.Q_1 \\ \mathit{Buy}_2 &= r_2 \rhd \mathsf{invoke}\ \mathit{offer}.\mathit{title}.\mathit{quote}.\mathit{bid}.Q_2 \\ \mathit{Sell} &= \mathit{sell} \Rightarrow \mathit{title}.\mathsf{install}[\mathit{offer} \Rightarrow \mathsf{merge}^+\ e.(\overline{\mathit{title}}.\overline{\mathit{quote}}.\overline{\mathit{quote}}.Q)].\mathsf{merge}^-\ e \end{aligned}$$

The service *sell* waits for a buyer to require a quote for a book (*title*). The buyer at l_1 invokes *sell* so that the new service *offer* is installed. Upon the invocation from the buyer at l_2, *offer* provides the book's *title* so that quotes are communicated to both buyers after the sessions are merged by the service. Finally, the buyers communicate over *bid* and the negotiation is concluded by the interactions among Q, Q_1 and Q_2 (not modeled here).

3 Types for Dynamic Multiparty Sessions

We consider types generated from the following CCS-like grammar:

$$\rho, \sigma ::= \mathsf{0} \quad | \quad c.\rho \quad | \quad \sigma|\rho \quad | \quad \beta \quad | \quad \mu\beta.\rho$$

$$\text{(TZERO)}\quad \mathit{\Gamma};\emptyset \vdash \mathbf{0} : \{\mathsf{0} \nearrow \mathsf{0}\} \qquad \text{(TACTION)}\quad \frac{\mathit{\Gamma};\mathit{\Delta} \vdash P : \{\sigma \nearrow \rho\}}{\mathit{\Gamma};\mathit{\Delta} \vdash c.P : \{c * \sigma \nearrow \rho\}} \qquad \text{(TVAR)}\quad \mathit{\Gamma}, X : \mathit{\Phi}; \mathit{\Delta} \vdash X : \{\mathit{\Phi}\}$$

$$\text{(TINVOKE)}\quad \frac{\mathit{\Gamma};\mathit{\Delta} \vdash P : \{\sigma_1|\sigma_2 \nearrow \rho\} \quad \mathit{\Gamma}(a^+) = \sigma \nearrow \rho' \quad \mathit{\Gamma}(a^-) = \mathsf{0} \nearrow \sigma_2}{\mathit{\Gamma};\mathit{\Delta} \vdash \mathsf{invoke}\ a.P : \{\sigma|\sigma_1 \nearrow \sigma_2|\rho\}} \qquad \text{(TREC)}\quad \frac{\mathit{\Gamma}, X : \mathit{\Phi}; \mathit{\Delta} \vdash P : \{\mathit{\Phi}\}}{\mathit{\Gamma};\mathit{\Delta} \vdash \mathsf{rec}\ X.P : \{\mathit{\Phi}\}}$$

$$\text{(TINSTALL)}\quad \frac{\mathit{\Gamma};\mathit{\Delta} \vdash P : \{\sigma_1|\sigma_2 \nearrow \rho\} \quad \mathit{\Gamma}(a^+) = \sigma_1 \nearrow \sigma_2 \quad \mathit{\Gamma}(a^-) = \mathsf{0} \nearrow \rho' \quad \mathit{\Gamma};\mathit{\Delta} \vdash Q : \{\mathit{\Phi}\}}{\mathit{\Gamma};\mathit{\Delta} \vdash \mathsf{install}[a \Rightarrow P].Q : \{\mathit{\Phi}\}}$$

$$\text{(TMERGE)}\quad \frac{\mathit{\Gamma};\mathit{\Delta} \vdash P : \{\sigma_1|\sigma_2|\sigma_3 \nearrow \rho\} \quad \mathit{\Gamma}(e^p) = \sigma|\sigma_2 \nearrow \sigma_3 \quad \mathit{\Gamma}(e^{\overline{p}}) = \sigma'|\sigma'' \nearrow \rho' \quad \sigma \approx \sigma''}{\mathit{\Gamma};\mathit{\Delta} \vdash \mathsf{merge}^p\ e.P : \{\sigma'|\sigma_1|\sigma'' \nearrow \sigma_2|\sigma_3|\rho\}}$$

$$\text{(TPAR)}\quad \frac{\mathit{\Gamma};\mathit{\Delta}_1 \vdash P : \{\sigma \nearrow \rho\} \quad \mathit{\Gamma};\mathit{\Delta}_2 \vdash Q : \{\sigma' \nearrow \rho'\}}{\mathit{\Gamma};\mathit{\Delta}_1|\mathit{\Delta}_2 \vdash P|Q : \{\sigma|\sigma' \nearrow \rho|\rho'\}} \qquad \text{(TSES)}\quad \frac{\mathit{\Gamma};\mathit{\Delta} \vdash P : \{\sigma \nearrow \rho\}}{\mathit{\Gamma};\mathit{\Delta}, r : \sigma \vdash r \rhd P : \{\mathsf{0} \nearrow \mathsf{0}\}}$$

$$\text{(TSERVICE)}\quad \frac{\mathit{\Gamma};\mathit{\Delta} \vdash P : \{\sigma_1|\sigma_2 \nearrow \rho\} \quad \mathit{\Gamma}(a^+) = \sigma_1 \nearrow \sigma_2 \quad \mathit{\Gamma}(a^-) = \mathsf{0} \nearrow \rho'}{\mathit{\Gamma};\mathit{\Delta} \vdash l :: a \Rightarrow P : \{\mathsf{0} \nearrow \mathsf{0}\}}$$

$$\text{(TLOC)}\quad \frac{\mathit{\Gamma};\mathit{\Delta} \vdash P : \{\mathit{\Phi}\}}{\mathit{\Gamma};\mathit{\Delta} \vdash l :: P : \{\mathit{\Phi}\}} \qquad \text{(TSPAR)}\quad \frac{\mathit{\Gamma};\mathit{\Delta}_1 \vdash S : \{\sigma \nearrow \rho\} \quad \mathit{\Gamma};\mathit{\Delta}_2 \vdash T : \{\sigma' \nearrow \rho'\}}{\mathit{\Gamma};\mathit{\Delta}_1|\mathit{\Delta}_2 \vdash S|T : \{\sigma|\sigma' \nearrow \rho|\rho'\}}$$

$$\text{(TNEW)}\quad \frac{\mathit{\Gamma}, n^+ : (\sigma \nearrow \rho), n^- : (\sigma' \nearrow \rho'); \mathit{\Delta} \vdash S : \{\mathit{\Phi}\} \quad \rho \approx \rho'}{\mathit{\Gamma};\mathit{\Delta} \vdash (\nu n)S : \{\mathit{\Phi}\}} \qquad \text{(TNEWR)}\quad \frac{\mathit{\Gamma};\mathit{\Delta}, r : \sigma \vdash S : \{\mathit{\Phi}\} \quad \sigma \in \Downarrow_{\mathsf{0}}}{\mathit{\Gamma};\mathit{\Delta} \vdash (\nu r)S : \{\mathit{\Phi}\}}$$

Fig. 4. Type system

with the expected structural properties and labeled transition relation $\overset{c}{\mapsto}$.

Type judgments for processes take the form $\mathit{\Gamma};\mathit{\Delta} \vdash P : \{\sigma \nearrow \rho\}$, meaning that: (i) P must perform communication activities in σ and ρ, (ii) it plans to interact as σ with the current participants of its session, (iii) it has delegated the interaction ρ to other endpoints that P itself will allow to join its session (via merge or service invocation). For a pair $\mathit{\Phi} = \sigma \nearrow \rho$, we call σ the *current* type, and ρ the *delegated* type.

The type environment $\mathit{\Gamma}$ is a finite partial mapping from variables X and polarized service / entry-point names n^p (with $p \in \{+,-\}$) to type pairs $\sigma \nearrow \rho$, with the understanding that actions in ρ are delegated to $n^{\overline{p}}$. The *linear* session environment $\mathit{\Delta}$ is a finite partial mapping from session names r to types σ, such that $\mathit{\Delta}(r)$ represents the parallel composition of the current types of all endpoints of r. We assume $\mathit{\Delta}(r) = \mathsf{0}$ when $r \notin \mathit{\Delta}$. We let $\mathit{\Delta}_1|\mathit{\Delta}_2$ denote the environment $\mathit{\Delta}$ such that $\mathit{\Delta}(r) = \mathit{\Delta}_1(r)|\mathit{\Delta}_2(r)$ for each $r \in \mathit{\Delta}_1 \cup \mathit{\Delta}_2$

Our type judgments are in Fig. 4. They are parametric w.r.t. three notions: (1) task *separation* $c * \sigma$, (2) *type compatibility* $\approx$, (3) *session completion* $\Downarrow_{\mathsf{0}}$.

Task separation is used to project the activities of P in separate threads (in case they must be delegated). Here we take the most relaxed form of separation, where $c * \sigma = c|\sigma$. Other possibilities exploit prefixes in types to take care of causality information.

Type compatibility $\sigma \approx \rho$ says that σ and ρ are complementary. Let $I(\sigma) = \{c \mid \exists \sigma' : \sigma \overset{c}{\mapsto} \sigma'\}$ denote the set of initial actions σ can perform. Here we take the largest relation on types such that whenever σ is compatible with ρ it holds that either $I(\sigma) = I(\rho) = \emptyset$, or $K = I(\sigma) \cap \overline{I(\rho)} \neq \emptyset$ and, for each $x \in K$ and for each σ' and ρ' such that $\sigma \overset{x}{\mapsto} \sigma'$ and $\rho \overset{\overline{x}}{\mapsto} \rho'$, then σ' and ρ' are compatible.

The completion set $\Downarrow_0$ contains those types σ that express admissible interactions of multiple endpoints. Here we define $\Downarrow_0$ as the largest set of types σ such that: (i) for each $c \in I(\sigma)$ such that $\overline{c} \notin I(\sigma)$ and for each $\sigma \overset{\tau}{\mapsto} \sigma'$ there exists σ'' such that $\overline{c} \in I(\sigma'')$ and $\sigma' \overset{\tau}{\mapsto}^{*} \sigma''$, (ii) if $\sigma \overset{\tau}{\mapsto} \sigma'$ then $\sigma' \in \Downarrow_0$.

We say that Γ is *well-formed* if: (i) whenever $\Gamma(n^p) = \sigma \nearrow \rho$, then $\Gamma(n^{\overline{p}}) = \sigma' \nearrow \rho'$ for some $\rho' \approx \rho$, and (ii) whenever $\Gamma(a^-) = \sigma \nearrow \rho$, then $\sigma = 0$. We say that Δ is *fully-formed* if whenever $\Delta(r) = \sigma$, then $\sigma \in \Downarrow_0$. We say that a system S is *self-typeable* if $\Gamma; \Delta \vdash S : \{0 \nearrow 0\}$ for some well-formed Γ and fully-formed Δ. For simplicity, we shall omit the type rules for explicit fusion of session names and restrict to self-typeable systems S without nested sessions and with no free session name, i.e. such that $\Gamma; \emptyset \vdash S : \{0 \nearrow 0\}$ for some well-formed Γ.

Rule (Tmerge) best illustrates the flavor of our type system. Note that if Γ is well-formed, it must be $\rho' \approx \sigma_3$. Then: (1) P matches σ_3 to ρ', (2) P delegates σ_2 to the other endpoints of the merged session, (3) the other endpoints of the session of P delegate σ to the merging endpoint, and symmetrically (4) the merging endpoint and its partners delegate $\sigma'|\sigma''$ to the partners of P.

The systems T_1 and T_2 from § 2 are self-typeable. The type derivation for T_1 is reported below, where we omit some labels and rules due to space limitation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; \emptyset \vdash \mathbf{0} : \{0 \nearrow 0\}}\ (\textsc{Tzero})}{\Gamma; \emptyset \vdash \overline{x} : \{\overline{x} \nearrow 0\}}\ (\textsc{Taction})}{\Gamma; \emptyset \vdash \mathsf{merge}^{+}\ e_2.\overline{x} : \{\overline{x}|y \nearrow 0\}}\ (\textsc{Tmerge})}{\Gamma; \emptyset \vdash \mathsf{merge}^{-}\ e_1.\mathsf{merge}^{+}\ e_2.\overline{x} : \{0 \nearrow \overline{x}|y\}}\ (\textsc{Tmerge})}{\Gamma; \Delta_2 \vdash P_2 : \{0 \nearrow 0\}}\ (\textsc{Tses})}{\Gamma; \Delta_2 \vdash l_2 :: P_2 : \{0 \nearrow 0\} \quad (\dagger)}\ (\textsc{Tloc})$$

$$\dfrac{\vdots}{\Gamma; \emptyset \vdash T_1 : \{0 \nearrow 0\}} \quad \text{with} \quad \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma; \emptyset \vdash \mathsf{merge}^{+}\ e_1.x.\overline{y} : \{0 \nearrow x|\overline{y}\}}}{\Gamma; \Delta_1 \vdash P_1 : \{0 \nearrow 0\}}}{\Gamma; \Delta_1 \vdash l_1 :: P_1 : \{0 \nearrow 0\}} \quad \dfrac{(\dagger) \qquad \overset{\dfrac{\vdots}{\Gamma; \emptyset \vdash \mathsf{merge}^{-}\ e_2.y : \{0 \nearrow y\}}}{\vdots}}{\Gamma; \Delta_2|\Delta_3 \vdash l_2 :: P_2 \mid l_3 :: P_3 : \{0 \nearrow 0\}}}{\Gamma; \Delta \vdash l_1 :: P_1 \mid l_2 :: P_2 \mid l_3 :: P_3 : \{0 \nearrow 0\}}$$

where $\Gamma = \{\ e_1^{+} : (0 \nearrow x|\overline{y}),\ e_1^{-} : (0 \nearrow \overline{x}|y),\ e_2^{+} : (0 \nearrow 0),\ e_2^{-} : (y \nearrow 0)\ \}$, $\Delta_1 = \{r_1 : 0\}$, $\Delta_2 = \{r_2 : 0\}$, $\Delta_3 = \{r_3 : 0\}$, and $\Delta = \{r_1 : 0, r_2 : 0, r_3 : 0\}$. The condition of $\approx$ for e_1 is satisfied since $x|\overline{y} \approx \overline{x}|y$ holds and the condition for e_2 is satisfied since $0 \approx 0$.

For T_2, we assume $Q \equiv Q_3|Q_4$ with $Q_1 \approx Q_3$ and $Q_2 \approx Q_4$, write b for *bid*, t for *title*, q for *quote*, and have $\Gamma;\emptyset \vdash T_2 : \{\mathsf{0} \nearrow \mathsf{0}\}$ for Γ shown below (together with some excerpts of type derivations).

$$\begin{aligned} \Delta = {} & \{\ r_1 : \mathsf{0},\ r_2 : \mathsf{0}\ \} \\ \Gamma = {} & \{\ sell^{+} : (\mathsf{0} \nearrow b|t|\overline{q}|Q_3), \\ & \quad sell^{-} : (\mathsf{0} \nearrow \overline{b}|\overline{t}.q|Q_1), \\ & \quad offer^{+} : (\mathsf{0} \nearrow \overline{b}|t|\overline{q}|Q_4), \\ & \quad offer^{-} : (\mathsf{0} \nearrow b|\overline{t}|q|Q_2), \\ & \quad e^{-} : (\overline{b} \nearrow \mathsf{0}), \\ & \quad e^{+} : (\overline{q}|b|Q_3 \nearrow \mathsf{0})\ \} \end{aligned}$$

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma;\emptyset \vdash \overline{t}.q.\overline{b}.Q_1 : \{\overline{b}|\overline{t}|q|Q_1 \nearrow \mathsf{0}\}}}{\Gamma;\emptyset \vdash \mathsf{invoke}\ sell.\overline{t}.q.\overline{b}.Q_1 : \{\mathsf{0} \nearrow \overline{b}|\overline{t}|q|Q_1\}}}{\Gamma;\Delta \vdash r_1 \triangleright \mathsf{invoke}\ sell.\overline{t}.q.\overline{b}.Q_1 : \{\mathsf{0} \nearrow \mathsf{0}\}}$$

Similarly, $\Gamma;\Delta \vdash r_2 \triangleright \mathsf{invoke}\ offer.\overline{t}.q.b.Q_2 : \{\mathsf{0} \nearrow \mathsf{0}\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma;\emptyset \vdash t.\overline{q}.\overline{q}.(Q) : \{\overline{q}|t|\overline{q}|Q \nearrow \mathsf{0}\}}}{\Gamma;\emptyset \vdash \mathsf{merge}^{+}\ e.(t.\overline{q}.\overline{q}.(Q)) : \{\overline{b}|t|\overline{q}|Q_4 \nearrow \overline{q}|Q_3\}}}{\Gamma;\emptyset \vdash offer \Rightarrow \mathsf{merge}^{+}\ e.(t.\overline{q}.\overline{q}.(Q)) : \{\mathsf{0} \nearrow \mathsf{0}\}} \quad \dfrac{\Gamma;\emptyset \vdash \mathbf{0} : \{\mathsf{0} \nearrow \mathsf{0}\}}{\Gamma;\emptyset \vdash \mathsf{merge}^{-}\ e : \{b|\overline{q}|Q_3 \nearrow \mathsf{0}\}}}{\Gamma;\emptyset \vdash sell \Rightarrow t.\mathsf{install}[offer \Rightarrow \mathsf{merge}^{+}\ e.(t.\overline{q}.\overline{q}.(Q))].\mathsf{merge}^{-}\ e : \{\mathsf{0} \nearrow \mathsf{0}\}}$$

This is still a preliminary study, but we conjecture that self-typeability enjoys subject reduction and that any non self-typeable system has at least one session that can deadlock on some pending communication. On the other hand, we point out that some self-typeable systems can deadlock, because, e.g., rule (Taction) relaxes the order of execution of actions, and the type system checks neither that the number of positive merges on an endpoint is the same as the number of negative merges on such endpoint nor that the number of invocations to a service is less than the number of times such service is available / installed. We guess that stronger guarantees can be obtained by restricting the syntax of endpoints, by tuning the definitions of $c * \sigma$, $\approx$ and $\Downarrow_0$, and by extending the types with capability / obligation level annotations à la Kobayashi [7].

References

1. E. Bonelli and A. Compagnoni. Multipoint session types for a distributed calculus. *TGC'07*, vol. 4912 of *LNCS*, pp. 240–256. Springer, 2008.
2. M. Boreale, *et al.* SCC: a service centered calculus. *WS-FM'06*, vol. 4184 of *LNCS*, pp. 38–57. Springer, 2006.
3. R. Bruni, I. Lanese, H. Melgratti, and E. Tuosto. Multiparty sessions in SOC. *COORDINATION'08*, vol. 5052 of *LNCS*, pp. 67–82. Springer, 2008.
4. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. *ESOP'07*, vol. 4421 of *LNCS*, pp. 2–17. Springer, 2007.
5. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. *ESOP'98*, vol. 1381 of *LNCS*, pp. 22–138. Springer, 1998.
6. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *POPL'08*, pp. 273–284. ACM, 2008.
7. N. Kobayashi. Typical tool. http://www.kb.ecei.tohoku.ac.jp/~koba/typical/.
8. I. Lanese, V. Vasconcelos, F. Martins, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. *SEFM'07*, pp. 305–314. IEEE, 2007.

Synchronous Multiparty Session Types

Extended Abstract

Andi Bejleri
Imperial College London

Nobuko Yoshida
Imperial College London

Abstract

This abstract continues the work on multiparty session types initiated by Honda et al. [5] for synchronous communications. Synchronous communications model control for timing events and strong sequentiality order of messages. The calculus proposed in this paper has a more relaxed syntax and operational semantics than the calculus of asynchronous communication, and the type soundness proof is therefore more straightforward. The calculus models non-arbitrary transmission of channels via multipolarity labels, multicasting and invisibility of delegation at global type definition. A new definition of linearity is provided with the modified causality analysis for ordering communications. The type system of the calculus is proved to be sound with respect to the operational semantics and coherent with respect to the global types. The full version of this abstract is found in [1].

1 Introduction

Multiparty session types for an asynchronous communication calculus have recently been introduced by Honda et al. [5] and Bonelli-Compagnoni [2]. The idea of multiparty session types of the first work is to represent the interactions between processes globally rather than by a list of binary sessions types as it is done in the second work. The system given in this abstract follows the typing theory of the first work.

In the context of multiparty sessions, controlling the timing of events becomes important: for example, in a fire control system for a building, we expect that all the fire alarms run before the elevators are blocked. This scenario would be modeled by the control process as a multicast send of an *ON* message to the fire alarms and a multicast send of a *BLOCK* message to the elevators. Synchronous communications support control for timing events. In the fire control example, the second multicast sending will happen only after the message is received by the multicast group of the first send, resulting therefore in the desired behavior of the fire control system.

Binary session types [4, 6] on their own are not rich enough to express dependencies between different interactions in a multiparty session. A notion of global type is therefore introduced in [5] to formalise the global behavior of a multiparty session. The example below illustrates the key ideas of multiparty session, dependencies between interactions and global description. In a Client-Addition-Successor- Predecessor session, the communication protocol is defined as: the Client sends two natural numbers to the Addition and waits to receive from him the sum of them. If the second operand is equal to 0 then the Addition sends to Client the first operand as result, otherwise it sends the first operand to the Successor and receives from him the successor of it, then it sends the second operand to the Predecessor and receives from him the predecessor of it; this behavior is repeated until the second operand is equal to 0.

The global description of the communication protocol in a name-arrow based representation is

$$
\begin{array}{l}
\texttt{Client} \rightarrow \texttt{Addition}\colon \langle \mathit{int} \rangle. \\
\texttt{Client} \rightarrow \texttt{Addition}\colon \langle \mathit{int} \rangle. \\
\mu\mathbf{t}.\texttt{Addition} \rightarrow \{\texttt{Successor}, \texttt{Predecessor}\} : \{ \\
\qquad\qquad \texttt{true} : \texttt{Addition} \rightarrow \texttt{Client}\colon \langle \mathit{int} \rangle.\mathrm{end}, \\
\qquad\qquad \texttt{false} : \texttt{Addition} \rightarrow \texttt{Successor}\colon \langle \mathit{int} \rangle. \\
\qquad\qquad\qquad \texttt{Successor} \rightarrow \texttt{Addition}\colon \langle \mathit{int} \rangle. \\
\qquad\qquad\qquad \texttt{Addition} \rightarrow \texttt{Predecessor}\colon \langle \mathit{int} \rangle. \\
\qquad\qquad\qquad \texttt{Predecessor} \rightarrow \texttt{Addition}\colon \langle \mathit{int} \rangle.\mathbf{t}\}
\end{array}
$$

where $\texttt{A} \rightarrow \texttt{B}, \texttt{C}\colon \langle t \rangle$ means that process A sends simultaneously a message of type t to process B and C, and $\texttt{A} \rightarrow \texttt{B}, \texttt{C}\colon \{l_1\colon \cdots, ..., l_j\colon \cdots\}$ means that process A sends simultaneously a label l_i where $i \in \{1, ..., j\}$ to process B and C. We have omitted channels from the example for simplicity.

In binary sessions, the Client-Addition-Successor-Predecessor protocol is represented by three sessions: Client-Addition$_1$, Successor-Addition$_2$ and Predecessor -Addition$_3$. The interactive structure of each of the processes is

$$
\begin{array}{rl}
\texttt{Client} = & !\langle \mathit{int} \rangle; !\langle \mathit{int} \rangle; ?\langle \mathit{int} \rangle; \mathsf{end} \\
\texttt{Addition}_1 = & ?\langle \mathit{int} \rangle; ?\langle \mathit{int} \rangle; !\langle \mathit{int} \rangle; \mathsf{end} \\
\texttt{Successor} = & \mu\mathbf{t}.\&\{\mathit{true} : \mathsf{end}, \mathit{false} :?\langle \mathit{int} \rangle; !\langle \mathit{int} \rangle; \mathbf{t}\} \\
\texttt{Addition}_2 = & \mu\mathbf{t}. \oplus \{\mathit{true} : \mathsf{end}, \mathit{false} :!\langle \mathit{int} \rangle; ?\langle \mathit{int} \rangle; \mathbf{t}\} \\
\texttt{Predecessor} = & \mu\mathbf{t}.\&\{\mathit{true} : \mathsf{end}, \mathit{false} :?\langle \mathit{int} \rangle; !\langle \mathit{int} \rangle; \mathbf{t}\} \\
\texttt{Addition}_3 = & \mu\mathbf{t}. \oplus \{\mathit{true} : \mathsf{end}, \mathit{false} :!\langle \mathit{int} \rangle; ?\langle \mathit{int} \rangle; \mathbf{t}\}
\end{array}
$$

where $!\langle t \rangle$ denotes an output of type t, $?\langle t \rangle$ denotes an input of type t, $\oplus\{l_1\colon \cdots, ..., l_j\colon \cdots\}$ denotes a choice of a label and $\&\{l_1\colon \cdots, ..., l_j\colon \cdots\}$ denotes branching on a set of labels. Addition consists of three interactive structures each corresponding to one of the three sessions. This representation of the interactions is well-typed by a binary session type system as the interaction structures Client-Addition$_1$, Successor-Addition$_2$ and Predecessor-Addition$_3$ are reciprocal between them, respectively. However, the binary session representation of the processes breaks the order of messages because in the case when the second operand is not 0, Addition should add the second operand to the first one before returning it to Client and this dependency between the sessions Client-Addition$_1$ and Addition$_{2,3}$-Successor- Predecessor can not be captured by binary session types.

In multiparty sessions, the protocol is represented by the global description given above and as a consequence the interactive structure of the Addition process is

$$?\langle \mathit{int} \rangle; ?\langle \mathit{int} \rangle; \mu\mathbf{t}. \oplus \{\mathit{true} :!\langle \mathit{int} \rangle; \mathsf{end}, \mathit{false} :!\langle \mathit{int} \rangle; ?\langle \mathit{int} \rangle; !\langle \mathit{int} \rangle; ?\langle \mathit{int} \rangle; \mathbf{t}\}.$$

The interactive structure of the other processes Client, Successor and Predecessor is the same. The Addition process has now only one interactive structure that defines its participation in the session and that respects the definition of the protocol. Hence, the use of global types allows a more complete and intelligible definition of communication protocols in multiparty sessions also in presence of programming features such as recursion, selection-branching and multicasting.

In synchronous communications calculi, the runtime sequence of interactions follows more strictly the one of the global behavior description than the asynchronous communication calculus with queue [5], resulting in a simpler typing system of the global behavior.
In the global type definition, a programmer does not specify only the communications of a protocol but also the channels where the communications take place. This is an important feature in multiparty session types as global types are not a simple human interpretable descriptive language; global types together with the projection algorithm and type-system represent a type-checking tool for communication-based processes.
In the context where different interactions can use the same channel, global types alone do not guarantee that the message-order at run-time of the session is the same as the one specified in the global behavior. A *linearity property* checks if the use of same channel in two different communications of a global type does not break the order of messages as specified in the global description. A precise casual analysis for sequencing two synchronous communications is provided in the next section. This analysis leads to the definition of causalities in a global type and the linearity property uses this analysis in a global type to define when it is safe in terms of non-ambiguity, in two communications to use the same channel.

Finally, each process is type-checked with the type obtained as the result of projecting the process identity on the global type. A flavor of the projecting algorithm is given in the next section. The type-checking algorithm analyses the process via a bottom-up strategy, starting to type check the process communication behavior (after the state of willing/waiting to initiate a session). If the type-checking algorithm ends at the stage where the analyser reaches the inaction process with a bottom session type then the processes satisfy its part of the communication protocol .

Contributions. The next paragraphs summarize the main technical contributions of this work.

A Simpler Calculus. The syntax of the calculus is more relaxed than the one introduced by Honda et al. [5]. We do not distinguish syntactically between a primitive value send and a session channel send following the idea firstly proposed in [3]. The syntax of the calculus does not introduce queues, neither at programmer code level nor at runtime code level, in contrast to the asynchrony communication calculus and in accordance to pioneering session calculi [4, 6].

Higher Order Communication. High-order communication, a.k.a. delegation, is defined $k!\langle k' \rangle \mid k?\langle k' \rangle$ in the first system [4] to support it, where the receiver posses the transmitted channel (k') at implementation time, so before the communication takes place. The calculus of this abstract allows the transmission of channels with the receiver not possessing the channels until the communication happens. This feature cannot be supported safely by simply adding substitution to the receiver end-point in the delegation rule of the operational semantics of Honda-Vasconcelos-Kubo systems because then one may build a well-typed term that makes use of the new rule and after some reductions is ill-typed. The solution proposed in this paper is similar to the one proposed in [3, 6] with the extension of having multipolarity labeled session channels rather than binary labeled ones.

Multicasting. The calculus supports the delivery of messages to a group of peers simultaneously (multicasting).

Invisibility of Delegation at Global Types. In the programming methodology of the calculus, the global type should define only the interactions of that session without knowing how this interactions might change due to other possible global types that formalise delegation. The global types of the Alice-Bob-Carol example in the asynchronous multiparty [5], $G_a = \mathtt{A} \to \mathtt{B}\colon\ t_1\langle s_1!\langle int\rangle @B\rangle.\mathsf{end}$ and $G_b = \mathtt{B} \to \mathtt{C}\colon\ s_1!\langle int\rangle.\mathsf{end}$ do not support invisibility. The *behavior global type* G_b expresses the changes of communications, from $\mathtt{A} \to \mathtt{C}$ to $\mathtt{B} \to \mathtt{C}$, due to the delegation in G_a expelling in this way A from being a participant of session started on b. This inconsistency of information between G_b and the implementation of session initiated on b makes process A not type-checked even though it is correct. Following the above methodology, the global types $G_a = \mathtt{A} \to \mathtt{B}\colon\ t_1\langle s_1!\langle int\rangle @A\rangle$ and $G_b = \mathtt{A} \to \mathtt{C}\colon\ s_1!\langle int\rangle.\mathsf{end}$ type-check all three processes.

2 Synchronous Multiparty Sessions

The Calculus. The two communication-based operation on session channels are values sending-receiving and label selection-branching. These two operations together with recursion represent the core of the calculus operational semantics. Also multicast session request-acceptance represents a communication idiom that is used only at session initiation.

The system given in this paper uses the same mechanism as in [3, 6] to support in a safe way non arbitrary transmission of channels with the only change of having multipolarity. In other word, their system uses a binary polarity (+, -) for the binary session calculus because sessions are represented by only two processes but for the multiparty calculus the number of processes participating in a session is generally more than two, so we introduce an index label ranging [1, ..., p], where p is the number of processes involved in a session, which is assigned to every channel of the session when substituted in a process.

The non capital Latin letters a, b, c, ... represent shared names; e, e', ... represent expressions; l, l_1, l_2, ... refer to branch labels; m_1, m_2, ..., p, q, ... range over naturals; κ_p, ... refer to session channel values; $x, y, z, ...$ refer to variables of the calculus. The capital Latin letter $P, P_1, P_2, ...Q$, ... refer to processes terms.

The operational semantics communication-based rules for the synchronous communication calculus are

$$\begin{array}{l}
\overline{a}_{[\mathtt{2..n}]}\,(\tilde{y}).P_1 \mid a_{[\mathtt{2}]}\,(\tilde{y}).P_2 \mid \cdots \mid a_{[\mathtt{n}]}\,(\tilde{y}).P_n \\
\;\to\;\; (\nu\tilde{\kappa})(P_1[\tilde{\kappa_1}/\tilde{y}] \mid P_2[\tilde{\kappa_2}/\tilde{y}] \mid ... \mid P_n[\tilde{\kappa_n}/\tilde{y}]) \qquad\qquad [\textsc{Link}] \\
\kappa_{i_1}[m_1, ..., m_n]!\langle\tilde{e}\rangle; P_1 \mid \kappa_{i_2}[m_1]?(\tilde{y}); P_2 \mid \cdots \mid \kappa_{i_{n+1}}[m_n]?(\tilde{y}); P_{n+1} \\
\;\to\;\; P_1 \mid P_2[\tilde{v}/\tilde{y}] \mid \cdots \mid P_{n+1}[\tilde{v}/\tilde{y}] \quad (i_1 \neq i_2 \neq \cdots \neq i_{n+1}, \tilde{e} \downarrow \tilde{v})\ [\textsc{Multicasting}] \\
\kappa_{i_1}[m_1, ..., m_n] \lhd l_i; P_1 \mid \kappa_{i_2}[m_1] \rhd \{l_j : P_{2_j}\}_{j\in I} \mid \cdots \mid \kappa_{i_{n+1}}[m_n] \rhd \{l_j : P_{n+1_j}\}_{j\in I} \\
\;\to\;\; P_1 \mid P_{2_i} \mid \cdots \mid P_{n+1_i} \qquad (i_1 \neq i_2 \neq \cdots \neq i_{n+1}, i \in I) \qquad [\textsc{MultiLabel}]
\end{array}$$

(II)	(IO)	(OI)	(OO)	(OO, II)	(IO, OI)
A, S	A, S	S	S	A, S	A, S
$\mathtt{P}_1 \rightarrow \mathtt{P} : k_1$	$\mathtt{P}_1 \rightarrow \mathtt{P} : k_1$	$\mathtt{P} \rightarrow \mathtt{P}_1 : k_1$	$\mathtt{P} \rightarrow \mathtt{P}_1 : k_1$	$\mathtt{P} \rightarrow \mathtt{P}_1 : k$	$\mathtt{P}_1 \rightarrow \mathtt{P} : k$
$\mathtt{P}_2 \rightarrow \mathtt{P} : k_2$	$\mathtt{P} \rightarrow \mathtt{P}_2 : k_2$	$\mathtt{P}_2 \rightarrow \mathtt{P} : k_2$	$\mathtt{P} \rightarrow \mathtt{P}_2 : k_2$	$\mathtt{P} \rightarrow \mathtt{P}_1 : k$	$\mathtt{P} \rightarrow \mathtt{P}_1 : k$

Figure 1: Causality Analysis

The other rules are the same as in other session calculi [4, 6, 5].

[Link] initiates a session between n peers. The result of the reduction is the generation of as many session channels as session variables and the substitution of them in processes. Note that the session channels for each process are labeled by a multipolarity label ranging [1, ..., n], where n is the number of processes involved in the session.

[Multicasting] actions the synchronous value sending-receiving between two or more peers. The result of the reduction is the substitution of the place holders with the values received by the receiver. Note that the reduction holds if the channel names are the same in both peers even though the polarity is different. The relation $\downarrow$ evaluates the expression e to the value v and the value v to itself.

[MultiLabel] actions the synchronous label selection-branching communication between two or more peers. The selector process sends the label l_i to the branching processes and the result of reduction are the process that were labeled by l_i ($P_{[2...n]i}$).

Linearity. Figure 1 presents all the causalities with concrete processes instances for illustration where $\mathtt{P}$, $\mathtt{P}_1$, $\mathtt{P}_2$ denote processes identities. The letter A and S represent respectively the asynchronous and synchronous communication calculus where these cases are considered. All the four causalities are considered for ordering in this calculus due to its synchronous communication nature, and there is an order between send-receive (OI), receive-send (IO), between two sends (OO) and between two receives (II). The send-receive (OI) and send-send (OO) are not considered in the asynchronous communication calculus [5]. The (OI) ordering states that process P sends a value before receiving one and the received value is different from the one sent. Due to independence of sent-received values, one can receive first and then sends the value without changing the semantics of the program. In asynchronous communications, (OI) does not introduce any dependency on messages order so this causality is not considered. In the second ordering (OO), the messages send are individual and the order when process P sends to P_1 and P_2 them does not alter the semantics of the program, therefore the order introduced by (OO) does not introduce any dependency on the order of messages.

The intuition behind linearity for synchronous communications is that two ordered communications $n_1 = \mathtt{p}_1 \rightarrow \mathtt{p}'_1 : m$ and $n_2 = \mathtt{p}_2 \rightarrow \mathtt{p}'_2 : m$ can use the same channel if the second communication happens only after the first one. Each of the two processes involved in n_2 have to undertake another communication ($\mathtt{p_j} \rightarrow \mathtt{p}_2/\mathtt{p}_2 \rightarrow \mathtt{p_j}$, $\mathtt{p_k} \rightarrow \mathtt{p}'_2/\mathtt{p}'_2 \rightarrow \mathtt{p_k}$) and these communications should happen after the first communication n_1. It is more easy in synchronous communications to establish when a certain communication has happened or not; i.e if the sender or receiver of that communication action another communication then we are sure that the former one has happened. In asynchronous communications only when the receiver of the communication actions another communication then we are sure that the former one has happened.

Type Preservation. The static type system rules are basically the ones for binary session calculi [4] with rules that type multicasting session initiation as in [5] and multipolarity channel restriction as in [6].

The projection algorithm mentioned in Section 1 is defined for primitive value sending in this section. Let G be linear then the *projection of G onto* $\mathtt{p}$, written $G \upharpoonright \mathtt{p}$, is inductively given as:

$$(\mathtt{p} \to \mathtt{p_1}, ..., \mathtt{p_j} : m_1, ..., m_j \, \langle \tilde{S} \rangle . G') \upharpoonright \mathtt{p_i} \quad \overset{\text{def}}{=}$$

$$\begin{cases} m_1, ..., m_j ! \langle \tilde{S} \rangle ; (G' \upharpoonright \mathtt{p_i}) & \textit{if } \mathtt{p_i} = \mathtt{p} \textit{ and } \mathtt{p_i} \notin \mathtt{p_1}, ..., \mathtt{p_j} \\ m_i ? \langle \tilde{S} \rangle ; (G' \upharpoonright \mathtt{p_i}) & \textit{if } \mathtt{p_i} \in \mathtt{p_1}, ..., \mathtt{p_j} \textit{ and } i \in \{1, ..., j\} \textit{ and } \mathtt{p_i} \neq \mathtt{p} \\ (G' \upharpoonright \mathtt{p_i}) & \textit{if } \mathtt{p_i} \notin \mathtt{p_1}, ..., \mathtt{p_j} \textit{ and } \mathtt{p_i} \neq \mathtt{p} \end{cases}$$

G is *coherent* if it is linear and $G \upharpoonright \mathtt{p}$ is well-defined for each process defined in G similarly for each carried global type inductively.

Preservation Theorem a.k.a. subject reduction ensures that the type of an expression is preserved during its evaluation.

Theorem 2.1 (preservation) *$\Gamma \vdash P \triangleright \Delta$ such that Δ is coherent and $P \to P'$ imply $\Gamma \vdash P' \triangleright \Delta'$ where $\Delta = \Delta'$ or $\Delta \to \Delta'$.*

$\Delta \to \Delta'$ denotes a reduction relation between session types which abstractly represents the interactions at session channels.

As future work, we plan to extend our system by disallowing at compile time the inner delegation problem and support safely delegating in multicast. These two issues are explained in more detail in [1].

References

[1] Andi Bejleri and Nobuko Yoshida. Synchronous multiparty session types. Available at `http://www.doc.ic.ac.uk/~ab406`.

[2] Eduardo Bonelli and Adriana Compagnoni. Multipoint session types for a distributed calculus. *Electr. Notes Theor. Comput. Sci.*, 2007.

[3] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.

[4] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, pages 122–138, 1998.

[5] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008.

[6] Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.

Session-based Choreography with Exceptions

Marco Carbone
Queen Mary University of London

Abstract

Choreography has recently emerged as a pragmatic and concise way of describing communication-based systems such as web services and financial protocols. Recent studies have investigated the transition from the design stage of a system to its implementation providing an automatic way of mapping a choreograhy into executable code.
In this note, we focus on an extension of choreography with a communication-based (interactional) exception mechanism and we give its formal semantics. In particular, we discuss through some examples how interactional exceptions at choreography level can be implemented into end-point code.

1 Introduction

Due to fast-changing technologies and exponential growth of Internet and world-wide web, applications based on communication are becoming vital in the practical world. Such communication-centred applications are often known as *Web Services*, the first major programming trend which positions communication as a key element of high-level programming.

An emerging paradigm for programming communication is the so-called *Choreography*. This discipline focuses on the fact that an architect, when designing a distributed system, no longer describes the behaviour of the single peers (*end-point behaviour*) but establishes how the various interactions between entities happen by giving a *global description* (choreography) of the system. In a traditional approach, the architect would describe the communication operations, e.g. an input, that must be performed at each peer. Unfortunately, this makes very difficult to have a global view of how the whole system being designed works. On the other hand, global descriptions can picture the whole scenario of where and when a communication has to happen. The architect will now decide that e.g. there will be a message from A to B and no longer think how this will be implemented at A (sending a message) or B (waiting to receive a message). Hence, choreography offers a vantage view of the system facilitating the design stage and leaving the implementation details to the (possibly automated) process of generating an (possibly sound) end-point code, called *end-point projection*.

Exceptions are a mechanism widely adopted in modern programming languages (e.g. Java, C#) for dealing with unwanted system behaviours i.e. they are designed to handle the occurrence of some conditions interrupting the normal flow of execution of a program. While the classical notion of exception is bound to the local flow of a process, in communication-centred programming exceptions are about the flow of interactions where a sudden interruption must involve all interacting participants. We shall call this kind of exception an *interactional exception* [5].

This note discusses an extension of choreography with exceptions. In particular, we will see through examples, how exceptions are naturally interactional in choreography (choreography is about interactions) while they require exception propagation when moving to end-point behaviour.

2 Extending Choreography with Exceptions

Syntax. The global calculus [3, 4] is a model of choreography based on WS-CDL [10], the most known language for choreography. We hereby extend it with the new terms for handling interactional exceptions. Its selected syntax is given by standard BNF. Below, I, J denote terms of the global calculus:

$$\begin{array}{rlrlr} I, J ::= & A \to B : b(s)[\tilde{t}, I, J] & \text{(init)} & \mid \mathbf{0} & \text{(inaction)} \\ & \mid A \to B : s\langle op, e, y\rangle \,.\, I & \text{(com)} & \mid I \mid J & \text{(par)} \\ & \mid \mathbf{throw} & \text{(throw)} & \mid I + J & \text{(sum)} \\ & \mid \mathbf{if}\ e@A\ \mathbf{then}\ I\ \mathbf{else}\ J & \text{(cond)} & \mid (\nu s)\ I & \text{(new)} \\ & \mid \mathbf{rec}\ X \,.\, I & \text{(rec)} & \mid X & \text{(recVar)} \end{array}$$

where $a, b, c, \ldots$ range over service channels which may be considered as e.g. shared channels of web services; s, t, r range over session channels, the communication channels freshly generated for each session; A, B, C range over participants who are equipped with their local variables denoted by $x, y, z \ldots$; e is an arithmetic or other first order expression; and X ranges over term variables used for recursion.

In the syntax above, terms (init) and (throw) are the novel constructs. (init) describes a system where participant A invokes a service b located at participant B and starts a session s. The novelty is in the triple $[\tilde{t}, I, J]$: choreography I, called the *default choreography*, describes the normal (or default) behaviour of the system, while J is the *exception handler* to be run whenever an exception is thrown. Vector $\tilde{t}$ has a pivotal role: an exception on any t_i has to be propagated (at implementation level) to the whole $\tilde{t}$ discarding any other handler previously instantiated on any t_i. As an example, the choreography $A \to B : b(s)[s, \quad B \to C : c(t)[t, I, J] \quad , J']$ describes a system different from the one described by $A \to B : b(s)[s, \quad B \to C : c(t)[(s,t), I, J] \quad , J']$. In the first case, the raising of an exception in I, would only concern the interactions on t between B and C while, in the second case, it would also affect the interactions on s. For consistency, we assume that

$$\begin{array}{l} \text{For any } C \to D : a(s')[\tilde{t}', I', J'] \text{ occurring in } A \to B : b(s)[\tilde{t}, I, J], \\ \text{ we have } t_i \in \tilde{t}' \text{ if and only if } \tilde{t} \subseteq \tilde{t}'. \end{array} \tag{1}$$

(throw) denotes the throwing of an exception. (com) models in-session communication between participants A and B: message e will be stored in variable y located at B while op indicates the type of operation [10]. (cond) is the standard conditional operator where the guard e is evaluated at A. (rec) and (recVar) are respectively recursion and recursion variable while (sum) is non-deterministic choice. The term (par) is the parallel composition of choreographies and (new) is session channel restriction. The syntax presented is not exhaustive and can be extended with other constructs [3, 4].

A simple Financial Protocol. This example and the following one are inspired by [5]. Let us consider a scenario where a buyer Buyer wishes to purchase a product from a seller Seller. Buyer starts a session with Seller who repeats sending quote updates about the product price. When Buyer decides to accept a particular quote, without explicitly notifying Seller, it throws an exception. At this point, Seller and Buyer move together to a new stage (exception stage) where they exchange information for successfully terminating the transaction e.g. credit card details for payment and receipt. We can write this protocol in the global calculus as:

$$\begin{array}{ll}
1. & \mathtt{Buyer} \to \mathtt{Seller}\ :\ \textbf{chSeller}(s)\ [s,\\
\begin{array}{l}2.\\ 3.\end{array} & \left.\begin{array}{l}\quad \textbf{rec}\ X\,\mathtt{.}\,\mathtt{Seller} \to \mathtt{Buyer} : s\langle \mathtt{update}, \mathit{quote}, y\rangle\,\mathtt{.}\\ \quad \textbf{if}\ (y < 100)@\mathtt{Buyer}\ \textbf{then throw else}\ X,\end{array}\right\}\text{default}\\
\begin{array}{l}4.\\ 5.\end{array} & \left.\begin{array}{l}\quad \mathtt{Seller} \to \mathtt{Buyer} : s\langle \mathtt{conf}, \mathit{cnum}, x\rangle\,\mathtt{.}\\ \quad \mathtt{Buyer} \to \mathtt{Seller} : s\langle \mathtt{data}, \mathit{credit}, x\rangle\ ]\end{array}\right\}\text{handler}
\end{array}$$

In line 1, Buyer invokes service **chSeller** from Seller. Line 2 and 3 compose the default choreography: the interaction $\mathtt{Seller} \to \mathtt{Buyer} : s\langle \mathtt{update}, \mathit{quote}, y\rangle$ models the sending of a quote *quote* from Seller to Buyer who will store the received value at variable y. In line 3, variable y is checked by Buyer and if its value is less than 100, an exception will be thrown otherwise the course of action will go back to line 2. Lines 4 and 5 describe how the system will handle an exception: Seller will send a confirmation *cnum* and Buyer will reply with its credit card *credit*.

A Financial Protocol with Broker. Following [5], we extend the protocol above including a third participant, a broker Broker whose role is to buy from Seller and resell to Buyer (a typical situation in financial protocols). In this scenario, Buyer will invoke Broker rather than Seller and act almost identically as in the previous example. On the other hand Broker, after being invoked by Buyer and checking his reputation, will invoke Seller and act as Buyer in the previous example. As before, Buyer can raise an exception in case of quote acceptance but also Broker can throw if Buyer's reputation is not satisfactory before even invoking Seller. In the global calculus:

$$\begin{array}{ll}
1. & \mathtt{Buyer} \to \mathtt{Broker}\ :\ \textbf{chBroker}(s)\ [s,\\
\begin{array}{l}2.\\ 3.\\ 4.\\ 5.\\ 6.\\ 7.\\ 8.\\ 9.\\ 10.\\ 11.\end{array} & \left.\begin{array}{l}
\quad \mathtt{Buyer} \to \mathtt{Broker} : s\langle \mathtt{identify}, \mathit{id}, x\rangle\,\mathtt{.}\\
\quad \textbf{if}\ \mathtt{bad}(x)@\mathtt{Broker}\ \textbf{then throw}\\
\quad \textbf{else}\ \mathtt{Broker} \to \mathtt{Seller}\ :\ \textbf{chSeller}(t)[(s,t),\ \textbf{rec}\ X\,\mathtt{.}\\
\left.\begin{array}{l}\qquad \mathtt{Seller} \to \mathtt{Broker} : t\langle \mathtt{update}, \mathit{quote}, y\rangle\,\mathtt{.}\\ \qquad \mathtt{Broker} \to \mathtt{Buyer} : s\langle \mathtt{update}, y + 10\%, y\rangle\,\mathtt{.}\\ \qquad \textbf{if}\ (y < 100)@\mathtt{Buyer}\ \textbf{then throw else}\ X,\end{array}\right\}\text{default}\\
\left.\begin{array}{l}\qquad \mathtt{Seller} \to \mathtt{Broker} : t\langle \mathtt{conf}, \mathit{cnum}, x\rangle\,\mathtt{.}\\ \qquad \mathtt{Broker} \to \mathtt{Buyer} : s\langle \mathtt{conf}, x, x\rangle\,\mathtt{.}\\ \qquad \mathtt{Buyer} \to \mathtt{Broker} : s\langle \mathtt{data}, \mathit{credit}, x\rangle\,\mathtt{.}\\ \qquad \mathtt{Broker} \to \mathtt{Seller} : t\langle \mathtt{data}, x, x\rangle],\end{array}\right\}\text{handler}
\end{array}\right\}\text{default}\\
12. & \left.\quad \mathtt{Seller} \to \mathtt{Buyer} : s\langle \mathtt{reject}, \mathit{reason}, x\rangle\,\mathtt{.}\,I\ ]\right\}\text{handler}
\end{array}$$

In lines 1 and 2, Buyer invokes service **chBroker** and sends its identity *id* to Broker who, in line 3, will check whether Buyer is bad or not. If Buyer is not trusted, Broker will raise an exception which will take both Buyer and Broker to an abortion procedure in line 12 (we leave I unspecified). Note that in this case, Buyer and Broker are the only participants involved so far and the only ones who will move to another conversation for handling the exception.

If Buyer can be trusted, Broker invokes service **chSeller** and forwards to Buyer all quotes received from Seller increasing them by 10%. As before, Buyer will throw an exception whenever s/he decides to accept a quote. In this case, as the participants involved are now Buyer, Broker and Seller, the handler to be executed is the inner one where Broker will forward messages between Buyer and Seller (lines 8-11). This event will also discard the handler in line 12 which, after session initiation in line 4, has become inactive.

Semantics. Above, we have shown how interactional exceptions can be exploited for expressing systems such as financial protocols. Now, we shall formalise the exception mechanism by defining a reduction semantics for the global calculus that can handle exceptions. This is done by enhancing the following notation from [3, 4]:

$$(I, \sigma) \xrightarrow{\tilde{s}} (J, \sigma')$$

which says a global description I in a state σ (which is the collection of all local states of the participants) will be changed into J with a new state σ'. Intuitively, the reduction semantics of choreography will change the state σ by updating the variables as a consequence of each interaction. Label $\tilde{s}$ is used for discarding already thrown exceptions on channels s_i (see below). We shall often omit the label when equal to $\emptyset$.

In order to give semantics to the new exception operations, we need to extend the syntax with the following run-time terms:

$$\mathbf{try}(\tilde{s})\ \{I\}\ \mathbf{catch}\ \{J\} \qquad\qquad\qquad \{\!\!\{I\}\!\!\}$$

The term on the left, called *try-catch block*, reads: "The system is behaving as choreography I; if an exception is thrown on any s_i then handler J will be run". This realises at run-time the default choreography and the handler in initialisation. When an exception is thrown, we need to make sure that handlers do not get brutally terminated by an embedding try-catch block. This is ensured by a *wrap* term $\{\!\!\{I\}\!\!\}$ which reads "the handler I is being executed".

We are now able to give the most relevant rules. The semantics transforms session initialisation into a try-catch term:

$$A \to B : b(\tilde{s})[\tilde{t}, I, J] \ \longrightarrow\ (\boldsymbol{\nu} s)\ \mathbf{try}(\tilde{t})\ \{I\}\ \mathbf{catch}\ \{J\}$$

The following rule handles the raising of an exception:

$$I \Searrow (I', S) \quad \Rightarrow \quad \mathbf{try}(\tilde{t})\ \{\mathbf{throw} \mid I\}\ \mathbf{catch}\ \{J\} \xrightarrow{\tilde{s}\cup S} \{\!\!\{J\}\!\!\} \mid I'$$

When a throw is top-level in a try-catch block then the default choreography terminates and the handler J is run (wrapped). The rule uses a new relation $I \Searrow (I', S)$ called meta reduction [5]. The initial choreography I is transformed into I', the result of erasing and wrapping embedded try-catch blocks; S denotes all those session channels affected by the exception from nested try-catch blocks. This relation is indispensable in interactional exceptions: choreography I may contain wraps or other try-catch blocks which should not be brutally erased and an exception should also be raised. The key rule for defining meta reduction is the following:

$$I \Searrow (I', S) \quad \Rightarrow \quad \mathbf{try}(\tilde{t})\ \{I\}\ \mathbf{catch}\ \{J\} \Searrow (\{\!\!\{J\}\!\!\} \mid I', S \cup \tilde{t})$$

We need two more rules to make our semantics sound. One is for removing try-catch blocks on which an exception has already been thrown:

$$\tilde{s} \subseteq \tilde{t}, \quad I \xrightarrow{\tilde{t}} I' \quad \text{and} \quad I' \Searrow I'' \qquad \Rightarrow \qquad \mathbf{try}(\tilde{s})\ \{I\}\ \mathbf{catch}\ \{J\} \xrightarrow{\tilde{t}\cup S} I''$$

Above, we need $I' \Searrow I''$ in case I is a parallel composition of more choregraphies. In fact, try-catch blocks in parallel with the choreography where an exception was thrown are left untouched while anything in I is supposed to be terminated.
The following rule removes session channels on the labels when there is a restriction:

$$I \xrightarrow{\tilde{s}} J \quad \Rightarrow \quad (\boldsymbol{\nu} t)\ I \xrightarrow{\tilde{s}\setminus\{t\}} (\boldsymbol{\nu} t)\ J$$

Semantics of the Financial Protocol with Broker. We now show how the rules work in our second example. When $\mathtt{bad}(x)$ holds, we have that

$$(\nu s) \quad \mathbf{try}\ (s)\ \{\ \mathbf{if}\ \mathtt{bad}(x)\ \mathbf{then\ throw\ else}\ \ldots\text{lines 4-9}\ldots\ \}\quad \mathbf{catch}\ \{\ J\ \} \qquad \longrightarrow \qquad \{\!\!\{\ J\ \}\!\!\}$$

where $J = \mathtt{Seller} \to \mathtt{Buyer} : s\langle \mathtt{reject}, \mathit{reason}, x\rangle\,\mathbf{.}\,I$. In the other case, when Buyer wishes to accept the quote ($y < 100$). we will have:

$$\begin{array}{lll} (\nu s)\ (\nu t)\ \ \mathbf{try}\ (s)\ \{ & & \\ \qquad\quad \begin{array}{l} \mathbf{try}\ (t,s)\ \{ \\ \quad \mathbf{if}\ (y < 100)\ \mathbf{then\ throw} \\ \quad \mathbf{else}\ \{\ldots\text{as line 4}\ldots\} \\ \}\ \mathbf{catch}\ \{\ldots\text{as lines 8-9}\ldots\} \end{array} & \longrightarrow & \{\!\!\{\ \{\ldots\text{as lines 8-9}\ldots\}\ \}\!\!\} \\ \qquad\ \ \}\ \mathbf{catch}\ \{\ J\ \} & & \end{array}$$

where we have applied the rule for removing the most external try-catch block.

Note that assumption (1) (together with the types defined below) plays a key role in the semantics when starting from non-run-time choreographies. In fact, this guarantees that no multiple try-catch blocks for the same session are in parallel, in an outer block.

Types. The type discipline for the calculus is based on session types as in [3, 4]. Writing θ for first-order value types, the grammar of types is as follows:

$$\alpha \ ::= \ \downarrow(\theta)\,\mathbf{.}\,\alpha \ \mid\ \uparrow(\theta)\,\mathbf{.}\,\alpha \ \mid\ \oplus\{l_i : \alpha_i\}_{i\in I} \ \mid\ \&\{l_i : \alpha_i\}_{i\in I} \ \mid\ \alpha\{\!\!\{\beta\}\!\!\} \ \mid\ \mathsf{end} \ \mid\ \mathbf{rec}\ \mathbf{t}\,\mathbf{.}\,\alpha \ \mid\ \mathbf{t}$$

α and θ are respectively called *session types* and *service types*. The grammar follows the standard session types [6], except for *try-catch type* $\alpha\{\!\!\{\beta\}\!\!\}$ [5], the abstraction of a try-catch block: in $\alpha\{\!\!\{\beta\}\!\!\}$, α denotes the type of a service in the default choreography while β the type in the exception handler.

As an example, the types of services **chBroker** and **chSeller** are $\mathbf{rec}\ t.\uparrow(\mathtt{int})\,\mathbf{.}\,\mathtt{t}\{\!\!\{\downarrow(\mathtt{int})\,\mathbf{.}\uparrow(\mathtt{int})\}\!\!\}$ and $(\downarrow(\mathtt{int})\,\mathbf{.}\,\mathbf{rec}\ \mathtt{t}.\uparrow(\mathtt{int})\,\mathbf{.}\,\mathtt{t})\{\!\!\{\oplus\{\mathtt{conf} :\downarrow(\mathtt{int})\,\mathbf{.}\uparrow(\mathtt{int}), \mathtt{reject} : \alpha\}\}\!\!\}$ respectively where α depends on the behaviour in I.

End-Point Projection. We can now conclude our discussion by translating a global description into its end-point counterpart (EPP). In the financial protocol, we would have the following behaviour for Buyer and Broker (we omit Seller) in pseudo-code[1]:

$$\begin{array}{ll} \overline{\mathtt{chBroker}}(s)[\ s, & \mathbf{out}(s)(\mathtt{id})\,\mathbf{.}\,\mathbf{rec}\ X\,\mathbf{.}\,\mathbf{in}(s)(y)\,\mathbf{.}\,\mathbf{if}\ \mathtt{ok}(y)\ \mathbf{throw\ else}\ X, \\ & \mathtt{conf} : \mathbf{in}(s)(x)\,\mathbf{.}\,\mathbf{out}(s)(\mathtt{credit})\ +\ \mathtt{reject} : \mathbf{in}(s)(x)\,\mathbf{.}\,P\ ] \end{array}$$

$$\begin{array}{ll} *\mathtt{chBroker}(s)[\ s, & \mathbf{in}(s)(x)\,\mathbf{.}\,\mathbf{if}\ \mathtt{bad}(x)\ \mathbf{then\ throw\ else} \\ & \overline{\mathtt{chSeller}}(t)[\ (t,s), \\ & \quad \ldots\text{fwd of quotes from } t \text{ to } s \text{ (inner default)}\ldots\ , \\ & \quad \mathbf{select}(s)(\mathtt{conf})\,\mathbf{.}\ldots\text{fwd between } t \text{ and } s \text{ (inner handler)}\ldots\ ], \\ & \mathbf{select}(s)(\mathtt{reject}) : P\quad] \end{array}$$

where $\mathbf{in}(s)(x)$ denotes an input, $\mathbf{out}(s)(e)$ an output and $\mathbf{select}(s)(l)$ a branch selection. Initialisation at end-point level is split into two dual operations (as in standard sessions [6]) but it is enriched with a default process and an exception handler. Note that the two throws in the choreography have been made local to Buyer and Broker. This follows the general idea of connectedness (like EPP of if-then-else [4]), where the last active participant will be the one executing the new action (like throw or guard evaluation).

[1] The formal semantics for an end-point calculus with exceptions is reported in [5] and it guarantees that propagation (e.g. from t to s in the example) is sound.

In general, the process of EPP can be tricky, because we can easily produce a global description which does not correspond to a realisable local counterpart. The descriptive principles of *Connectedness* (a basic local causality principle), *Well-threadedness* (a stronger locality principle based on session types) and *Coherence* (a consistency principle for description of each participant) guarantee a sound EPP [4].

3 Conclusions and Related Work.

We have introduced the notion of exception for choreography. In particular, we have extended the syntax of the global calculus [3, 4] with the exception mechanism and given its formal semantics. The aim of this note was to show with simple but practical examples how exceptions can be used at choreography level and how they can be mapped to end-points. In the global calculus, exceptions are a simple form of transferring execution to a different choreography. But, together with a sound end-point projection, choreography becomes a powerful tool for designing end-point behaviour where the raising of an exception will transfer the execution of all end-points to an exception handling interaction.

We believe that obtaining end-point behaviour from choreography with exceptions through end-point projection will ensure/relax many of those conditions required for sound exception propagation in [5] (e.g. liveness comes automatically), though details are to be seen. Also, principles required for making end-point projection sound such as connectedness, well-threadedness and coherence [4] should be easily extendable to the global calculus with exceptions. Another future subject is to implement this framework for languages for web services and business protocols such as WS-CDL [10].

The notions of choreography and end-point projection have already been formalised in literature (e.g. [2, 4]). Exceptions for concurrent systems have already been studied in [1, 7–9] but they are purely local to processes (end-points). To the best of our knowledge, none of these works discuss exceptions for choreography nor exceptions for interactions.

References

[1] L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *FMOODS'03*, LNCS, pages 124–138. Springer, 2003.

[2] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *Coodination*, volume 4038 of *LNCS*, pages 63–81, 2006.

[3] M. Carbone, K. Honda, and N. Yoshida. A calculus of global interaction based on session types. In *2nd Workshop on Developments in Computational Models (DCM)*, ENTCS, 2006.

[4] M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[5] M. Carbone, K. Honda, and N. Yoshida. Structured Interactional Exception in Session Types. Submitted for publication. Available at `http://www.dcs.qmul.ac.uk/~carbonem/exception`, Mar 2008.

[6] K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In C. Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.

[7] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *ESOP '07*, pages 33–47, 2007.

[8] J. Misra and W. Cook. ORC - an orchestration language. `www.cs.utexas.edu/users/wcook/projects/orc/`, 2007.

[9] H. Vieira, L. Caires, and J. Seco. The conversation calculus: A model of service oriented computation. In *ESOP'08*, LNCS. Springer, 2008.

[10] Web Services Choreography Working Group. `http://www.w3.org/2002/ws/chor/`.

Compiling the π-calculus into a Multithreaded Typed Assembly Language

Tiago Cogumbreiro[*] Francisco Martins[*]
Vasco T. Vasconcelos[*]

Introduction. Current trends in hardware made available multi-core CPU systems to ordinary users, challenging researchers to devise new techniques to bring software into the multi-core world. However, shaping software for multi-cores is more envolving than simply balancing workload among cores. In a near future (in less than a decade) Intel prepares to manufacture and ship 80-core processors [6]; programmers must perform a paradigm shift from sequential to concurrent programming and produce software adapted for multi-core platforms.

In the last decade, proposals have been made to compile formal concurrent and functional languages, notably the π-calculus [14], typed concurrent objects [9], and the λ-calculus [13], into assembly languages. The last work goes a step further and presents a series of type-preserving compilation steps leading from System F [5] to a typed assembly language. Nevertheless, all theses works are targeted at sequential architectures.

This paper proposes a type-preserving translation from the π-calculus into MIL, a multithreaded typed assembly language for multi-core/multi-processor architectures [16]. We start from a simple asynchronous typed version of the π-calculus [1, 8, 12] and translate it into MIL code that is then linked to a run-time library (written in MIL) that provides support for implementation of the π-calculus primitives (*e.g.*, queuing messages and processes). In short, we implement a message-passing paradigm in a shared-memory architecture.

Source language. Our starting point is a simple, typed, monadic, asynchronous π-calculus, equipped with integer values, generated by the below grammar.

$$\begin{aligned} P, Q &::= \mathbf{0} \mid \overline{x}\langle v\rangle \mid x(y).P \mid P \mid Q \mid (\nu\, x\colon T)\, P \mid\, !x(y).P \\ v &::= x \mid \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \\ T &::= \mathsf{int} \mid (T) \end{aligned}$$

Processes, P and Q, comprise the inactive process $\mathbf{0}$; the output process $\overline{x}\langle v\rangle$ that sends datum v on channel x; the input process $x(y).P$ that receives a value via channel x and proceeds as P, binding variable y to the received value in process P; the parallel composition process running concurrently $P \mid Q$; the restriction process $(\nu\, x\colon T)\, P$ that creates a new channel definition local to process P; and, finally, the replicated process $!x(y).P$ that represents an

[*] University of Lisbon, Faculty of Sciences, Department of Informatics.

infinite number of active processes $x(y).P$ running in parallel. Identifiers x and y are taken from a denumerable set of names. Symbol T is the type of a value: *int* represents integer values and (T) denotes a channel type carrying values of type T. The operational semantics and the type system for the π-calculus are the standard and can be easily found in the literature [15].

Target language. The target language is a small multithreaded typed assembly language generated by the grammar depicted below, parametric on the number of registers R and on the number of processors [16]. Another possible choice for the target language is CMAP [3] which, unlike MIL, is targeted at mono-processors.

$$
\begin{array}{lrcl}
\textit{registers} & r & ::= & \mathsf{r}_1 \mid \dots \mid \mathsf{r}_\mathsf{R} \\
\textit{integer values} & n & ::= & \dots \mid \text{-1} \mid 0 \mid 1 \mid \dots \\
\textit{lock values} & b & ::= & \textbf{-1} \mid \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \dots \\
\textit{values} & v & ::= & r \mid n \mid b \mid l \mid \mathsf{pack}\ \tau, v\ \mathsf{as}\ \tau \mid \mathsf{packL}\ \alpha, v\ \mathsf{as}\ \tau \mid \\
 & & & v[\tau] \mid ?\tau \\
\textit{instructions} & \iota & ::= & r := v \mid r := r + v \mid \mathsf{if}\ r = v\ \mathsf{jump}\ v \mid \\
 & & & r := \mathsf{malloc}\ [\vec{\tau}]\ \mathsf{guarded\ by}\ \alpha \mid r := v[n] \mid r[n] := v \mid \\
 & & & \alpha, r := \mathsf{newLock}\ b \mid \alpha := \mathsf{newLockLinear} \\
 & & & r := \mathsf{tslE}\ v \mid r := \mathsf{tslS}\ v \mid \mathsf{unlockE}\ v \mid \mathsf{unlockS}\ v \mid \\
 & & & \alpha, r := \mathsf{unpack}\ v \mid \mathsf{fork}\ v \\
\textit{basic blocks} & I & ::= & \iota; I \mid \mathsf{jump}\ v \mid \mathsf{yield} \\
\textit{heaps} & H & ::= & \{l_1 \colon h_1, \dots, l_n \colon h_n\} \\
\textit{heap values} & h & ::= & \langle v_1 \dots v_n \rangle^\alpha \mid \tau\{I\}
\end{array}
$$

New to typed assembly languages is the inclusion of threads and the use of locks to discipline shared-memory access by multiple threads running in parallel. Threads are started using the fork instruction and execute until the processor is (voluntarily) released with a yield instruction. Lock value **0** designates an open lock; values **1**, **2**, ..., **n** denote a lock held by n threads with reading capabilities (shared reading); and value **-1** indicates a lock held exclusively. Lock manipulation is performed by newLock, newLockLinear for lock creation, tslE, tslS for lock acquisition, and unlockE, unlockS for lock disposal (suffix E for exclusive, S for shared).

The syntax for types is generated by the following grammar.

$$
\begin{array}{lrcl}
\textit{types} & \tau & ::= & \mathsf{int} \mid \langle \vec{\sigma} \rangle^\alpha \mid \forall[\vec{\alpha}].(\Gamma\ \mathsf{requires}\ \Lambda) \mid \mathsf{lock}(\alpha) \mid \\
 & & & \mathsf{lockE}(\alpha) \mid \mathsf{lockS}(\alpha) \mid \exists \alpha.\tau \mid \exists^{\mathsf{L}} \alpha.\tau \mid \mu\alpha.\tau \mid \alpha \\
\textit{init types} & \sigma & ::= & \tau \mid ?\tau \\
\textit{lock permissions} & \Lambda & ::= & (\vec{\alpha}; \vec{\alpha}; \vec{\alpha}) \\
\textit{register file types} & \Gamma & ::= & \mathsf{r}_1 \colon \tau_1, \dots, \mathsf{r}_n \colon \tau_n
\end{array}
$$

To control concurrent memory access we protect every tuple $\langle v_1 \dots v_n \rangle$ with a lock α (yielding a value $\langle v_1 \dots v_n \rangle^\alpha$ of type $\langle \sigma_1 \dots \sigma_n \rangle^\alpha$) and enforce statically, by means of a type system, that a thread acquires the locks for the tuples it eventually reads or writes, and that all locks are released before the thread yields the processor. We follow [4] and represent locks by singleton types ($\mathsf{lock}(\alpha)$, where α is a type variable), splitting locks usage into shared, $\mathsf{lockS}(\alpha)$, and exclusive, $\mathsf{lockE}(\alpha)$, thus allowing us to implement multiple readers for the same heap tuple. (Notice that we require every single tuple to be protected by a lock.) Code blocks are of type $\forall[\vec{\alpha}].(\Gamma \ \mathsf{requires}\ \Lambda)$, where the universal abstraction $\forall[\vec{\alpha}]$ affects the register file Γ and the triple of required permissions Λ, which corresponds, respectively, to the required exclusive, shared, and linear locks. The type system enforces that well-typed programs are free from race conditions. Details can be found in references [2, 16].

The π run-time library. To implement asynchronous π-calculus channels in MIL we use *channel queues* to store messages and input processes waiting for reduction. We follow the design of Lopes of et al. [10]. A channel queue comprises a state (empty, with messages, or with processes), a queue for messages to be delivered, and a queue for processes waiting for a message. When enqueuing messages we simply record its argument value. For (replicated) input processes we record a flag indicating whether the process is to be kept in the channel after reduction, and its *closure* that stores the set of variables known by the process—the *environment*—and a pointer to its basic block—the *continuation*. The following definition makes precise the notion of a channel queue.

$$\mathsf{ChannelQueue}(\tau,\beta) \stackrel{\text{def}}{=} \langle \mathsf{State}, \mathsf{Queue}(\tau,\beta), \mathsf{Queue}(\mathsf{Proc}(\tau,\beta),\beta)\rangle^{\beta}$$

Channel queue types are parametric on the type of the messages τ and on the lock β that enforces mutual exclusion on channel queue operations.

A *channel* packs together a channel queue and a lock specific for each channel (*i.e.*, we implement channels as monitors [7]). Notice the usage of the existential quantifier over locks $\exists^{\mathrm{L}}$ in the channel type.

$$\mathsf{Channel}(\tau) \stackrel{\text{def}}{=} \exists^{\mathrm{L}}\beta.\langle \mathsf{ChannelQueue}(\tau,\beta), \langle \mathsf{lock}(\beta)\rangle^{\beta}\rangle^{\gamma}$$

Channel types are parametric on the type of the messages τ. Henceforth, we use a *global* shared lock γ to protect $\mathsf{Channel}$s, and local shared locks α to protect the different environments. Notice that the run-time API encapsulates lock usage, keeping API clients from manipulating locks directly.

The API makes available two operations for sending and receiving messages through channels. Operation send transmits a message (in register r_1 of type τ) through a channel (in register r_4). In case there are no receptors waiting on the channel queue, the message is enqueued. The send code block signature is as follows:

$$\mathsf{send}\ \forall[\tau](r_1 : \tau,\ r_4 : \mathsf{Channel}(\tau))$$

Operation $\mathsf{receive}$ places a process, described by its closure (continuation in register r_1, environment in register r_2, and environment's lock α in register r_3), in a channel (in register r_4). Register r_6 contains a flag indicating whether the input process is replicated. As for the case of message sending, input processes are blocked when there is no pending message for the channel. The code block signature follows.

$$\begin{aligned}\mathsf{receive}\ \forall[\alpha,\tau_m,\tau_e](&r_1 : \mathsf{Cont}(\tau_m,\tau_e),\ r_2 : \tau_e,\ r_3 : \langle \mathsf{lock}(\alpha)\rangle^{\alpha},\\ &r_4 : \mathsf{Channel}(\tau),\ r_6 : \mathsf{int})\end{aligned}$$

Compiling π into MIL. The translation from the π-calculus into MIL comprises the translation of types $\mathcal{T}[\![\cdot]\!]$, of values $\mathcal{V}[\![\cdot]\!](\vec{x}, r)$, and of processes $\mathcal{P}[\![\cdot]\!](\vec{x}, l, \Gamma)$. The translation of types is straightforward.

$$\mathcal{T}[\![int]\!] \stackrel{\text{def}}{=} \mathsf{int} \qquad\qquad \mathcal{T}[\![(T)]\!] \stackrel{\text{def}}{=} \mathsf{Channel}(\mathcal{T}[\![T]\!])$$

For simplicity we create a new environment whenever a new name is defined. The motivation is twofold. First, immutable environments may be shared without contention between multiple threads. Second, processors may optimise the execution through caching, taking advantage of locality. The type of environments may be defined parametrically as follows:

$$\mathsf{Env}(x_1 \dots x_n, \Gamma, \alpha) \stackrel{\text{def}}{=} \langle \mathcal{T}[\![\Gamma(x_1)]\!], \dots, \mathcal{T}[\![\Gamma(x_n)]\!]\rangle^{\alpha}$$

The translation of values follows, where r is the register holding the values $x_1 \dots x_n$ in the environment.

$$\mathcal{V}[\![v]\!](x_1 \dots x_i \dots x_n, r) \stackrel{\text{def}}{=} \begin{cases} v & \text{if } v \text{ is an integer value} \\ r[i] & \text{if } v = x_i \end{cases}$$

We are now ready to define the translation of processes, a procedure simplified by the existence of the API. All processes have access to the environment $\vec{x}$ in register r_2 and to the lock of the environment in register r_3. The code block requires the permission to manipulate tuples protected by shared lock α.

$$\mathsf{ProcBlock}(\Gamma, \vec{x}) \stackrel{\text{def}}{=} \forall[\alpha].((r_2 \colon \mathsf{Env}(\vec{x}, \Gamma, \alpha),\ r_3 \colon \langle \mathsf{lock}(\alpha)\rangle^{\alpha})\ \mathsf{requires}\ (;\alpha;))$$

The translation of processes is parametric on the names $\vec{x}$ representing the environment. The translation of output and of parallel processes is as follows.

```
$\mathcal{P}[\![\overline{x_i}\langle v\rangle]\!](\vec{x}, l, \Gamma) \stackrel{\text{def}}{=}$
$l$ ProcBlock$(\Gamma, \vec{x})$ {
    $r_1 := \mathcal{V}[\![v]\!](\vec{x}, r_2)$
    $r_4 := r_2[i]$
    unlockS $r_3$
    jump send$[\tau]$
} where $\tau = \mathcal{T}[\![\Gamma(v)]\!]$
```

```
$\mathcal{P}[\![P \mid Q]\!](\vec{x}, l, \Gamma) \stackrel{\text{def}}{=}$
$l$ ProcBlock$(\Gamma, \vec{x})$ {
    fork $l_1[\alpha]$
    $r_4 := l_2$
    fork grabLock$[\tau_e][\alpha]$
    yield
} where $\tau_e = \mathsf{Env}(\vec{x}, \Gamma, \alpha)$
$\mathcal{P}[\![P]\!](\vec{x}, l_1, \Gamma)$
$\mathcal{P}[\![Q]\!](\vec{x}, l_2, \Gamma)$
$l_1$ and $l_2$ are fresh labels
```

For the output process, we prepare the registers as expected by code block send, by moving message v into register r_1 and fetching the channel from the environment into register r_4. When translating the parallel process, we fork the execution of the process on the left, and, because we loose the permission α to use the environment, we have to acquire it before continue executing the process on the right. Finally, we show the translation of the input process.

```
$\mathcal{P}[\![x_i(y).P]\!](\vec{x}, l, \Gamma) \overset{\text{def}}{=}$
$l$ $\mathsf{ProcBlock}(\Gamma, \vec{x})$ {
    $r_4 := r_2[i]$
    $\mathsf{unlockS}\ r_3$
    $r_1 := l_1$
    $r_6 := 0$
    $\mathsf{jump\ receive}[\tau_e][\tau][\alpha]$
}
```

```
$l_1$ $\mathsf{Cont}(\tau, \tau_e)$ {
    $CreateEnvironment(\vec{x}y, r_2, r_4)$
    $r_2 := r_4$
    $\mathsf{jump}\ l_2[\alpha]$
}
$\mathcal{P}[\![P]\!](\vec{x}y, l_2, \Gamma)$
where $\tau = \mathcal{T}[\![\Gamma(y)]\!]$,
        $\tau_e = \mathsf{Env}(\vec{x}, \Gamma, \alpha)$,
        $l_1$ and $l_2$ are fresh labels
```

In code block l we prepare the parameters for the receive operation, by moving the channel x_i into register r_4 and setting the continuation as l_1. In the continuation l_1, we create the environment for P and, after that, we jump to the body of the input in l_2. The interested reader may refer to the companion technical report for details [2].

Implementation. The run-time code consists of 19 code blocks and 34 type definitions, summing a total of 322 lines of MIL code. To execute the run-time, we depend on 8 registers. The type-checker and the interpreter for MIL, as well as the compiler for the π-calculus are available at the MIL website [11].

Main (expected) results. The contribution of our work is the translation of the π-calculus into MIL. In order to substantiate the translation we pursue two technical results: type-preservation, in the sense that well-typed π processes are translated into well-typed MIL programs; and operational correspondence, meaning that the semantics of the source language is preserved by the resulting MIL programs. For the former result, although we did not check all the proof details yet, we are confident that such a result holds; our experiments with the compiler have confirmed such intuition. The latter result we leave entirely for future work.

Related and future work. All related work we are aware off is geared towards sequential machines. Pict [14] translates the π-calculus into the C programming language. Lopes *et al.* [9] present a framework for compiling process calculi; contrary to MIL, the target language is untyped. Morrisett *et al.* [13] present a translation from the System F into a typed assembly language, two sequential languages.

We are currently working on a translation from the π-calculus into MIL equipped with compare and swap rather than locks. Our aim is to obtain a wait-free implementation of the π-calculus.

References

[1] G. Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, 1992.

[2] T. Cogumbreiro, F. Martins, and V. T. Vasconcelos. Compiling the π-calculus into a Multithreaded Typed Assembly Language. DI/FCUL TR 08–13, Department of Computer Science, University of Lisbon, 2008.

[3] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proceedings of ICFP '05*, pages 254–267. ACM Press, 2005.

[4] C. Flanagan and M. Abadi. Types for Safe Locking. In *Proceedings of ESOP '99*, volume 1576 of *LNCS*, pages 91–108. Springer, 1999.

[5] J.-Y. Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192, 1986.

[6] J. Held, J. Bautista, and S. Koehl. From a few cores to many: A tera-scale computing research overview. White paper, 2006.

[7] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[8] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *Proceedings of ECOOP '91*, volume 512 of *LNCS*, pages 133–147. Springer, 1991.

[9] L. Lopes, F. Silva, and V. T. Vasconcelos. A Virtual Machine for the TyCO Process Calculus. In *Proceedings of PPDP '99*, volume 1702 of *LNCS*, pages 244–260. Springer, 1999.

[10] L. Lopes, V. T. Vasconcelos, and F. Silva. Fine Grained Multithreading with Process Calculi. *IEEE Transactions on Computers*, 50(9):229–233, 2001.

[11] Multithreaded Intermediate Language. `http://gloss.di.fc.ul.pt/mil/`.

[12] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Part I/II. *Journal of Information and Computation*, 100:1–77, 1992.

[13] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programing Language and Systems*, 21(3):527–568, 1999.

[14] B. C. Pierce and D. N. Turner. Pict: A Programming Language Based on the Pi-Calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing. MIT, 2000.

[15] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

[16] V. T. Vasconcelos and F. Martins. A multithreaded typed assembly language. In *Proceedings of TV '06*, 2006.

Type-Directed Compilation for Multicore Programming

Kohei Honda
Queen Mary Univ. of London

Vasco T. Vasconcelos
University of Lisbon

Nobuko Yoshida
Imperial College London

Preamble. In this abstract we outline a general picture of our ongoing work on compilation into multicore CPUs [8, 14, 15]. Our focus is to harness the power of concurrency and asynchrony in one of the major forms of multicore CPUs based on noncoherent shared memory, using the well-known technology of type-directed compilation [13]. The key idea is to regard explicit asynchronous data transfer among local caches as a realiser of communication among processes. By typing processes with a variant of session types [9, 19], we obtain both type-safe and efficient compilation into processes distributed over multiple cores.

Concurrency at the Cores of Computing. In spite of the increasing reliance on distributed components in the Internet and the world-wide web, the basic computing paradigm for our applications had been centring on monolithic, predominantly sequential code. This fits our hardware, which is a virtually monolithic Von Neumann Machine (VNM), even though interactions with the distributed services often necessitate the use of concurrent threads inside a program.

It is only during the last decade that limiting physical parameters in VLSI manufacturing process [8, 14, 16] started to push a fundamental change in the internal environment of computing machinery, from monolithic Von Neumann architectures to concurrent ones, the so-called chip-level multiprocessing (CMP from now on), giving rise to CPUs with multiple cores. A multicore CPU is most effectively utilised by having multiple programs running concurrently, even inside a single application. Combined with the increasing reliance on distributed components through web services and sensor networks, computing is now becoming concurrent inside out.

A Machine Model for CMP. Following the standard dichotomy in parallel computer architecture [4], a multicore CPU can be based on either coherent cache (or SMP), cf. [11], or non-coherent cache (or non-cache-coherent NUMA), cf. [15]. In the former, memory coherence is maintained across multiple cores, while in the latter, sharing of data among cores is performed in non-uniform memory space. This second form is often found in multiprocessor system-on-chips (MPSoCs) for embedded systems, one of the areas where multicore CPUs are being effectively deployed centring on a flexible on-chip interconnect.

A non-uniform cache access can be realised by different methods such as cache-line locking. One basic method employs direct asynchronous data transfer, or Direct Memory Access (DMA), to an on-chip memory local to each core. A central observation underlying this approach is that trying to annihilate distance (i.e. to maintain strict coherence) is too costly, just as coherent distributed shared memory over a large number of nodes is unfeasible. Thus we regard CMP as distributed VNMs, along the lines of the LogP model [3] and PGAS [2].

Because of its efficiency and versatility, this framework is widely used in MPSoC for embedded systems, as noted already, including a major multicore chip [15]. It is a natural model when we consider CMP as a microscopic form of distributed computing, suggesting its potential scalability when the number of cores per chip increases. Further it can realise arbitrary forms of data sharing among cores, and in that sense it is general-purpose. Being efficient and general-purpose, however, this computing model is also known to be extremely hard and unsafe to program. Indeed, the very element that makes the major mode of data sharing in this model, DMA, fast and general-purpose, also makes it unwieldy and dangerous: it involves raw writes of one memory area to another, asynchronously issued and asynchronously performed, which can easily destroy the works being conducted in multiple cores.

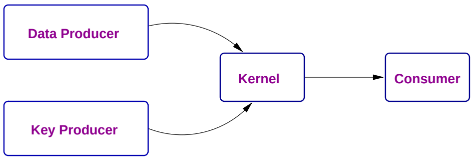
It is through the use of types for interaction, as we shall argue, that we can make the best of this computing model without losing high-level abstraction nor efficiency. For clarity, we consider an idealised model along the lines of [3], where a chip consists of multiple isomorphic VNMs, of the same ISA and each with its own memory. Data sharing is through asynchronous copy of possibly multiple words from one memory to another, or DMA. For simplicity in this abstract we do not take into consideration either the size of local memory or the maximum unit of transfer [3], and consider only a so-called "push" version of DMA (cf. [15]).

A Type-Directed Compilation Framework. One of the key features of CMP is its versatility to host a variety of applications, in size, in granularity of parallelism, and in the shape of control and data flows. Such applications may be written using domain specific languages [12, 18]. How can we translate these applications to executables for CMP? The basic idea of our approach is to stipulate *typed communicating processes* at an intermediate compilation step, and perform a *type-directed compilation* [13] onto a typed machine language for CMP. Schematically:

$$\text{DSL}\ \ (\mathsf{L2})\ \overset{\texttt{pi}}{\longmapsto}\ \text{typed processes}\ \ (\mathsf{L1})\ \overset{\texttt{asm}}{\longmapsto}\ \text{typed assembly language for CMP}\ \ (\mathsf{L0})$$

$\mathsf{L0}$, $\mathsf{L1}$, $\mathsf{L2}$ refer to abstraction levels. $\mathsf{L2}$ is a (type-safe) domain specific language, whose description is compiled into typed communicating processes ($\mathsf{L1}$, which may as well include imperative features). This is further translated into $\mathsf{L0}$, a typed low-level language for asynchronous CMP. We illustrate the key ideas of this approach using a simple example.

Streaming Example We take a simple program for stream cipher [17].



Data Producer and KeyProducer continuously send a data stream and a key stream respectively to Kernel. Kernel calculates their XOR and sends the result to Consumer. A high-level specification of such an example — specifying kernels and their connections through asynchronous streams as Kahn's networks — can be written using a DSL for streaming [12, 18], which we omit. Our purpose is to translate this program to a type-safe multicore program.

Processes with Session Types We use processes with session types [9, 19] as an intermediate language. Our motivations are two-fold. First it offers an effective *source* language for compilation into a typed assembly language for CMP, as we shall discuss soon. Secondly it offers an expressive *target* language into which we can efficiently and flexibly translate different kinds of high-level programs. Many concurrent and potentially concurrent programs (such as a streaming example above) may be represented as a collection of structured conversations, where we can abstract their structures as types for conversations.

Below we show a simple process representation of the streaming algorithm given above. The kernel initiates a session:

$$\begin{aligned}\text{Kernel} \overset{\text{def}}{=} \mathsf{def}\ \mathrm{K}(d,k,c) \ =\ & d!\langle\rangle;\ k!\langle\rangle;\ d?(x);\ k?(y);\ c?();\ c!\langle x\ \mathsf{xor}\ y\rangle;\ \mathrm{K}\langle d,k,c\rangle\\ \mathsf{in}\ \ & \overline{a}(d,k,c).\mathrm{K}\langle d,k,c\rangle\end{aligned}$$

The channels d and k are used for Kernel to receive data and keys from Data Producer and Key Producer, respectively, where Kernel notifies Data/Key Producers that it is ready before receiving data/keys (such an insertion of a notification message before the reception of datum is needed for safe translation into DMA operations, and follows a simple discipline which is statically verifiable). The channel c is used for Consumer to receive the encrypted data from Kernel, which is also used for notifying its readiness to receive the data. The keyword def denotes the recursive agent; $\overline{a}(d,k,c)$ is a session initiation which establishes the session between the three parties; $d?(x)$ is an input action at d; and $c!\langle x\ \mathsf{xor}\ y\rangle$; is an output action at c.

DataProducer can be given as follows.

$$\begin{aligned}\text{DataProducer} &\overset{\text{def}}{=} \mathsf{def}\ \mathrm{P}(d,k,c) \ =\ d?();\ d!\langle data\rangle;\mathrm{P}\langle d,k,c\rangle\ \ \mathsf{in}\ \ a(d,k,c).\mathrm{P}\langle d,k,c\rangle\\ \text{Consumer} &\overset{\text{def}}{=} \mathsf{def}\ \mathrm{C}(d,k,c) \ =\ c!\langle\rangle;\ c?(data);\mathrm{C}\langle d,k,c\rangle\ \ \mathsf{in}\ \ a(d,k,c).\mathrm{C}\langle d,k,c\rangle\end{aligned}$$

KeyProducer is identical to DataProducer except that it outputs at k rather than at d.

In all these processes, we assume that output actions of these processes are asynchronous (no blocking), and that input actions are synchronous. When these three processes are composed, messages are always consumed in the order they are produced because of the linearised usage of each channel.

The exchange of messages as above forms a "conversation" among processes, with a precise structure: this structure we abstract below as a type. The session type of the Kernel is given as:

$$T_K \quad = \quad \mu\mathbf{t}.d!\ \langle\rangle; k!\ \langle\rangle; d?\ \langle\mathsf{bool}\rangle; k?\ \langle\mathsf{bool}\rangle; c?\ \langle\rangle; c!\ \langle\mathsf{bool}\rangle; \mathbf{t}$$

Above $\mu\mathbf{t}.T$ represents a recursive type, $k?\ \langle\mathsf{bool}\rangle$ (resp. $k!\ \langle\mathsf{bool}\rangle$) denotes the input (resp. output) of a value of bool-type, and $T;T'$ denotes a sequencing. The type of the DataProducer is given as $\mathbf{t}.d?\ \langle\rangle; d!\ \langle\mathsf{bool}\rangle; \mathbf{t}$. Similarly for KeyProducer and Consumer. Safe parallel composition of communicating code is guaranteed by checking duality of types: the type of the Kernel and one of the DataProducer are dual to each other at d, so that there is no communication error occurs at d. Similarly for k and c.

Type-Directed Compilation Processes with session types are guaranteed to follow rigorous communication structures, given as types. By tracing this session type, we know beforehand what and when processs will send and receive as messages. Using this information, we can replace message passing in typed processes with direct memory write to a multicore chip.

```
main: {
  main: {
    r1 := getIdleCore
    r2 := getIdleCore
    r3 := getIdleCore
    r4 := getIdleCore
    fork dataProducer at r1
    fork keyProducer at r2
    fork kernel at r3
    fork consumer at r4
    yield
  }
}

dataProducer: {
  data: byte [128]
  ack: byte [0]
  main: {
    // produce data
    get ack
    put data in r3.data
    jump main
  }
}
```

```
keyProducer: {
  key: byte [128]
  ack: byte [0]
  main: {
    // produce key
    get ack
    put key in r3.key
    jump main
  }
}

consumer: {
  buf: byte [128]
  ack: byte [0]
  main: {
    get buff
    // consume buf
    put ack in r3.ack
    jump main
  }
}
```

```
kernel: {
  data: byte [128]
  key: byte [128]
  buf: byte [128]
  ackD: byte [0]
  ackK: byte [0]
  ackC: byte [0]
  main: {
    put ackD in r1.ack
    put ackK in r2.ack
    get data; get key
    r5 := 128; jump loop
  }
  loop: {
    when r5 < 0 jump done
    r6 := data[r4]; r7 := key[r4]
    buf[r4] := r7 xor r6;
    jump loop
  }
  done: {
    get ackS
    put sum in r1.arg
    jump main
  }
}
```

Figure 1: L0 code for the stream example

Since our purpose is to have type-safe compilation, we use a typed assembly language [13] targeted at distributed memory CMP and NoC [1,5], which we call L0 for brevity. L0 is built on top of MIL [20], which in turn is a multi-threaded extension of TAL [13]. Task scheduling is accomplished by loading a program into a core. This includes copying from the main memory the code and the data required for a run of the core, as well as a snapshot of the current register values.

Figure 1 presents one possible result of compiling our running example into L0. As we observed, all typed message passing is replaced by DMA primitives, using addresses of the variables in the local memory of a target core for remote asynchronous writes, where the addresses are shared at the time a thread is launched.

The block associated with identifier main defines a program comprising, in this case, a single basic block, also named main. The program is intended to be uploaded at some core and its execution launched. Cores terminate their execution with a special instruction **yield**, thus joining the pool of available cores. Cores requiring extra workers get hold idle cores by issuing an instruction of the form r_1:= getIdleCore. The first **fork** instruction in main.main copies program dataProducer to the core in register r_1, copies a snapshot of its registers to the target core, and launches the execution of basic block dataProducer.main. Notice that by getting first the number of required cores and then forking the threads we guarantee that each thread knows all other cores (including its own) via registers r_1 to r_4.

The program associated with identifier dataProducer defines a program comprising two buffer declarations (named data and ack) and a basic block (named main). The core running this program writes its data buffer into the kernel's data buffer, but first needs to make sure it can overwrite the latter. We do all this with L0's support for DMA operations. Instruction **get** ack blocks the core until a corresponding **put** instruction is issued, namely via instruction **put** ackD **in** r_1.ack in basic block kernel.done and the data is safely written. After **put**, the producer asynchronously writes its buffer (with **put** data **in** r_3.data), for which the kernel waits with a**get** data instruction.

Program kernel declares three buffers (two incoming, one outgoing), and another three (empty) buffers used for acknowledgements, signals the data and the key producers that the respective buffers can be written, waits for the completion of the write oper-

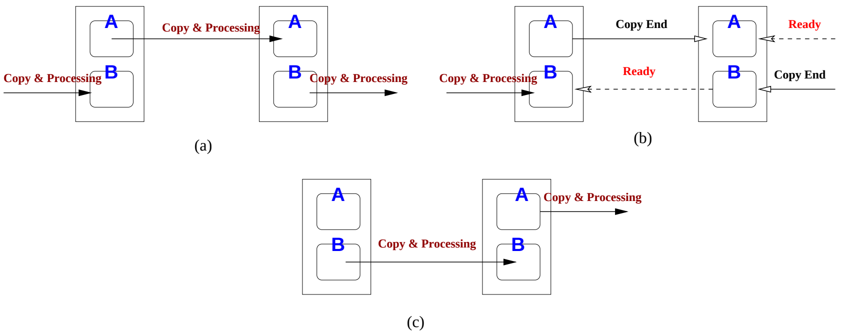


Figure 2: Double-Buffering

ations, and embarks on a loop to fill the outgoing (buf) buffer. Finally it asynchronously writes this buffer into the arg buffer at core consumer before restarting the process. Instruction **when** $\mathsf{r}_4 < 0$ **jump** done is expanded into the two instructions $\mathsf{r}_4 := \mathsf{r}_4 - 1$; **if** $\mathsf{r}_4 == 0$ **jump** done, providing for loops.

Shared Channels. The example under consideration does not use shared access to main memory. However, it is natural that a program which accepts multiple requests at a shared channel (located at main memory), receives a request, then forks a thread to one of the available cores. Generally this demands multiple clients to invoke a shared channel concurrently. In this and related schemes, a shared initial channel can be effectively realised by the combination of traditional load and store instructions together with mutual exclusion primitives (lock [20] or compare and swap) and DMAs.

Further Topics Our approach is based on a simple premise: session types offer rigorous abstraction of conversation structures, and, as far as concurrent programs can be represented as a collection of conversations, we can use their types in order to realise the same conversations through asynchronous data transfers among local memories of multiple cores, instead of message passing. Processes offer readable, transparent program structures, as well as a target of translation, and types guarantee type-safety of compiled code.

There are several topics which we could not discuss in this abstract. We however briefly touch one topic, which is important for practical implementation. The process-based representation of stream can be made more asynchronous, by transforming the protocol structure slightly. The transformation is simple. For brevity we consider three-party interactions, from a single source to the kernel to the consumer, and only present the session type of the kernel. Let s be a channel used for data-transfer with the right-hand side, while k is with the left-hand side; and "$s \lhd \mathsf{ReadyA}$;" (resp. "$s \rhd \mathsf{ReadyA}$") sends (resp. receives) a signal which tells A is empty.

$$
\begin{array}{l}
s \lhd \mathsf{ReadyA}; s \lhd \mathsf{ReadyB}; \\
\quad \mu\mathbf{t}.s?\,\langle T\rangle; k \rhd \mathsf{ReadyA}; k!\,\langle T\rangle; s \lhd \mathsf{ReadyA};\,; s?\,\langle T\rangle; k \rhd \mathsf{ReadyB}; k!\,\langle T\rangle; \mathbf{t}
\end{array}
$$

This type says: first it sends signal to s; then it gets the data into A from s; once the data transfer is completed *and* it gets the signal to tell A is free from k, then it starts transferring the data to k; similarly for B. This scheme is close to so-called double buffering technique used in multicore processors [10], as shown in Figure 2: indeed, by the same translation scheme, this conversation structure is compiled into a (type-safe) double-buffering implementation of streams, which is much more efficient than the original version due to the exploitation of asynchrony.

This alternative presentation suggests inherent flexibility in compilation and execution of concurrent programs in CMP and other extremely concurrent computing environments, opening new opportunities and challenges. For example we may need more flexibility and generality in type structures (as in the case of, for example, the typing for the process representing double buffering discussed above), new compilation and static analysis techniques, new runtime architectures, and new abstractions. Research from multiple directions (here we only refer to [2, 6, 7] among many closely related and/or complementary works) will be needed to explore this rich field of structured concurrent programming.

References

[1] L. Benini and G. D. Micheli. Networks on chip: a new SoC paradigm. *IEEE Computer*, 35:1, 2002.

[2] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[3] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, 1993.

[4] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[5] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *DAC*, pages 684–689, 2001.

[6] R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. In *ESOP*, pages 204–218, 2004.

[7] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, , and S. Levi. Language support for fast and reliable message-based communication in singularity os. In *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.

[8] P. Gelsinger, P. Gargini, G. Parker, and A. Yu. Microprocessors circa 2000. *IEEE SPectrum*, pages 43–47, 1989.

[9] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[10] IBM. ALF double buffering. `http://www.ibm.com/developerworks/blogs/page/powerarchitecture?entry=ibomb_alf_sdk30_5`.

[11] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[12] C.-K. Lin and A. P. Black. DirectFlow: A domain-specific language for information-flow systems. In *ECOOP*, volume 4609 of *LNCS*, pages 299–322. Springer, 2007.

[13] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[14] K. Olukotun, B. A. Nayfeh, L. Hammond, K. G. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS*, pages 2–11, 1996.

[15] D. Pham et al. The design and implementation of a first-generation cell processor. In *ISSCC Dig. Tech. Papers*, pages 184–185. IEEE, February 2005.

[16] F. J. Pollack. New microarchitecture challenges in the coming generations of cmos process technologies. In *MICRO*, 1999.

[17] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1993.

[18] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: high-throughput stream programming in java. In *OOPSLA*, pages 211–228. ACM, 2007.

[19] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[20] V. T. Vasconcelos and F. Martins. A multithreaded typed assembly language. In *Proceedings of TV06 - Multithreading in Hardware and Software: Formal Approaches to Design and Verification*, 2006.

Synchronization as a Special Case of Access Control

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

Abstract

Synchronization ensures exclusive access to shared variables at runtime, and unique-access control gives similar guarantees at compilation time. We propose to integrate synchronization into access control in a Java-like language: Synchronization is based on lockable shared variables accessible only in the presence of corresponding (partially unique) tokens. We get more freedom in expressing synchronization at appropriate points, and the influence of concurrency on the program structure becomes weaker.

1 Motivation

Concurrency is an important aspects of programming where we urgently need more advanced support. Independently, there is work on language concepts for describing software architectures at a higher level [1] where techniques allowing us to constrain access to objects in certain ways play an important role.

In this abstract we consider synchronization to be just a special case of a technique to ensure unique access. These aspects fit together quite naturally. However, while access control mechanisms are static we regard synchronization as inherently dynamic. To overcome this discrepancy we add a dynamic quality to access control. This approach promises to give us a number of advantages:

- Programmers think in terms of software architecture and accessibility of methods and variables, and the compiler ensures proper synchronization. Concurrency need not dominate the program structure.
- Unique access does not depend on threads. Even when using a conventional thread model the identity of threads does not play any role.
- Interfaces specify accessibility and (implicitly) synchronization. Clients get all information they need to avoid synchronization conflicts.
- There is more freedom in ensuring synchronization at appropriate places. We can safely move synchronization from servers to clients.

In one aspect the proposed concept resembles the SCOOP model of Eiffel [6] where preconditions represent synchronization conditions: Synchronization depends only on current values of variables in objects. There is no need for `wait` and `notify` as in Java because variable values can represent waiting conditions.

$!x{:}\tau$	(variable x is of type τ; exclusive access; x cannot be locked)
$x{:}\tau$	(variable x is of type τ; exclusive access; x can be locked)
$*x{:}\tau$	(variable x is of type τ; shared read-access; x can be locked))
$x{:}\sigma?\tau$	(if x of type σ, execute method with [$x{:}\sigma$->$x{:}\tau$] while x locked)

Table 1: Kinds of tokens representing knowledge about instance variable x

2 Accessibility and Synchronization

In most language concepts ensuring unique access, programmers specify quite directly that some object is accessible only through a specific reference, this is, there cannot exist aliases [3]. To support aliasing we take a different approach expressing accessibility of specific object parts: Programmers annotate

- references with tokens expressing partially unique knowledge about (otherwise usually not visible) variables of the referenced objects[1];
- methods with required tokens (this is, what clients must know before method invocation) and ensured tokens (what clients will know on return).

We distinguish between four kinds of tokens as shown in Table 1. The first two ensure unique access to variables protected by the variables x encoded in tokens (see Sect. 3) while the third one supports shared read-access. Using the last kind we must acquire a lock to get exclusive access. A literal can occur where the syntax requires a type σ or τ, the type with this literal as its only instance.

We show the use of tokens by an example in a Java-like pseudo-language:

```
class Window {
  public void iconify()[icon:true->icon:false]{..icon=false;...}
  public void uniconify()[icon:false->icon:true]{..icon=true;...}
  public void update(...)[sync:Unit->sync:Unit]{...}
  public Window(...)[->sync:Unit?Unit, !icon:false]{...}
  private boolean icon = false;
  private Unit sync = new Unit();
}
```

A client can invoke `iconify` only with unique knowledge about `icon` to be of value `true` because of a required token `icon:true` to the left of `->` in the annotation within square brackets. While executing `iconify` the value of `icon` must change to `false` according to the ensured token to the right of `->`. A new instance of `Window` gets a token `!icon:false`, and according to it we can invoke `iconify` without synchronization. Tokens can move from one reference to another as side-effects of parameter passing and assignment:

```
Window[icon:true] foo(Window[icon:false->] w)
   { if(...) {w.iconify(); return w;} else {return null;} }
```

On invocation of `foo` a token `icon:false` moves from the argument to the parameter `w`, and on return `icon:true` moves from `w` or `null` to the result of `foo`. We assume `null` to be associated with every token since no methods are

[1] Programmers annotate only formal parameters and results of methods. A compiler can infer corresponding annotations of local variables, instance variables, and class variables.

$$a \equiv a \qquad \frac{a \equiv b \quad b \equiv c}{a \equiv c} \qquad \frac{a \equiv b}{b \equiv a} \qquad \frac{a \equiv a' \quad b \equiv b'}{a, b \equiv a', b'}$$

$$a, b \equiv b, a \qquad a, (b, c) \equiv (a, b), c \qquad a \equiv a, \epsilon$$

$$*x{:}\tau \equiv *x{:}\tau, *x{:}\tau \qquad x{:}\sigma?\tau \equiv x{:}\sigma?\tau, x{:}\sigma?\tau \qquad x{:}\sigma?\tau \equiv x{:}\sigma?\sigma, x{:}\sigma?\tau$$

$$\frac{a \equiv b}{a \leqq b} \qquad \frac{a \leqq b \quad b \leqq c}{a \leqq c} \qquad \frac{a \leqq a' \quad b \leqq b'}{a, b \leqq a', b'} \qquad \frac{a \text{ optional}}{a \leqq \epsilon}$$

$$\frac{\sigma \le \tau}{!x{:}\sigma \leqq !x{:}\tau} \qquad \frac{\sigma \le \tau}{x{:}\sigma \leqq x{:}\tau} \qquad \frac{\sigma \le \tau}{*x{:}\sigma \leqq *x{:}\tau} \qquad \frac{\sigma \le \sigma' \quad \tau' \le \sigma'}{x{:}\sigma?\tau', x{:}\tau'?\tau \leqq x{:}\sigma'?\tau}$$

$$!x{:}\tau \leqq x{:}\tau \qquad !x{:}\tau \leqq x{:}\tau?\tau \qquad x{:}\tau \leqq *x{:}\tau \qquad x{:}\sigma?\tau \leqq x{:}\sigma?\tau', x{:}\tau'?\tau$$

Table 2: Equivalence $\equiv$ and subsumption $\leqq$ of token sequences

invokable through `null`. When executing `w.iconify()` the token `icon:false` moves from `w` to `this` of the window referenced by `w`, and then `icon:true` moves from `this` to `w`. The value of `icon` can be modified only if `this` has a corresponding unique token, and as a side-effect this token is modified, too.

Table 2 shows an equivalence and subsumption relation on comma-separated token lists. Tokens a can be used where tokens b are expected if $a \leqq b$ holds. For example, we can use `!icon:false` where `icon:false` is expected. Tokens of the forms $*x{:}\tau$ and $x{:}\sigma?\tau$ can be duplicated while the compiler prevents duplication of unique tokens of the forms $!x{:}\tau$ and $x{:}\tau$.

Tokens of the form $x{:}\sigma?\tau$ build the basis for synchronization. Several clients can have the same token. Through a reference annotated with $x{:}\sigma?\tau$ we invoke a method requiring $x{:}\sigma$ and ensuring $x{:}\tau$. However, execution is delayed until variable x has a value of type σ and is not locked; then, during execution there is an exclusive write-lock on x, and the token $x{:}\sigma$ of `this` must be modified to $x{:}\tau$ as side-effect of an assignment to x if τ differs from σ. When invoking a method requiring $*x{:}\sigma$ it is sufficient to acquire a shared read-lock. Table 3 shows essential parts of type checking rules for invocations to ensure that clients have all required tokens and to infer locks to be acquired: In $a \xrightarrow{b \to b'} a'/l$ the list a contains available tokens and a' tokens remaining after invocation, b are the required and b' the ensured tokens, and l is the list of needed locks.

In our example, the constructor ensures `sync:Unit?Unit` (where `Unit` need not support any method). Several references to the same window can be annotated with this token. When invoking `update` we get mutual exclusion as with synchronized methods in Java, but there is an important difference: While Java's monitor concept allows the thread holding a lock on an object to invoke further methods of this object, locks in our approach do not belong to threads, and in the body of `update` (and all methods invoked by `update`) we must not use `sync:Unit?Unit` again to avoid deadlocks. Instead, we make use of `sync:Unit` associated with `this` in the body of `update`, this is, we apply access control instead of synchronization. We get more safety and possibly more efficiency at the cost of more verbose annotations needed to ensure unique access.

Using tokens of the form $!x{:}\tau$ (instead of $x{:}\tau$) clients can decide if they prefer access control or synchronization. According to Table 2 we can replace `!icon:true` with `icon:true?false` and `icon:false?true` allowing two clients

$$a \xrightarrow{\epsilon\rightarrow\epsilon} a/\epsilon \qquad \frac{a \xrightarrow{b\rightarrow b'} a'/l}{c, a \xrightarrow{c,b\rightarrow b'} a'/l} \qquad \frac{a \xrightarrow{b\rightarrow b'} a'/l}{a \xrightarrow{b\rightarrow c,b'} c, a'/l}$$

$$\frac{a \xrightarrow{b\rightarrow b'} a'/l}{x{:}\sigma?\tau, a \xrightarrow{x:\sigma,b\rightarrow x:\tau,b'} a'/\mathrm{wlock}(x{:}\sigma), l} \qquad \frac{a \xrightarrow{b\rightarrow b'} a'/l}{x{:}\tau?\tau, a \xrightarrow{*x:\tau,b\rightarrow *x:\tau,b'} a'/\mathrm{rlock}(x{:}\tau), l}$$

Table 3: Token checking and lock inference for method invocation (simplified)

to independently and repeatedly invoke `iconify` and `uniconify`. Synchronization is implicit by waiting until `icon` is of appropriate value. Such rules are applicable only if the types in ?-tokens build cycles, this is, method invocations cause the variable again and again to get values of each type in the cycle. This concept presumes clients to repeatedly invoke methods corresponding to all ?-tokens in their scopes. Hence, ?-tokens must not get lost. Token lists not containing ?-tokens are *optional*, and a rule in Table 2 allows us to remove them.

To break cycles built by ?-tokens we put the corresponding variable into a stop mode causing clients to get exceptions instead of a lock on the variable.

3 Static Guarantees

Static type checking gives the following guarantees:

- Always at most one thread can write to a variable y. All methods writing to a shared variable y must require a token $!x{:}\tau$ or $x{:}\tau$ – all tokens with the same x and with arbitrary type τ. The instance or class variable x protects y. No two tokens of the form $!x{:}\tau$ or $x{:}\tau$ must exist simultaneously with the same x belonging to the same object. We ensure uniqueness by avoiding token duplication wherever aliases may be introduced.

- A variable y is readable only while no other thread can simultaneously write to y. All methods possibly reading from an instance or class variable y must require a token $!x{:}\tau$, $x{:}\tau$, or $*x{:}\tau$ where variable x protects y.

- Synchronization is continuous: There are sequences of method invocations causing all invoked methods to become executable (if all methods terminate in finite time). We require cycles on the types in tokens of the form $x{:}\sigma?\tau$ and avoid loss of such tokens. Cycles can occur only if methods changing tokens according to each step in the cycles actually exist. Tokens in annotations of class and instance variables easily get lost. Such variables must not be annotated with ?-tokens.

- As a simple approach to ensure deadlock freeness we create a global ordering of variable names and check for all tokens used as required tokens or associated with formal parameters (to the left of `->`) in a method if each x in tokens of the form $x{:}\sigma?\tau$ precedes each y in tokens of the form $y{:}\tau$ or $*y{:}\tau$. We can get false positives because of missing static information and propose to issue a warning instead of an error message if a check fails.

- Subtyping considers tokens and, therefore, synchronization and access control according to the principle of substitutability.

Annotations of variables (but not parameters) can depend on current values of variables in tokens. Such annotations are lengthy and instable when expressed explicitly. Fortunately, a compiler can infer them: It collects information about available as well as required tokens from specifications of invoked methods. All tokens available for a variable at the end of a method are regarded as annotations of the variable depending on the ensured tokens of the method. These tokens are assumed as available at the begin of methods requiring corresponding tokens.

4 Discussion and Related Work

In the proposed approach, concurrency is based on threads, locks, and values of shared variables. In this respect there are similarities with the SCOOP model [6]. Both models disclose information on the variables used for synchronization. The way how and the time when such information becomes available to clients is different: Every client can get access to such variables at runtime in the SCOOP model while access control in our model causes much information to be available at compilation time and usually only few clients can access the variables. The access control mechanism adds a new dimension to concurrent programming and reduces the importance of dynamic synchronization.

There are many approaches to ensure unique access [4], most of them by avoiding aliases. The token-based approach used in this work was developed from process types [7], an object-oriented variant of linear types. This concept restricts the way how objects can be accessed. Tokens express all information clients need to provide required and avoid conflicting synchronization [8].

In the author's previous work, tokens always have been separate entities without relationship to variables. It is a new contribution to regard tokens as static abstractions of concrete variables that can be locked. The kinds of tokens in Table 1 were developed as a consequence. They turned out to be useful for simple programs. Some other kinds of tokens were rejected because they either were not consistent with more important tokens or turned out to be less useful.

This work has the following goals: Program structures implied by concurrency shall be independent from those implied by object-oriented principles, and program code shall be stable when refactoring concurrency. It is important that method specifications do not distinguish between access control and synchronization. We can invoke a method in many different contexts. The code is stable because most likely we need not change the method if we replace synchronization with access control or move the point of synchronization.

By locking variables instead of objects we get finer granularity for synchronization and keep concurrency independent from object-oriented factorization.

We can create new threads as in Java or as side-effect of invoking asynchronous methods as in Polyphonic C# [2]. In contrast to those in Java, threads in our approach need not have any identity: The thread that acquired a lock is in no way privileged compared to other threads because only the (statically checked) availability of tokens counts, not the thread identity.

The proposed approach exposes more information to clients than necessary: Synchronization used only to ensure simple mutual exclusion (as expressed by `sync:Unit?Unit`) need not be visible and can be added later on without changing interfaces [8]. We can avoid exposure of such information (and thereby soften dependences between concurrency and object factorization) by making tokens implicit through special cases in our mechanism. However, even implicit tokens can suffer from deadlocks and become visible in corresponding warnings.

There are approaches to concurrent object-oriented programming that avoid low-level locking and still ensure atomicity [5]. It is an open question how to combine such techniques with access control.

5 Conclusions

Synchronization and access control ensuring unique access to shared variables fit together quite naturally. We explored an approach to integrate synchronization resembling that in the SCOOP model into a token-based access control mechanism. It turned out to be possible and beneficial to annotate methods with access information not distinguishing between static access control and dynamic synchronization. Static type checking ensures unique access to shared variables as well as continuity of synchronization.

References

[1] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, Orlando, Florida, May 2002. ACM.

[2] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):269–804, September 2004.

[3] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In Erik Ernst, editor, *ECOOP – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, Berlin, Germany, July 2007. Springer-Verlag.

[4] Sophia Drossopoulou, David Clarke, and James Noble. Types for hierarchic shapes. In *ESOP*, pages 1–6, 2006.

[5] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA'03*, pages 388–402, Anaheim, California, USA, October 2003. ACM.

[6] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition edition, 1997.

[7] Franz Puntigam. Coordination requirements expressed in types for active objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 367–388, Jyväskylä, Finland, June 1997. Springer-Verlag.

[8] Franz Puntigam. Internal and external token-based synchronization in object-oriented languages. In *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006*, volume 4228 of *Lecture Notes in Computer Science*, pages 251–270, Oxford, UK, September 2006. Springer-Verlag.

Towards a Symbolic Semantics for Service-oriented Applications*

Rosario Pugliese Francesco Tiezzi Nobuko Yoshida

Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze
{pugliese,tiezzi}@dsi.unifi.it

Department of Computing
Imperial College London
yoshida@doc.ic.ac.uk

Abstract

We introduce a symbolic characterisation of the operational semantics of COWS, a formal language for specifying and combining service-oriented applications, while modelling their dynamic behaviour. This alternative semantics avoids infinite representations of COWS terms due to the value-passing nature of communication. Finite representations can pave the way for the development of efficient analytical tools, such as e.g. behavioural equivalences and model checkers. We illustrate our approach through a 'translation (web) service' scenario.

1 Introduction

The recent success of e-business, e-learning, e-government, and other similar emerging models, has led the World Wide Web, initially thought of as a system for human use, to evolve towards an architecture for *service-oriented computing* (SOC) supporting automated use. SOC advocates the use of loosely coupled 'services', to be understood as autonomous, platform-independent, computational entities that can be described, published, discovered, and assembled, as the basic blocks for building interoperable and evolvable applications. The most successful instantiation of the SOC paradigm are probably the *web services*, that are sets of operations that can be invoked through the Web via XML messages complying with given standard formats. To support the web service approach, several new languages and technologies have been designed and many international companies have invested a lot of efforts.

Current software engineering technologies for SOC, however, remain at the descriptive level and lack rigorous formal foundations. We are still experiencing a gap between practice (programming) and theory (formal methods and analysis techniques) in the design of SOC applications. The challenges come from the necessity of dealing at once with issues like communication, co-operation, resource usage, security, failures, etc. in a setting where demands and guarantees can be very different for the many different components. Many researchers have hence put forward the idea of using *process calculi* that, due to their algebraic nature, convey in a distilled form the compositional programming style of SOC. Thus, many process calculi have been designed (see e.g. [3, 2, 8, 6, 1, 4]), addressing one aspect or another of SOC and aiming at assessing the adequacy of diverse sets of primitives w.r.t. modelling, combining and analysing SOC applications.

*This work has been partially supported by the EU project SENSORIA, IST-2005-016004.

By taking inspiration from well-known process calculi and from the standard language for orchestration of web services WS-BPEL [12], in [10] we have designed COWS (*Calculus for Orchestration of Web Services*), a process calculus for specifying and combining service-oriented applications, while modelling their dynamic behaviour. We have shown that COWS can model distinctive features of web services, such as, e.g., correlation-based communication, compensating activities, service instances and interactions among them.

Equivalence and model checkers, and other similar verification tools, do not work directly on syntactic specifications but rather on abstract representations of the behaviour of processes. Thus, for value-passing languages, such as COWS, using an inappropriate representation can lead to unfeasible verifications. Indeed, according to the COWS's original operational semantics, if the communicable values range over an infinite value set (e.g. natural numbers, strings), the behaviour of a service that performs a receive activity is modelled by an infinite abstract representation. Such representation is a Labelled Transition System whose initial state has infinite outgoing edges, each labelled with an input label having a different value as argument and leading to a different state. Hence, by taking inspiration from Hennessy and Lin [7], we are currently defining a *symbolic* operational semantics for COWS. The new semantics associates a finite representation to each COWS term and can therefore pave the way for devising efficient reasoning mechanisms and tools to analyse COWS terms.

In the rest of this abstract, we presents syntax and main features of COWS 'at work' on modelling an Italian-English translation (web) service (Section 2), and discuss verification problems and the major intuitions underlying the symbolic operational semantics for COWS (Section 3).

2 COWS: a Calculus for Orchestration of Web Services

In this section, we present COWS main features and syntax in a step-by-step fashion while modelling an Italian-English translation (web) service. Due to lack of space, here we only provide an informal account of the semantics of COWS and refer the interested reader to [10, 9] for a formal presentation, for examples illustrating peculiarities and expressiveness of the language, and for comparisons with other process-based and orchestration formalisms.

Let us consider a web service that provides to its customers an Italian-English translation service. Specifically, when the service is invoked by a customer, that communicates first her partner name and then an Italian word, it replies to the request with either the corresponding English word or the string "*unknown word*". A high-level specification of the service can be rendered in COWS as follows:

$$[x]\, t \bullet \mathit{req}?x \,.\, [y]\, t \bullet \mathit{word}?y \,.\, x \bullet \mathit{resp}!\mathit{trans}(y) \qquad (1)$$

where t is the translation service partner name, *req*, *word* and *resp* are operation names, x and y are variables that store the customer partner name and the Italian word to be translated respectively, and $\mathit{trans}(\_)$ is a total function that maps a large subset of Italian words to the corresponding English ones and returns the string "*unknown word*" for all words that do not appear in the Italian words set. The service simply performs a sequence of two *receive* activities $t \bullet \mathit{req}?x$ and $t \bullet \mathit{word}?y$, corresponding to reception of a request and of an Italian word sent by a customer, and replies with the translated word, by invoking the operation *resp* of the customer by means of the *invoke* activity $x \bullet \mathit{resp}!\mathit{trans}(y)$. Receives and invokes are the basic communication activities provided by COWS. Besides input parameters and sent values, they indicate the endpoint, i.e. a pair $p \bullet o$ made of a partner name p and an operation name o, through which the communication should occur. Differently from most process calculi, receive activities in COWS bind neither names nor variables. The only binding construct is *delimitation*: $[d]\, s$ binds the delimited object d in the scope s (the notions of bound and free occurrences of a delimited object are defined accordingly). For example, the service (1) uses the delimitation operator to declare the scope of variables x and y. An inter-service communication takes place when the arguments of a receive and of a concurrent invoke along the same endpoint do match[1], and causes replacement of the variables arguments of the receive with the corresponding values arguments of the invoke (within the scope of variables declarations). For example, variable x will be initialised by the first receive activity with data provided by a customer.

At a lower level, the service could be described in terms of three entities composed by using the *parallel composition* operator $\_ \mid \_$ that allows them to be concurrently executed and to interact with each other. Then, the COWS specification of the translation service can be

$$[\mathit{reqDB1}, \mathit{reqDB2}, \mathit{respDB1}, \mathit{respDB2}]\, (\, \mathit{Translator} \;\mid\; \mathit{DB1} \;\mid\; \mathit{DB2}\,) \qquad (2)$$

The delimitation operator is used here to declare that *reqDB1*, *reqDB2*, *respDB1* and *respDB2* are private operation names known to the three components *Translator*, *DB1* and *DB2*, and only to them (at least initially, since during a computation private names can be exported exactly as in π-calculus). The three subservices are defined as follows:

$$\begin{array}{rcl} \mathit{Translator} & \triangleq & [x]\, t \bullet \mathit{req}?x\,.\,[y]\, t \bullet \mathit{word}?y\,. \\ & & \quad [k]\,(\, t \bullet \mathit{reqDB1}!y \;\mid\; [x_1]\, t \bullet \mathit{respDB1}?x_1\,.\,(\,\mathbf{kill}(k) \mid \{\!|\, x \bullet \mathit{resp}!x_1 \,|\!\}\,) \\ & & \qquad \mid t \bullet \mathit{reqDB2}!y \;\mid\; [x_2]\, t \bullet \mathit{respDB2}?x_2\,.\,(\,\mathbf{kill}(k) \mid \{\!|\, x \bullet \mathit{resp}!x_2 \,|\!\}\,)\,) \end{array}$$

$$\begin{array}{rcl} \mathit{DB1} & \triangleq & t \bullet \mathit{reqDB1}?\text{“}a\text{”}.\, t \bullet \mathit{respDB1}!\text{“}to\text{”} \\ & & +\, t \bullet \mathit{reqDB1}?\text{“}albero\text{”}.\, t \bullet \mathit{respDB1}!\text{“}tree\text{”} \\ & & +\, \ldots\, +\, t \bullet \mathit{reqDB1}?\text{“}zucca\text{”}.\, t \bullet \mathit{respDB1}!\text{“}pumpkin\text{”} \end{array}$$

$$\begin{array}{rcl} \mathit{DB2} & \triangleq & [z]\,(\, t \bullet \mathit{reqDB2}?z.\, t \bullet \mathit{respDB2}!\text{“}unknown\ word\text{”} \\ & & \quad +\; t \bullet \mathit{reqDB2}?\text{“}a\text{”}.\, t \bullet \mathit{respDB2}!\text{“}to\text{”} \\ & & \quad +\; t \bullet \mathit{reqDB2}?\text{“}abate\text{”}.\, t \bullet \mathit{respDB2}!\text{“}abbot\text{”} \\ & & \quad +\; \ldots\, +\; t \bullet \mathit{reqDB2}?\text{“}zuppo\text{”}.\, t \bullet \mathit{respDB2}!\text{“}soaked\text{”}\,) \end{array}$$

Service *Translator* is publicly invocable and can interact with customers other than with the ‘internal’ services *DB1* and *DB2*. These latter two services, instead, can only be invoked by *Translator* (indeed, all the operations used by them are restricted) and have the task of looking up in databases the English word corresponding to a given Italian one and replying accordingly. In particular, *DB1* performs a quick search in a small database of commonly used words, while *DB2* performs a slower search in a bigger database (that exactly corresponds to that modelled by the function $\mathit{trans}(\_)$). For performance purposes, after the two initial receives, *Translator* invokes services *DB1* and *DB2* concurrently. When one of them replies, *Translator* immediately stops the other search. This is done by executing the *kill* activity $\mathbf{kill}(k)$, that forces termination of all unprotected parallel terms inside the enclosing $[k]$, that stops the killing effect. Then, *Translator* forwards the response to the customer and terminates. Kill activities

[1] The pattern-matching mechanism permits correlating messages logically forming a same interaction ‘session’ by means of their same contents.

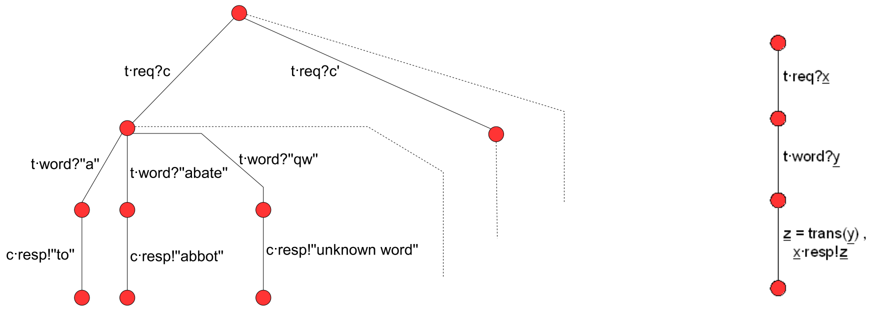


Figure 1: LTS and symbolic LTS for the translation service (high-level specification)

are executed eagerly with respect to the other parallel activities but critical code can be protected from the effect of a forced termination by using the *protection* operator $\{\!|\_|\!\}$; this is indeed the case of the response $x \bullet resp!x_1$ in our example. Services *DB1* and *DB2* use the *choice* operator $\_ + \_$ to offer alternative behaviours: one of them can be selected by executing an invoke matching the receive leading the behaviour. In case the word to be translated is unknown, *DB1* does not reply, while *DB2* returns the string "*unknown word*". Indeed, the semantics of parallel composition avoids that *DB2* returns "*unknown word*" in case of known words. This is done by assigning the receive $t \bullet reqDB2?z$ less priority than the other receive activities, so that it is only executed when none of the other receives matches the word to be translated.

3 A symbolic operational semantics for COWS

In this section, we discuss verification problems and present the major intuitions underlying the symbolic operational semantics for COWS. A more detailed presentation of the symbolic semantics can be found in [13].

Verification problems. When the considered specification language is a value-passing process algebra and the value-space is infinite, using standard Labelled Transition Systems (LTSs) for the semantics can lead to infinite representations. For example, the operational behaviour of service (1) can be represented by the infinite LTS in the left-hand side of Figure 1. Notably, for the sake of presentation, the LTSs shown in the figures rely on an operational semantics in *early* style, where substitutions are applied when receive actions are inferred. However, the problem of infinite representations remains also in case of *late* semantics, due to the fact that the continuation of a receive action with argument variables $\bar{x}$ has to be considered under all substitutions for $\bar{x}$.

The symbolic approach. To tackle the problems above, in [7] Hennessy and Lin have introduced the so-called *symbolic LTSs* and used them to define finite semantical representations of terms of the value-passing CCS. For example, the symbolic LTSs corresponding to the COWS service (1) is shown in the right-hand side of Figure 1. The symbolic actions $t \bullet req?\underline{x}$ and $t \bullet word?\underline{y}$ denote reception of generic values $\underline{x}$ and $\underline{y}$ along endpoints $t \bullet req$ and $t \bullet word$, respectively; the condition-guarded symbolic action $(\underline{z} = trans(\underline{y})\,,\ \underline{x} \bullet resp!\underline{z})$ denotes sending of a generic value $\underline{z}$ such that $\underline{z} = trans(\underline{y})$. Of course, for the same reasons, also the LTS representing the behaviour of service (2) is infinite, while the corresponding symbolic LTS is finite. Indeed, if for the sake of presentation we assume that database *DB1* contains only the association for word "*a*" and database *DB2* contains only the associations for "*a*" and "*abate*", the symbolic LTS representing (2) is that shown in Figure 2.

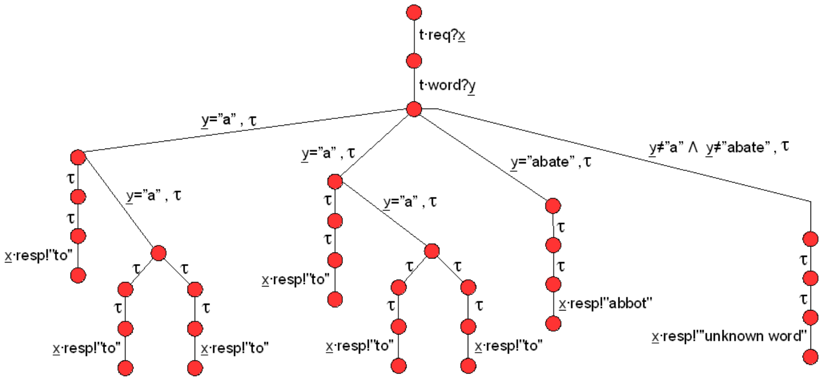


Figure 2: Symbolic LTS for the (simplified) translation service (low-level specification)

***Applying the symbolic approach to* COWS.** We are currently defining a symbolic operational semantics for COWS. To achieve this goal, the main issue is to give receive activities a proper semantics, because variables in their arguments are placeholders for something to be received. For example, let us consider the service $p \bullet o?x.s$. If $p \bullet o?x.s \xrightarrow{p \bullet o?x} s$ then the behaviour of the continuation service s must be considered under all substitutions of the form $\{x \mapsto v\}$ (i.e. the semantics of s can intuitively be thought of as a function $\lambda x\, s$ from values to services). In case of standard semantics for π-calculus [11], for example, this problem is not tackled at the operational semantics level, but it is postponed to the observational semantics level. In fact, in the definition of late bisimulation for π-calculus, whenever P is bisimilar to Q, if $P \xrightarrow{a(x)} P'$ then there is Q' such that $Q \xrightarrow{a(x)} Q'$ and $P'\{u/x\}$ is bisimilar to $Q'\{u/x\}$ for every u. Thus, continuations P' and Q' are considered under all substitutions for x. Instead, here we aim at defining an operational semantics for COWS that properly handles input transitions, while allowing finite state LTSs to be associated to COWS terms.

The basic idea is to allow receive activities to evolve by performing a communication with the 'external world' (i.e. a COWS context), this way they do not need to synchronise with invoke activities within the considered term. To avoid infinite branching (as in the case of early operational semantics), we replace variables with *generic values* rather than with specific values. We denote by $\underline{x}$ the generic value for the variable x. This way, the term $[x]\,(\, p \bullet o?x.\, q \bullet o'!x\,)$ can evolve as follows:

$$[x]\,(\, p \bullet o?x.\, q \bullet o'!x\,) \xrightarrow{p \bullet o\,?[x]} q \bullet o'!\underline{x} \xrightarrow{q \bullet o'!\underline{x}} \mathbf{0}$$

Also receive activities having a value as argument (e.g. $p \bullet o?v$) and invoke activities (e.g. $p \bullet o!v$) can evolve by communicating with the external world. Of course, these kinds of communication do not produce substitutions.

When an external communication takes place, the behaviour of the continuation service depends on the *admittable values* for the generic value. To take care of the real values that the generic values can assume, we define a *symbolic semantics* for COWS, where the label on each transition has two components: the *condition* that must hold for the transition to be enabled and, as usual, the *action* of the transition. Moreover, to store the conditions that must hold to reach a state and the names exported along the path, we define the semantics over configurations of the form $\Phi, \Delta \vdash s$, called *constrained services*, where the condition Φ and the set of names Δ are used to determine the actions that s can perform. Thus, the symbolic transitions are of the form $\Phi, \Delta \vdash s_1 \overset{\Phi', \alpha}{\rightarrowtail} \Phi', \Delta' \vdash s_2$, meaning "if the conditions Φ' (such that Φ is a subterm of Φ') holds then s_1 can perform the action α leading to s_2 by extending the set of exported private names Δ to the set Δ'".

All in all, a symbolic LTS associated to a COWS term conveys in a distilled form all the semantics information of the term's behaviour. More specifically, besides receive transitions, symbolic representations take into account generation and exportation of fresh names, pattern-matching, expressions evaluation, and priorities among conflicting receives. Dealing at once with all the above features at operational semantics level makes the development of a symbolic semantics for COWS more complex than for more standard calculi, such as value-passing CCS or π-calculus.

Work in progress. The symbolic operational semantics for COWS can pave the way for the development of efficient equivalence and model checkers. In fact, the model checking approach for COWS specifications presented in [5] does not support a fully compositional verification methodology. It permits to analyse systems of services 'as a whole', i.e. we cannot analyse isolated services (e.g. a provider service without a proper client). This is somewhat related to the original semantics of COWS that, although based on an LTS, follows a reduction style. Symbolic operational semantics should permit to overcome this limitation. We are also defining a notion of *bisimilarity* to characterise COWS terms that have the same behaviour, and intend to develop an efficient symbolic characterisation. The behavioural equivalence above could be also exploited to improve performance of model checking analyses, by reducing the size of large LTSs while preserving the intended behaviour.

References

[1] M. Boreale et al. SCC: a Service Centered Calculus. In *WS-FM*, *LNCS* 4184, pp. 38–57. Springer, 2006.

[2] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *COORDINATION*, *LNCS* 4038, pp. 63–81. Springer, 2006.

[3] M.J. Butler, C.A.R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, *LNCS* 3525, pp. 133–150. Springer, 2005.

[4] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP*, *LNCS* 4421, pp. 2–17. Springer, 2007.

[5] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying COWS specifications. In *FASE*, *LNCS* 4961, pp. 230–245. Springer, 2008.

[6] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: a calculus for service oriented computing. In *ICSOC*, *LNCS* 4294, pp. 327–338. Springer, 2006.

[7] M. Hennessy and H. Lin. Symbolic bisimulations. *Theor. Comput. Sci.*, 138(2):353–389, 1995.

[8] C. Laneve and G. Zavattaro. Foundations of web transactions. In *FoSSaCS*, *LNCS* 3441, pp. 282–298. Springer, 2005.

[9] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services (full version). Technical report, Università degli Studi di Firenze, 2006. `http://rap.dsi.unifi.it/cows`.

[10] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *ESOP*, *LNCS* 4421, pp. 33–47. Springer, 2007.

[11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Inf. Comput.*, 100(1):1–40, 41–77, 1992.

[12] OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, April 2007.

[13] R. Pugliese, F. Tiezzi, and N. Yoshida. Towards a symbolic semantics for service-oriented applications (full version). Technical report, Università degli Studi di Firenze and Imperial College London, 2008. `http://rap.dsi.unifi.it/cows`.

Seamlessly Distributed & Mobile Workflow or: The right processes at the right places[*]

Position Paper

Mikkel Bundgaard Thomas Hildebrandt
Espen Højsgaard

IT University of Copenhagen, Denmark
{mikkelbu, hilde, espen}@itu.dk

Abstract

We briefly outline some of the paths being explored within two recently initiated research projects on Computer Supported Mobile Adaptive Business Processes (CosmoBiz) and Trustworthy Pervasive Healthcare Services (TrustCare) to provide flexible process languages and models that allow seamless, trustworthy distribution and mobility of workflows and business process. In particular, we consider higher-order mobile embedded business processes, declarative versus imperative process specifications, and communication-centric architectures versus distributed shared storage.

1 Introduction

Concurrent and distributed processes are becoming increasingly present, taking many different forms: as machine code deployed on multi-core processors, as parallel algorithms executed on grid computing networks, as service orchestrations, or even as computer supported workflows and global business processes. A common challenge is that the level of concurrency and distribution is *not* statically fixed or a priori known. It typically depends on the availability of resources and capabilities of different processors, servers or localities.

Below we briefly outline some of the paths being explored within two recently initiated research projects on Computer Supported Mobile Adaptive Business Processes (CosmoBiz) and Trustworthy Pervasive Healthcare Services (TrustCare) respectively, in addressing the challenge to provide process languages and models that allow for seamless distribution and re-distribution of workflows and business process. In particular, we consider higher-order mobile embedded business processes, declarative versus imperative process specifications, and communication-centric architectures versus distributed shared storage.

[*] This work was funded in part by the Danish Research Agency (grant no.: 2106-07-0019, no.: 274-06-0415 and no.: 2059-03-0031) and the IT University of Copenhagen (the TrustCare, CosmoBiz and BPL projects).

2 Higher-order Mobile Embedded BPEL

Services implemented and orchestrated by processes written in languages such as WS-BPEL are being put forward as a means to achieve loosely coupled and highly flexible computer supported business and work processes.

In the current architectures, the service topology is *flat*, i.e. no service is a priori administrated by other services. In many applications however, an instance of a service will be acting as a sub-instance of another service, which e.g. should be reflected in the situation when one of the services terminates abnormally. Another limitation of current architectures is that services are deployed and managed by *meta-level* tools, i.e. one cannot write a business process that automate deployment and management of processes.

In the paper [1] we propose and formalize a higher-order WS-BPEL-like language called *Higher-order mobile embedded BPEL* (HomeBPEL), where processes are values that can be stored in variables and dynamically instantiated as embedded sub-instances. A sub-instance is similar to a WS-BPEL scope, except that it can be dynamically frozen during a session and stored as a process in a variable. When frozen in a variable, the process instance can be sent to remote services as any other content of variables and dynamically re-instantiated as a local sub-instance continuing its execution. This conceptually relatively simple idea results in a very powerful higher-order business process language allowing to express a nested hierarchy of processes and business process management processes.

We exemplify the use of HomeBPEL by an example of pervasive healthcare, where instances of treatment workflows are moved between and executed locally on mobile devices belonging to either the doctor or the patient, depending on whether the treatment workflow requires actions by the doctor or it prescribes actions carried out as self-treatment by the patient.

The investigation is part of the Computer Supported Mobile Adaptive Business Processes (CosmoBiz) project [10], which aims to provide a fully formalized runtime engine for a business process language extended to allow for flexible mobile and disconnected operation of Enterprise Resource Planning (ERP) systems as developed by Microsoft Development Center Copenhagen [12].

A key concern is to limit the gap between the source language, its formalization, and the implementation. To this end we currently work within the model of bigraphical reactive systems on proving operational correspondence between the concrete bigraphical semantics of WS-BPEL (which is close to the concrete WS-BPEL syntax) and a more abstract semantics given by a second bigraphical reactive system closer to process calculi. Another key concern is to use the formalizations as basis for the development of type systems that can be used to statically guarantee safe and reliable behavior. To this end we plan to examine the approaches done for Boxed Ambients [6] and for the higher-order π-calculus [13] on the safe integration of higher-order mobility and sessions.

Another interesting path for future research will be to examine different primitives for management and manipulation of processes, such as sub-process reflection and general manipulation, e.g. editing or joining of frozen sub-processes. This relates to the work on Higher-Order (Petri) Nets and applications to workflow studied in [11].

3 Trustworthy Pervasive Healthcare Services

The Trustworthy Pervasive Healthcare Services (TrustCare) project [8] is a strategic interdisciplinary research collaboration between IT University of Copenhagen, Department of Computer Science, Copenhagen University and Resultmaker[1], a Copenhagen based software company developing workflow management systems for e.g. the public sector. The project combines experiences in developing workflow management systems with research in programming languages technology, concurrency theory, logical frameworks, pervasive computing and human computer interaction. The aim of the project is to contribute to the foundations of IT-systems able to support trustworthy pervasive workflows and services within the healthcare sector. This is an extremely challenging application domain, since by nature, healthcare services involve coordination of a heterogeneous set of professionals and patients, across different locations, organizations, and sectors. Moreover, healthcare services are highly safety critical; deal with sensitive medical data; and must be able to support dynamic changes to adapt to inevitable evolution of treatment processes and unforeseen events.

One path being explored in the project is the use of *declarative* process descriptions and *shared storage* architectures as opposed to *imperative* process descriptions and *message passing/communication-centric* architectures such as WS-BPEL. The current process model employed in the Resultmaker Online Consultant workflow management system is indeed based on a patented declarative process model based on the specification of dynamically evaluated conditions for inclusion of activities in workflows (e.g. different kinds of predecessor constraints between activities) and a shared storage architecture.

This is in line with a recent proposal by Van der Aalst and Pesic in [15] to use LTL as the foundation for flexible declarative process languages. Van der Aalst and Pesic argue that the use of imperative process languages often leads to *over-specification*, which imposes too many constraints on the flows and consequently amplifies the need for changes to the specified process. Based on this, the authors propose a paradigm shift replacing the imperative process languages with *declarative* process languages, in which one specifies only the required constraints between work activities rather than a receipt for how the constraints are resolved. (This is in fact a rebirth of 20 year old proposal by Gabbay described in [5].)

So far, we have described in [14] how to formalize the key primitives of the Online Consultant process model as Linear time Temporal Logic (LTL) formulas. A workflow process is then described as the conjunction of temporal constraints, e.g. specifying that an activity A must happen before another activity B or that some activity B must be re-executed every time the activity A has been executed. A concrete example of the latter is when B is the activity of signing a contract and activity A is the activity of changing the content of the contract.

Van der Aalst and Pesic [15] and our work in [14] focus on the *temporal* constraints. However, we would like to stress that this approach could equally well be applied to the *spatial* aspects, that is, the specification of distribution and uses of resources. Part of our future work will thus be to investigate the use of declarative models, e.g. adding spatial [4] and so-called independence (or

[1] See ⟨`http://www.resultmaker.com`⟩

true concurrency) modalities [7] to the specification logics, to describe flexible distributed workflow processes that may be seamlessly dynamically distributed, changed or re-distributed.

Related to this we are investigating the development and use of (timed) concurrent constraint programming for workflow and business process management, which to our surprise is as yet almost unexplored.

As a possible architecture for a distributed declarative process engine we will investigate distributed shared storages. A key point here is that the interaction happens through a (possibly transparently) distributed storage and not as explicit communications between localities. However, at some level one has to consider the communication between different localities. This raises the natural question of how to map from a global description to a distributed end-point description, in particular to specify and implement reliable and secure interfaces between end-points. In addressing the former question we plan to investigate the research on communication-centric computation and translations between global descriptions to local end-point descriptions in [2, 3] and in general the use of behavioral types.

A related question is how to support reliable and trustworthy interfaces between services and *human* actors/end-points. We plan to address this question by investigating extensions of pervasive user interfaces based on the paradigm of activity-based computing, e.g. to include dynamically generated user interface components based on behavioral types and end-point projections.

An interesting challenge to the existing methods is to be able to deal with dynamic changes in behavioral interfaces, i.e. by dynamic end-point projections, which may be needed to cope with the fact that workflow processes may need to be dynamically changed. We intend to investigate the use of proof carrying code techniques and (concurrent) logical frameworks as a foundation for trustworthy dynamic changes of interfaces.

4 Conclusions

To summarize, the two projects CosmoBiz and TrustCare outlined above both address the challenge to provide flexible process languages and models that allow seamless, trustworthy distribution and mobility of workflows and business process. But so far from two different sides. The former project so far considers *imperative* process languages extended with primitives for higher order mobile embedded processes and formalized as bigraphical reactive systems. This has resulted in a proposal and formalization of HomeBPEL, forming the foundation for further studies of type systems for higher order processes. On the other hand, the latter project focus on *declarative* process languages formalized in temporal logics and implemented using logical frameworks. While declarative process languages has been proposed as a means to achieve more flexible process descriptions with respect to the *logical* ordering of actions, we point out that it may well also be used to achieve more flexible process descriptions with respect to the *spatial* distribution and truly concurrent execution of actions. The two projects will of course not run in isolation — in particular, we intend to study the use of (declarative) higher order primitives in the latter and the use of declarative languages and distributed shared storage in the former. (Research on the use of bigraphical reactive systems as the foundation for an XML-centric distributed shared storage coordination middleware has in fact already been initiated in [9].)

Finally, we expect to contribute to the study of projections from global descriptions to local end-points and interfaces, in particular in researching how to support changes to processes, dynamic generation and verification of interfaces, and the generation of human user interfaces.

References

[1] M. Bundgaard, A. J. Glenstrup, T. Hildebrandt, E. Højsgaard, and H. Niss. Formalizing higher-order mobile embedded business processes with binding bigraphs. In *Proceedings of COORDINATION 08*, Lecture Notes in Computer Science. Springer Verlag, 2008.

[2] M. Carbone, K. Honda, and N. Yoshida. A calculus of global interaction based on session types. *Electronic Notes in Theoretical Computer Science*, 171(3):127–151, 2007.

[3] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In R. De Nicola, editor, *Proceedings of the 16th European Symposium on Programming (ESOP 2007)*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer Verlag, 2007.

[4] G. Conforti, D. Macedonio, and V. Sassone. Static BiLog: a unifying language for spatial structures. In W. Penczek and G. Rozenberg, editors, *Half a century of inspirational research, honouring the scientific influence of Antoni Mazurkiewicz*, volume 80 of *Fundamenta Informaticae*, pages 91–110. IOS Press, 2007.

[5] D. M. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In *Temporal Logic in Specification*, pages 409–448, London, UK, 1987. Springer-Verlag.

[6] P. Garralda, A. B. Compagnoni, and M. Dezani-Ciancaglini. BASS: Boxed ambients with safe sessions. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 61–72. ACM Press, 2006.

[7] J. Hayman and G. Winskel. Independence and concurrent separation logic. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 147–156. IEEE Computer Society, 2006.

[8] T. Hildebrandt. Trustworthy pervasive healthcare processes (TrustCare) research project. Webpage, 2008. ⟨http://www.trustcare.dk/⟩.

[9] T. Hildebrandt, H. Niss, M. Olsen, and J. W. Winther. Distributed Reactive XML. In *Proceedings of the 1st International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord'05)*, volume 150 of *Electronic Notes in Theoretical Computer Science*, pages 61–80, 2006.

[10] T. Hildebrandt, H. Niss, and K. Schmidt. Cosmobiz research project. Webpage, 2007. ⟨http://www.cosmobiz.org/⟩.

[11] K. Hoffmann and T. Mossakowski. Algebraic higher-order nets: Graphs and petri nets as tokens. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Proceedings of the 16th International Workshop on Recent Trends in Algebraic Development Techniques (WADT'02)*, volume 2755 of *Lecture Notes in Computer Science*, pages 253–267. Springer Verlag, 2003.

[12] Microsoft. Microsoft dynamics mobile development tools white paper - extending business solutions to the mobile workforce. Webpage, June 2007. ⟨http://dynamicsuser.net/files/folders/94158/download.aspx⟩.

[13] D. Mostrous and N. Yoshida. Two session typing systems for higher-order mobile processes. In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications (TLCA'07)*, volume 4583 of *Lecture Notes in Computer Science*, pages 321–335. Springer Verlag, 2007.

[14] M. R. Rao, T. Hildebrandt, K. Nørgaard, and J. B. Tøth. The Resultmaker Online Consultant: From declarative workflow management in practice to LTL. Submitted for publication, 2008.

[15] W. M. P. van der Aalst and M. Pesic. A declarative approach for flexible business processes management. In *Proceedings of Workshop on Dynamic Process Management (DPM'06)*, volume 4103 of *Lecture Notes in Computer Science*, pages 169–180. Springer Verlag, 2006.