PLACES'10: The 3rd Workshop on Programmng Language Approaches to concurrency and Communication-Centric Software

Honda, Kohei; Mycroft, Alan

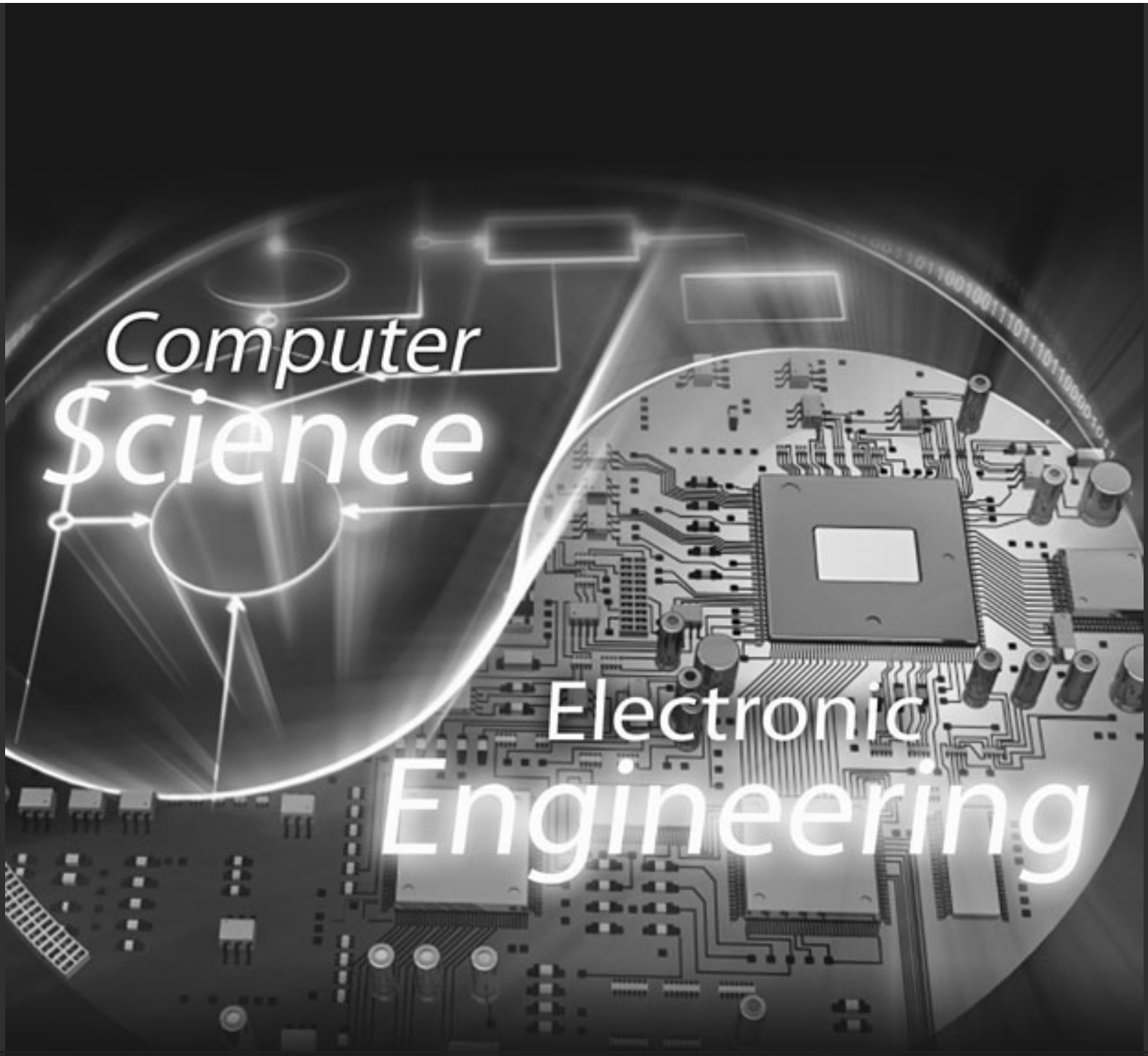
ISSN 2043-0167

PLACES'10: The 3rd Workshop on Programming Language Approaches to concurrency and Communication-Centric Software

Editors: Kohei Honda and Alan Mycroft

Queen Mary
University of London
School of Electronic Engineering
and Computer Science

EECSRR-10-02 March 2010



PLACES'10

The 3rd Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software

Paphos, Cyprus

March 2010

Preface

This is the pre-proceedings of PLACES'10, the 3rd Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, to be held in Paphos, Cyprus, on March 21, 2010, co-located with ETAPS'10. 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.

PLACES'10 welcomed William Cook for the invited talk. We are excited to be able to hear William's talk on concurrency, and sincerely thank him for agreeing to come from Texas Austin.

For the submitted contributions, the Program Committee, after a careful and thorough reviewing process, selected for the inclusion in the programme 10 papers out of 13 submissions. Each submission was evaluated by at least two referees, and the accepted papers were selected during one week electronic discussions. This volume contains all of the 10 contributed papers.

PLACES'10 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. Secondly we would like to thank our invited speaker. We would also like to thank the members of the Program Committee for their careful reviews. We appreciate continuous help by Anna Philippou (ETAPS General Co-Chair) for her continuous help and George A. Papadopoulos (ETAPS Local Chair) for his valuable assistance.

March 2010

Kohei Honda
Alan Mycroft

Conference Organization

Programme Chairs

Kohei Honda
Alan Mycroft

Programme Committee

Alastair Beresford
Marco Carbone
Simon Gay
Joshua Guttman
Hanne Nielson
John Reppy
Konstantinos Sagonas
Vivek Sarkar
Vasco Vasconcelos
Jan Vitek
Nobuko Yoshida

External Reviewers

Fossati, Luca

Table of Contents

Secure Execution of Distributed Session Programs

Nuno Alves
Freelance Consultant
nma08@imperial.ac.uk

Raymond Hu Nobuko Yoshida Pierre-Malo Deniélou
Imperial College London
{rhu,yoshida,malo}@doc.ic.ac.uk

Abstract

The development of the SJ Framework for session-based distributed programming is part of recent and ongoing research into integrating *session types* and practical, real-world programming languages. SJ programs featuring session types (protocols) are statically checked by the SJ compiler to verify the key property of *communication safety*, meaning that parties engaged in a session only communicate messages, including higher-order communications via *session delegation*, that are compatible with the message types expected by the recipient.

This extended abstract presents current work on security aspects of the SJ Framework. Firstly, we discuss our implementation experience from improving the SJ Runtime platform with security measures to protect and augment communication safety at runtime. We implement a transport component for secure session execution that uses a modified TLS connection with authentication based on the Secure Remote Password (SRP) protocol. The key technical point is the delicate treatment of *secure session delegation* to counter a previous vulnerability. We find that the modular design of the SJ Runtime, based on the notion of an Abstract Transport for session communication, supports rapid extension to utilise additional transports whilst separating this concern from the application-level session programming task. In the second part of this abstract, we formally prove the target security properties by modelling the extended SJ delegation protocols in the π-calculus.

1 Session Programming in SJ

It has become increasingly important to understand and specify the behaviour of applications across many domains as sequences of communications and interaction between concurrently executing components, as opposed to independent black boxes that simply return a final output from a given input. Unfortunately, the most common technologies and programming techniques for communication-based programming in use today do not provide the level of support and safety guarantees enjoyed for traditional single-threaded, "localised" programming. For example, low-level network socket APIs provide only the minimal mechanisms for exchanging untyped data in an unstructured manner, and higher-level RPC/RMI libraries are typically coupled to the synchronous call-return pattern and lack the facility to encapsulate a series of such exchanges as one complete unit of interaction.

SJ [9, 11] is an extension of Java that uses *session types* [7] to address the above issues. Programmers use session types, which can be thought of as communication protocols, to specify the abstract communication behaviour of a program. Concrete communication behaviour is implemented using special session primitives; in particular, SJ supports higher-order session types, implemented by *session delegation* actions, that express the migration of ongoing sessions to new parties. The SJ compiler performs static session type checking, guaranteeing that session implementations conform to their declared types, i.e. a SJ session between compatible peers will never reduce to a communication error other than premature termination due to e.g. network failure. Other works have presented SJ implementations of communications-based applications from widely different domains, such as Internet Web services [9] and parallel algorithms for cluster computing [3], demonstrating competitive performance in both cases.

Securing session execution. To enforce the statically verified communication safety property at runtime, the SJ Runtime (SJR) validates peer compatibility at session initiation and provides a session monitoring service that dynamically tracks session progress against the expected type. Nevertheless, the development of the SJ Framework as a research project, until now, had yet to focus on *session security*

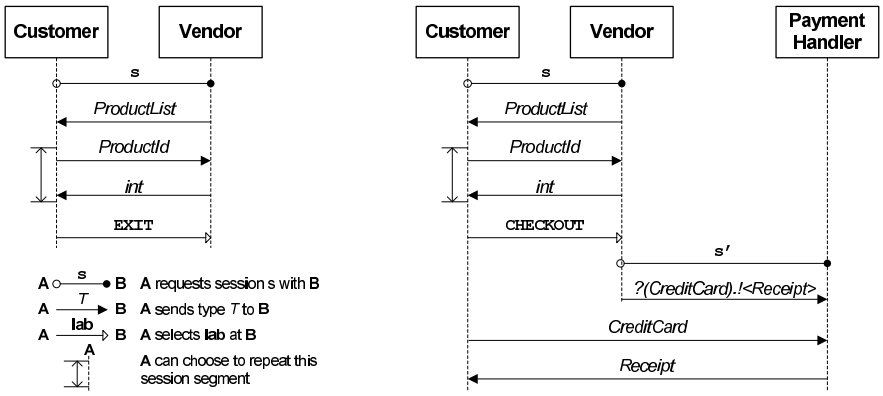


Figure 1: The interaction between a Customer, the Vendor and Payment Handler service in an online purchase session.

as a primary objective. Indeed, the security of session delegation has been a frequently posed question. In this abstract, we identify a potential vulnerability in the existing SJ session delegation protocols; we then implement and formalise SJR extensions for secure session execution, solving the above problem and strengthening runtime communication safety. § 2 starts with an example SJ application (a modified version of the main example in [9]) that illustrates both the usual security concerns and issues specific to session delegation. § 3 then briefly discusses the design and implementation of a new *Transport Module* that enables secure session execution over a modified TLS connection with authentication based on the Secure Remote Password (SRP) protocol. In § 4, we formalise the extended delegation protocols encapsulated by the new Transport Module, and prove the target security properties. Finally, § 5 concludes by describing our ongoing and future work.

2 An Example SJ Application

To illustrate the session security issues, in particular regarding session delegation, we examine the interaction in a modified version of the main Web services example from [9], an implementation of Use Case C-UC-001 from the W3C Web Services Choreography Requirements [5].

The basic scenario is an online purchase session involving three parties, a Customer (**C**), the Vendor (**V**) and a Payment Handler service (**H**). The interaction between these parties constituting one session is depicted in Figure 1, where each side represents one of the two ways to complete the session. Both start by **C** connecting to **V**, and **V** sending a list of the products for sale. In the next part, **C** adds a product to the basket and **V** returns the updated total cost of the basket; this segment can be repeated an arbitrary number of times by **C**. After this, **C** has two choices. On the left-hand side, **C** cancels the purchase and ends the session by selecting the `EXIT` branch. On the right-hand side, **C** proceeds by selecting the `CHECKOUT` branch. At this point, **V** enters a session with the third party, **H**. The single action between **V** and **H** is an example of *session delegation*: the message type itself is a session type that specifies the remaining session actions to be completed by **H** on behalf of **V**. After **V** delegates its side of the session to **H**, the interaction proceeds between **C** and **H**: **C** sends her credit card details, and **H** issues a receipt.

Session types for this application. Figure 2 lists the session types, declared as SJ protocols, for the interaction between **C** and **V** from the perspective of each party. The `customerToVendor` protocol starts

```
protocol customerToVendor {
  cbegin. // Client session request.
  ?(ProductList). // Get product list.
  ![ // Can repeat this segment.
    !<ProductId>. // Add product to basket.
    ?(int) // Get updated basket total.
  ]*.
  !{ // Two branch options.
    CHECKOUT: // Proceed to checkout.
      !<CreditCard>.
      ?(Receipt),
    EXIT: // Cancel purchase.
  }
}
```

```
protocol vendorToCustomer {
  sbegin.
  !<ProductList>.
  ?[
    ?(ProductId).
    !<int>
  ]*.
  ?{
    CHECKOUT:
      ?(CreditCard).
      !<Receipt>,
    EXIT:
  }
}
```

Figure 2: Session types (declared as SJ protocols) for the interaction depicted in Figure 1.

with the `cbegin` element to denote the client side of the session. Receiving and sending messages have the syntax, e.g. `?(ProductList)` and `!<ProductId>` respectively. The repeated segment of the session is specified as a session iteration type, `![...]`. Here, the `!` signifies that **C** controls the termination condition of the loop. Finally, the two session branches are collected within the `!{...}` constructor and labelled, e.g. `CHECKOUT`; the `!` again signifies that **C** makes the branch decision. The `vendorToCustomer` protocol is the *dual* session type that describes the reciprocal behaviour, in this case given by simply inverting the output (`!`) and input (`?`) symbols. Note that the `?[...]` (resp. `?{...}`) type specifies that **V** should follow the iteration (resp. branch) decision made by **C**.

Recall that **V** will not actually perform the final `?(CreditCard).!<Receipt>` exchange itself, but will delegate these actions to **H**. These actions must still be specified in `vendorToCustomer` to achieve duality between the behaviours of **C** and **H** (in SJ, session initiation between non-dual parties raises an exception). However, the delegation action from **V** to **H**, specified by the protocols

```
protocol vendorToHandler {
  cbegin.
  !<?(CreditCard).!<Receipt>>
}
```

```
protocol handlerToVendor {
  sbegin.
  ?(?(CreditCard).!<Receipt>)
}
```

ensures that **V** indeed fulfils the `vendorToCustomer` protocol contract. This Use Case demonstrates how session types provide a type-safe discipline for communications programming, including higher-order communications that evolve the shape of the session network.

Due to space limitations, we omit the application-level implementations of these protocols in order to focus on the runtime security of delegation. The full source code for this example can be found in the `tests/src` directory (see the `places.purchase` package) of the SJ Google Code repository (linked from [11]); see [9] for further explanation of the SJ session primitives. The types and implementation of this application can be readily extended (e.g. to pass additional information from **V** to **H** before the delegation, and for **H** to return an acknowledgment after completing the session with **C**) by adding the required session interactions.

A vulnerability in session delegation. Needless to say, conducting the above session over an insecure transport connection jeopardises message confidentiality and integrity, an especial concern for highly sensitive messages like `CreditCard`. For this purpose, the SJR includes SSL and HTTPS variants of the basic TCP and HTTP Transport Modules, implemented using the standard Java APIs for these features. However, the current version of SJ does not have dedicated support for mutual peer authentication outside

1.	$V \rightarrow H$:	START_DELEGATION			
2.	H:	Open server socket on free port p_H; accept connection on p_H			
3.	$H \rightarrow V$:	p_H			
4.	$V \rightarrow C$:	$DS_C^V(H) = \langle ST_{\mathbf{C}}^{\mathbf{V}}, IP_H, p_H \rangle$			
5.	$C \rightarrow V$:	ACK_{CV}			
6.	C:	Close s	6'.	V:	Close s
7.	C:	Connect to $IP_H : p_H$			
8.	$C \rightarrow H$:	$LM(ST_{\mathbf{V}}^{\mathbf{C}} - ST_{\mathbf{C}}^{\mathbf{V}})$			

Figure 3: Operation of the original Resending Protocol between the Vendor, Handler and Customer.

of TLS certificate-based authentication (which is used to accomplish only unilateral authentication in most typical TLS authentication scenarios).

In addition to the general attacks mentioned above, we identify a specific vulnerability in the SJ session delegation protocols. The first work on implementing session delegation [9] presented three alternative SJR protocols with varying tradeoffs for coordinating the three (or four) parties involved in the delegation of an ongoing session. Two of these protocols, the Resending Protocol and Bounded Forwarding Protocol, are termed as *reconnection-based*. As the name suggests, these protocols involve closing the original transport-level connection underlying the application-level session being delegated, and establishing a new connection to reflect the session migration. A detailed recap of the delegation protocols is beyond the scope of this abstract, but Figure 3 lists an instance of the Resending Protocol for the above application: V (resp. H, C) denotes the SJR supporting **V** (resp. **H**, **C**), $DS_C^V(H)$ denotes the Delegation Signal for the delegation from **V** to **C** with passive party **H**, $ST_{\mathbf{C}}^{\mathbf{V}}$ the remaining type of the session between **V** and **C** from the former's side, ACK_{CV} the Delegation Acknowledgement from C to V, and $LM(ST_{\mathbf{V}}^{\mathbf{C}} - ST_{\mathbf{C}}^{\mathbf{V}})$ the "lost messages" corresponding to the difference between the two session types. The reconnection itself is performed by the C SJR over steps 6 and 7. The crucial point is that the lack of mutual authentication between all *three* peers, and hence the inability to confirm that the party accepted by H (step 2) is the same as the original C, allows an attacker to infiltrate the session, masquerading as C, at this weak point. This attack also applies to the Bounded Forwarding delegation protocol [9] for the same reasons.

For secure session delegation, we extend the reconnection-based protocols with additional authentication message exchanges: Figure 4 lists the new secure version of the original protocol instance from Figure 3. Our extended protocol is different from the original in step 1, the creation of a fresh credential by V, and steps 2 and 5, which send the credential to H and C respectively. H then stores the credential and waits for a connection on p_H. The key action in the extended protocol lies on step 9, where C sends the credential to H after connecting: if H can validate that the credentials match, the connection is successfully established, otherwise the delegation has failed and the session is aborted.

The above protocol listings correspond to Case 1 of the Resending Protocol [9]. A complete analysis of all four Delegation Cases for our Secure Resending and Forwarding Protocols is specified in [2].

3 Implementation of a Secure Transport Module

We first summarise the SJ Framework and the structure of the SJ Runtime (SJR). We then briefly explain the design and implementation of a secure Transport Module plugin for the SJR that solves the security issues described in § 2.

The SJ Framework comprises the compiler and the SJ Runtime (SJR). The compiler transforms SJ programs into a *transport-independent* form in standard Java, translating the application-level SJ session

1.	V:	Credential creation			
2.	$V \rightarrow H$:	`START_DELEGATION::CRED`			
3.	H:	Open server socket on free port p_H; accept connection on p_H			
4.	$H \rightarrow V$:	p_H			
5.	$V \rightarrow C$:	$DS_C^V(H) = \langle ST_{\mathbf{C}}^{\mathbf{V}}, IP_H, p_H, \mathtt{CRED} \rangle$			
6.	$C \rightarrow V$:	ACK_{CV}			
7.	C:	Close s	7'.	V:	Close s
8.	C:	Connect to $IP_H : p_H$			
9.	$C \rightarrow H$:	`CRED`	9'.	H:	`CRED` checking
9a.	-pass:	Connection successful	9a'.	-pass:	Connection established
9b.	-fail:	Credential rejected, close s	9b'.	-fail:	Authentication error, close p_H
10.	$C \rightarrow H$:	$LM(ST_{\mathbf{V}}^{\mathbf{C}} - ST_{\mathbf{C}}^{\mathbf{V}})$			

Figure 4: Operation of the *Secure* Resending Protocol between the Vendor, Handler and Customer.

primitives in terms of Java control flow and calls to *Interaction Services* hosted by the SJR, also implemented in Java. Thus, SJ programs can be executed on any standard JVM, where the purpose of the SJR is to perform the requested Interaction Services as actions on an underlying transport connection. The key element in the SJR is the *Abstract Transport Interface*, which represents an idealised asynchronous, reliable and order-preserving message-oriented transport for session communication. Interaction Service components are implemented as actions on the Abstract Transport, which are in turn implemented by *Transport Modules* that encapsulate the communication mechanisms of specific "concrete" transports, such as TCP and shared memory. The Abstract Transport thus serves to decouple the realisation of session interaction semantics in the SJR from the provision of the underlying communication mechanisms.

Design. Our primary goal was to provide a SJR Transport Module that incorporates a means of peer authentication in addition to message confidentiality and integrity, and thereby solve the security concern exposed by the reconnection-based delegation protocols. After evaluating the available options (see [2] for the omitted details), our chosen design adapts TLS to use the Secure Remote Password (SRP) [12] authentication method [6]. Although standard usage of TLS provides confidentiality and integrity, authentication primarily relies on certificates and external authorities: our approach instead combines TLS and SRP to provide session security without requiring trusted third-parties or certificates, whilst still being able to use these mechanisms if available. Retaining session independence from external third-parties is also in line with the established session theory (which our formalism in § 4 follows), where scope restriction of each session to just the two parties involved is an important invariant in proving communication safety [10].

Implementation and the extended delegation protocols. TLS consists of three basic phases: negotiation of the algorithms supported, key exchange and authentication, and symmetric cipher encryption/message authentication. Unfortunately, none of the publically available Java implementations of SSL/TLS fulfilled all of our requirements: pure Java implementations either lacked support for SRP integration or were otherwise incomplete, and using JNI to interface e.g. C libraries would break SJ portability [2]. As a consequence, we decided to implement SRP as an initial mutual authentication phase before the standard TLS phases. At the client side, we use a modified SJ transport negotiation protocol to activate the SRP; at the server side, we override the behaviour of the `accept` method of the standard Java `SSLServerSocket` API.

With the SRP authentication in place, the resending-based delegation protocols are strengthened by using the (successfully authenticated) session-sender (i.e. the party delegating the session — in our example, **V**) to generate and distribute fresh, secure credentials on behalf of the party performing the

reconnection (**C**), as illustrated in § 2. By presenting these credentials to the session-receiver (**H**) over the *new* TLS connection (ruling out replay attacks), the latter can confirm the authenticity of the connecting party. The detailed protocol specifications for the extended delegation protocols can be found in [2].

4 A Process Model for Secure Sessions

To prove the design of the new Transport Module satisfies the intended security properties, we model the extended delegation protocols in the π-calculus and formalise each property. The three original properties for correct session delegation are Linearity, Liveness and Session Consistency, which were proved by case analysis for each of the four valid delegation scenarios (referred to as *Delegation Cases 1–4*) [9]. To these, we add a new *Session Security* property to rule out attacks on the delegation protocols: there are three aspects to this property, corresponding to the key security mechanisms that the delegation protocols depend on. In the following, we focus on the three aspects of the latter; the other properties (plus the cases omitted from below) are fully documented in [2]. As a convention, we use A, B and C to respectively denote the passive party, session-sender and session-receiver in the three-party delegation scenarios (Delegation Cases 1–3 in [2]). (In the previous example of § 2, the passive party, session-sender and session-receiver are respectively the Customer, the Vendor and the Payment Handler; we now use a general notation for the delegation roles rather than the parties from that specific example.) Our approach has been influenced by [7, 10]. Below, we use "∀protocols" to mean all of the protocols formalised in [2].

Freshness *(The credentials are fresh for only the current session.)*

$$\forall \text{protocols}.\exists P \equiv \mathsf{make}(Cred); Q \text{ s.t. } P \text{ is a session-sender}$$

where $\mathsf{make}(Cred); Q$ is a binder for $Cred$ in Q whose reduction instantiates $Cred$ to a fresh value in Q. Freshness means that every session-sender P (defined as $P \equiv s\,!\langle Cred\rangle; P_1; s'\,!\langle ..., Cred, ...\rangle; P_2$)) in the protocols creates a new credential for that specific session. Freshness in the Resending Protocol is verified for Delegation Case 1 as follows (we omit the parallel composed processes):

$$\begin{array}{c} B = \mathsf{make}(CredA); s'_{\mathrm{BC}}\,!\langle CredA\rangle; B_1 \longrightarrow s'_{\mathrm{BC}}\,!\langle CredA\rangle; B_1 \longrightarrow \\ B_1 \longrightarrow^* \overline{s_{\mathrm{AB}}}\,!\langle S', IP_{\mathrm{C}}, x_{p_C}, CredA\rangle; B_2 \longrightarrow B_2 \end{array}$$

B creates a new credential specific to the current session before the actual delegation action, which it sends to both C and A (for A, it is sent together with other important session information). So, we can conclude that Freshness holds in this case since the credential is fresh by the semantics of make.

Credential Checking *(Session delegation only succeeds if the credentials of the passive party matches that of the session-receiver. Otherwise, a delegation error has occurred and the session is terminated.)*

$$\begin{array}{ll} & \forall \text{protocols}.\exists P_C \equiv s\,?(x_{Cred}); P; port\,(x\colon \overline{S}).x\,?(y_{Cred}); \\ & \qquad\qquad \texttt{if}\ x_{Cred} == y_{Cred}\ \texttt{then}\ x \triangleleft \textbf{Success}; Q\ \texttt{else}\ x \triangleleft \textbf{Fail}; \mathsf{close}(x) \\ \text{and} & \exists P_A \equiv \overline{port}\,(x\colon S).x\,!\langle y_{Cred}\rangle; x \triangleright \{\textbf{Success} : Q, \textbf{Fail} : \mathsf{close}(x)\} \\ \text{s.t.} & P_C \text{ is the session-receiver and } P_A \text{ is the passive party} \end{array}$$

where P is defined to be a session-receiver if $P \equiv s\,?(Cred); P_1; s'\,?(Cred); P_2$, and a passive party if $P \equiv s'\,?(..., Cred, ...); P_1; s\,!\langle Cred\rangle; P_2$. This property states that in all protocols, the session-receiver must start by receiving a set of credentials from the session-sender. The session-receiver then accepts a new connection on the open *port*, and receives another set of credentials. If the credentials match, then the session is continued by selecting **Success** branch; otherwise the new connection is closed, aborting the session. The passive party starts by requesting a connection on *port* followed by sending the credential it received from the session-sender. If the credential is accepted it similarly follows the **Success** branch; otherwise it goes to the **Fail** branch, which closes the new connection from the other end.

Below we verify Credential Checking for the Resending Protocol Delegation Case 2 in [2]. Due to space limitations, we omit B's specification and interactions, and focus on the interactions between A and C.

$$\begin{array}{lcl} \text{Let } P & = & x \triangleright \{\textbf{Success} : x!\langle LM(S-S')\rangle, \textbf{Fail} : \mathsf{close}(x)\} \\ \text{and } Q & = & \texttt{if}\ x_{CredA} == y_{CredA}\ \texttt{then}\ x \triangleleft \textbf{Success}; x?(x_{\mathrm{LM}})\ \texttt{else}\ x \triangleleft \textbf{Fail}; \mathsf{close}(x); \mathsf{close}(p_C) \end{array}$$

$$\begin{array}{lcl} A \mid B \mid \overline{s'_{BC}}?(x_{CredA}); C_1 & \longrightarrow^* & \overline{p_C}(x\colon S).x!\langle z_{CredA}\rangle; P \mid B' \mid p_C(x\colon \overline{S}).x?(y_{CredA}); Q \\ & \longrightarrow^* & x!\langle z_{CredA}\rangle; P \mid B' \mid x?(y_{CredA}); Q \\ & \longrightarrow^* & (P \mid B' \mid Q) \longrightarrow^* \mathbf{0} \end{array}$$

The delegation protocol starts with $C = \overline{s'_{BC}}?(x_{CredA}); C_1$ receiving a credential from B. Then A connects to C via p_C and establishes a new session (bound to x). To finalise the delegation, A sends its credentials to C who compares this value with the one received from B: if they match, the delegation is successful; otherwise the session is closed along with the port p_C.

Attack Protection *(Protection against attacks from the Network, ensuring message authentication, confidentiality and integrity.)*

The formalism assures protection of sessions from Network attacks due to the firm restriction that each session channel can only be accessed by the processes involved in that session, i.e. each channel is private to those parties and cannot be interfered by others. To fully prove this property, an additional layer is needed to model the lower level protocols underneath the session calculus; one suitable approach would be the Applied π-Calculus [1], which we leave as future work. As a simple demonstration, however, we can model a Network Attacker E that attempts to interfere with the genuine session parties.

$$\begin{array}{l} \mathrm{E} = s_{ABC}?(S', x_{IP_C}, y_{p_C}, z_{CredA}); s_{ABC}!\langle ACK\rangle; \overline{y_{P_C}}(z\colon S).z!\langle z_{CredA}\rangle; \\ \qquad z \triangleright \{\textbf{Success} : z!\langle LM(S-S')\rangle, \textbf{Fail} : \mathbf{0}\} \end{array}$$

If E is able to intercept the delegation information, then she will be able to masquerade as A without C knowing. Note that E does not close the session after the ACK as the attack may benefit from remaining connected to all parties. However, since there is no interaction with any external s_{ABC} in any of the delegation protocols, we can infer that E will remain blocked forever. This is in line with the design choice to keep SJ close to the original session types theory by restricting the scope of all sessions from foreign parties.

5 Conclusion

The aim of this work was to examine and improve session security in SJ. In addition to standard security concerns, we identified a vulnerability in the resending-based Resending and Bounded Forwarding protocols by modelling and analysing the delegation protocols in the π-calculus. To overcome this problem, we implemented a new Transport Module for secure session execution that combines TSL with SRP authentication. We found that the structure of the SJ Framework cleanly decouples application-level logic related to implementing specific sessions from the provision of general, lower-level communication mechanisms. Hence, we were able to readily extend the SJR so that all existing SJ programs can immediately utilise this new Transport Module without any modifications to the application source code or other SJR components. To verify the correctness of our approach, we incorporated our new extensions into the formal model to formalise and prove the intended security properties.

Future Work. We are currently looking at modifying a TLS implementation to include the SRP protocol in the cipher suite list. This way, the additional key exchange could be avoided and the SRP would be part of the key exchange phase of the TLS protocol. We are also working on timestamping and revocation of credentials. Extension to multiparty session types [8] is an interesting future direction for

SJ and our session security work. One idea is to automatically generate a cryptographic protocol from the session specification to secure the multiparty session executions despite compromised participants [4]. However, the work in [4] has yet to be extended to support delegation and be adapted to the SJ Framework. In order to prove the security properties in more complex contexts, we also plan to model the delegation protocols at a lower level using the applied π-calculus.

References

[1] Martín Abadi and Cédric Fournet. Mobile Values, New Names, and Secure Communication, 2001.

[2] Nuno Alves. A Secure Session-Based Distributed Programming Language. *Imperial College London - MSc Thesis*, 2009. `http://www.doc.ic.ac.uk/teaching/distinguished-projects/2010/n.alves.pdf`.

[3] Andi Bejleri, Raymond Hu, and Nobuko Yoshida. Session-based programming for parallel algorithms. In *PLACES*, 2009. To appear in EPTCS.

[4] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic Protocol Synthesis and Verification for Multiparty Sessions. *CSF 09*, 2009.

[5] Web Services Choreography Requirements. `http://www.w3.org/TR/ws-chor-reqs/`.

[6] D. Taylor, T. Wu, N. Mavrogiannopoulos, and T. Perrin. Using the Secure Remote Password (SRP) Protocol for TLS Authentication. *RFC 5054*, 2007.

[7] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP '98: Proceedings of the 7th European Symposium on Programming*. Springer-Verlag, 1998.

[8] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284, New York, NY, USA, 2008. ACM.

[9] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 516–541, 2008.

[10] Dimitris Mostrous and Nobuko Yoshida. Session-based communication optimisation for higher-order mobile processes. In *TLCA '09*, pages 203–218. Springer, 2009.

[11] SJ homepage. `http://www.doc.ic.ac.uk/~rhu/sessionj.html`.

[12] Thomas Wu. The secure remote password protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, 1998.

Channels as Objects in Concurrent Object-Oriented Programming

Joana Campos
Department of Informatics
University of Lisbon

Vasco T. Vasconcelos
Department of Informatics
University of Lisbon

Abstract

Session types have been proposed to specify the interactions in communication protocols, allowing channel implementations to be verified by static type checking. Motivated by a recent approach for distributed object-oriented languages that abstracts from the details of the communication protocol by hiding channel primitive operations in an API, we present a prototype compiler for a small concurrent object-oriented language that extends previous work by eliminating channel operations and defining method call as the single communication primitive in both sequential and concurrent settings. In contrast to previous works, we define a single category for objects, instead of distinct categories for linear and for shared objects. We qualify types as linear or shared, and let the status of an object to be governed by its type, allowing linear objects to evolve into shared ones. Finally, we introduce a sync qualifier to enforce mutual exclusion in concurrent access to certain operations of shared objects.

Keywords: Session types, object-oriented programming, concurrency, type systems, aliasing control

Motivation Session types, introduced in [8, 9, 15], have been proposed to enhance the verification of programs at compile-time by specifying the sequences and types of messages in communication protocols. Traditionally associated with communication channels, session types provide a means to enforce that channel implementations obey the requirements stated by their types, and are thus of great assistance to programmers who want to verify the correctness of their programs. Originally developed for dyadic sessions, the concept was extended to multi-party sessions, although this is a feature beyond the scope of this paper. Programming languages that implement session types come in all flavours: pi calculus, an idealised concurrent programming language in the context of which the original concepts were developed, functional languages [7, 12, 17, 19], CORBA [16], object-oriented languages [3, 4, 6, 11, 20].

Channels as conceived in session type theory are special entities that carry messages of different types, bi-directionally, in a specific sequence between two end points. These channels are usually implemented in a socket-like style, and this involves having to work with request and accept primitive operations for creating a fresh channel and, once a connection is established, with send and receive operations for message passing between processes. To our knowledge, none of the previous attempts to integrate session types into object-oriented programming ever abstracted this notion of communication channels. The work on Moose [3, 4], a multi-threaded object-oriented calculus with session types, was the first attempt to marriage the (concurrent) object-oriented paradigm and session types. This and subsequent work have kept distinct mechanisms for local and remote communication, in the form of message passing and channel operations, respectively.

In this paper, we present our on-going work on the language MOOL, a mini object-oriented language in a Java-like style that is being designed in the context of a prototype compiler targeting Mono, the open-source *clone* of the CLR (Common Language Runtime). Our language offers a simple concurrency mechanism for thread spawning and mutual exclusion control, and a type system that applies ideas taken from session type theory. The design of our language is being guided by the attempt to make it not only type-safe at compile-time, but also simple and intuitive for object-oriented practitioners. To achieve simplicity, our communication model relies exclusively on remote method invocation in the style of Java RMI, which we believe to be much easier to work with than sockets.

In other words, we do not claim to present a novel approach to session types, or an innovative object-oriented language implementation. Rather, taking session types, and the object-oriented paradigm as given, we investigate the problem of designing a concurrent object-oriented language that relies on method call as the only primitive for structured communication, both in sequential and concurrent settings. Our approach aims at extending the flexible style introduced by previous work on modular session types [6, 20] where session-typed communication channels are treated as objects. One of the main differences between our approaches is that we replace all channel operations by method calls, which is the natural mechanism for code-level message passing between objects. Previous work hides channel primitive operations in an API from where clients can call methods. We take one step further and reduce all interaction between threads, and within a single thread, to the method call primitive.

From session types, we borrow the idea of attaching a session type at the class level, using what we call a *global usage declaration* (the keyword *usage* is taken from [10]) that defines the sequence of permitted method invocations. We spread the implementation of a session over separate methods and classes, following a modular programming trend. This means that we have to handle *non-uniform objects*, that is, objects that dynamically change the set of available methods. Nierstrasz [13] was the first to study the behaviour of non-uniform, or active, objects in concurrent systems.

The use of method invocation as the single mechanism for communication made us aware of several problems related to integrating linear objects into the object-oriented paradigm. The usage declaration, because it prescribes a call sequence, allows us to define linear methods that can be called only once. A linear method thus defined is *consumed* after being called, which means that the method is removed from the object type, and this is handled implicitly by our type-state approach. But in shared objects, the same cannot be done. The problem is closely related to the difficulty of controlling aliasing in this paradigm, because of state held in instance fields. Consider an object o shared by multiple client objects o_i, each one holding a reference to o in its fields $o_i.f$. Changes to o will change the type of the object, and thus affect all client classes, resulting that the specification in the usage declaration no longer can be guaranteed by the type system.

In our approach to object aliasing control, we define a single category for objects, as opposed to distinct categories for linear and for shared objects. We let the current status of an object to be governed by its type, allowing linear objects to evolve into shared ones (cf. [17, 18]). The opposite is not possible, as we do not keep track of the number of references to a given object. We propose that an annotation should be explicitly introduced by the programmer in the usage descriptor, indicating a **lin** type or **un** type, and this will allow us to control the object current status. The **lin** qualifier describes the status of an object that can be referenced once in exactly one thread object. The **un** qualifier stands for unrestricted, or shared, and governs the status of an object referenced in multiple threads. To lighten the syntax, the **un** qualifier can be omitted as it is the default type of every object. Foregoing work on session types for objects deals with linear types only. Even though the introduction of shared objects into session type theory looks unproblematic (and is dealt with in the implementation mentioned in [6]), the details have not yet been worked out.

Still, this does not solve the problem of concurrent access in mutual exclusion. Since we are eliminating the channel communication model, we cannot rely on a shared channel playing the role of a monitor with threads waiting on its queue. Thus, some additional mechanism must be used. Because our focus are linear objects, to enforce serialised access to certain methods that manipulate shared data, we adopt a standard and straight-forward solution similar to the *synchronized* mechanism used in Java, and specified it in the class usage descriptor by an additional **sync** qualifier.

Summing up the main contributions of our approach:

- In contrast to other work on session types, we elect method invocation as the only communication model in both concurrent and sequential programming;

- We annotate classes with an usage descriptor to structure method invocation by client objects, and we enhance it with **lin**/**un** qualifiers for aliasing control, thus defining a single category for objects that may evolve from a linear status into an unrestricted one;

- In contrast to other work on session types, we replace the well-known shared channel primitive by a conventional synchronization primitive for mutual exclusion access. We introduce a **sync** qualifier in the usage descriptor to describe those operations in shared objects that must be accessed without thread interference.

The Auction System We now present an example adapted from [16, 17]. It is a simple auction system, featuring three kinds of participants: the auctioneer, the sellers and the bidders. Sellers sell items for a minimum price. Bidders place bids in order to buy some item for the best possible price. The auctioneer controls these interactions.

The two scenarios of our auction system are best described by the UML sequence diagrams in Figures 1 and 2. The first diagram describes the scenario for a seller, while the second one describes the scenario for a bidder. Following UML notation, we specify concurrent threads using an arrow with only the upper half of the arrow showing. We are mainly interested in visualising interactions. Synchronization is not represented here as this type of diagram is not suitable for describing concurrency control.

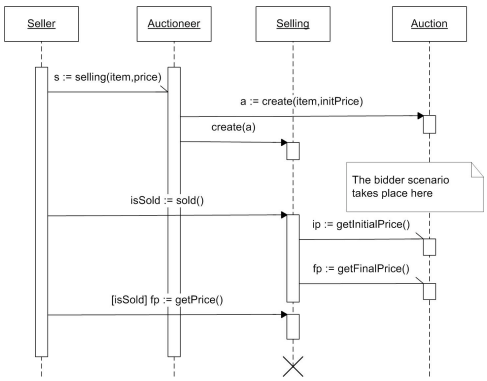


Figure 1: The scenario for a seller

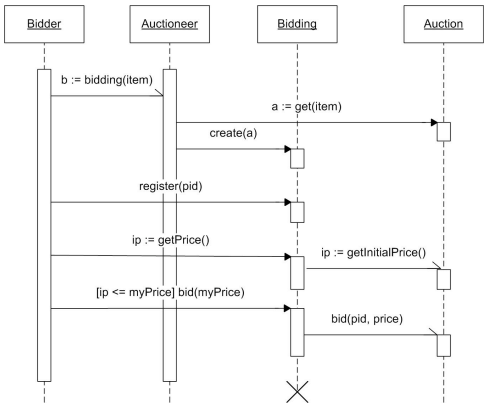


Figure 2: The scenario for a bidder

It is not difficult to conclude from the diagrams that the Selling and Bidding objects implement the old linear channels, and provide the services to the client objects. A seller will start the interaction with the auctioneer, who will delegate the service to the new Selling object. In the interaction initiated by a bidder, the service is delegated to the Bidding object. Notice that a fresh Selling (and Bidding) object is created at the start of each selling (and bidding) interaction, and is destroyed at the end. We signal object destruction in the diagram to increase the expressiveness of the representation; the type system described below provides crucial information to (remote) object deallocation.

Given the design of our auction system, it is straight-forward to write the code for each object. Figures 3–6 implement the scenario for a seller. Because the implementation for a bidder follows a similar structure, we have omitted it from this paper. Each class begins by specifying the method availability

through the usage declaration. There is no need for further specification because, as conventionally established, method signatures convey all the information a client class needs (number and type of parameters, and return type).

Apart from the usage at the beginning of each class, our language presents a typical Java-like syntax. Before explaining the implementation in detail, we briefly introduce some less obvious syntactic details in the usage descriptor. If a program defines a conventional class named C with methods m1, m2 and m3, and no usage declaration, our compiler will insert **usage** $*\{$m1 + m2 + m3$\}$ as the class default usage type, where each method (m1, m2 and m3) is always available. Formally, this usage defines a recursive branch type of the form μ X$\{$m1;X + m2;X + m3;X$\}$. In the example, a choice between calling one of the three available methods is indicated by (+). Because of the particular form of the recursive type, calling any of these methods on an instance of class C will not change the object state nor the set of available methods.

A typical usage declaration for a linear object is a sequential composition of available methods. If class C is linear, **usage lin** m1; **lin** m2; **lin** m3; **end**; is a possible usage declaration. Calling methods in the prescribed order on an instance of this class will change the object state and the set of available methods. When an object is in state **end**, it means it has no further available methods. Formally, **end** is short for the empty branch type **un**$\{\}$. A variant type, denoted $\langle \ldots + \ldots \rangle$, is indexed by the two values of the **boolean** type returned by the method preceding the variant in the usage declaration: if **true** is returned, the new object state, and the available methods, will be found in the left-hand side of the variant; if **false** is returned, it will be the right-hand side to dictate the object state and available methods. For simplicity, we use binary-only variants as in typestates [14], but more generous variants using enumerations can be found in the literature [6].

Consider now the usage specification in Figure 3. When an object of class Auctioneer is created, only one reference exists to it, but then the object evolves into a shared type, as several sellers and bidders will hold references to this object. Notice that we have defined a recursive (shared) type. Each client object can do one of two things: (1) it can call method selling to obtain an object that provides an implementation of the selling activity on the auctioneer; or (2) it can call method bidding and obtain an object that implements the buying activity on the auctioneer. Then, it can repeat the interaction all over again: a seller can lower the price of an item with no bids, and start a new sale; a bidder can bid a higher price. The type never ends, and this illustrates why, in any program, we cannot keep track of the number of references to a shared type.

Notice, still in Figure 3, that when a new selling request is made, a new Auction object is created (line 10). This instance is then added to the AuctionMap object (line 12), where the Auctioneer keeps all the auctions, and is passed to the constructors of both the Selling and Bidding objects. In the first case, the reference already exists (line 15); in the second one, it must be fetched from the AuctionMap object (line 20). It is through reading and writing to this shared Auction object that the protocol takes place.

The usage declaration in class Auction (Figure 5) shows another example of a recursive type in a shared object. Notice also the **sync** qualifier in line 3 that is used to control concurrent bids made by separate Bidder threads. This annotation also qualifies the put and get operations on the usage descriptor of the AuctionMap class (omitted).

Figures 4 and 6 implement two linear types. The usage declaration of class Seller (Figure 4) is an abbreviation for the nested composition of branch types **lin**$\{$ init ; **lin**$\{$run; **un**$\{\}\}\}$. Also notice an example of a variant type in class Selling (line 3 of Figure 6), where (;) binds stronger than (+). A variant type is always linear, so the redundant **lin** qualifier can be omitted. This type requires that the client object evaluates the returned boolean value of method sold in order to determine the next available method: if the value evaluates to **true**, then the caller can obtain the price (because the item was sold) via method getPrice, otherwise the interaction ends.

```
class Auctioneer {
  usage lin init;
        *{selling + bidding};
  AuctionMap map;
  unit init() {
    map = new AuctionMap();
  }
  Selling selling(string item,
                  int initPrice) {
    Auction a = new Auction();
    a.init(item, initPrice);
    map.put(item, a);
    Selling s = new Selling();
    s.init(a);
    s; // return s
  }
  Bidding bidding(string item) {
    Bidding b = new Bidding();
    b.init(map.get(item));
    b; // return b
  }
}
```

Figure 3: An auctioneer

```
class Seller {
  usage lin init; lin run; end;
  string item; int price;
  Auctioneer a;
  unit init(Auctioneer a,
        string item, int price) {
    ... // initialize fields
  }
  unit run() {
    Selling s =
          a.selling(item, price);
    if(s.sold())
      print("made " +
        s.getPrice() + " euros!");
    else if(lowerPrice())
      run();
  }
  boolean lowerPrice() {
    ... // implementation omitted
  }
}
```

Figure 4: A seller

```
class Auction {
  usage lin init;
        *{sync bid + getInitialPrice +
                getMaxBid + getBidder};
  string item; int initPrice;
  int bidder; int maxBid;
  unit init(string item, int initPrice) {
    ... // initialize fields
  }
  unit bid(int pid, int bid) {
    if(maxBid <= bid) {
      bidder = pid;
      maxBid = bid;
    }
  }
  ... // the getters
}
```

Figure 5: An auction

```
class Selling {
  usage lin init; lin sold;
        ⟨getPrice; end + end⟩;
  Auction a; int finalPrice;
  unit init(Auction a) {
    ... // initialize fields
  }
  boolean sold() {
    finalPrice = a.getMaxBid();
    finalPrice >= a.getInitialPrice();
  }
  int getPrice() {
    finalPrice;
  }
}
```

Figure 6: A selling protocol

```
class Main {
  usage lin main; end;
  unit main() {
    Auctioneer a = new Auctioneer();
    a.init();
    Seller seller = new Seller();
    seller.init(a, "psp", 100);
    spawn seller.run();
    Bidder bidder1 = new Bidder();
    bidder1.init(a, 1, 100);
    spawn bidder1.run();
    Bidder bidder2 = new Bidder();
    bidder2.init(a, 2, 100);
    spawn bidder2.run();
  }
}
```

Figure 7: The main class

The method lowerPrice of class Seller (Figure 4) is not referred in the usage specification. Although our language does not support method qualifiers, this can be regarded as a *private* method since the type system ensures that it cannot be called from a client (cf. [6, 20]).

Finally, the Main class in Figure 7 creates an Auctioneer object that controls the auction, and spawns a separate thread for a Seller and two Bidder objects, using a Java-like technique for thread creation.

The Language The main simplification in our language, as opposed to previous works that use session types, is that it does not include channels; its syntax is straight-forward for an object-oriented language: typical class and method declarations, and expressions. For brevity sake, we omit here the details. From the technical point of view, all the tools that we need have already been developed in previous work, and we do not anticipate technical difficulties. The core language of modular session types [6] describes most of the typing rules and operational semantics that we need to implement our type system, and the reconstruction of session types in [18] gives us insight into how to approach **lin/un** qualifiers. We have been building our ideas mostly on these two works. Our main challenge is the reuse of previous work, which involves a great deal of adaptation to fit our attempt to converge channels and objects.

Concluding Remarks In the literature, several lines of research can be found that reveal similarities with session type theory. One of these lines introduces the concept of typestate [14] in which the state of the object in some particular context determines the set of available operations in that context, based on pre- and post-conditions. Objects, by nature, can be in different states throughout their life cycle. The concept involves static analysis of programs at compile-time so that all the possible states of an object and associated legal operations can be tracked at each point in the program text. Typestate checking has been incorporated in several programming languages [1, 2, 5], and some ideas relate very closely to session type recent approach on modularity.

In this paper, we have described our on-going work on defining a more intuitive approach to handle session types within the object-oriented paradigm: we replace communication channels by method invocations, we allow objects to change from a linear status to a shared one and we control concurrency in shared objects through a standard synchronization mechanism, because we no longer have the well-known shared channel. The existing work on modular session types has been the inspiration for our specification language and type system. So far, we have developed a fully functional prototype compiler that uses a conventional type checker, and we are currently in the process of implementing the type-checking system described above. The type-checking algorithm defined in the foregoing work gives us some insight into the work we are doing, and we hope to present our results in a near future.

References

[1] Kevin Bierhoff and Jonathan Aldrich. PLURAL: checking protocol compliance under aliasing. In *ICSE Companion '08*, pages 971–972. ACM Press, 2008.

[2] Robert DeLine and Manuel Fähndrich. The fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research, 2003.

[3] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopolou. Session types for object-oriented languages. *ECOOP, Springer LNCS*, 4067:328–352, 2006.

[4] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopolou. A distributed object-oriented language with session types. *TGC, Springer LNCS*, 3705:299–318, 2005.

[5] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*. ACM Press, 2006.

[6] Simon Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, pages 299–312. ACM Press, 2010.

[7] Simon J. Gay, António Ravara, and Vasco T. Vasconcelos. Session types for inter-process communication. Technical Report TR-2003-133, Comp. Sci., Univ. Glasgow, 2003.

[8] Kohei Honda. Types for dyadic interaction. *CONCUR, Springer LNCS*, 715:509–523, 1993.

[9] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. *ESOP, Springer LNCS*, 1381:122–138, 1998.

[10] Filipe Militão. Design and implementation of a behaviorally typed programming system for web services. Master's thesis, New University of Lisbon, 2008.

[11] Dimitris Mostrous. Moose: a minimal object oriented language with session types. Master's thesis, University of London, 2005.

[12] Matthias Neubauer and Peter Thiemann. An implementation of session types. *PADL, Springer LNCS*, 3057:56–70, 2004.

[13] Oscar Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[14] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[15] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. *PARLE, Springer LNCS*, 817:398–413, 1994.

[16] Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of objects and components using session types. *Fundamenta Informaticæ*, 73(4):583–598, 2006.

[17] Vasco T. Vasconcelos. Session types for linear multithreaded functional programming. In *PPDP*, pages 1–6. ACM Press, 2009.

[18] Vasco T. Vasconcelos. *SFM*, volume 5569 of *LNCS*, chapter Fundamentals of Session Types, pages 158–186. Springer-Verlag, 2009.

[19] Vasco T. Vasconcelos, Simon J. Gay, and António Ravara. Typechecking a multithreaded functional language with session types. *Theoret. Comp. Sci.*, 368(1–2):64–87, 2006.

[20] Vasco T. Vasconcelos, Simon J. Gay, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Dynamic interfaces. FOOL, 2009.

Towards a Modal Logic for the Global Calculus*

Marco Carbone Thomas T. Hildebrandt
Hugo A. López

IT University of Copenhagen, Copenhagen, Denmark
{carbonem,hilde,lopez}@itu.dk

Abstract

We explore logical reasoning for the global calculus, a coordination model based on the notion of choreography, with the aim to provide a methodology for the specification and verification of structured communications. Starting with an extension of Hennessy-Milner logic, we propose a proof system for the logic that allows for verification of properties among participants in a choreography. Additionally, some examples of properties on service specifications are drawn, and we provide hints on how this work can be extended towards a full verification framework.

1 Introduction

Due to the continuous growth of technologies, software development is recently shifting its focus on communication, giving rise to various research efforts for proposing new methodologies dealing with higher levels of complexity. A new software paradigm, known as *choreography*, has emerged with the intent to ease programming of communication-based protocols. Intuitively, a choreography is a description of the global flow of execution of a system where the software architect just describes which interactions *might* take place. This idea differs from the standard approach where the communication primitives are given for each single entity separately. A good illustration can be seen in the way a soccer match is planned: the coach has an overall view of the team, and organizes (a priori) how players will interact in each play (the role of a choreography); once in the field, each player performs his role by interacting with each of the members of his team by throwing/receiving passes. The way each player synchronize with other members of the team represents the role of an orchestration.

The work in [3] formalises the notion of choreography in terms of a calculus, dubbed the *global calculus*, which pinpoints the basic features of the choreography paradigm. Although choreography provides a good abstraction of the system being designed allowing to *forget* about common problems that can arise when programming communication e.g. races over a channel, it can still have complex structures hence being often error prone. Additionally, choreography can be non-flexible in early design stages where the architect might be interested in designing only parts of a system as well as specifying only parts of a protocol (e.g. initial and final interactions). In this view, we believe that a logical approach can allow for more modularity in designing systems e.g. providing partial specification of a system using the choreography paradigm.

In this document, we attempt to provide a link between choreography and logics. Starting with an extension of Hennessy-Milner logic [6], we provide the syntax and the semantics of a logic for the global calculus as well as several examples pointing out the usefulness of the method. Moreover, we provide a proof system that allows for property verification of choreographies and show that it is sound.

2 The Global Calculus

The Global Calculus (GC) [3, 4] allows for the description of choreographies as interactions between participants by means of message exchanges. The description of such interactions is centered around the notion of *session*, in which two interacting parties first establish a private connection k, often referred to as *session channel*, and then interact on k, possibly interleaved with other sessions. Each session k is unique, therefore communication activities will be clearly separated.

*Authors are listed alphabetically by last name.

2.1 Syntax.

The syntax of GC [3, 4] is given by the following grammar.

$$
\begin{array}{rll}
C ::= & A \to B : a(k).\, C & \text{(init)} \\
\mid & A \to B : k\langle e, y\rangle.\, C & \text{(com)} \\
\mid & A \to B : k[l_i : C_i]_{i \in I} & \text{(choice)} \\
\mid & C_1 \mid C_2 & \text{(par)} \\
\mid & \textbf{if}\, e@A\ \textbf{then}\, C_1\ \textbf{else}\, C_2 & \text{(cond)} \\
\mid & (\nu k)\, C & \text{(newL)} \\
\mid & 0 & \text{(inaction)}
\end{array}
$$

Terms $C, C', \ldots$ are called *choreographies*; $A, B, \ldots$ range over *participants*; $k, k', \ldots$ are *session channels*; $a, b, c, \ldots$ *shared channels*; $v, w, \ldots$ variables; $X, Y, \ldots$ process variables; $l, l_i, \ldots$ labels for branching; and $e, e', \ldots$ denote arithmetic and other first-order expressions over variables and some values.

Above, (init) denotes a session initiation by A via B's service channel a, with fresh session channels k and continuation C. (com) denotes an in-session communication of e over a session channel k. Note that y does not bind in C. (choice) denotes a labelled choice over session channel k and set of labels I. $C_1 \mid C_2$ denotes the parallel product between C_1 and C_2. $(\nu k)\, C$ works the same as the name restriction operator in the π -calculus, binding k in C. Since such a hiding is only generated by session initiation, we assume that a hiding never occurs inside a prefix or a conditional. In the standard conditional operator (cond), $e@A$ indicates that e is located at participant A. 0 denotes termination. The free and bound session channels and term variables are defined in the usual way. The calculus is equipped with a standard structural congruence $\equiv$ [3].

2.2 Semantics.

We equip GC with a standard labelled transition semantics obtained by enriching the one in [3, 4]. Formally, actions in the semantics are defined using the notation $(\sigma, C) \xrightarrow{\ell} (\sigma', C')$ which says that a choreography C in a state σ (which maps participants to variable assignments) executes an action ℓ and evolves into C' with a new state σ' . Actions (or labels) are defined as $\ell = \{\mathsf{init}\ A \to B\ \mathsf{on}\ a(k),\ \mathsf{com}\ A \to B\ \mathsf{over}\ k,\ \mathsf{sel}\ A \to B\ \mathsf{over}\ k : l_i\}$, denoting initiation, in-session communication and branch selection. We write $C \longrightarrow C'$ when the states σ, σ' and ℓ are irrelevant, and $\longrightarrow^*$ for its transitive closure. A variable x located at A's is written as $x@A$. The same variable name labelled with different participant names denotes different variables (hence $\sigma@A(x)$ and $\sigma@B(x)$ may differ). The transition relation $\longrightarrow$ is defined as the minimum relation on pairs state/interaction satisfying the rules of Table 1.

In (G-INIT), after A initiates a session with B on service channel a, A and B share k locally. This is denoted by the restriction of k over the continuation C. As for communication, in (G-COM), the expression e is evaluated into v in the A-portion of the state σ and then assigned to the variable x located at B resulting in the new state $\sigma[x@B \mapsto v]$. (G-CHOICE) chooses the evolution of a choreography resulting from a labelled choice over a session key k. (G-IFT) and (G-IFF) show the possible paths that a deterministic evolution of a choreography can produce. (G-PAR), (G-RES) (G-REC) and (G-STRUCT) behave as the standard rules for parallel product, restriction, recursion and structural congruence.

Remark 1 (Global Parallel)**.** Parallel composition in the global calculus differs from the notion of parallel found in standard concurrency models based on input/output primitives. In the latter, a term $P_1 \mid P_2$ may allow *interactions* between P_1 and P_2. However, in the global calculus, the parallel composition of two choreographies $C_1 \mid C_2$ concerns two parts of the described system where *interactions* may occur in C_1 and C_2 but never across the parallel operator $\mid$. This is because an interaction $A \to B \ldots$ abstracts from

$$\text{(G-INIT)}\ \frac{}{(\sigma,\, A \to B : a(k).\, C) \xrightarrow{\text{init } A\to B \text{ on } a(k)} (\sigma, (\nu k)\, C)}$$

$$\text{(G-COM)}\ \frac{\sigma' = \sigma[x@B \mapsto v] \qquad \sigma \vdash e@A \Downarrow v}{(\sigma,\, A \to B : k\langle e,x\rangle.\, C) \xrightarrow{\text{com } A\to B \text{ over } k} (\sigma',\, C)}$$

$$\text{(G-CHOICE)}\ \frac{}{(\sigma,\, A \to B : k[l_i : C_i]_{i\in I}) \xrightarrow{\text{sel } A\to B \text{ over } k:l_i} (\sigma,\, C_i)}$$

$$\text{(G-IFT)}\ \frac{\sigma \vdash e@A \Downarrow \texttt{tt} \qquad (\sigma, C_1) \xrightarrow{\ell} (\sigma', C_1')}{(\sigma, \textbf{if}\, e@A\ \textbf{then}\, C_1\ \textbf{else}\, C_2\,) \xrightarrow{\ell} (\sigma', C_1')}$$

$$\text{(G-STRUCT)}\ \frac{C \equiv C'' \qquad (\sigma,\, C) \xrightarrow{\ell} (\sigma',\, C') \qquad C' \equiv C'''}{(\sigma,\, C'') \xrightarrow{\ell} (\sigma',\, C''')}$$

$$\text{(G-IFF)}\ \frac{\sigma \vdash e@A \Downarrow \texttt{ff} \qquad (\sigma, C_2) \xrightarrow{\ell} (\sigma', C_2')}{(\sigma, \textbf{if}\, e@A\ \textbf{then}\, C_1\ \textbf{else}\, C_2\,) \xrightarrow{\ell} (\sigma', C_2)}$$

$$\text{(G-RES)}\ \frac{(\sigma,\, C) \xrightarrow{\ell} (\sigma',\, C') \qquad u \in \{a,k\}}{(\sigma, (\nu u)\, C) \xrightarrow{\ell} (\sigma',\, (\nu u)\, C')}$$

$$\text{(G-PAR)}\ \frac{(\sigma,\, C_1) \xrightarrow{\ell} (\sigma',\, C_1')}{(\sigma, C_1 \mid C_2) \xrightarrow{\ell} (\sigma',\, C_1' \mid C_2)}$$

Table 1: Operational Semantics for the Global Calculus

the actual end-point behaviour i.e. how A sends and B receives. In our model, dependencies between two choreographies can be expressed by using variables (in the state σ).

Example 1 (Online Booking)**.** We consider a simplified version of the online booking scenario presented in [9]. Here, the customer establishes a session with the airline company AC using service *ob* (online booking) and creating session keys k_1, k_2. Once sessions are established, the customer will request the company about a flight offer with his booking data, along the session key k_1. The airline company will process the customer request and will send a reply back with an offer using the session key k_2. The customer will eventually accept the offer, sending back an acknowledgment to the airline company using k_1. The following specification in the global calculus represents the protocol:

$$\begin{aligned} C_{OB} \;=\; & Cust \to AC : ob(k_1,k_2). \\ & Cust \to AC : k_1\langle booking, x\rangle. \\ & AC \to Cust : k_2\langle offer, y\rangle. \\ & Cust \to AC : k_1\langle accept, z\rangle.\ 0 \end{aligned} \tag{1}$$

2.3 Session Types for the Global Calculus.

We use a generalisation of session types [7] for global interactions, first presented in [4]. Session types in *GC* are used to structure sequence of message exchanges in a session.

A typing judgment has the form $\Gamma \vdash C : \Delta$, where Γ, Δ are *service type* and *session type* environments, respectively. Typically, Γ contains a set of type assignments of the form $a@A : (\vec{k})\alpha$, which says that a service a located at participant A may be invoked with a fresh $\vec{k}$ followed by a session α. Δ contains type assignments of the form $\vec{k}[A,B] : \alpha$ which says that a vector of session channels $\vec{k}$, all belonging to the same session between participants A and B, has the session type α when seen from the viewpoint of A. The typing rules are omitted, and we refer to [5] for the full account of the type discipline. Returning to the example presented in Equation 1, the service type of the airline company at channel *ob* can be described as:

$$ob@AC : (k_1,k_2).\ k_1 \downarrow booking(\texttt{string}).\ k_2 \uparrow offer(\texttt{int}).\ k_1 \downarrow accept(\texttt{int}).\ \texttt{end} \tag{2}$$

3 A Logic for the Global Calculus

In this section we provide the main ingredients of our logic for choreographies *GL* (Global Logic). In Section 3.1, we introduce the reader to the syntax of *GL* and give several examples. In Section 3.2,

we present the semantics of *GL*, inspired by the modal logic presented in [1]. The logical language comprises assertions for equality, value/name passing and existential quantifiers, plus modalities for timed execution of actions.

3.1 Syntax and Examples.

Choreography assertions (ranged over by $\phi, \chi, \ldots$) give a logical interpretation of the global calculus introduced in the previous section. The grammar of assertions used in *GL* is defined as follows:

$$\phi \ ::= \quad \langle \ell \rangle \phi \quad | \quad \exists t.\phi \quad | \quad \mathtt{tt} \quad | \quad e_1 = e_2 \quad | \quad \phi \wedge \chi \quad | \quad \neg\phi \quad | \quad \circ\phi \quad | \quad \Diamond\phi \quad | \quad \phi \mid \chi$$

In $\exists t.\ \phi$, the variable t is meant to range over service and session channels, participants, labels and basic placeholders for expressions. Accordingly, it works as a binder in ϕ. In addition to the standard operators, we include an unspecified (decidable) equality on expressions $(e_1 = e_2)$ as in [1]. Our operators depend on the labels of the labelled transition system of the global calculus: $\langle \ell \rangle \phi$ represents the execution of a labelled action ℓ followed by the assertion ϕ; $\circ\phi$ and $\Diamond\phi$ denote the standard next and evenutally operators respectively. The parallel operator in $\phi \mid \chi$ denotes composition of formulae: because of the unique nature of parallel composition in choreographies, we use the symbol $\mid$ in order to stress the fact that there is no interference between two choreographies running in parallel. As usual, we can get the full account of the logic by deriving the standard set of modal operators from the syntax presented above. For example, $\mathtt{ff} = \neg\mathtt{tt}$, $(e_1 \neq e_2) = \neg(e_1 = e_2)$, $\phi \vee \chi = \neg(\neg\phi \wedge \neg\chi)$, $\phi \Rightarrow \chi = \neg\phi \vee \chi$, $\forall x.\ \phi = \neg\exists x.\ \neg\phi$, $\Box\phi = \neg\Diamond\neg\phi$, $[\ell]\phi = \neg\langle \ell \rangle \neg\phi$.

Example 2 (Availability, Service Usage and Coupling)**.** *The logic above allows to express that, given a service invoker (known as A in this setting) requesting the service a, there exists another participant (called B in the example) providing a with A invoking it. This can be formulated in GL as*

$$\exists B.\ \langle \mathit{init}\ A \rightarrow B\ \mathit{on}\ a(k) \rangle \mathtt{tt}$$

Assume now, that we want to ensure that services available are actually used. We can use the dual property for availability i.e. for a service provider B offering a, there exists someone invoking a:

$$\exists A.\ \langle \mathit{init}\ A \rightarrow B\ \mathit{on}\ a(k) \rangle \mathtt{tt}$$

Verifying that there is a service pairing two different participants in a choreography can be done by existentially quantifying over the shared channels used in an initiation action. A formula in GL representing this can be seen below:

$$\exists a.\ \langle \mathit{init}\ A \rightarrow B\ \mathit{on}\ a(k) \rangle \mathtt{tt}$$

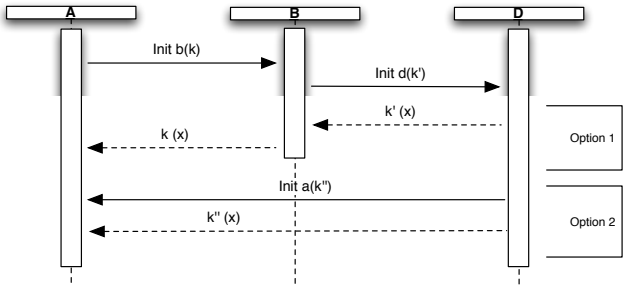
□

Example 3 (Causality Analysis)**.** *We can use the modal operators of the logic in order to perform studies of the causal properties that our specified choreography can fulfill. For instance, we can specify that given an expression e evaluated to true at participant A, there is an eventual firing of a choreography that satisfies property ϕ_1, and ϕ_2 will never be satisfied. Such a property can be specified as follows:*

$$(e@A = \mathtt{tt}) \wedge \Diamond(\phi_1) \wedge \Box\neg\phi_2$$

□

Example 4 (Response Abstraction)**.** *An interesting aspect of our logic is that it allows for the declaration of partial specification properties regarding the interaction of the participants involved in a choreography. Take for instance the interaction diagram below:*



Above, participant A invokes service b at B's and then B invokes D's service d. At this point, D can send the content of variable x to A in two different ways: either by using those originally established sessions, or by invoking a new service at A's. However, at the end of the either computation path, variable z (located at A's) will contain the value of x. In the global calculus:

$$C \;=\; A \to B : b(k).\, B \to D : d(k').\, \textbf{if}\, e@D\, \textbf{then}\, C_1\, \textbf{else}\, C_2 \quad \textit{where} \quad \begin{array}{ll} C_1 = & D \to B : k'\langle x, y_B\rangle.\, B \to A : k\langle y_B, z\rangle \\ C_2 = & D \to A : a(k'').\, D \to A : k''\langle x, z\rangle \end{array}$$

We argue that, under the point of view of A, both options are sufficiently good if, after an initial interaction with B is established, there is an eventual response that binds variable z. Such a property can be expressed in the logic by the formula:

$$\exists X, k''.\, \langle \mathit{init}\, A \to B\ \mathit{on}\ a(k)\rangle\, \Diamond\Big(\langle \mathit{com}\, X \to A\ \mathit{over}\, k''\rangle\ (z@A = x@D)\Big) \tag{3}$$

Note that a third option for the protocol above is to use delegation. However, the current version of the global calculus does not feature such an operation and we leave it as future work. □

Example 5 (Connectedness)**.** *The work in [4] proposes a set of criteria for guaranteeing a safe end-point projection between global and local specifications (note that the choreography in the previous example does not respect such properties). Essentially, a valid global specification have to fulfill three different criteria, namely Connectedness, Well-threadedness and Coherence. It is interesting to see that some of this criteria relate to global and local causality relations between the interactions in a choreography, and can be easily formalized as properties in the choreography logic here presented. Below, we consider the notion of connectedness and leave the other cases as future work. Connectedness dictates a global causality principle in interaction. If A initiates any action (say sending messages, assignment, etc) as a result of a previous event (e.g. message reception), then such a preceding event should have taken place at A. In the following, let* $\mathsf{Interact}(A,B)\phi$ *be a predicate which is true whenever* $\langle \ell \rangle \phi$ *holds for some* ℓ *with an interaction from A to B. Connectedness can then be specified as follows:*

$$\forall A, B.\, \Box\Big(\mathsf{Interact}(A,B)\mathtt{tt} \;\Rightarrow\; \exists C.\, \big(\mathsf{Interact}(A,B)\ \mathsf{Interact}(B,C)\ \mathtt{tt} \vee\ \mathsf{Interact}(A,B)\ \neg\langle \ell \rangle \mathtt{tt}\big)\Big)$$

□

3.2 Semantics of the Logic.

We now give a formal meaning to the assertions introduced above with respect to the semantics of the global calculus introduced in the previous section. In particular we introduce the notion of satisfaction.

$$[\mathsf{P}_{\mathsf{init}}]\ \frac{C \vdash_\sigma \phi}{A \to B : a(k).\, C \vdash_\sigma \langle \mathsf{init}\, A \to B \ \mathsf{on}\ a(k)\rangle \phi}$$

$$[\mathsf{P}_{\mathsf{sel}}]\ \frac{\forall_{i\in I}\quad C_i \vdash_\sigma \phi_i}{A \to B : k[l_i : C_i]_{i\in I} \vdash_\sigma \bigwedge_{i\in I}\langle \mathsf{sel}\, A \to B \ \mathsf{over}\ l_i : s\rangle \phi_i}$$

$$[\mathsf{P}_{\mathsf{com}}]\ \frac{C \vdash_\sigma \phi}{A \to B : k\langle e, y\rangle.\, C \vdash_\sigma \langle \mathsf{com}\, A \to B \ \mathsf{over}\ s\rangle \phi}$$

$$[\mathsf{P}_{\mathsf{if}}]\ \frac{C_1 \vdash_\sigma e \Rightarrow \phi \qquad C_2 \vdash_\sigma \neg e \Rightarrow \phi}{\textbf{if}\, e\ \textbf{then}\, C_1\ \textbf{else}\, C_2 \ \vdash_\sigma \phi}$$

$$[\mathsf{P}_{\mathsf{and}}]\ \frac{C \vdash_\sigma \phi \qquad C \vdash_\sigma \chi}{C \vdash_\sigma \phi \wedge \chi}$$

$$[\mathsf{P}_{\mathsf{exp}}]\ \frac{\sigma(e_1) = \sigma(e_2)}{C \vdash_\sigma e_1 = e_2}$$

$$[\mathsf{P}_{\mathsf{Inact}}]\ \frac{}{0 \vdash \mathtt{tt}}$$

$$[\mathsf{P}_{\mathsf{res}}]\ \frac{C \vdash_\sigma \phi \qquad k \text{ is fresh}}{(\nu k)\, C \vdash_\sigma \nu k.\ \phi}$$

$$[\mathsf{P}_{\mathsf{sub}}]\ \frac{C \vdash_\sigma \phi \qquad \phi \Rightarrow \chi}{C \vdash_\sigma \chi}$$

$$[\mathsf{P}_{\mathsf{par}}]\ \frac{\forall_{i\in\{1,2\}}\quad C_i \vdash_\sigma \phi_i}{C_1 \mid C_2 \vdash_\sigma \phi_1 \mid \phi_2}$$

$$[\mathsf{P}_{\exists}]\ \frac{C \vdash_\sigma \phi\{w \mapsto t\}}{C \vdash_\sigma \exists t \phi}$$

Table 3: Proof system for the Global Calculus

To guarantee the correctness of our logic, we shall prove the correspondence between the assertion semantics and the proof system. The details of the proof here presented can be found at [8].

Theorem 1 (soundness)**.** *for any given choreography C, if* $C \vdash \phi$*, then* $C \models \phi$*.*

4 Conclusion and Related Work

The ideas hereby presented constitutes just the first step towards a verification framework of structured communications. As a future work, our main concerns relate to establishing a completeness relation between the choreography logic and its proof system and the ability of integrating our framework into other end-point models and logical frameworks for the specification of sessions. In particular, our next step will focus on relating the logic to the end-point projection [4], the process of automatically generating end-point code from choreography. Other improvements to the system proposed include the use of fixed points, essential for describing state-changing loops, and auxiliary axioms describing structural properties of a choreography.

This work can be fruitfully nourished by related work in types and logics for session-based communication. In [9] the authors proposed a mapping between the calculus of structured communications and concurrent contraint programming, allowing them to establish a logical view of session-based communication and formulae in First-Order Temporal Logic. In [1], Berger et al. presented proof systems characterizing May/Must testing preorders and bisimilarities over typed π -calculus processes. The connection between types and logics in such system comes in handy to restrict the shape of the processes one might be interested, allowing us to consider such work as a suitable proof system for the calculus of end points. Finally, [10] studies a logic for choreographies in a model without services and sessions while [2] proposes notion of global assertion for enriching multiparty session types with simple formula describing changing in the state of a session.

Acknowlegments

This research has been partially supported by the Trustworthy Pervasive Healthcare Services (TrustCare) project. Danish Research Agency, Grant # 2106-07-0019 (www.TrustCare.eu).

References

[1] Martin Berger, Kohei Honda, and Nobuko Yoshida. Completeness and logical full abstraction in modal logics for typed mobile processes. In Luca Aceto, editor, *ICALP'08*, number 5126 in LNCS, pages 99–111. Springer-Verlag, Berlin Germany, 2008.

[2] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. Available at `http://www.cs.le.ac.uk/people/lb148/AssertedTypes/assertedTypesExtended.pdf`, January 2010.

[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'2007)*, volume 4421 of *LNCS*, pages 2–17. Springer, Berlin Heidelberg, March 24–April 1 2007.

[5] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A Theoretical Basis of Communication-Centred Concurrent Programming. *Web Services Choreography Working Group mailing list, to appear as a WS-CDL working report*, 2009.

[6] M. Hennessy and R. Milner. On Observing Nondeterminism and Concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309. Springer-Verlag London, UK, 1980.

[7] K. Honda, V.T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP'1998*, volume 1381 of *LNCS*, pages 122–138. Springer-Verlag London, UK, 1998.

[8] Hugo A. López. Formal models for trustworthy process and service oriented systems. Master's thesis, IT University of Copenhagen, Copenhagen, January 2009.

[9] Hugo A. López, Carlos Olarte, and Jorge A. Pérez. Towards a Unified Framework for Declarative Structured Communications. In *Programming Language Approaches to Concurrency and Communication-cEntric Software: PLACES'09*, February 2009.

[10] Carlo Montangero and Laura Semini. A logical view of choreography. In *COORDINATION*, pages 179–193, 2006.

$$
\begin{array}{lcl}
C \models_\sigma e_1 = e_2 & \iff & \sigma(e_1) = \sigma(e_2) \\
C \models_\sigma \mathtt{tt} & \iff & \mathsf{tt} \\
C \models_\sigma \phi \wedge \chi & \iff & C \models_\sigma \phi \text{ and } C \models_\sigma \chi \\
C \models_\sigma \neg\phi & \iff & C \not\models_\sigma \phi \\
C \models_\sigma \exists t.\ \phi & \iff & C[w/t] \models \phi \quad \text{(for some appropriate } w\text{)} \\
C \models_\sigma \Diamond\phi & \iff & (\sigma, C) \longrightarrow^* (\sigma', C') \text{ and } C' \models_{\sigma'} \phi \\
C \models_\sigma \langle\ell\rangle\phi & \iff & (\sigma, C) \xrightarrow{\ell} (\sigma', C') \text{ and } C' \models_{\sigma'} \phi \qquad \ell \notin \{\mathsf{init}\ A \to B \ \mathsf{on}\ a(k)\} \\
C \models_\sigma \circ\phi & \iff & (\sigma, C) \longrightarrow (\sigma', C') \text{ and } C' \models_{\sigma'} \phi \\
C \models_\sigma \langle\ell\rangle\phi & \iff & (\sigma, C) \xrightarrow{\ell} (\sigma', C') \text{ and } C' \models_{\sigma'} (\nu)\, k\phi \qquad \ell = \mathsf{init}\ A \to B \ \mathsf{on}\ a(k) \\
C \models_\sigma (\nu k)\ .\ \phi & \iff & C \equiv (\nu k) C' \text{ and } C' \models_\sigma \phi \\
C \models_\sigma \phi \mid \chi & \iff & C \equiv C_1 \mid C_2 \text{ s.t. } C_1 \models_\sigma \phi \text{ and } C_2 \models_\sigma \chi
\end{array}
$$

Table 2: Assertions of the Choreography Logic

We write $C \models_\sigma \phi$ whenever an environment σ and a choreography C satisfy a *GL* formula ϕ. The relation $\models_\sigma$ is the maximum relation satisfying the rules given in Table 2.

Above, we assume that variables occurring in an expression e are always located e.g. $x@A$. In the $\exists t.\ \phi$ case, w should be an appropriate value according to the type of t e.g. a participant if t is a participant placeholder. A formula ϕ is a "logical consequence" of a formula χ if every interpretation that makes ϕ true also makes χ true. In this case one says that χ is logically implied by ϕ ($\phi \Rightarrow \chi$). Moreover, let ϕ and χ be two formulae in the choreography logic. We say that ϕ is semantically equivalent to χ (denoted by $\phi \equiv_{\models} \chi$) if $\phi \models_\sigma \chi$ if and only if $\chi \models_\sigma \phi$. A formula is satisfiable if there is some choreography under which it is true. A formula with free variables is said to be satisfied by a choreography if the formula remains true regardless which participants, names/values are assigned to its free variables. A formula ϕ is *valid* if it is true in every choreography, that is $C \models_\sigma \phi$ for any C.

Proof System. Here, the proof system is presented. In order to reason about judgments $C \models_\sigma \phi$, we propose a proof (or inference) system for assertions of the form $C \vdash \phi$. Intuitively, we want $C \vdash_\sigma \phi$ to be as approximate as possible to $C \models_\sigma \phi$ (ideally, they should be equivalent). We write $C \vdash_\sigma \phi$ for the provability judgement where C is a process and ϕ contains a choreography formula.

We say that a choreography C *exhibits* a formula ϕ under an environment σ (written $C \vdash_\sigma \phi$) iff the assertion $C \vdash_\sigma \phi$ has a proof in the proof system given in Table 3.

Let us now describe some of the inference rules of the proof system. $\mathsf{P_{inact}}$ and is the standard rule for inaction $\mathsf{P_{sel}}$ can be explained as follows: suppose we are given a process $P = A \to B : k[l_i : C_i]_{i \in I}$, a set of branch labels $\{l_i\}$ (determined by typing) and we are given a proof that each C_i satisfies ϕ_i, then we certainly have a proof saying that every derivation of P should satisfy a guard l_i followed by a formula ϕ_i. The initiation and interaction rule $\mathsf{P_{init}}, \mathsf{P_{com}}$ behave similarly to $\mathsf{P_{sel}}$: given an initiation/communication process in P and a proof that its continuation satisfies the proof term ϕ, we can derive a proof that P will first exhibit an initiation/communication action followed by ϕ. The rules for existential quantification, evaluation of expressions and conditional operators $\mathsf{P_\exists}$, $\mathsf{P_{exp}}$, $\mathsf{P_{if}}$ are standard. The subsumption rule $\mathsf{P_{sub}}$ is the standard consequence rule as found in Hoare logic. The rules for parallel composition and hiding are represented in $\mathsf{P_{par}}$ and $\mathsf{P_{res}}$ respectively, and they do not indicate the behaviour of a given choreography, but hint information about the structure of the process: $\mathsf{P_{par}}$ juxtaposes the behaviour of two processes and combines their respective formulae by the use of a separation operator, $\mathsf{P_{res}}$ hides a variable x in a formula ϕ; the intuition is that since $(\nu k)\, C$ is a choreography C with x restricted, then if C proves ϕ and k is a fresh, then $(\nu k)\, C$ should satisfy ϕ with hidden x.

Analysing DMA Races in Multicore Software

Alastair F. Donaldson, Daniel Kroening, Philipp Rümmer*
Oxford University Computing Laboratory, Oxford, UK

Abstract

We present ongoing work on applying model checking techniques to automatically analyse multicore software where data is managed explicitly by the programmer via direct memory access (DMA) operations. We describe SCRATCH, a DMA race analysis tool for the Cell BE processor which employs bounded model checking and k-induction. We then outline our plans to extend this work.

1 Introduction

In this position paper, we present ongoing work on applying model checking techniques to automatically verify software for multicore processors. Our focus is on multicore architectures with multiple memory spaces, where it is the programmer's responsibility to orchestrate movement of data between memories in an efficient manner, to achieve high performance. Such architectures are extremely difficult to program correctly, and there is great scope for formal verification techniques to be of use.

The Cell Broadband Engine (BE) architecture [5] consists of a host core, the Power Processor Element (PPE), and a number of accelerator cores, the Synergistic Processor Elements (SPEs). The PPE is a regular processor connected to main memory, whereas each SPE is equipped with a private, 256 K "scratch-pad" memory, also known as *local memory*. The SPE local memories are not coherent with each other, or with main memory. As a result, an SPE can access its local memory very efficiently, without contention. However, an SPE thread can only access main memory using direct memory access (DMA). A DMA operation is a request to copy a chunk of data between host and local memory, and is handled asynchronously by dedicated hardware. High performance can be achieved by organising DMA operations so that computation and communication are overlapped, via buffering techniques. The price for performance is programming complexity: writing data-movement code is error-prone. Programmer errors related to DMA operations can result in memory corruption, due to multiple DMA operations simultaneously accessing the same memory (DMA races). Errors due to misuse of DMA can result in nondeterministic bugs that are difficult to consistently reproduce and fix.

In prior work, we have designed a tool, SCRATCH, to detect, or prove absence of, DMA races in SPE programs [4]. The tool uses a combination of bounded model checking [2] and k-induction [9] to automatically analyse SPE threads in isolation, checking for DMA races on local memory. After describing DMA operations in more detail (§2) we summarise our existing work (§3). We then discuss plans to extend this work (§4). Due to lack of space, we do not discuss related work in detail. Please refer to [4] for such a discussion.

2 Direct memory access operations

We consider three DMA primitives: $\mathsf{get}(l,h,s,t)$, which issues a transfer of s bytes from host memory address h to local memory address l, and is identified by tag t; $\mathsf{put}(l,h,s,t)$, which analogously transfers data from local to host memory; and $\mathsf{wait}(t)$, which blocks until all DMA operations identified by tag t have completed. On the Cell BE processor a tag is an integer in the range $0,\ldots,31$, and it is legal for

*Alastair F. Donaldson is supported by EPSRC grant EP/G051100. Daniel Kroening and Philipp Rümmer are supported by EPSRC grant EP/G026254/1, the EU FP7 STREP MOGENTES, and the EU ARTEMIS CESAR project.

```
#define CHUNK 16384 // Process data in 16K chunks

float buffers[3][CHUNK/sizeof(float)]; // Triple-buffering requires 3 buffers

void process_data(float* buf) { ... }

/* 'in' and 'out' are pointers to host memory */
void triple_buffer(char* in, char* out, int num_chunks) {
    unsigned int tags[3] = { 0, 1, 2 }, tmp, put_buf, get_buf, process_buf;

(1) get(buffers[0], in, CHUNK, tags[0]); // Get triple-buffer scheme rolling
    in += CHUNK;
(2) get(buffers[1], in, CHUNK, tags[1]);
    in += CHUNK;
(3) wait(tags[0]); process_data(buffers[0]); // Wait for and process first buffer
    put_buf = 0; process_buf = 1; get_buf = 2;
    for(int i = 2; i < num_chunks; i++) {
(4)   put(buffers[put_buf], out, CHUNK, tags[put_buf]); // Put data processed
      out += CHUNK;                                     //   last iteration
(5)   get(buffers[get_buf], in, CHUNK, tags[get_buf]);  // Get data to process
      in += CHUNK;                                      //   next iteration
(6)   wait(tags[process_buf]);              // Wait for and process data
      process_data(buffers[process_buf]);   //   requested last iteration

      tmp = put_buf; put_buf = process_buf; // Cycle the buffers
      process_buf = get_buf; get_buf = tmp;
    }
    ... // Handle data processed/fetched on final loop iteration
}
```

Figure 1: Triple-buffering example, adapted from an example provided with the IBM Cell SDK [6]

one SPE to issue up to 32 concurrent DMAs (thus each DMA can, in principle, be identified by a distinct tag). Note that DMA operations are always issued from the point of view of an accelerator core (SPE), *e.g.* get always denotes movement of data into local memory. It is usual for all DMA operations to be initiated by the SPE cores, and we restrict our attention to this scenario.

Two DMA operations are said to *race* if they are pending simultaneously, operate on a common region of (host or local) memory, and at least one modifies this region.

Example: triple-buffering. Figure 1, adapted from an example provided with the IBM Cell SDK [6], is part of an SPE program, and illustrates the use of DMA operations to stream data from host memory to local store to be processed, and to stream results back to host memory. Triple-buffering is used to overlap communication with computation: each iteration of the loop in `triple_buffer` puts results computed during the previous iteration to host memory, gets input to be processed next iteration from host memory, and processes data which has arrived in local memory.

If `num_chunks` is greater than three, this example exhibits a local memory DMA race, which we can observe by logging the first six DMA operations. To the right of each operation we record its source code location and, if appropriate, its loop iteration. We omit host address parameters, irrelevant to the race:

```
      get(buffers[0], ..., CHUNK, tags[0])   (1)
      get(buffers[1], ..., CHUNK, tags[1])   (2)
      wait(tags[0])                          (3)
(*)   put(buffers[0], ..., CHUNK, tags[0])   (4), i=2
      get(buffers[2], ..., CHUNK, tags[2])   (5), i=2
      wait(tags[1])                          (6), i=2
      put(buffers[1], ..., CHUNK, tags[2])   (4), i=3
(*)   get(buffers[0], ..., CHUNK, tags[0])   (5), i=3
```

At this point in execution the operations marked (*) race with one another: they operate on the same

local memory, the second operation modifies the memory, and is not protected by an intervening wait. The race can be avoided by inserting a wait with tag `tags[get_buf]` before the get at (5).

We discovered this bug using SCRATCH, our automatic DMA analysis tool, which can also show that the fix is correct. The bug occurs in an example provided with the IBM Cell SDK, and was, to our knowledge, previously unknown. Our bug report has been confirmed by an engineer at IBM.

3 Scratch: an automatic DMA race analyser for Cell BE software

SCRATCH (so called because it analyses scratch-pad memory) takes as input a C program written for one of the SPE cores of the Cell BE processor. The program is transformed so that DMA operations are replaced with statements to check for local memory races. In principle, the transformed program can be analysed by any tool capable of checking assertions in C programs augmented with assume statements and nondeterministic choice. In practice, SCRATCH is built on top of the bounded model checker CBMC [3], which unwinds the transformed program to check for DMA races up to a user-specified depth, using SAT techniques.

Translating DMA statements. This is achieved via an array of *DMA entry* records, called the *tracker* array. A DMA entry represents a pending DMA operation, and has the form $(valid, local, size, tag)$. The *valid* field is a bit determining whether the entry represents a pending DMA, or is unused. If $valid = 1$ then the remaining fields store the local address, size, and tag associated with the DMA, otherwise they are ignored. Note that host memory locations are *not* tracked, since SCRATCH does not currently analyse DMA races on host memory; we discuss this further in §4. The size of the tracker array is, by default, 32, the maximum number of DMAs which may be simultaneously issued by an SPE.

At the start of the program, *valid* is set to 0 for each DMA entry in the tracker array.

A DMA command of the form $\mathsf{op}(l,h,s,t)$, where op is get or put, is translated into:

1. $\mathsf{assert}([l,l+s) \cap [local, local+size) = \emptyset)$ for all DMA entries matching the pattern $(1, local, size, \_)$, *i.e.* the new DMA does not operate on the same local memory as any pending DMA;
2. an assertion that some DMA matches the pattern $(0, \_, \_, \_)$, *i.e.* at least one entry is not valid;
3. a statement replacing a nondeterministically chosen DMA entry matching the pattern $(0, \_, \_, \_)$ with $(1, l, s, t)$, *i.e.* an entry for the new DMA operation is added to the tracker array.

A DMA wait operation of the form $\mathsf{wait}(t)$ is translated into statements that replace any DMA entry matching the pattern $(1, \_, \_, t)$ with $(0, \_, \_, t)$.

In the translated program, issuing a DMA that races with an already pending DMA results in an assertion failure due to 1 above; an attempt to issue more than the maximum number of allowed DMAs also results in a failed assertion due to 2. The above translation is actually too strict: it prohibits concurrent, overlapping put operations, which cannot lead to local memory races. To avoid this limitation, each DMA entry is extended with a flag indicating whether an operation is a put, and the assertion in 1 is modified to ignore simultaneous put operations.

Proving absence of DMA races. While bounded model checking is good at finding bugs, it cannot be used in isolation to prove the absence of bugs. As well as detecting DMA races, we are interested in proving their absence. SCRATCH achieves this goal on many practical examples using a novel formulation of k-induction, a technique first introduced in [9].

To verify absence of DMA races for a transformed program consisting of a single *while* loop with prologue α, condition c, body β and epilogue γ, where β does not contain nested loops, SCRATCH solves a series of verification problems using bounded model checking. For increasing values of k, starting with $k = 0$, a base case and a step case are checked:

- **Base case:** α; $\underbrace{\mathsf{if}(c)\ \{\beta\}\ldots\mathsf{if}(c)\ \{\beta\}}_{k \text{ times}}$ $\mathsf{if}(!c)\ \{\gamma\}$
- **Step case:** havoc; $\underbrace{\mathsf{assume}(c);\ \beta_{\mathsf{assume}};\ldots;\ \mathsf{assume}(c);\ \beta_{\mathsf{assume}}}_{k \text{ times}}$; $\mathsf{if}(c)\ \{\beta\}\ \mathsf{else}\ \{\gamma\}$

The base case consists of the loop unwound k times. A base case failure yields a counterexample exposing a DMA race; otherwise we know that a DMA race cannot occur within k loop iterations.

In the step case, havoc sets every program variable to a nondeterministic value. For a boolean expression e, $\mathsf{assume}(e)$ at program point p tells the model checker to cease exploring an execution trace if e does not hold at p; β_{assume} denotes the sequence β with assert replaced by assume throughout. The step case succeeds if, from *any* potential state, if it is possible to execute k loop iterations without encountering a DMA race then it must be possible to execute one more iteration (if c still holds), or execute the loop epilogue (if c does not hold), without a DMA race occurring.

If there is some k for which both the base case and step case hold, the theory of k-induction guarantees that a DMA race can never occur. If the base case holds but the step case fails, a larger value of k must be considered. Our formulation of k-induction is the first to operate at the loop level; other presentations of the technique work at a finer granularity by unwinding the transition relation.

There is no guarantee that k-induction will terminate with a conclusive result for a feasibly small value of k. However, we find that the technique works well in practice for DMA race analysis. Intuitively, this is because DMA operations in loops are typically designed to be pending for only a bounded number of loop iterations, allowing k-induction to succeed with a value of k proportional to the bound. This is analogous to the intuition that k-induction works well for sequential hardware circuits with pipelines, where the k required for induction to succeed is proportional to the pipeline depth [1].

Experimental summary. We have evaluated SCRATCH using a set of 22 benchmarks, adapted from the IBM Cell SDK [6]. For correct and buggy version of each example, SCRATCH is able to find the DMA race, or prove absence of DMA races. Bug finding is fast, as one would expect from bounded model checking. More interestingly, for 15 of the benchmarks, correctness can be proved using k-induction in less than 10 seconds on a state-of-the-art platform, with $k \leq 5$ in all cases. One benchmark requires a verification time of 7 minutes with $k = 10$. A full discussion of experimental results is given in [4], including a comparison with model checking approaches based on predicate abstraction, and with a run-time race checking tool from IBM.

4 Further opportunities for formal verification

We plan to extend this work in several ways.

Multi-loop k-induction. SCRATCH can currently only employ k-induction to programs consisting of a *single*, non-nested loop. While we were able to verify many interesting examples with this restriction (in some cases by manually slicing away inner data-processing loops) the limitation is, in general, rather restrictive. It is always possible to transform a nest of loops into one monolithic loop, using a program counter variable and conditional statements to simulate the nesting. We have applied this transformation manually on Cell BE examples containing one nested loop, and found that k-induction succeeds, thus we plan to automate the transformation and explore larger examples. The drawback is that if one nested loop would require a large value of k to prove correct in isolation then this value will dominate the necessary unwinding of the (potentially very large) monolithic loop. Alternatively, if k-induction is viewed as a proof rule operating on loops, as in [4], the proof rule can be applied recursively to a loop nest. This approach involves solving a possibly large series of verification problems, but has the advantage that each problem can be solved using the smallest possible k. We plan to explore, and compare, both approaches.

Other extensions related to k-induction include using abstract interpretation to strengthen loop invariants, and exploring the success of k-induction in software verification more generally.

Inter-thread interference. SCRATCH currently checks for DMA races on local memory only. This simplifies the verification problem, allowing the tool to analyse a single SPE thread in isolation. We are keen to tackle the more challenging problem of checking for interference between threads running on separate cores. For the Cell BE processor this corresponds to checking for DMA races on host memory. One approach to verifying these kinds of properties is to use abstract interpretation to under-/over-approximate the memory regions accessed by individual threads, determining presence/absence of DMA races if these regions do/do not overlap. A related, but possibly more precise strategy for proving non-interference is to derive invariants for the individual threads (or an invariant for the system) that imply the absence of DMA races. Such invariants could be inferred by means of interpolation-based model checking [8]: showing via bounded model checking that threads do not interfere up to a certain search depth, using the unsatisfiability proof of the BMC formula to derive an interpolant, and iterating the BMC/interpolant-computation process until an interpolant is computed that is an inductive invariant.

Pointer validity and alignment. The Cell BE processor requires that source and target pointers for DMA operations are aligned to appropriate byte boundaries (16 for correctness, 128 for efficiency). Misalignment is a common source of DMA errors, and it can be hard to track down the cause of a misaligned pointer. In addition, the lack of separation between host and local pointers at the type level means that errors can occur, *e.g.* if the programmer accidentally passes a host pointer where a local pointer is required. We plan to consider two solutions to this problem: using type reconstruction to infer alignment and memory space properties of pointers, and tracking pointer information during bounded model checking, automatically adding assertions to check for invalid or misaligned pointer parameters to DMA operations.

Support for OpenCL. OpenCL [7] is a new, open standard language for programming heterogeneous multicore architectures including the Cell BE, and graphics processing units. OpenCL includes asynchronous memory copy primitives, which are similar to DMA transfers: the same programming problems arise, and it appears that similar solutions should be applicable. Thus, we plan to build a verification tool for OpenCL programs based on the technology behind SCRATCH.

References

[1] Roy Armoni, Limor Fix, Ranan Fraer, Scott Huddleston, Nir Piterman, and Moshe Y. Vardi. SAT-based induction for temporal safety properties. *Electr. Notes Theor. Comput. Sci.*, 119(2):3–16, 2005.

[2] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[3] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[4] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *TACAS*, LNCS. Springer, 2010. To appear.

[5] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA*, pages 258–262. IEEE Computer Society, 2005.

[6] IBM. Cell BE resource center, 2009. http://www.ibm.com/developerworks/power/cell/.

[7] Khronos Group. The OpenCL specification. http://www.khronos.org/opencl.

[8] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[9] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, volume 1954 of *LNCS*, pages 108–125. Springer, 2000.

A Type System for Unstructured Locking that Guarantees Deadlock Freedom without Imposing a Lock Ordering

Prodromos Gerakios Nikolaos Papaspyrou Konstantinos Sagonas
School of Electrical and Computer Engineering, National Technical University of Athens, Greece
{pgerakios,nickie,kostis}@softlab.ntua.gr

Abstract

Deadlocks occur in concurrent programs as a consequence of cyclic resource acquisition between threads. In this paper we present a novel type system that guarantees deadlock freedom for a language with references, unstructured locking primitives, and locks which are implicitly associated with references. The proposed type system does not impose a strict lock acquisition order and thus increases programming language expressiveness.

1 Introduction

Lock-based synchronization may give rise to deadlocks. Two or more threads are deadlocked when each of them is waiting for a lock that is acquired by another thread. Several type systems have been proposed [5, 2, 9, 10, 11] that prevent deadlocks by imposing a strict (non-cyclic) lock-acquisition order that must be respected throughout the entire program. This approach greatly limits programming language expressiveness as many correct programs are rejected unnecessarily. Boudol has recently proposed a type system that avoids deadlocks and is more permissive than existing approaches [1]. However, his system can only deal with programs that use lexically-scoped locking primitives.

In this paper we sketch a simple language with functions, mutable references, explicit (de-)allocation constructs and unstructured (i.e., non lexically-scoped) locking primitives. To avoid deadlocks, we propose a type system for this language based on Boudol's idea. We argue that the addition of unstructured locking primitives makes Boudol's system unsound and show that it is possible to regain soundness by preserving more information about the order of events both statically and dynamically.

Our work is part of a more general effort to design a low-level language suitable for systems programming [6, 7] that not only guarantees deadlock freedom but also memory safety, race freedom and definite release of resources such as memory and locks.

2 Deadlock Freedom and Related Work

We start by providing a concrete definition of deadlocks and compare our work with existing static approaches to deadlock freedom. According to Coffman *et al.* [4], a set of threads reaches a *deadlocked state* when the following conditions hold:

- *Mutual exclusion*: Threads claim exclusive control of the locks that they acquire.
- *Hold and wait*: Threads already holding locks may request (and wait for) new locks.
- *No preemption*: Locks cannot be forcibly removed from threads; they must be released explicitly by the thread that acquired them.
- *Circular wait*: Two or more threads form a circular chain, where each thread waits for a lock held by the next thread in the chain.

Therefore, deadlock freedom can be guaranteed by denying at least one of the above conditions *before* or *during* program execution. Coffman has identified three strategies that guarantee deadlock-freedom:

- *Deadlock prevention*: At each point of execution, *ensure* that at least one of the above conditions is not satisfied. Thus, programs that fall into this category are correct by design.
- *Deadlock detection and recovery*: A dedicated observer thread *determines* whether the above conditions are satisfied and preempts some of the deadlocked threads, releasing (some of) their locks, so that the remaining threads can make progress.
- *Deadlock avoidance*: Using advance information regarding thread resource allocation, *determine* whether granting a lock will bring the program to an *unsafe* state, i.e., a state which can result in deadlock, and only grant locks that lead to safe states.

The majority of literature for language-based approaches to deadlock freedom falls under the first two strategies. In the deadlock prevention category, one finds type and effect systems [5, 2, 9, 10, 11] that guarantee deadlock freedom by statically enforcing a global lock-acquisition ordering that must be respected by all threads. In this setting, starting with the work of Flanagan and Abadi [5], lock handles are associated with type-level lock names via the use of singleton types. Thus, handle lk_ι is of type $\mathtt{lk}(\iota)$. The same applies to lock handle variables. The effect system tracks the order of lock operations on handles or variables and determines whether all threads acquire locks in the same order.

Using a strict lock acquisition order is a constraint we want to avoid. It is not hard to come up with an example that shows that imposing a partial order on locks is too restrictive. The simplest of such examples can be reduced to program fragments of the form:

$$(\mathtt{lock}\ x\ \mathtt{in}\ \ldots\ \mathtt{lock}\ y\ \mathtt{in}\ \ldots) \parallel (\mathtt{lock}\ y\ \mathtt{in}\ \ldots\ \mathtt{lock}\ x\ \mathtt{in}\ \ldots)$$

In a few words, there are two parallel threads which acquire two different locks, x and y, in reverse order. When trying to find a partial order $\leq$ on locks for this program, the type system or static analysis tool will deduce that $x \leq y$ must be true, because of the first thread, and that $y \leq x$ must be true, because of the second. Thus, the program will be rejected, both in the system of Flanagan and Abadi which requires annotations [5] and in the system of Kobayashi which employs inference [9] as there is no single lock order for *both* threads. Similar considerations apply to the more recent works of Suenaga [10] and Vasconcelos *et al.* [11] dealing with non lexically-scoped locks.

Recently, Boudol developed a type and effect system for deadlock freedom [1], which is based on *deadlock avoidance*. The effect system calculates for each expression the set of acquired locks and annotates lock operations with the "future" lockset. The runtime system utilizes the inserted annotations so that each lock operation can only proceed when its "future" lockset is unlocked. The main advantage of Boudol's type system is that it allows a larger class of programs to type check and thus increases the programming language expressiveness as well as concurrency by allowing arbitrary locking schemes.

The previous example can be rewritten in Boudol's language as follows, assuming that the only lock operations in the two threads are those visible:

$$(\mathtt{lock}_{\{y\}}\ x\ \mathtt{in}\ \ldots\ \mathtt{lock}_{\emptyset}\ y\ \mathtt{in}\ \ldots) \parallel (\mathtt{lock}_{\{x\}}\ y\ \mathtt{in}\ \ldots\ \mathtt{lock}_{\emptyset}\ x\ \mathtt{in}\ \ldots)$$

This program is accepted by Boudol's type system which, in general, allows locks to be acquired in *any* order. At runtime, the first lock operation of the first thread must ensure that y has not been acquired by the second (or any other) thread, before granting x (and symmetrically for the second thread). The second lock operations need not ensure anything special, as the future locksets are empty.

The main disadvantage of Boudol's work is that locking operations have to be lexically-scoped. As it will be shown, his type and effect system cannot guarantee deadlock freedom for unscoped locking operations. In the section that follows, we discuss a novel type system for a simple language with mutable references, that is intended to guard against deadlocks and, taking advantage of our previous work [6], against race conditions and memory violations as well.

```
let  f = λx.λy.λz.  lock_{y} x;     x := x + 1;
                    lock_{z} y;     y := y + x;
                    unlock x;
                    lock_∅ z;       z := z + y;
                    unlock z;
                    unlock y
in   f a a b
```

(a) before substitution

```
lock_{a} a;   a := a + 1;
lock_{b} a;   a := a + a;
unlock a;
lock_∅ b;     b := b + a;
unlock b;
unlock a
```

(b) after substitution

Figure 1: An example program, which is well typed before substitution (a) but not after (b).

3 Type System Overview

In this section, we sketch a type system that guarantees absence of deadlocks in a language supporting non lexically-scoped locking operations. As mentioned earlier, Boudol's proposal does not support unstructured locking; even if his language had `lock`/`unlock` constructs, instead of `lock`...`in`..., Boudol's type system is not sufficient to guarantee deadlock freedom. The example program in Figure 1(a) will help us see why: It updates the values of three shared variables, x, y and z, making sure at each step that only the strictly necessary locks are held.

In our naïvely extended (and broken, as will be shown) version of Boudol's type and effect system, the program in Figure 1(a) will type check. The future lockset annotations of the three locking operations in the body of f are $\{y\}$, $\{z\}$ and $\emptyset$, respectively. (This can be easily verified by observing the lock operations between a specific `lock` and `unlock` pair.) Now, function f is used by instantiating both x and y with the same variable a, and instantiating z with a different variable b. The result of this substitution is shown in Figure 1(b). The first thing to notice is that, if we want this program to work in this case, locks have to be *re-entrant*. This roughly means that if a thread holds some lock, it can try to acquire the same lock again; this will immediately succeed, but then the thread will have to release the lock *twice*, before it is actually released.

Even with re-entrant locks, however, it is easy to see that the program in Figure 1(b) does not type check with the present annotations. The first `lock` for a now matches with the *last* (and not the first) `unlock`; this means that a will remain locked during the whole execution of the program. In the meantime b is locked, so the future lockset annotation of the first `lock` should contain b, but it does not. (The annotation of the second `lock` contains b, but blocking there if lock b is not available does not prevent a possible deadlock; lock a has already been acquired.) So, the technical failure of our naïvely extended language is that the preservation lemma breaks. From a more pragmatic point of view, if a thread running in parallel already holds b and, before releasing it, is about to acquire a, a deadlock can occur. The naïve extension also fails for another reason: Boudol's system is based on the assumption that calling a function cannot affect the set of locks that are held. This is obviously not true, if non lexically-scoped locking operations are to be supported.

The type and effect system proposed in this paper supports unstructured locking, by preserving more information at the effect level. Instead of treating effects as unordered collections of locks, our type system precisely tracks effects as an order of `lock` and `unlock` operations, without enforcing a strict lock-acquisition order. A *continuation effect* of a term represents the effect of the function code succeeding that term. In our approach, lock operations and application terms are annotated with a continuation effect. At runtime, when a function application redex is evaluated, its continuation effect is pushed on the stack. When a `lock` operation is evaluated, the future lockset is calculated by inspecting its continuation effect and (if necessary) the lookup proceeds with the continuation effects of the enclosing context. The `lock` operation succeeds only when both the lock and the future lockset are available.

Figure 2 illustrates the same program as in Figure 1, except that locking operations are now annotated

```
let  f = λx.λy.λz.  lock[y+,x−,z+,z−,y−] x;   x := x + 1;
                    lock[x−,z+,z−,y−] y;      y := y + x;
                    unlock x;
                    lock[z−,y−] z;            z := z + y;
                    unlock z;
                    unlock y
in   f a a b
```

(a) before substitution

```
lock[a+,a−,b+,b−,a−] a;   a := a + 1;
lock[a−,b+,b−,a−] a;      a := a + a;
unlock a;
lock[b−,a−] b;            b := b + a;
unlock b;
unlock a
```

(b) after substitution

Figure 2: The program of Figure 1 with continuation effect annotations; now well typed in both cases.

Expression	e	::=	$x \mid \mathtt{c} \mid f \mid (e\ e)^{\xi} \mid (e)[\rho] \mid e := e$
		|	$\mathtt{deref}\, e \mid \mathtt{let}\, \rho, x = \mathtt{ref}\, e\, \mathtt{in}\, e$
		|	$\mathtt{share}\, e \mid \mathtt{release}\, e \mid \mathtt{lock}_{\gamma}\, e$
		|	$\mathtt{unlock}\, e \mid ()$
Function	f	::=	$\lambda x.e\ \mathtt{as}\ \tau \xrightarrow{\gamma} \tau \mid \Lambda\rho.f$
Type	τ	::=	$b \mid \langle\rangle \mid \tau \xrightarrow{\gamma} \tau$
		|	$\forall\rho.\tau \mid \mathtt{ref}(\tau,\rho)$
Calling mode	ξ	::=	$\mathsf{seq}(\gamma) \mid \mathsf{par}$
Capability	κ	::=	$n, n \mid \overline{n,n}$
Effect	γ	::=	$\emptyset \mid \gamma, \rho^{\kappa}$

Figure 3: Language syntax.

with continuation effects. For example, the annotation $[y+, x-, z+, z-, y-]$ at the first `lock` operation means that in the future (i.e., after this lock operation) y will be acquired, then x will be released, and so on. If x and y were different, the runtime system would deduce that between this `lock` operation on x and the corresponding `unlock` operation, only y is locked, so the future lockset in Boudol's sense would be $\{y\}$. On the other hand, if x and y are instantiated with the same a, the annotation becomes $[a+, a-, b+, b-, a-]$ and the future lockset that is calculated is now the correct $\{a, b\}$. In a real implementation, there are several optimizations that can be performed (e.g., pre-calculation of effects) but we do not deal with them in this paper.

4 Formalism

The syntax of our language is illustrated in Figure 3, where x and ρ range over term and "region" variables, respectively. Similarly to our previous work [6, 7], a region is thought of as a memory unit that can be shared between threads and whose contents can be atomically locked. In this paper, we make the simplistic assumption that there is a one-to-one correspondence between regions and memory cells, but this is of course not necessary. The language supports explicit location polymorphism. Monomorphic functions must be annotated with their type, which carries their overall effect. Application is annotated with a *calling mode* which differentiates normal (sequential) application from parallel application, i.e., the spawning of a new thread. Sequential application is further annotated with the *continuation effect*, as mentioned earlier. The construct $\mathtt{let}\ \rho, x = \mathtt{ref}\ e_1\ \mathtt{in}\ e_2$ allocates a fresh cell, initializes it to e_1, and associates it with variables ρ and x within expression e_2. As in other approaches, we use ρ as the type-level representation of the new cell. The type of reference variables x is the singleton type $\mathtt{ref}(\rho, \tau)$, where τ is the type of the cell's contents. This allows the type system to connect x and ρ and thus to statically track uses of the new cell. Assignment and dereference operators are standard. Notice that our language does not support recursion.[1]

At any given program point, each cell is associated with a *capability*, which roughly consists of two

[1] Even if our language supported recursion, the type and effect system described in this paper would only be able to type-check recursive functions that do not contain lock and unlock primitives. We are currently working on an extended language that fully supports recursion, whose type and effect system is significantly different in the way that a function's effect is calculated.

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2 \,\&\, (\gamma_3;\gamma') \qquad \xi \vdash \gamma_a \qquad \gamma_2 = \gamma \oplus \gamma_a \\ \Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_2;\gamma_3) \qquad \xi = \mathsf{seq}(\gamma) \vee (\xi = \mathsf{par} \wedge \tau_2 = \langle\rangle)\end{array}}{\Delta;\Gamma \vdash (e_1\ e_2)^{\xi} : \tau_2 \,\&\, (\gamma;\gamma')} \ \textit{(T-A)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \mathtt{ref}(\tau,\rho) \,\&\, (\gamma_1;\gamma') \\ \Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma_1) \qquad \gamma(\rho) \geq (1,1)\end{array}}{\Delta;\Gamma \vdash e_1 := e_2 : \langle\rangle \,\&\, (\gamma;\gamma')} \ \textit{(T-AS)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \tau_1 \,\&\, (\gamma_2 \setminus \rho;\gamma') \qquad \gamma_1 = \gamma_2, \rho^{1,1} \\ \Delta \vdash \tau \qquad \Delta,\rho;\Gamma, x : \mathtt{ref}(\tau_1,\rho) \vdash e_2 : \tau \,\&\, (\gamma, \rho^{0,0};\gamma_1)\end{array}}{\Delta;\Gamma \vdash \mathtt{let}\ \rho, x = \mathtt{ref}\ e_1\ \mathtt{in}\ e_2 : \tau \,\&\, (\gamma;\gamma')} \ \textit{(T-NG)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,\rho) \,\&\, (\gamma, r^{\kappa-(0,1)};\gamma') \\ \kappa \geq (1,1) \qquad \gamma(\rho) = \kappa\end{array}}{\Delta;\Gamma \vdash \mathtt{lock}_{\gamma}\, e : \langle\rangle \,\&\, (\gamma;\gamma')} \ \textit{(T-LK)}$$

Figure 4: Selected typing rules.

natural numbers: the *capability counts*. The first number is the *cell reference* count (n_1), which denotes whether the cell is live, and the second is the *cell lock* count (n_2), which denotes whether the cell has been locked by the current thread. (We use natural numbers, instead of booleans, to support sharing and re-entrant locks.) Furthermore, a capability can be *impure* (denoted by $\overline{n_1, n_2}$), which allows for cell aliasing in the same spirit as in fractional permissions [3]. This aliasing information is required to determine whether it is safe to pass lock capabilities to new threads. The remaining language constructs operate on a cell reference and modify its capability: `share` and `release` increase and decrease n_1, respectively, whereas `lock` and `unlock` do the same for n_2. As mentioned in the previous section, the runtime system inspects the annotation on `lock` to determine whether it is safe to lock a cell.

We now briefly discuss the most interesting parts of our type and effect system. Effects are used to statically track cell capabilities. An effect is an *ordered list* of elements of the form ρ^{κ} and represents a sequence of operations that affect the capabilities of various cells. The typing relation is denoted by $\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$, where Δ and Γ form the typing context, γ is the input effect, and γ' is the output effect. The reader should bear in mind two deviations from standard practice in type and effect systems. First, as each lock operation must be annotated with the future lockset, *effects flow backwards* through typing: the input effect to expression e represents the operations that follow after e is evaluated, and the output effect is the combined effect of the expression *and* its future. Second, as effects must reflect the exact order of cell operations, typing rules do not update effects, but rather append to them. Therefore, the input effect is *always* a prefix of the output effect.

A few selected typing rules are given in Figure 4. The typing rule for function application (*T-A*) joins the input effect γ and the function's effect γ_a, which contains the entire history of events occurring in the function body. In the case of parallel application, the function's return type must be unit, whereas in sequential application the annotation is checked against the input effect. The premise $\xi \vdash \gamma_a$ enforces a number of soundness restrictions, e.g., that pure capabilities are not aliased. In the rule for assignment (*T-AS*), the premises ensure that the referenced cell has positive reference and lock counts; in other words, that ρ is live and locked after the evaluation of e_1 and e_2. The rule for the `lock` operator (*T-LK*) checks that the annotation matches the input effect. It also checks that the cell is locked *after* the `lock` operation and makes sure to remove one from the lock count, in the output effect. Finally, the rule for creating new cells (*T-NG*) checks that the new cell is properly released (and unlocked) in the input effect, and makes sure to initialize the new cell with capability $(1,1)$, before the evaluation of e_2 starts.

For well typed programs, the safety theorem in our system guarantees three things: *memory safety* (and definite release of memory resources), *race freedom* (and definite release of locks), and *deadlock freedom*. A full formalization for our language, containing the operational semantics and a proof sketch, are given in the companion technical report [8].

References

[1] G. Boudol. A deadlock-free semantics for shared memory concurrency. In M. Leucker and C. Morgan, editors, *Proceedings of the International Colloquium on Theoretical Aspects of Computing*, volume 5684 of *LNCS*, pages 140–154. Springer, 2009.

[2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, New York, NY, USA, Nov. 2002. ACM Press.

[3] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[4] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[5] C. Flanagan and M. Abadi. Object types against races. In J. C. M. Baeten and S. Mauw, editors, *Concurrency Theory: Proceedings of the 10th International Conference*, volume 1664 of *LNCS*, pages 288–303. Springer, 1999.

[6] P. Gerakios, N. Papaspyrou, and K. Sagonas. A concurrent language with a uniform treatment of regions and locks. In A. R. Beresford and S. Gay, editors, *PLACES 2009*, volume 17 of *EPTCS*, pages 79–93, 2010.

[7] P. Gerakios, N. Papaspyrou, and K. Sagonas. Race-free and memory-safe multithreading: Design and implementation in Cyclone. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 15–26, New York, NY, USA, 2010. ACM Press.

[8] P. Gerakios, N. Papaspyrou, and K. Sagonas. A type system for unstructured locking that guarantees deadlock freedom without imposing a lock ordering. Technical report, National Technical University of Athens, 2010.

[9] N. Kobayashi. A new type system for deadlock-free processes. In C. Baier and H. Hermanns, editors, *CONCUR 2006*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

[10] K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In G. Ramalingam, editor, *Asian Symposium on Programming Languages and Systems*, volume 5356 of *LNCS*, pages 155–170. Springer, 2008.

[11] V. Vasconcelos, F. Martin, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In A. R. Beresford and S. Gay, editors, *PLACES 2009*, volume 17 of *EPTCS*, pages 95–109, 2010.

Distributed Dynamic Condition Response Structures

Thomas Hildebrandt Raghava Rao Mukkamala
{hilde,rao}@itu.dk
IT University of Copenhagen
Programming, Logic and Semantics Group
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark

Abstract

We present *distributed dynamic condition response structures* as a declarative process model inspired by the workflow language employed by our industrial partner and conservatively generalizing labelled event structures. The model adds to event structures the possibility to 1) finitely specify repeated, possibly infinite behavior, 2) finitely specify fine-grained acceptance conditions for (possibly infinite) runs based on the notion of responses and 3) distribute events via roles. We give a graphical notation inspired by related work by van der Aalst et al and formalize the execution semantics as a labelled transition system. Exploration of the relationship between dynamic condition response structures and traditional models for concurrency, application to more complex scenarios, and further extensions of the model is left to future work.

1 Introduction

A key difference between declarative and imperative process languages is that the control flow for the first kind is defined *implicitly* as a set of constraints or rules, and for the latter is defined *explicitly*, e.g. as a flow diagram or a sequence of state changing commands.

There is a long tradition for using declarative logic based languages to schedule transactions in the database community. Several authors have noted that it could be an advantage to also use a declarative approach to specify workflow and business processes [4, 8, 9, 5, 1]. An important motivation for considering a declarative approach is to achieve more flexible process descriptions [11]. The increased flexibility is obtained in two ways: Firstly, imperative descriptions tend to over-constrain the control flow, since one does not think of all possible ways of fulfilling the intended constraints. Secondly, adding a new constraint to an imperative process description may require that the process code is completely rewritten, while the declarative approach just requires the extra constraint to be added.

As a simple example, consider a hospital workflow with a single rule stating that the doctor must sign after having added a prescription of medicine to the patient record. A naive imperative process description may instruct the doctor first to prescribe medicine and then sign it. In this way the possibility of adding several prescriptions before or after signing is lost, even if it is perfectly legal according to the declaratively given rule. With respect to the second type of flexibility, consider adding the rule that a nurse should give the prescribed medicine to the patient, but it is not allowed before the patient record has been signed. For the simple imperative solution, one may be led to just adding a command in the end of the program instructing the nurse to give the medicine. Perhaps we remember to insert a loop to allow that the nurse give the medicine repeatedly. But the nurse should be allowed to give medicine as soon as the first signature is put and the doctor should also be allowed to add new prescriptions after or even at the same time as the nurse gives the medicine. So, the most flexible imperative description should in fact spawn a new thread for the nurse after the first signature has been given. One may argue that the rules are too lax in this setting, i.e. that one would need stricter rules to govern the medication. However, besides the fact that this example is indeed extracted from a real-life study of paper-based oncology workflow at danish hospitals [6, 7], the main point is that this is an example of how workflows in general often are intended to be lax and flexible, not this workflow in particular.

A drawback of the declarative approach however, is that the implicit definition of the control flow makes the flow less easy to perceive for the user or compute by the execution engine. At each state, one has to solve the set of constraints to figure out what are the next possible events. It becomes even worse if you are not only interested in knowing the immediate next event, but also want to get an overview of the complete run of the process.

This motivates researching the problem of finding an expressive declarative process model language that can be easily visualized by the end user, allows an effective run-time scheduling and can be mapped easily to a state based model if an overview of the flow graph is needed. In this paper, we propose a new such declarative process model language called *dynamic condition response structures*. The model is inspired by the declarative *process matrix* model language [6, 7] used by our industrial partner and (labelled) prime event structures [12]. Indeed, it is formally a conservative generalization and strict extension of both event structures and the core primitives of the process matrix model language.

An (labelled, prime) event structure in some sense can be regarded as a minimal, declarative model for concurrent processes. It consists of a set of events, a causality (partial order) relation between events stating which events are caused by the previous events (or dually, which events must have preceded the execution of an event), a conflict relation stating which events can not happen in the same execution and finally a labeling function describing the observable action name of each event.

To be used as an execution language for workflow or concurrent (multi-processor) systems several aspects are missing however. In this paper we consider three of these aspects: Firstly, we need some compact, still declarative, way to model *repeated*, possibly infinite behavior. In an event structure each event can only be executed once. Secondly, it must be possible to specify that only *some* of the partial (or infinite) computations are *acceptable*. Event structures have no notion of acceptance condition. Finally, we need to be able to describe a *distribution* of events on agents/persons/processors.

To address these aspects, we propose a number ways to generalize event structures. Firstly, we allow each event to happen many times and replace the symmetric conflict relation by an asymmetric relation which *dynamically* determines which events are included in or excluded from the structure. Secondly, the causality relation is split in two relations (not necessarily partial orders): A *condition* relation stating which events must have happened before an event and a *response* relation stating which events must happen after (as a response to) an event. We can then define runs to be acceptable if no response event from some point in the execution is executable continuously without ever being executed. This relates to the elegant definition of fair runs in true concurrency models investigated in [2]. Finally, we define *distribution* by adding a set of roles assigned to persons/processors and actions.

Being based on essentially only four relations between events, the model can be simply visualized as a directed graph with events (labelled by activities and roles) as nodes and four different kinds of arrows. We found that our condition and response relations were two of the core LTL templates used in [11] and thus decided to base our graphical notation on the one suggested in [11].

We also provide a relatively simple mapping to the state based model of labelled transition systems, which formalizes the semantics. We show how run-time scheduling for workflows with finite runs can easily be supported by identifying accepting states in the labelled transition system This gives a finite state automaton that reflects the run-time scheduling of the process matrix model used by our industrial partner. We leave the treatment of *infinite* runs for future work.

The main advantage of the dynamic condition response structures compared to the related work based on Event logics, Concurrent transactional logic and temporal logics such as LTL explored in [10, 11, 4, 3] is that the latter logics are more general and thus, we claim, more complex to visualize and understand by people not trained in logic.

2 Distributed Dynamic Condition Response Structures

Let us first recall the definition of a prime event structure and configurations of such [12].

Definition 1. *A labeled* prime event structure *(*ES*) over an alphabet Act is a 4-tuple* $(E, \leq, \#, l)$ *where*

(i) E is a (possibly infinite) set of events

(ii) $\leq \subseteq E \times E$ is the causality relation between events which is partial order

(iii) $\# \subseteq E \times E$ is a binary conflict relation between events which is irreflexive and symmetric

(iv) $l : E \to Act$ is the labeling function mapping events to actions

Action names $a \in Act$ represent the actions the system might perform, an event $e \in E$ labelled with a represents occurrence of action a during the possible run of the system. The causality relation $e \leq e'$ means that event e is a prerequisite for the event e' and the conflict relation $e\#e'$ implies that events e and e' both can not happen in the same run, more precisely one excludes the occurrence of the other. The causality and conflict relations satisfy the conditions that $e\#e' \leq e'' \implies e\#e''$ and $\{e' \mid e' \leq e\}$ is finite for any $e \in E$. A configuration c is a set of events such that,

(i) conflict-free: $\forall e, e' \in c. \neg e\#e'$

(ii) downwards-closed: $\forall e \in c, e' \in E. e' \leq e \implies e' \in c$

We define a run of a labelled event structure to be a sequence of labelled events $(e_0, l(e_0)), (e_1, l(e_1)), \ldots$ such that $\{e \mid e \leq e_0\} = \emptyset$ and for all $i \geq 0. \cup_{0 \leq j \leq i} \{e_j\}$ is a configuration.

As an intermediate step towards dynamic condition response structures we generalize prime event structures to (prime) condition response event structures by replacing the causality relation with two relations: the condition and the response relation, as described in the introduction.

Definition 2. A labeled *condition response event structure* (CRES) over an alphabet *Act* is a tuple $(E, \leq_C, \leq_R, \#, l)$ where

(i) E is a (possibly infinite) set of events

(ii) $\leq_C \subseteq E \times E$ is the *condition* relation between events which is partial order

(iii) $\leq_R \subseteq E \times E$ is the *response* relation between events, satisfying that $\leq = \leq_C \cup \leq_R$ is a partial order

(iv) $\# \subseteq E \times E$ is a binary *conflict* relation between events which is irreflexive and symmetric

(v) $l : E \to Act$ is the labeling function mapping events to actions

The *condition* relation imposes a precedence relation between events. For example, if two events are related by the *condition* relation $e \leq_C e'$, then event e must have happened before event e' can happen. As for the causality relation in prime event structures we require that $e\#e' \leq e'' \implies e\#e''$ and $\{e' \mid e' \leq e\}$ is finite for any $e \in E$. We define configurations and runs as for prime event structures, except that a configuration of a CRES is only required to be downwards closed with respect to the *condition* relation. That is, a configuration c of a CRES is a set of events such that,

(i) conflict-free: $\forall e, e' \in c. \neg e\#e'$

(ii) downwards-closed: $\forall e \in c, e' \in E. e' \leq_C e \implies e' \in c$

The *response* relation is in some sense dual to the condition relation and allows for defining an acceptance condition for runs: We define a run $(e_0, l(e_0)), (e_1, l(e_1)), \ldots$ to be accepting if $\forall i \geq 0. e_i \leq_R e \implies \exists j \geq 0. (e \# e_j \vee (i < j \wedge e = e_j)$. In words, any pending response event must eventually happen or be in conflict.

If one as it is usually the case consider any run of a prime event structure to be accepting, a prime event structure can trivially be regarded as a condition response event structure with empty response relation. This provides an embedding of prime event structures into condition response event structures which preserves configurations and runs.

Proposition 1. *The labelled prime event structure* $(\mathsf{E}, \leq, \#, l, \mathsf{Act})$ *has the same runs as the accepting runs of the* CRES *structure (*$\mathsf{E}, \mathsf{Act}, \leq_C, \leq_R, \#, l, \mathsf{Act}$*) where* $\leq_C = \leq$, $\leq_R = \emptyset$

We now go on to generalize the model to allow events to be executed several times. This also leads to a relaxation of the constraints on the condition and response relations and changing the conflict relation to a dynamic exclusion and inclusion of events.

Definition 3. A *dynamic condition response structure* (DCR) is a tuple $D = (\mathsf{E}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, l)$ where

(i) E is the set of events

(ii) Act is the set of actions

(iii) $\rightarrow\bullet \subseteq \mathsf{E} \times \mathsf{E}$ is the *condition* relation

(iv) $\bullet\rightarrow \subseteq \mathsf{E} \times \mathsf{E}$ is the *response* relation

(v) $\pm : \mathsf{E} \times \mathsf{E} \rightarrow \{+, \%, *\}$ is the *dynamic inclusion/exclusion* relation.

(vi) $l : \mathsf{E} \rightarrow \mathsf{Act}$ is a labelling function mapping events to actions.

The condition and response relations in DCR are the same as corresponding relations from CRES, except that they are not constrained in any way. In DCR, we have used a slightly different symbols for condition and response relations in order to be consistent with the graphical notation of DCR model. The *dynamic inclusion/exclusion* relation allows events to be included and excluded dynamically in the process. We will use the notation $e \rightarrow + e'$ for $\pm(e, e') = +$ and similarly write $e \rightarrow \% e'$ for $\pm(e, e') = \%$. The relation $e \rightarrow + e'$ expresses that, whenever event e happens, it will include e' in the process. On the other hand, $e \rightarrow \% e'$ expresses that when e happens it will exclude e' from the process.

We make the execution semantics precise below by giving a mapping to a labelled transition system with an acceptance condition on runs defined as described in the introduction.

A CRES can be represented as a DCR by making every event excluding itself and encoding the conflict relation by making any two conflicting events mutually exclude each other.

For example, consider a CRES with two conflicting events e, e' as shown in figure 1(a). This CRES can be represented as a DCR using the *exclude* relation as shown in the figure 1(b). The mutual *exclude* relation on events e, e' will ensure that, only one of the events can happen and similarly self *exclude* relation on the events will enforce that any event can happen only once.

Finally, we define *distributed* dynamic condition response structures by adding roles and principals.

Definition 4. A *distributed* dynamic condition response structure(DDCR) is a tuple

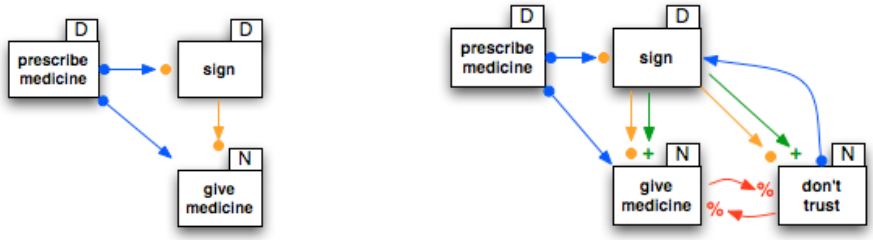
$$(\mathsf{E}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, l, \mathsf{R}, \mathsf{P}, \mathsf{as})$$

where $(\mathsf{E}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, l)$ is a dynamic condition response structure, R is a set of *roles*, P is a set of *principals* (e.g. persons/processors/agents) and $\mathsf{as} \subseteq (\mathsf{P} \cup \mathsf{Act}) \times \mathsf{R}$ is the role assignment relation to executors and actions.

(a) # relation in CRES (b) Encoding of # in DCR

Figure 1: Conflict relation in graphical notation

For a *distributed* DCR, the role assignment relation indicates the roles of principals and which roles gives permission to executed which actions. As an example, if *Peter* as *Doctor* and *Sign* as *Doctor* (for *Peter* ∈ P and *Doctor* ∈ R, then *Peter* can do the *Sign* action having the role as *Doctor*.



(a) Prescribe Medicine Example (b) Prescribe Medicine Example With Check

Figure 2: DCRS example in graphical notation

Now, figure 2(a) shows the small example workflow from the introduction graphically. It contains three events uniquely labelled (and thus identified) by the actions: prescribe medicine (the doctor calculates and writes the dose for the medicine), sign (the doctor certifies the correctness of the calculations) and give medicine (the nurse administers medicine to patient). The events are also labelled by the assigned roles (D for Doctor and N for Nurse).

The arrow •→• between prescribe medicine and sign indicates that the two events are related by both the condition relation and the response relation. The condition relation means that the prescribe medicine event must happen at least once before the sign event. The response relation enforces that, if the prescribe medicine event happen, subsequently at some point the sign event must happen for the flow to be accepted. Similarly, the response relation between prescribe medicine and give medicine enforces that, if the prescribe medicine event happen, subsequently at some point the give medicine event must happen for the flow to be accepted. Finally, the condition relation between sign and give medicine enforces that the signature event must have happened before the medicine can be given. Note the nurse can give medicine many times, and that the doctor can at any point chose to prescribe new medicine and sign again. (This will not block the nurse from continue to give medicine. The interpretation is that the nurse may have to keep giving medicine according to the previous prescription).

The dynamic inclusion and exclusion of events is illustrated by an extension to the scenario (also taken from the real case study): If the nurse distrusts the prescription by the doctor, it should be possible to indicate it, and this action should force either a new prescription followed by a new signature or just a new signature. As long the new signature has not been added, medicine must not be given to the patient.

This scenario can be modeled as shown in Figure 2(b), where one more action don't trust is added. Now, the nurse have a choice to indicate distrust of prescription and thereby avoid give medicine until

the doctor re-execute sign action. Executing the don't trust action will exclude give medicine and makes the sign as pending response. So the only way to execute give medicine action is to re-execute sign action which will then include give medicine. Here the doctor may choose to re-do prescribe medicine followed by sign actions (new prescription) or simply re-do sign.

We now define the semantics of distributed DCRs by giving a map to a labelled transition system and define the set of accepting runs. The states of the transition semantics will be triples (E,I,R) where $E \subseteq \mathsf{E}$ represents the set of happened events, $I \subseteq \mathsf{E}$ represents the set of currently included events, and R represents the set of pending responses.

Definition 5. For a distributed DCR $D = (\mathsf{E}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, l, \mathsf{R}, \mathsf{P}, \mathsf{as})$ we define the corresponding labelled transition systems $T(D)$ to be the tuple $(S, (\emptyset, \mathsf{E}, \emptyset), \rightarrow \subseteq S \times \mathsf{Act} \times S)$ where $S = \mathscr{P}(\mathsf{E}) \times \mathscr{P}(\mathsf{E}) \times \mathscr{P}(\mathsf{E})$ is the set of states, $(\emptyset, \mathsf{E}, \emptyset) \in S$ is the initial state, $\rightarrow \subseteq S \times (\mathsf{P} \times \mathsf{Act} \times \mathsf{R}) \times S$ is the transition relation given by

$$(E,I,R) \xrightarrow{(e,(p,a,r))} (E \cup \{e\}, I', R') \text{ where}$$

(i) $e \in I$, $l(e) = a$, $p \,\mathsf{as}\, r$, and $a \,\mathsf{as}\, r$

(ii) $\{e' \in I \mid e' \rightarrow\bullet\, e\} \subseteq E$

(iii) $I' = (I \cup \{e' \mid \pm(e,e') = +\}) \setminus \{e' \mid \pm(e,e') = \%\}$

(iv) $R' = (R \setminus \{e\}) \cup \{e' \mid e \bullet\!\rightarrow e'\}$

We define the runs $(e_0,(p_0,a_0,r_0)),(e_1,(p_1,a_1,r_1)),\ldots$ of the transition system to be the sequences of labels of a sequence of transitions $(E_i,I_i,R_i) \xrightarrow{(e_i,(p_i,a_i,r_i))} (E_{i+1},I_{i+1},R_{i+1})$ from the initial state. We define such a run to be accepting if $\forall i \geq 0. e \in R_{i+1} \implies \exists j. i < j \wedge (e = e_j \vee e \notin I_j)$. In words, a run is accepting if no pending response event from one point in the run is continuously included without happening.

The first item in the above definition expresses that, only events e that are currently included, can be executed, and to give the label (p,a,r) the label of the event must be a, p must be assigned to the role r, which must be assigned to a. The second item requires that all condition events to e which are currently included should have been executed previously. The third and fourth items are the updates to the sets of included events and pending responses respectively.

If one only want to consider finite runs, which is sometimes the case in the workflow community, the acceptance condition degenerates to requiring that no pending response is included at the end of the run. This corresponds to defining all states where $R \cap I = \emptyset$ to be accepting states and define the accepting runs to be those ending in an accepting state. If infinite runs are also of interest (as e.g. for reactive systems and the LTL logic) the acceptance criteria can be captured by a mapping to a Büchi-automaton. The construction is not straightforward and we leave it for future work to study it in detail.

(We define the transition system, runs and acceptance condition for a non-distributed DCR as for a distributed DCR except there are no principals and roles.)

We can then state the result that the representation of CRES as DCR exemplified in figure 1(b) provides an embedding preserving accepting runs.

Proposition 2. *The condition response event structure* $(\mathsf{E}, \leq_C, \leq_R, \#, l, \mathsf{Act})$ *has the same accepting runs as the accepting runs of the* DCR *structure (*$\mathsf{E}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, l$*) where* $\rightarrow\bullet = \leq_C$, $\bullet\rightarrow = \leq_R$, $\forall e,e' \in \mathsf{E}. \pm(e,e') = \%$ *if* $e = e'$ *or* $e\#e'$ *and otherwise* $\pm(e,e') = *$.

3 Conclusion and Future Work

We presented a declarative process model derived as a sequence of relatively simple generalizations of labelled event structures inspired by the workflow language employed by our industrial partner. The first generalization is to split the causality relation of event structures into two dual relations, a *condition* relation $\rightarrow\bullet$ such that $\{e' \mid e' \rightarrow\bullet\, e\}$ is the set of events required to have happened before the event e can happen and a *response* relation $\bullet\rightarrow$, such that $\{e' \mid e \bullet\rightarrow e'\}$ is the set of events that must happen (or be in conflict) after the event e has happened. The final extension allows to finitely specify repeated, possibly infinite behavior and acceptance conditions for runs by allowing *multiple execution*, and *dynamic inclusion* and *exclusion* of events and allows for distribution of events via roles. We presented a graphical notation inspired by related work by van der Aalst et al, and gave a mapping to labelled transition systems with an acceptance condition on runs based on the response relation. We remarked that if one only considers finite runs, the acceptance condition can be captured by defining a set of accepting states in the labelled transition system and defining a run to be accepting if it ends in an accepting state. Moreover, we remarked that for infinite runs the accepting condition can be captured by a mapping to a Büchi-automaton, but leave the detailed study of this construction to future work. Also, future work will consider a more detailed comparison between dynamic condition response structures and existing models for concurrency, including the relation to the work in [2]. We also plan to study more complex scenarios and workflow patterns, other acceptance conditions, distributed scheduling, and extensions of the model, notably with time, nested sub structures, soft constraints, and compensation/exceptions.

Acknowlegments

This research is supported by the Trustworthy Pervasive Healthcare Services (TrustCare) project. Danish Research Agency, Grant # 2106-07-0019 (www.TrustCare.eu).

References

[1] Christoph Bussler and Stefan Jablonski. Implementing agent coordination for workflow management systems using active database systems. In *Research Issues in Data Engineering, 1994. Active Database Systems. Proceedings Fourth International Workshop on*, pages 53–59, Feb 1994.

[2] Allan Cheng. Petri nets, traces, and local model checking. In *Proceedings of AMAST*, pages 322–337, 1995.

[3] Nihan Kesim Cicekli and Ilyas Cicekli. Formalizing the specification and execution of workflows using the event calculus. *Information Sciences*, 176(15):2227 – 2267, 2006.

[4] Hasam Davulcu, Michael Kifer, C. R. Ramakrishnan, and I.V. Ramakrishnan. Logic based modeling and analysis of workflows. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–3. ACM Press, 1998.

[5] Alvaro A. A. Fernandes, M. Howard Williams, and Norman W. Paton. A logic-based integration of active and deductive databases. *New Gen. Comput.*, 15(2):205–244, 1997.

[6] Karen Marie Lyng, Thomas Hildebrandt, and Raghava Rao Mukkamala. From paper based clinical practice guidelines to declarative workflow management. In *Proceedings of 2nd International Workshop on Process-oriented information systems in health- care (ProHealth 08)*, pages 336–347, Milan, Italy, September 2008.

[7] Raghava Rao Mukkamala, Thomas Hildebrandt, and Janus Boris Tøth. The resultmaker online consultant: From declarative workflow management in practice to LTL. In *Proceeding of 1st International Workshop on Dynamic and Declarative Business Processes, 2008*, pages 36–43, 2008.

[8] Pinar Senkul, Michael Kifer, and Ismail H. Toroslu. A logical framework for scheduling workflows under resource allocation constraints. In *In VLDB*, pages 694–705, 2002.

[9] Munindar P. Singh, Greg Meredith, Christine Tomlinson, and Paul C. Attie. An event algebra for specifying and scheduling workflows. In *Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 53–60. World Scientific Press, 1995.

[10] Wil M. P. van der Aalst and Maja Pesic. A declarative approach for flexible business processes management. In *Proceedings of Workshop on Dynamic Process Management (DPM 2006)*, volume 4103 of *LNCS*, pages 169–180. Springer Verlag, 2006.

[11] Wil M. P. van der Aalst, Maja Pesic, and Helen Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - R&D*, 23(2):99–113, 2009.

[12] Glynn Winskel. Event structures. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer, 1986.

Session Type Inference in Haskell

Keigo Imai Shoji Yuen Kiyoshi Agusa
Graduate School of Information Science, Nagoya University, Japan
imai@nagoya-u.jp, yuen@is.nagoya-u.ac.jp, agusa@is.nagoya-u.ac.jp

1 Introduction

In order to efficiently develop reliable communicating distributed software, designing communication protocols between components is one of the key issues to ensure correct behavior of the software. *Session-type systems*[4][11][3][10] provide static checking of communication protocols. These systems enforce communication descriptions to conform to certain protocols. Incorporating session types in programming languages eases the communication centric programming in that session typed components are guaranteed to behave correctly by their types.

We show an implementation of binary session types equipped with a session type inference in Haskell.[1] One of our contributions is the treatment of *multiple channels* annotated less than the existing implementations[7][8][9]. This makes the session types easier to work with.

Our technique for embedding session types in Haskell is to maintain an extra type environment for session types using type-level programming. However, there is yet a problem of how accessors, i.e., names, of a type environment should be generated and compared in the type-level. To resolve this problem, we introduced a canonical indexing of names by natural numbers, as in the *de Bruijn notation*. Providing access to a type environment by natural numbers enabled us to implement the current library without the bookkeeping operations[8] nor manual construction of session types[9].

Our implementation also supports network programming based on TCP. We show an example of a SMTP client based on our implementation. The reader is assumed to have a basic knowledge of programming in Haskell.

1.1 Related work

Neubauer and Thiemann[7] implemented session types on a single communication channel in Haskell. Their implementation avoids aliasing by prohibiting explicit use of a channel.

Pucella and Tov[8] have shown a general technique to encode session types in languages like Haskell, ML, and C#. They provide communications with multiple channels. To track the identity of a channel on type-level, fresh universally quantified type variables are assigned to each channel. Since type checker in those languages does not support type level operations on such type variables, their implementation requires inserting bookkeeping operations like `swap` and `dig` on type environments in source code.

The implementation proposed by Sackman and Eisenbach[9] supports full functionality of session types. However, their library requires a manual construction of session types. There are trade-offs between such a manual handling and annotated type-inference approach in that while type-inference reduces unneeded annotations, explicit annotation with a rich set of syntax increases readability and expressiveness of types. We will discuss this aspect in the later section.

The difference of our implementation from the previous work is summarized in the following table.

[1] A working implementation, `full-sessions` , which can be compiled by the Glasgow Haskell Compiler 6.10.2 or higher is available at: `http://hackage.haskell.org/package/full-sessions/`. Typing `cabal install full-sessions` in a shell will install `full-sessions` in your environment.

	channel passing	annotation	portable
Neubauer et al.[7]	no	auto	no
Pucella et al.[8]	yes in a limited context[2]	stack based channel handling	yes
Sackman[9]	yes	manual construction of session types	no
Our implementation	*yes*	*auto*	*no*

2 Session-type inferencer on Haskell

We first introduce concurrency primitives and session types in our implementation. Then we present a few techniques to embed session types in Haskell as in [7] and [8]. Finally we show the session type reconstruction for multiple channels based on de Bruijn levels.

2.1 Concurrency primitives and session types in `full-sessions`

Our implementation, `full-sessions` , provides primitives for starting a new thread, generating a channel, and communicating values and channels synchronously through it. They are summarized in the Table 1. For readability, we use the `ixdo` notation[8], which provides a syntactic sugar to write programs in an imperative style. All the primitives return a *session* of type `Session`. They can be composed sequentially by the notation s_1 ; s_2, or be composed in parallel using the primitive `fork` which starts a new thread. Session types in our implementation are summarized in the Table 2. These types are inferred at compile time for each channel according to the typing rules in the session-type system[4].

	Syntax	Meaning
Channel Creation	$c \leftarrow$ `new`	Create a new channel and bind it to c
Value Output	`send` c e	Output an expression e on c
Value Input	$x \leftarrow$ `recv` c	Input a value on c and bind it to x
Selection	`sel`i c $(i \in \{1,2\})$	Output a label i on c
Offering	`offer` c p_1 p_2	Input a label i on c, then continue p_i
Session Delegation	`sendS` c c'	Output a channel c' on c
Session Reception	$c' \leftarrow$ `recvS` c	Input a channel on c and bind it to c'
Fork	`fork` p	Create a new thread and run with the session p
Calling Haskell I/O	`io` m	Execute Haskell's `IO` action m
Recursion of a session	`recur1` f c	Recursive call of a session $(f\ c)$ where c is a channel
Recursive use of a channel	`unwind` c	Unwind a recursive session type `Rec` n u into $u[\texttt{Var}\ n \mapsto \texttt{Rec}\ n\ u]$ on c

Table 1: Primitives in the `full-sessions` library

2.2 Session type inference for a single channel

2.2.1 A single-threaded participant

Let us begin with a case of single channel in a single-threaded participant. In such a case a session type *advances* as a session proceeds. For example a type `Send Int End` advances to `End` when a channel of that type is used to send an integer. To track such an advance of a session type, we assign a pair of session

[2]Since in our implementation, terms are represented in a de Bruijn form, channel are free from renaming. Thus, any form of channel passing is allowed whereas in [8] the scope of a channel has to be fixed.

Session Types	Meaning
`Send` v u	Output a value of type v then u
`Recv` v u	Input a value of type v then u
`Select` u_1 u_2	Internal choice between u_1 and u_2
`Offer` u_1 u_2	External choice between u_1 and u_2
`Throw` u_1 u_2	Output a channel which has a session u_1 then u_2
`Catch` u_2 u_2	Input a channel which has a session u_1 then u_2
`End`	The channel cannot be used any more
`Bot`	The channel is already owned by two communicating processes
`Rec` n u	Recursive session which binds occurrences of `(Var` n`)` in u
`Var` n	Occurrence of a recursion variable

Table 2: Session Types in the `full-sessions` library

types, called a *pre-type* and a *post-type*, to each occurrence of a channel. A pre-type denotes the session type *before* a session starts. Similarly, a post-type is the session type *after* a session ends.

In many cases post-types act as a *placeholder*, which allows concatenation of two session types. For example, consider one of the simplest sessions, `send` c `True`. The pre-type of the channel c in this session is `Send Bool` u and the post-type of it is u, where u is a type variable. This means that an another session which uses the channel c can be further concatenated after this session.

The concatenation of two session types are done by unification. In a concatenation $s_1;s_2$ of two sessions, the post-types of channels in s_1 is unified with the pre-types of ones in s_2. The pre-types of channels in the concatenated session $s_1;s_2$ is same as the ones in s_1. The post-types of channels in $s_1;s_2$ is the ones of s_2. Accordingly, `(send` c `True; send` c `"abc")` has `(Send Bool (Send String` u_2`))` as the pre-type and u_2 as the post-type on the channel c, where u_2 is a type variable distinct from u.

For a more complex example, the code below describes a simple calculator server.

```
server c = ixdo x ← recv c; y ← recv c; offer c (send c (x+y::Int)) (send c (x<y))
```

The `server` firstly receives two values of type `Int` and a branch label (here the label is either 1 or 2), then sends an answer either of type `Int` or of `Bool` according to the label.

The pre/post-type of the channel `c` in the `server` can be inferred by the GHC's typechecker via auxiliary function `channeltype1`. By showing the type of `(channeltype1 server)` using GHC's interactive environment, users will obtain the following response:

```
prompt> :t channeltype1 server
channeltype1 server :: (Recv Int (Recv Int (Offer (Send Int a) (Send Bool a))), a)
```

2.2.2 Duality of two session types

The `fork` primitive requires the *duality* between pre-types of two sessions. Here we explain it by using the previous example of a calculator server. Firstly, a client of the server would be like this:

```
client c = ixdo send c 123; send c 456; sel2 c; ans ← recv c;
                io (putStrLn (if ans then "Lesser" else "Greater or Equal"))
```

The pre-/post-type of `c` in `client` is `(Send Int (Send Int (Select` u_1 `(Recv Bool` u`)))` and u, respectively. By putting `server` and `client` in parallel by `fork`, and by generating a channel by `new`, we obtain the code below:

```
calc c    = ixdo fork (server c); client c;
startCalc = ixdo c ← new; calc c
```

The above code typechecks because the two usages of `c` in `client` and `server` are dual. The resulting pre-type is `Bot`, as the session-type algebra of [4] implies. The post-type is `End` since `fork` requires the usage of channels in the given session to be ended. [3] Here we confirm it:

```
prompt> :t channeltype1 calc
channeltype1 calc :: (Bot, End)
```

A session can be run by the function `runS`. Typing `runS startCalc` will produce the result "`Lesser`" on the console. The following is the result of the execution using the interpreter:

```
prompt> runS (channeltype1 calc)
Lesser
```

2.3 Tracking sessions with multiple channels by De Bruijn indexing

To track usages of multiple channels in type-level, a natural number of *de Bruijn level* is assigned to each channel. De Bruijn level represents the nesting depth of a variable binder. For example, in a λ-calculus term $\lambda x.\lambda y.x$ the level of the variable x is 0 whereas y is 1. Figure 1 shows the de Bruijn level indexing of a session. In the figure, the de Bruijn level of a variable is denoted by a superscript at the binding position. Note that we need to count on only channels, hence each variable c, d, e and f have an index but x does not.

```
ixdo c^n ← new; d^(n+1) ← new;
     fork (ixdo e^(n+2) ←catch c; ...)
     x ← recv d;
     f^(n+2) ← catch d;
     ...
```

Figure 1: De Bruijn *level* indexing in a session

De Bruijn levels are assigned to the type of channels. A channel has the type of the form `Channel` t n where n is a de Bruijn level of the channel and t is a "type-tag"[6]. We do not explain the type-tag, since it is out of scope of our paper. Natural numbers are represented by combinations of the two types representing peano-numerals `Z` and `S` n where each of them denotes 0 and $n+1$ respectively. For example, a channel which has de Bruijn level 2 has type `Channel` t `(S (S Z))`. Each number points to a certain position of a type environment.

Session types of multiple channels are recorded in extra type environments. We need two type environments for pre-types and post-types. Hereafter we call them *pre-environment* and *post-environment*, respectively.

Such an environment is represented by a list of session types, and its elements are accessed by specifying the number of de Bruijn level. Figure 2 is an example of a session `send` c `"abc"; send` d `True` and its pre-/post-environment. Assuming that c and d have (Haskell-) type `Channel` t `Z` and `Channel` t `(S Z)` respectively, the pre- and post-environment of the session is inferred as shown in the figure. c and d has pre-type `Send String` u_1 and `Send Bool` u_2, and post-types of them are u_1 and u_2,

[3] Such discipline can also be observed in session-type systems equipped with a thread-spawning construct. the "ended" condition of **Spawn** rule in [2].

respectively. Note that the figure also depicts the session-types in an intermediate step after `send` c `"abc"`. In that state, c has type u_1 and d has type `Send Bool` u_2.

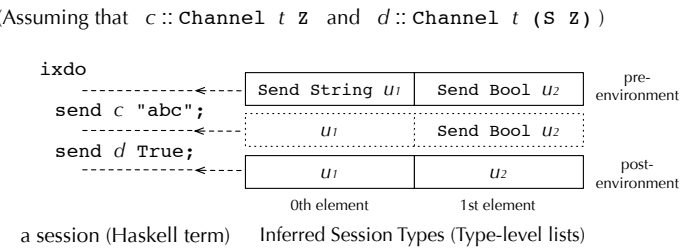


Figure 2: Tracking session types by numbers

The type of a session has the form of `Session` t ss tt a. The pre-/post-environments are at the position of ss and tt respectively. The parameter a is the type of a value returned by a session, and t is a type-tag.

The pre-/post-environments ss and tt are actually represented by *type-level lists*[5]. A type-level list is either ss `:>` u or `Nil`, where ss is another type-level list and u is a session type, and `Nil` is a empty list. Note that the type constructor `:>` is left-associative, for example ss`:>`a`:>`b is interpreted as (`(`ss`:>`a`)`:>b. Also note that the type environment is counted from left to right order. For example, the 0-th element of `Nil:>`a`:>`b`:>`c is a.

Provided that the type of c is `Channel` t `Z` and the type of d is `Channel` t `(S Z)`, a session of the previous example `(send` c `"abc"; send` d `True)` has type `Session` t `(Nil:>Send String` u_1`:> Send Bool` u_2`) (Nil:>`u_1`:>`u_2`) ()`.

In general, the de Bruijn levels can be a non-constant value, like $n+1$, $n+2$ and so on. For example, if the length of a session-type environment ss is n, and the type of c and d is `Channel` t n and `Channel` t `(S` n`)` respectively, a session `(send` c `1; send` d `True)` has type `Session` t `(`ss`:>Send Int` u_1`:> Send Bool` u_2`) (`ss`:>`u_1`:>`u_2`) ()`. Constraints for the length of a session-type environment is represented in the type-level by the type-class `SList` ss n, which represents that the length of ss is n. Observe that the existence of the placeholder ss in each of session-type environments. This makes possible to handle arbitrary numbers of channels by concatenation of sessions which introduce new channels, which involves unification between the post-environment of the earlier session and the pre-environment of the later session.

When a new channel is introduced, post-environments are *extended* to store the session type of the introduced channel. The primitive `new` and `catch` involve such a mechanism. `new` has pre-environment ss and post-environment ss`:>Bot`. At the same time `new` returns a channel of type `Channel` t n, where n is equal to the length of ss and points to the leftmost position of the post-environment, namely `Bot`. Hence the index of a generated name is assured to be fresh.

Figure 3 shows the pre-/post-environments of a session (`c ← new; fork (send` c `True)`). The post-environment has an extra entry for the newly created channel. The post-type of the newly created channel is dual of `Send Bool End`, which is required to communicate with the forked session.

Similarly, `catch` c has the pre-/post-environment ss and tt`:>`u', where n-th element of ss is `Catch` u' u and that of tt is u. Figure 4 shows such use of `catch` and the inferred session types.

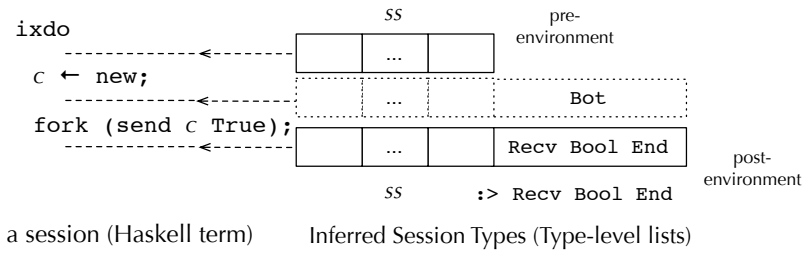


Figure 3: Extension of a type environment by `new` operation

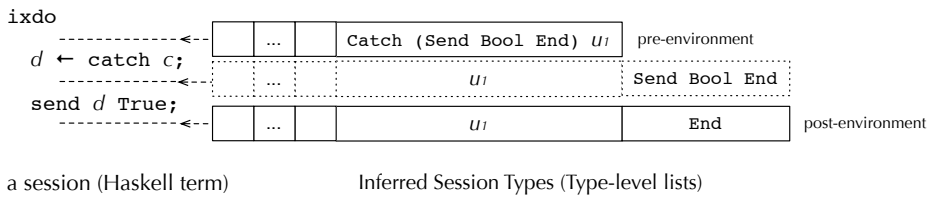


Figure 4: Extension of a type environment by `catch` operation

2.4 Comparison of existing Haskell implementations of session types

Our encoding based on de Bruijn indexing reduces most of annotations which are required in the other works. We show that by giving a few examples of sessions.

Stack-based implementation The implementation by Pucella and Tov[8] applies a stack of session types as the representation of a type-environment. Communication primitives can only access the top of the stack, hence explicit manipulation of stack is required. The combinator `dig` and `swap` is provided for such purpose. The `swap` combinator swaps the top two channels on the stack. On the other hand, `dig` combinator converts a given session to operate on a deeper channel stack. Provided that the session type for *c* and *d* is on the top of the stack in this order, the code below is equivalent to `(send c "abc"; send d True)`:

```
ixdo send c "abc"; swap; send d True
```

or

```
ixdo send c "abc"; dig (send d True)
```

As a number of channels increases, more stack operations will be required. In [8] a few approaches to this problem are discussed, however the problem had been left open, and our number-based approach is not covered.

Manual construction of session types The implementation by Sackman and Eisenbach[9] provides a very rich set of communication primitives, at a cost of manual construction of session types. There

are two communication media, channels and *Pid*s. We show the simplest case of communication via Pid. The example below passes an integer 10 to the other thread and terminates. It is equivalent to `runS (ixdo c <- new; fork (send c 10); recv c)`.

```
(s, a) = makeSessionType (
           newLabel ~>>= λa →
           a ∘= send (undefined :: Int) ~>> end
           ~>> sreturn a)

p = run s a (ssend 10) srecv
```

Here `makeSessionType` returns a collection of session types s and its fragment a. In the argument of `makeSessionType` the construction of a session type is described procedurally. Again, as a number of threads with different protocol increases, the more construction of session types will be required. The case of channel-based communication is similar.

3 An example SMTP client

We show the network functionality of the `full-sessions` by the example of a SMTP client with multiple channels. A single-channel version of SMTP client with session types has its origin from [7].

Table 3 shows additional primitives for network communication. To model network protocols, the *type-based branching*, `sel`i`N` and `offerN`, is provided in addition to the previous *label-based branching*. Note that the `sel`i`N` does nothing, but we need them to infer the session types for type-based selections.

	Syntax	Meaning
Connect to a service	$c \leftarrow$ `connectNw` s	Connect to a service s and bind a session channel to c
Type-based offering	`offerN` c p_1 p_2	Offer two receiving session p_1 and p_2 on c
Selection annotation	`sel`i`N` c $(i \in \{1,2\})$	Determine which branch of `Select` u_1 u_2 will be selected on c

Table 3: Additional primitives for network programming

Here we show our implementation of SMTP client in the simplest form. Firstly, the types for SMTP commands and replies are defined as follows:

```
-- Types for SMTP commands.
newtype EHLO = EHLO String
newtype MAIL = MAIL String
newtype RCPT = RCPT String
data    DATA = DATA
data    QUIT = QUIT
newtype MailBody = MailBody [String]

-- Types for SMTP server replies (200 OK, 500 error and 354 start mail input)
newtype R2yz = R2yz String; newtype R5yz = R5yz String; newtype R354 = R354 String
```

To deal with the stream-based communication of TCP, either a parser or a printer for each type of communicated values must be prepared. Provided such functions exist, the SMTP client is described as follows:

```
-- auxiliary functions
send_receive_200 c mes = ixdo send c mes; (R2yz _) ← recv c; ireturn ()
send_receive_354 c mes = ixdo send c mes; (R354 _) ← recv c; ireturn ()
```

```
sendMail c d = ixdo  -- the body of our SMTP client
  (R2yz _) ← recv c                   -- receive 220
  send_receive_200 c (EHLO "mydomain") -- send EHLO, then receive 250
  unwind0 c; sel1N c                  -- (annotation) branch to send 'MAIL FROM'
  from ← recv d                          -- (1) input the sender's address on d
  send_receive_200 c (MAIL from)      -- send 'MAIL FROM', then receive 250
  unwind1 c; sel1N c                  -- (annotation) branch to send 'RCPT TO'
  to ← recv d                            -- (2) input the recipient's address on d
  send c (RCPT to)                    -- send 'RCPT TO'
  offerN c (ixdo                      -- branch the session according to the reply
    (R2yz _) ← recv c                 -- if 250 OK is offered
    sel1 d; mail ← recv d                -- (3) input the content of the mail on d
    unwind1 c; sel2N c                -- (annotation) branch to send 'DATA'
    send_receive_354 c DATA           -- send 'DATA' and receive 354
    send_receive_200 c (MailBody mail) -- send the content of the mail
    unwind0 c; sel2N c                -- (annotation) branch to send 'QUIT'
    send c QUIT; close c              -- send 'QUIT' and close the connection
   ) (ixdo
    (R5yz errmsg) ← recv c;           -- if 500 ERROR is offered
    sel2 d; send d errmsg;               -- (4) output the error message on d
    send c QUIT; close c)             -- send 'QUIT' and close the connection
  close d
```

The `sendMail` takes two channels `c` and `d` as its parameters. The former is used to communicate with the SMTP server while latter is used to prepare necessary information for sending a mail. By checking the type of `typecheck2 sendMail`, the following type is answered by GHC:

```
typecheck2 sendMail :: (SList ss l, IsEnded ss b1) ⇒ Session t
  (ss :> Recv R2yz (Send EHLO (Recv R2yz (Rec Z (SelectN
          (Send MAIL (Recv R2yz (Rec (S Z) (SelectN
            (Send RCPT (OfferN (Recv R2yz (Var (S Z))) (Recv R5yz (Send QUIT Close))))
            (Send DATA (Recv R354 (Send MailBody (Recv R2yz (Var Z))))))))
          (Send QUIT Close)))))
      :> Recv String (Recv String (Select (Recv [String] Close) (Send [String] Close))))
  (ss :> End :> End) ()
```

The SMTP protocol is successfully represented in the pre-type of `c`. A server that have the dual of this type can communicate with this client.

Observe that the two channels are used with no annotation. On the other hand, the implementation of [8] requires the `swap` operation before and after the each occurrence of `d`, namely at `(1)`, `(2)`, `(3)` and `(4)`, and if we add more channels, more complicated bookkeeping operations will be required.

4 Discussion

Here we discuss a few aspects of session type implementation.

Trade-offs between type inference and manual construction of session types As we have shown in Section 2.4, annotations required by our implementation is not more than any of the other implementations. However, there seems to be a few advantages in [9] in a few points. (1) Recursion of a session type is treated more naturally in [9]. By using term-level operation for constructing session types, [9] offers more readable formulation of recursion via labels. As you can observe in the SMTP example of

the previous section, recursion on a session type require a few of not so intuitive annotations $\texttt{unwind}_i$ on the term-level to represent a recursive protocol. (2) Manual construction of session types in term-level offers chance of subtyping. It is difficult to allow subtyping of session types in the parallel composition, because of our bijective encoding of duality to extract more information in a parallel composition of a session.

Readability of type error messages If the duality check of two session types fails, the type error would be reported. For example, by replacing the occurrence of an integer `456` in Section 2.2.2 with a string `"456"`, the following error is obtained :[4]

```
examples/calc.hs:<xx>:0:
    Couldn't match expected type '[Char]' against inferred type 'Int'
      Expected type: tt' :> Send [Char] a
      Inferred type: tt' :> Send Int (Select (Recv Int End) (Recv Bool End))
    When generalising the type(s) for 'plus'
```

The error reports that the inferred pre-type of `client` is not compatible with the expected one. The position `<xx>` of the reported error is not at `send c "456"` itself, but at the position where the dual of the session type is calculated, namely the occurrence of the `fork`. Thus, this error message directly shows which session types are not compatible. Even the type-level hackery we depend tends to produce large type signatures, the type error itself can be concisely represented.

5 Concluding remarks

This paper showed a Haskell implementation of session types equipped with a session-type inference. Our implementation improves the other existing implementations of session types in that our session type inference requires no manual bookkeeping as in [8].

The treatment of binders is the key issue in the technology for embedding an language into another, as stated in [1]. In our implementation, we have taken separated approach for *term-level computation* and *type-level* (compile-time) computation. In term-level, the fresh channels are represented by λ-abstraction (the technique usually called Higher order abstract syntax), which utilizes the power of variable-bindings in the host language Haskell. In type-level, *de Bruijn levels* which are encoded in channel types to compare of names, which considerably automates session-type inference.

Our technique using de Bruijn level can be applied to other substructural type systems for the π-calculus, such as linear type systems. However, since the present technique depends on a rich type-level programming functionality of Haskell, it is not easy to export this scheme to the other programming languages.

Acknowledgments This work was partially supported by the Grant-in-Aid (Scientific Research (B) 20300009 and Scientific Research(C) 19500026) from the Ministry of Education, Culture, Sports, Science, and Technology of Japan.

References

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized Metatheory

[4] Here, `[Char]` is a type synonym of `String`.

for the Masses: The POPLMARK Challenge. In *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 2005.

[2] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous Session Types and Progress for Object Oriented Languages. In *Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *Lecture Notes in Computer Science*, pages 1–31. Springer-Verlag, 2007.

[3] Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi Calculus. *Acta Informatica*, 42(2):191–225, November 2005.

[4] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP '98: Proceedings of the 7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer-Verlag, 1998.

[5] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly Typed Heterogeneous Collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[6] Oleg Kiselyov and Chung C. Shan. Lightweight monadic regions. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 1–12, New York, NY, USA, 2008. ACM.

[7] Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In *PADL'04 : Practical Aspects of Declarative Languages*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer-Verlag, 2004.

[8] Riccardo Pucella and Jesse A. Tov. Haskell Session Types with (Almost) No Class. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM Press, 2008.

[9] Matthew Sackman and Susan Eisenbach. Session Types in Haskell: Updating Message Passing for the 21st Century. Technical report, Imperial College London, June 2008. Available at http://pubs.doc.ic.ac.uk/session-types-in-haskell/.

[10] Vasco T. Vasconcelos, Simon J. Gay, and Antonio Ravara. Type checking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1-2):64–87, December 2006.

[11] Nobuko Yoshida and Vasco T. Vasconcelos. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. In *SecReT 2006: Proceedings of the First International Workshop on Security and Rewriting Techniques*, volume 171 of *Electronic Notes in Theoretical Computer Science*, pages 73–93. Elsevier Science Publishers B. V., 2007.

A Modular Toolkit for Theories of Distributed Interactions

Julien Lange
University of Leicester, UK
jl250@le.ac.uk

Emilio Tuosto
University of Leicester, UK
et52@le.ac.uk

Abstract

We discuss the design principles of an architecture for a toolkit which will implement the theories of distributed interactions. The main design principles of our architecture are *flexibility* and *modularity*. In fact, the toolkit is inspired by the existing theories of distributed interactions recently introduced by several authors. Our main goal is to provide an easily extensible workbench to encompass current algorithms and incorporate future developments of the theory of distributed interactions.

1 Introduction

With the emergence of distributed systems, communication has become one of the most important elements of today's programming practice. Nowadays, distributed applications typically build up from (existing) components that (sometimes dynamically) are glued together to form more complex pieces of software. It is hence natural to model such applications as units of computation interacting through suitable communication models. An intricacy of communication-centred applications is that interactions are distributed. Here, the acceptation of distribution has to be taken in a very general sense since interactions are physically and logically distributed; as a matter of fact, components may run remotely and, for instance, components may belong to different administrative domains.

In order to ensure predictable behaviours of communication-centred applications, it is necessary that software development is based on solid methodologies. Besides the theoretical results that guarantee the interesting properties of software, it is also desirable to provide practitioners with a set of tools to support them in addressing the most common problems (e.g. avoiding synchronisation bugs).

Session types (ST) stand out as an effective mathematical foundation for designing/analysing distributed interactions. For instance, dyadic ST [7] have been proposed as a structuring method and a formal basis for verification of distributed interactions of two participants (e.g., in client-server architectures). Dyadic ST has been recently generalised to multiparty sessions [5, 6, 8] where sessions have more than two participants and in [6] dynamic sessions have been considered. On top of multiparty sessions, in [3] a theory of design-by-contract for distributed interactions has been introduced. Basically, ST are extended with *assertions* acting as pre-/post-conditions or invariants of interactions.

The aim of this paper is to describe the design principles for the architectures of a modular toolkit which puts in practice the theories of distributed interactions based on ST. Arguably, most of the research around ST has been mainly devoted to give a precise description of verification and validation frameworks. In fact, only very few and ad-hoc implementations have been developed (e.g., [9, 10, 11, 12]).

We aim to develop a toolkit that accommodates a few main requirements. Firstly, the toolkit has to provide a workbench for theoretical studies so to permit (*i*) to experiment with potentially more realistic examples and (*ii*) to possibly combine several of these methodologies. Secondly, our toolkit has to be easily extensible so to allow researchers to explore new directions as the theory of distributed interactions develops. Finally, albeit being a prototype for research, our toolkit has to shape the basic implementations that can be used in more realistic frameworks for the development of communication-centric software.

Synopsis § 2 gives background information and a motivating example. § 3 gives more details on the design principles of our toolkit and its architecture. § 4 highlights the main advantages of our modular approach. § 5 concludes and highlights our future plans.

2 Background and motivating example

We briefly describe the distinguished aspects common to several theories of distributed interactions.

The key ingredients of the theories of distributed interactions based on ST are as follows [7].

Sessions are sets of *structured* interactions which correspond, for instance, to a complex communication protocol (from the communication initialisation to the termination). The basic idea is that a computation consists of several concurrent sessions that involve some participants. A main concern is that participants acting in different sessions do not interfere. For instance, a desirable property to enforce is that a message sent in a session from A and meant for B is not received by a participant C of (another) session.

Interaction primitives basically include communication mechanisms à-la π-calculus that deal with sessions as first-class values. Another kind of interaction primitives often present features a *select/branch* mechanism which resembles a simplified form of method invocation. For instance, communication interaction and select/branch in the global calculus [8] notation are

$$A \to B : k\langle \texttt{sort} \rangle \qquad \text{and} \qquad A \to B : k\{l_i : G_i\}_{i \in J}$$

In the former, participant A sends a message of type `sort` to B on the channel k; in the latter A selects one of the labels l_i (sending it on k) and, correspondingly, B executes its i-th branch G_i.

Communication primitives typically permit *delegation*, namely the fact that sessions can be exchanged so to allow a process to delegate another process to continue the computation.

Typing disciplines guarantee properties of computations. For example, in dyadic ST [7] the *duality* principle guarantees that, in a session, the actions of a participant have to be complemented by the other participant (or its delegate). Among the properties checked by type systems, *progression* and some form of correctness properties are paramount. For instance, in [3] a well-typed system is guaranteed to respect the contract specified by its assertions and, once projected, the program is guaranteed to be free from "communication-errors".

Type systems are sometime subject to *well-formedness* conditions. For instance, global types in [8] have to be linear in order to be projected to *local types*.

We illustrate some theoretical aspects with an example adapted from [8] to the global assertions in [3]. The following is a global assertion (cf. [3]) specifying a protocol[1] with two buyers (B_1 and B_2) and a seller (S). The buyers B_1 and B_2 want to purchase a book from S by combining their money.

$$\begin{array}{rlll} G = & B_1 \to S: & s\langle t : \texttt{string}\rangle.(\!|\textbf{assert } t \neq \text{""}|\!) & (1) \\ & S \to B_1: & b_1\langle q : \texttt{int}\rangle.(\!|\textbf{assert } q > 0|\!) & (2) \\ & B_1 \to B_2: & b_2\langle c : \texttt{int}\rangle.(\!|\textbf{assert } 0 < c \leq q|\!) & (3) \\ & B_2 \to S: & s\{\mathsf{ok}(\!|\,|\!) : D, \mathsf{quit}(\!|\,|\!) : \mathit{end}\} & (4) \end{array}$$

Interactions (1÷4) are decorated with assertions $(\!|\textbf{assert } \phi|\!)$ stating a condition ϕ on the variables of the protocol ($(\!|\,|\!)$ abbreviates $(\!|\textbf{assert } \mathsf{true}|\!)$). In (1), B_1 and S interact (through s) and exchange the book title; B_1 (resp. S) guarantees (resp. relies on) the title is (resp. being) a non-empty string. In (2), S gives B_1 a quote q; $(\!|\textbf{assert } q > 0|\!)$ constraints the price to a positive value. In (3), B_1 tells B_2 its non-negligible contribution c to the purchase (as B_1 guarantees $(\!|\textbf{assert } 0 < c \leq q|\!)$). In the last step, B_2 may refuse

[1] For the sake of this extended abstract, global assertions may be thought of as global types decorated with formulae of a (decidable) logic.

(selecting label quit) or accept[2] the deal (selecting label ok); in the former case the protocol just finishes otherwise it continues as:

$$D = B_2 \to S : s\langle a : \texttt{string}\rangle (\!|\textbf{assert } a \neq \text{''''}|\!).S \to B_2 : b_2\langle d : \texttt{date}\rangle (\!|\,|\!)$$

namely B_2 and S exchange delivery address and date.

Linearity is a (typically decidable) property ensuring that communications on a common channel are ordered temporally. Linear types can be *projected* so to obtain the local types of each participant. Similarly, global assertions have to be well-asserted, namely the assertions have to guarantee satisfiability in a decidable logic. The projections of our example are:

$$\begin{array}{l} pB_1 = s!\langle t : \texttt{string}\rangle\ (\!|\textbf{assert } t \neq \text{''''}|\!); \\ \quad b_1?\langle q : \texttt{int}\rangle\ (\!|\textbf{assert } q > 0|\!); \\ \quad b_2!\langle \texttt{c:int}\rangle\ (\!|\textbf{assert } q > 0 \wedge 0 < c \leq q|\!) \end{array}$$

$$\begin{array}{l} pB_2 = b_2?\langle \texttt{c:int}\rangle\ (\!|\textbf{assert } \exists q : \mathsf{int} | 0 < c \leq q|\!); \\ \quad s \oplus \{\mathsf{ok}\ (\!|\,|\!) : s!\langle a : \texttt{string}\rangle\ (\!|\textbf{assert } a \neq \text{''''}|\!); \\ \qquad b_2?\langle d : \texttt{date}\rangle\ (\!|\,|\!), \\ \quad \mathsf{quit}\ (\!|\,|\!) : end\} \end{array}$$

$$\begin{array}{l} pS = s?\langle \texttt{string}\rangle\ (\!|\textbf{assert } t \neq \text{''''}|\!); \\ \quad b_1!\langle q : \texttt{int}\rangle\ (\!|\textbf{assert } q > 0|\!); \\ \quad s\&\{\mathsf{ok}\ (\!|\,|\!) : s?\langle a : \texttt{string}\rangle\ (\!|\textbf{assert } a \neq \text{''''}|\!); b_2!\langle d : \texttt{date}\rangle\ (\!|\,|\!), \mathsf{quit}\ (\!|\,|\!) : end\} \end{array}$$

Finally, a type-checking algorithm can be used to check the type of a program. For instance,

```
cB1 =  ā[2,3](s,b1,b2).                          // Session initialization
       s!⟨``The art of computer programming''⟩;  // Send title to Seller
       b1?⟨quote⟩;                               // Receive quote from Seller
       b2!⟨quote/2⟩                              // Send contribution to Buyer2
```

can be proved to have type pB_1.

3 An architecture for a toolkit

3.1 Objectives

The objective of this work is to describe the architecture of a modular toolkit realising algorithms as those described in § 2. The toolkit we want to realise has to support the following development methodology (see [1] for a concrete realisation). A team of software architects specifies a global description of the distributed interactions which specifies the intended behaviour of the whole system. The global description is checked and projected onto each participant. Then, each part of the system is developed (possibly independently) by a group of programmers. Finally, the pieces of programs are checked, validated, and possibly monitored during the execution. This methodology is supported by the theories drafted in § 2 whereby (1) global descriptions are given by a global assertions, (2) projections yield the parts of the systems to be realised, and (3) compliance of code with the specification is obtained by typing systems (to be matched against the projection). It is therefore possible to statically verify properties of designs/implementations and to automatically generate monitors that control the execution in untrusted environments. Our main driver is that the architecture has to easily allow our toolkit to be adapted to change in the theories; for instance, it has to consistently integrate the two (equivalent) projection algorithms described in [3], or be parametric wrt the logic used in the assertion predicates. Note that our approach distinguishes itself from other works such as [9, 10, 11, 12] by focusing on the tools accompanying the theories and not on the integration of ST in a programming language.

[2]For simplicity, it is not specified how B_2 takes the decision; this can easily be done with suitable assertions on c and q.

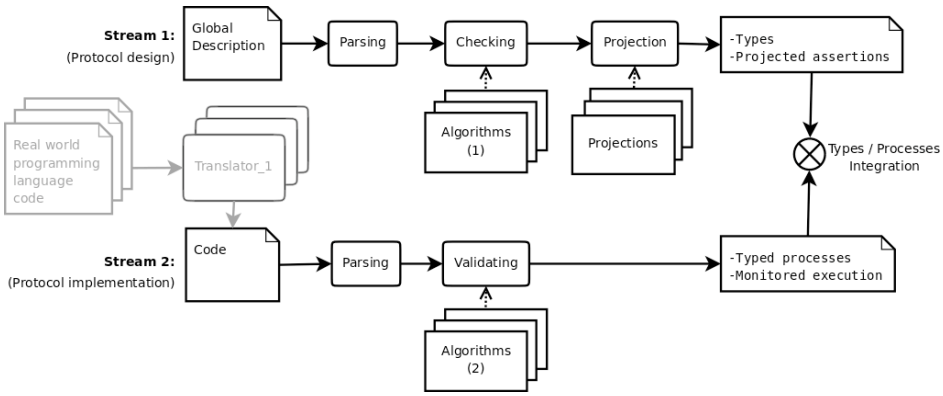


Figure 1: Implementation high-level architecture.

3.2 Architecture

Figure 1 gives a high-level view of the implementation architecture. The toolset has two main inputs. On the one hand, the global description of the interactions, on the other hand the program code of each component (i.e. participant) of the system, written in a π-calculus-like language (e.g., Scribble [2]). Two streams originate from the two inputs. In Stream 1 (top of Figure 1), global descriptions are parsed, checked then projected on each participant. Stream 2 (bottom of Figure 1) takes the code of the programs which is then parsed, typed and validated.

We give a walkthrough of both streams to illustrate the main components of the toolkit. Taking a global description, such as G in § 2, a parser constructs an abstract tree of the distributed interactions, while interacting with the user in case there are syntax errors in the description. The checking module applies a series of algorithms (see (1) in Figure 1) on the tree to check that some properties are guaranteed. At least, the following algorithms will be executed: *one-time unfolding* (unfolding the recursive calls one time is necessary before checking for linearity), *linearity* (ensuring there is no races on the communication channels), *well-assertedness* (ensuring that the assertions respect the constraints of history-sensitivity, locality and temporal-satisfiability). Each algorithm must tell the user in case the description does not satisfy the properties, and accordingly, stop the execution of the process. If the global description is "valid", the projections (like pB_1, pB_2, and pS in § 2) can be calculated. This will be done according to the map defined in [3, §4] which is based on the one in [8, §4.2] with predicates (an implementation of the former shall be able to run the latter, with empty predicates).

In Stream 2, a program code (such as cB_1 in § 2) is parsed to check for syntax errors and build an abstract tree, similarly to the first stream. Then, the tree is given as input to a "validator" which *types* the processes, *validates* their assertions, and checks their *compatibility* (see (2) in Figure 1). The typing module will be able to type a piece of code written in a π-calculus-like language, according to a parameterisable set of typing rules. This will ease the adaptation of the toolkit to support possible extensions of the theory. As in the first stream, each step of the stream will interact with the user in case errors are detected.

Once both streams have been executed successfully, both outputs can be, e.g., compared, and should be compatible since the inputs have passed all the checks. In addition, monitors generated in Stream 1 can be integrated in the code originally input in Stream 2.

An implementation of the toolkit in Haskell is ongoing. Haskell has been chosen because a functional language will keep the implementation close to the underlying theories and is more suitable for a

large class of algorithm in the toolkit (e.g., the typing and projection algorithms can be straightforwardly implemented by exploiting the pattern matching featured by Haskell). Moreover, Haskell provides a convenient means to build a modular architecture; in fact, each component of Figure 1 will be implemented in a different module and polymorphic typing allows to re-use different functions in many different contexts (for instance, to realise the parametricity of the toolkit wrt the assertion logic). Also, stable parsing tools for Haskell are available (e.g. Happy and Alex, which are conveniently combined in BNF Converter[3] [4]).

4 On featuring modularity

In this section we argue on how modularity is featured in our implemantation. We mainly envisage four possible degree of modularity:

1. **Notation.** All inputs and outputs of the implementation (e.g. global assertions, projections, etc.) are encoded in Haskell data types which specify an abstract syntax of the supported languages. This allows to possibly support other notations than the ones originally supported. Notably, the implementation exhibits four data structures to/from which other languages can be translated: *global assertions*, *end-point assertions* (projections), *π-calculus-like language* (participants implementation), *assertion logic*.

2. **Languages.** An important requirement of our modular approach, is that it has to feature the parameterisation of the implementation with respect to the languages used to describe the distributed interations and the associated type systems. For instance, the theory described in [3] abstracts from the actual logical language used to express asserted interactions. Notably, depending on the the chosen language, ad-hoc optimisations can be applied (see 3 below).

3. **Algorithms.** As stated before, the tool will consist of several algorithms than can be used in a modular way (i.e. the user will be able to choose which algorithms s/he needs). For instance, several algorithms can be used in [3, §3.3] to check well-assertedness of assertions; in fact, depending on the adopted logic several formulae manipulation could be applied. Notably, the well-assertedness notion defined in [3] could be replaced by equivalent ones which exploit optimizations on logical formulae. In this way, one could use the close-to-theory algorithms in theoretical experimentation on simple scenario, while more efficient algorithms could be used when considering realistic cases.

4. **Theory.** As the toolkit is developed in a functional language, it allows the theory to be straightforwardly mapped into the programming language. This means that, most of the time, when one wants to change a rule or a definition this can be done by changing only a few lines of code. We illustrate this with an example. The definition of the *dependency relations* ([8, §3.3]) is translated as showed in Figure 2. In the conclusions of [8], the authors comment the adaptation of the theory to support synchrony. Following their idea, this could be done by taking into account output-output dependencies between different names and adding a new dependency from output to input. In our implementation this change could be implemented simply by a few modifications of the code in Figure 2. In particular, we would relax the condition `k1 /= k2` in `OO` and add a new `dep_OI` function for output-input dependencies, similar to the other rules.

[3] BNF Converter generates the (Haskell) skeleton of a parser from a BNF grammar.

II	$n_1 < n_2$ and
	$n_i = p_i \to p : k_i$ (i=1,2)
		`dep_ii :: Prefix -> Prefix -> Bool`
		`dep_ii (Prefix p1 p k1) (Prefix p2 q k2)`
		`| k1 /= k2 = (q == p)`
		`dep_ii _ _ = False`

IO	$n_1 < n_2$,
	$n_1 = p_1 \to p : k_1$ and
	$n_2 = p \to p_2 : k_2$
		`dep_io :: Prefix -> Prefix -> Bool`
		`dep_io (Prefix p1 p k1) (Prefix q p2 k2)`
		`| k1 /= k2 = (q == p)`
		`dep_io _ _ = False`

OO	$n_1 < n_2$,
	$n_i = p \to p_i : k_i$ (i=1,2)
		`dep_oo :: Prefix -> Prefix -> Bool`
		`dep_oo (Prefix p p1 k1) (Prefix p2 q k2)`
		`| k1 /= k2 = (q == p)`
		`dep_oo _ _ = False`

where each dep_** p_1 p_2 assumes that $p_1 < p_2$.

Figure 2: Dependency relations implementation.

5 Conclusions

We have described the architecture of a toolkit for distributed interactions currently under development. The distinguished design principles of the architecture are flexibility and modularity to meet the changes in the theories underlying the toolkit.

The toolkit aims to support a development methodology of communication-centric software based on formal theories of distributed interactions. It is worth mentioning that a similar methodology has recently been adopted in the SAVARA project [1] where *global* and *local* model are used in the development process to validate requirements against implementations. Noteworthy, SAVARA combines state of the art design techniques with ST and provides an open environment where tools based on formal theories can be integrated. It is our intention to integrate (part of) our toolkit in SAVARA. In particular, our toolkit could be used to project the choreography model onto individual services. Namely, SAVARA uses WS-CDL[4] to represent the choreography model, from which it can generates WS-BPEL[5] implementations of indivual services and BPMN[6] diagrams that may be used to guide the implementation. We believe the integration of our tool in SAVARA is feasible and would require the following three main components. Firstly, a mapping from WS-CDL to our Global Assertions. This should be quite straightforward as the Global Types were designed with WS-CDL in mind. However, the support for assertions may require more work as it would require to extend WS-CDL with pre-/post-conditions on messages. Secondly, we need to translate the projections output by our tool to BPEL, BPMN and/or another language, such as the ones used in SAVARA to design/implement services. Finally, we need means to type check the conformance of services against a choreography. This means that we need a tool which translates the (partly) implemented services to a language compliant with (the abstract syntax of) our π-calculus-like language. Technically, these three mappings should be relatively easy to implement as it amounts to transform an XML tree to Haskell data types (and vice-versa). However, careful attention is needed when including the assertions in the notations supported by SAVARA.

Another interesting implementation perspective would be to integrate the algorithms featured by our toolkit in a full-fledged programming language (e.g. Java, similarly to [9]). For example, we conjecture that global assertions could be implemented in two phases. Firstly, a language-independent part could take care of the verification and validation tools which guarantee the good behaviour of programs (i.e.

[4]W3C Web Service Choreography Description Language.
[5]Web Services Business Process Execution Language.
[6]Business Process Modeling Notation.

the implementation of the toolkit described here). Secondly, a language-dependent part could extend a programming language by developing an API which implements the communication primitives (session initialisation, value passing, branching/selection and delegation); while a translator to an abstract language (such as the π-calculus-like language we use) links the API to our toolkit (see faded boxes in Figure 1).

References

[1] SAVARA and the "testable architecture" methodology. `http://www.jboss.org/savara`.

[2] Scribble. `http://sourceforge.net/apps/trac/pi4scribble/wiki`.

[3] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. `http://www.cs.le.ac.uk/people/lb148/assertedtypes.html`.

[4] Björn Bringert, Markus Forsberg, and Aarne Ranta. BNF Converter. `http://www.cs.chalmers.se/Cs/Research/Language-technology/BNFC/`.

[5] Roberto Bruni, Ivan Lanese, Hernán Melgratti, and Emilio Tuosto. Multiparty sessions in SOC. In Doug Lea and Gianluigi Zavattaro, editors, *COORDINATION'08*, volume 5052 of *LNCS*, pages 67–82. Springer Verlag, 2008.

[6] Luís Caires and Hugo Torres Vieira. Conversation types. In *ESOP'09*, pages 285–300, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *In ESOP'98, volume 1381 of LNCS*, pages 122–138. Springer-Verlag.

[8] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284, New York, NY, USA, 2008. ACM.

[9] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. *ECOOP, Springer LNCS*, 5142:2008.

[10] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *In PADL, volume 3057 of LNCS*, pages 56–70. Springer, 2004.

[11] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 25–36, New York, NY, USA, 2008. ACM.

[12] Matthew Sackman and Susan Eisenbach. Session Types in Haskell: Updating Message Passing for the 21st Century. Technical report, June 2008.

Inference of Conversation Types for Distributed Multiparty Systems

Luísa Lourenço Luís Caires

CITI e Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Portugal

Abstract

The Conversation Calculus is a model for distributed communication-centric systems based on the notion of conversation, a generalisation of binary sessions for multi-party interactions over a single shared communication channel. As sessions are disciplined by session types, conversations are disciplined by conversation types, an extension of session types for the conversation interaction model, developed in previous work. Given the fairly rich structure of the underlying type structure, it may not be immediately clear from the proposed type system how types may be inferred for a system, given partial annotations. In this paper, we propose a solution to the conversation type inference problem, proving soundness, completeness and decidability of our algorithm.

1 Introduction

In recent years, there has been an increasing interest in the study and analysis of multiparty service-based system. Several process calculi were designed to model and reason about these systems, namely [5] (based on previous work [4]), and [1]. On top of such models, type systems have been proposed for studying the local and global behavioural correctness of participants in a service-based system. Although type inference for session types has been considered in several works [6, 7], when considering conversation types, given the fairly rich structure of the underlying type structure, and the heavy dependence on a behavioural merge relation, it may not be immediately clear how types may be inferred for a system, given partial annotations.

In this work we present a type inference algorithm for a form of conversation types and show decidability, soundness, and completeness results. Our solution uses standard techniques based on constraint solving (unification). The most challenging aspects of our proposal is the formulation of the particular constraint language used and its combination with the type rules, which has benefited from a direct representation of sequential composition at the level of types.

In section 2 we make a brief introduction to the CC followed by a small example using our language. Section 3, presents our type inference algorithm, an example of its execution, and the correctness results we have obtained. We discuss in section 4 how we can accommodate (iso)recursive types in our type inference algorithm and show that our correctness results are preserved. Finally, we outline some concluding remarks and future work in section 5.

2 Conversation Calculus

The Conversation Calculus (CC) was first introduced in [8] and later refined in [1] and consists in an extension of the π-calculus to allow multiparty conversations (interactions between two or more partners) through a conversation access operator $n \triangleleft [P]$ (P is a process in the context of the conversation n) and context-sensitive communication operators, $l^{\mathsf{d}}!(n)$ and $l^{\mathsf{d}}?(n)$ for output and input, respectively, in either the current conversation context ($\downarrow$) or the enclosing conversation context ($\uparrow$). The syntax is presented in

$$
\begin{array}{lcl}
\mathtt{P,\ Q} & ::= & \mathbf{0} \mid \mathtt{P} \mid \mathtt{Q} \mid (\nu \mathtt{n})\mathtt{P} \mid \mathbf{rec}\,\mathtt{X.P} \\
 & & \mid \mathtt{X} \mid \mathtt{n} \triangleleft [\mathtt{P}] \mid \sum_{i \in I} \alpha_i.\mathtt{P}_i \\
\mathtt{d} & ::= & \uparrow \mid \downarrow \\
\alpha & ::= & \mathtt{l}^{\mathtt{d}}!(\mathtt{n}) \mid \mathtt{l}^{\mathtt{d}}?(\mathtt{n}) \mid \mathbf{this}(\mathtt{x})
\end{array}
$$

Figure 1: Conversation Calculus Syntax

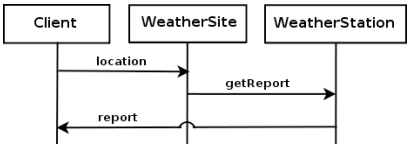


Figure 2: Weather Forecast Message Sequence Chart

Figure 1. We have been developing a concrete distributed language based on CC, which has motivated this work, in this paper we will use sometimes the syntax of our language rather than the formal calculus.

We illustrate our language's syntax through a simple services' use case scenario: a weather forecast service and its client. Upon invocation, the service awaits for the client's location through label location. Then, from the received location, it will ask the nearest weather station to join the on-going conversation (established by the client when invoking the service via **invoke**) and request the desired weather report. The weather station service will, in turn, generate a weather report and send it directly to the client via label report. Notice that the weather station service is capable to communicate directly to the client because it was invoked through the **join** primitive by one of the participants of the conversation, thus is able to join the invoker's conversation instead of creating a new conversation with him. So we have a conversation involving three participants in which one of them dynamically joins. Figure 2 describes the message sequence of our example while Figure 3 shows the code on each participant's site. The message sequence gives a global view of the protocol that every conversation generated by invoking this particular service must comply to (a choreography). Thus, each participant of the conversation must comply with its part of the protocol. Conversation types have been introduced with the aim of statically enforcing correctness of global protocol compliance, given types describing the behaviour of the several participants and of the whole system, for example, $n : [s](B)$ states that site n has a service s whose behaviour is described by behavioural type B.

3 Type Inference

In this section, we will present our type inference algorithm for conversation types and the results obtained, namely we show that it is sound, complete and decidable. Our conversation type system, based on the system of [1], uses judgments of the form $\Gamma \vdash P : T$ where Γ is a set of type declarations, P is the program to be typed and T is a type. In general Γ contains types for remote services, declared in program P using the **remoteType** primitive, and a declaration of the form $this : B$ that describes the current conversation's behaviour B. We show the typing rules for the communication centric fragment of our language in Figure 4. We briefly explain some of the key typing rules. In rule (INVOKE), to typecheck a service invocation we must verify if the body of the invocation has a dual behaviour with the invoked service's behaviour. Then the service invocation is well-typed under the conversation that has the invocation's upper behaviour localised, $loc(\uparrow B)$, i.e. all the message types in the invocation's behaviour that have a up direction correspond to the behaviour of a conversation that invokes the service. In rule (SEND),

```
remoteType WeatherStation: [weatherReport](getReport?(String);report!(String))
site WeatherSite {
  def forecastWeather as {
    val loc = receive(location);
    join weatherReport in
        http://localhost:8000/WeatherStation as { send(getReport); }
  }
};;

site WeatherStation {
  def weatherReport as {
    receive(getReport);
    send(report, generatedReport);
  }
};;

invoke forecastWeather in http://localhost:8000/WeatherSite as {
  send(location, my_location);
  val my_weather_report = receive(report);
  println my_weather_report
};;
```

Figure 3: Code for WeatherSite, WeatherStation, and Client.

we say a send typechecks under the conversation with message type $l!(\beta)$ if the value sent has type β.

The type inference algorithm takes as input a program P, a set of remote types declarations (in a typing environment Γ), an initially empty set of constraints on types R, and an initially empty set of apartness restrictions A. The algorithm outputs the type of program P, the typing environment Γ' where we can typecheck program P, and a set of constraints R' that respects the set of apartness restrictions A'

$$\textbf{typecheck}(\mathrm{P}, \Gamma, \mathrm{R}, \mathrm{A}) = (\mathrm{T}, \Gamma', \mathrm{R}', \mathrm{A}')$$

The algorithm consists of the following steps. First, it transverses the abstract syntax tree, applying typing rules backward if possible. Whenever a type needs to be inferred, a constraint is generated and added to the set R. Finally, the system of equations, represented by all constraints of R is manipulated by applying transformation rules until the system is either in solved form, or type inference fails. During constraint solving, matching labels may need to be synchronised (for e.g., when we have a merge constraint on two dual labels). To ensure linearised usage, we have introduced a new kind of constraint (checked on each transformation step) to state that a label cannot occur in a given type. We denote this as an apartness restriction $l\#B$ (added to the apartness set A), with l being the label that can not occur in type B. As expected, if at any moment a step can not be executed the algorithm aborts since the program must be ill-typed.

The unification algorithm receives as input a constraint set R whose constraints represent a system of equations, and a set of apartness restrictions A. After solving all the constraints in R, the algorithm outputs a set of constraints R' (in particular, a substitution) respecting the set of apartness restrictions A'

$$\textbf{solve}(\mathrm{R}, \mathrm{A}) = (\mathrm{R}', \mathrm{A}')$$

The constraints generated by the type inference algorithm have the form $< E, E' >$ according to the

$$\frac{\Gamma,\ this : B_1 \vdash P_1 \qquad \ldots \qquad \Gamma,\ this : B_n \vdash P_n}{\Gamma,\ this : B_1 \bowtie \ldots \bowtie B_n \vdash \mathrm{P}_1 || \ldots || \mathrm{P}_n}\ \text{(PAR)} \qquad \frac{\Gamma,\ this : B_1 \vdash P_1 \qquad \Gamma,\ this : B_2 \vdash P_2}{\Gamma,\ this : B_1;B_2 \vdash \mathrm{P}_1;\mathrm{P}_2}\ \text{(SEQ)}$$

$$\frac{\Gamma,\ this : B,\ n : [s](B_1) \vdash P}{\Gamma,\ this : B_1 \bowtie B,\ n : [s](B_1) \vdash \textbf{join}\ \mathrm{s}\ \textbf{in}\ \mathrm{n}\ \textbf{as}\ \{\ \mathrm{P}\ \}}\ \text{(JOIN)} \qquad \frac{}{\Gamma,\ this : l?(\beta) \vdash \textbf{receive}(\mathrm{l}){:}\beta}\ \text{(RECV)}$$

$$\frac{\Gamma,\ this : B \vdash P}{\Gamma,\ this : [s](\downarrow B);loc(\uparrow B) \vdash \textbf{def}\ \mathrm{s}\ \textbf{as}\ \{\ \mathrm{P}\ \}}\ \text{(DEF)} \qquad \frac{\Gamma \vdash E : \beta}{\Gamma,\ this : l!(\beta) \vdash \textbf{send}(\mathrm{l},\mathrm{E})}\ \text{(SEND)}$$

$$\frac{\Gamma,\ this : B,\ n : [s](B_1) \vdash P \qquad B_1 = \overline{\downarrow B}}{\Gamma,\ this : loc(\uparrow B),\ n : [s](B_1) \vdash \textbf{invoke}\ \mathrm{s}\ \textbf{in}\ \mathrm{n}\ \textbf{as}\ \{\ \mathrm{P}\ \}}\ \text{(INVOKE)} \qquad \frac{\Gamma,\ this : B \vdash P}{\Gamma,\ n : B \vdash \textbf{site}\ \mathrm{n}\ \{\mathrm{P}\}}\ \text{(SITE)}$$

$$\frac{\Gamma,\ this : B_1 \vdash E : \beta \qquad \Gamma,\ this : B_2,\ x : \beta \vdash P}{\Gamma,\ this : B_1;B_2 \vdash \textbf{let}\ \mathrm{x} = \mathrm{E}\ \textbf{in}\ \{\ \mathrm{P}\ \}}\ \text{(LET)}$$

$$\frac{\Gamma,\ this : B_1 \vdash P_1 \qquad \ldots \qquad \Gamma,\ this : B_n \vdash P_n}{\Gamma,\ this : \&\{l_1 : B_1;\ \ldots;\ l_n : B_n\} \vdash \textbf{select}\ \{\ \mathrm{l}_1{:}\ \mathrm{P}_1;\ \ldots;\ \mathrm{l}_n{:}\ \mathrm{P}_n\}}\ \text{(SELECT)}$$

$$\frac{\begin{array}{l} if\ l_i = l_j\ then\ B_i = B_j \quad i,j \in \{1,\ \ldots,\ n\} \quad m <= n \\ \Gamma \vdash E_1 : Bool \quad \ldots \quad \Gamma \vdash E_n : Bool \quad \Gamma,\ this : B_1 \vdash P_1 \quad \ldots \quad \Gamma,\ this : B_n \vdash P_n \quad \Gamma,\ this : B_d \vdash P_d \end{array}}{\Gamma,\ this : \oplus\{l_1 : B_1;\ \ldots;\ l_m : B_m\} \vdash \textbf{switch}\ \{\ \textbf{case}\ (\mathrm{E}_1)\textbf{do}\ \mathrm{l}_1{:}\ \mathrm{P}_1;\ldots;\ \textbf{case}\ (\mathrm{E}_n)\textbf{do}\ \mathrm{l}_n{:}\ \mathrm{P}_n;\ \textbf{default do}\ \mathrm{l}_i{:}\ \mathrm{P}_d\}}\ \text{(SWITCH)}$$

Figure 4: Some Typing Rules of our Type System.

syntax presented in Figure 5b. We have standard constraints like $< x, T >$, stating that a type variable x has type T (either behavioural type B or a basic type β). In the unification algorithm, these are treated using the standard transformation rules for variable elimination and type equality [3], a solvable system terminates in a system in solved form, that corresponds to a substitution.

A distinguishing aspect of our constraint structure is a merge constraint on types of the form $< x, \bowtie(B, B') >$, that constrains type variable x to be a "composition" of behavioural types B and B' (this operation is defined by a *merge relation*, see [1]). Merge constraints are necessary to approximate the type of a parallel composition, rule (PAR) in Figure 4, (where synchronisation can happen) or when we invoke a service via the join primitive, rule (JOIN) in Figure 4, (since we merge the behaviours of the invoked service with the client's). Thus we need to be able to represent the merge of all behaviour in the composition such that casual ordering is kept and interleaves are avoided unless there is a synchronisation: this way, the most general (less serialised) behaviour is computed. Merge constraints are solved using a set of transformation rules that represent the merge relation on behavioural types. We now present the transformation rules.

Definition 3.1 (Non Interference)**.** *We say two behavioural types, B_i and B_j, do not interfere with each other, denoted as $B_i \# B_j$, if B_i has no label that can synchronise with some label in B_j, and conversely.*

$$
\begin{array}{lcl}
C & ::= & [s](B) \\
B & ::= & B_1 \mid B_2 \mid \mathbf{0} \mid B_1;B_2 \mid M \\
 & & \mid rec\ X.B \mid X \\
 & & \mid \oplus\{l_1 : B_1;\ \ldots;\ l_n : B_n\} \\
 & & \mid \&\{l_1 : B_1;\ \ldots;\ l_n : B_n\} \\
M & ::= & lp^d(\beta) \\
\beta & ::= & Int \mid Bool \mid String \mid Array(\beta) \\
 & & \mid \beta \xrightarrow{B} \beta' \mid Ref(\beta) \mid Unit \\
p & ::= & ! \mid ? \mid \tau \\
d & ::= & \uparrow \mid \downarrow
\end{array}
$$

(a) Syntax of Types

$$
E \quad ::= \quad B \mid \beta \mid \bowtie(E,E') \mid x
$$

(b) Syntax of constraints on types

Figure 5: Syntax of Types and Constraints

Definition 3.2 (Transformations Rules)**.** *Let R denote a system of equations, t a term, A an apartness constraint set, and T a type. We define the transformations rules, $R \Longrightarrow^A R'$ (if R does not violate any apartness constraint in A, then we can transform to a system R' that also complies with A), as follows:*

Trivial:

$$\{<t,t'>\} \cup R \Longrightarrow^A_{triv} R$$

where $t \equiv t'$

Variable Elimination:

$$\{<x,T>\} \cup R \Longrightarrow^A_{elem} \{<x,T>\} \cup R[^T/_x]$$

such that $x \notin Var(T)$

Merge Trivial:

$$\{<x,\bowtie(B)>\} \cup R \Longrightarrow^A_{merge\_trivial} \{<x,B>\} \cup R$$

Merge Inact:

$$\{<x,\bowtie(B_1,\ldots,B_{i-1},0,B_{i+1},\ldots,B_n)>\} \cup R \Longrightarrow^A_{merge\_inact}$$
$$\{<x,\bowtie(B_1,\ldots,B_{i-1},B_{i+1},\ldots,B_n)>\} \cup R$$

Merge Parallel:

$$\{<x,\bowtie(B,\ldots,B_1|B_2,\ldots,B')>\} \cup R \Longrightarrow^A_{merge\_par}$$
$$\{<x,\bowtie(B,\ldots,B_1,B_2,\ldots,B)>\} \cup R$$

Merge Sync:

$$\{<x,\bowtie(B_1,\ldots,B_i;lp_1^d;B_i',\ldots,\ B_j;lp_2^d;B_j',\ldots,B_n)>\} \cup R \Longrightarrow^A_{merge\_sync}$$

$$\{<x,(B_i \mid B_j);l\tau^d;y>\} \cup \{<y,\bowtie(B_1,\ldots,B'_i,\ldots,B'_j,\ldots,B_n)>\} \cup \sigma(R)$$

where p_1 is the opposite polarity of p_2, $A = A \cup \{l\#y\}$, and $B_i\#B_j$ and for all B_k, $B_i\#B_k$ and $B_j\#B_k$ with $k \in \{1,\ldots,n\}$ and $k \neq i \neq j$, and $\sigma = [^{(B_i \mid B_j);l\tau^d;y}/_x]$.

Merge Choice Sync:

$$\{<x,\bowtie(B_1,\ldots,B_i;C;B'_i,\ldots,\ B_j;D;B'_j,\ldots,B_n)>\} \cup R \Longrightarrow^A_{merge\_sync2}$$
$$\{<x,(B_i \mid B_j);\oplus\{l_1:y_1,\ldots,l_n:y_n\};y>\}\cup$$
$$\{<y,\bowtie(B_1,\ldots,B'_i,\ldots,B'_j,\ldots,B_n)>\}\cup$$
$$\{<y_1,\bowtie(B_{c1},B'_{c1})>\}\cup\ldots\cup\{<y_n,\bowtie(B_{cn},B'_{cn})>\}\cup\sigma(R)$$

where $C = \&\{l_1 : B_{c1},\ \ldots,\ l_n : B_{cn}\}$ and $D = \oplus\{l_1 : B'_{c1},\ \ldots,\ l_n : B'_{cn}\}$, and $B_i\#B_j$ and for all B_k, $B_i\#B_k$ and $B_j\#B_k$ with $k \in \{1,\ldots,n\}$ and $k \neq i \neq j$, and $\sigma = [^{(B_i \mid B_j);\oplus\{l_1:y_1,\ldots,l_n:y_n\};y}/_x]$.

We will now explain one of the rules that represent our merge relation, namely the Merge Sync rule. This rule is applied when a synchronisation is possible between the types the terms represent. Its application ensures that when we synchronise a label then the preceding type on each side, B_i and B_j, must not interfere with each other, $B_i\#B_j$ and thus can be safely composed into a parallel composition of types, $B_i|B_j$. A new equation is generated to merge the remaining types of the merge along with the remainder of both sides' types, B'_i and B'_j. Since labels must be used linearly, we impose an apartness restriction, $l\#y$, stating that the synchronised label, l, can not occur in the new equation, y. Lastly, we eliminate the solved variable x in the remaining constraints in R by applying the substitution σ to R.

The unification algorithm is confluent. This implies that our type inference algorithm is deterministic since the application of typing rules is syntax driven, and also due to the non-interference conditions on the merge relation mentioned above. The solution returned is represented as a mapping on variables to types, i.e. as a substitution for the variables of the system. The type inference algorithm then succeeds if such a solution exists. Furthermore, our algorithm always determines the most general type since we serialise the composition of two types only when there is a synchronisation between them, thus keeping them as general as possible by composing into a parallel composition of types.

We illustrate the application of our algorithm to the code of the weather forecast service's site in Figure 3.

As input we have

P = code of weather forecast's site as shown in Figure 3
Γ = { WeatherStation:[weatherReport](getReport?(String);report!(String)) }
R = A = ∅

So **typecheck**(P, Γ, R, A) takes the following steps:

1. Inductively, applies type inference rules;
2. When typechecking the join primitive, a type variable **x** is created as well as a constraint to represent the merge of the primitive's body's behaviour with the behaviour of the invoked service, <x, ⋈(getReport!(String), getReport?(String);report!(String))> that is added to R;
3. The constraint is solved upon the typecheck of the site primitive and the algorithm terminates.

The algorithm outputs (Γ ∪ WeatherSite:[*weatherForecast*](location?(z);B), R, A′) where B is the type obtained by the unification algorithm (function **solve**) when solving the constraint on **x**, and A' the resulting apartness set.

In step 3 the unification algorithm is called with the following input:

R = { <x, ⋈(getReport!(String), getReport?(String);report!(String))> }
A = ∅

So **solve**(R, A) takes the following steps:

<x, ⋈(getReport!(String), getReport?(String);report!(String))> $\Longrightarrow^{A}_{merge\_sync}$

<x, getReportτ(String);y> ∪ <y, ⋈(∅, report!(String))> $\Longrightarrow^{A'}_{merge\_inact}$
with A' = { getReport#y}

<x, getReportτ(String);y> ∪ <y, ⋈(report!(String))> $\Longrightarrow^{A'}_{merge\_trivial}$

<x, getReportτ(String);y> ∪ <y, report!(String)>

We then obtain B = getReportτ(String);report!(String), R = { <x, getReportτ(String);y>, <y, report!(String)> }, and A$'$ = { getReport#y }.

For a negative case, suppose that we make use of label **getReport** instead of the label **report** to transmit the requested report. This would violate the condition of labels being used linearly because then, at one point, we would have two receivers for the empty message sent by the **forecastWeather** service. This type of errors are detected by the algorithm when solving the constraints. In our example, the unification algorithm would instead solve variable **y** with type **getReport!(String)** which would violate the apartness restriction **getReport#y** and therefore the unification algorithm would abort. Thus typechecked programs always comply with linear usage of labels inside conversations.

We denote a substitution application as the application of a constraint set to a typing environment, R(Γ), and define it as being the substitution of all occurrences of a type variable in the input environment with its corresponding type if, and only if, the constraint associated with the type variable is solved in R.

We conclude by presenting the main results for the type inference algorithm which are the decidability, soundness and completeness of the algorithm. The first states that if a program *P* can be typechecked by the typechecking algorithm, then there is a type derivation for any type instance of the generated inferred type.

Theorem 3.3 (Soundness of Typechecking Algorithm)**.** *Let P be a program,* Γ, Γ' *type environments , T a type, A a set of apartness constraints, and R a constraint set.*

Assume $\texttt{typecheck}(\texttt{P}, \Gamma, \emptyset, \emptyset) = (\texttt{T}, \Gamma', \texttt{R}, \texttt{A})$. *Then* $\texttt{R}(\Gamma') \vdash \texttt{P} : \texttt{R}(\texttt{T})$ *and there is a substitution* θ *such that* $\theta(\texttt{R}(\Gamma')) \vdash \texttt{P} : \theta(\texttt{R}(\texttt{T}))$

The second results states that no typing is lost; if a program *P* is typable then the type-checking algorithm terminates with a typing, of which the given typing is an instance.

Theorem 3.4 (Completeness of Typechecking Algorithm)**.** *Let P be a program,* Γ *a typing context,* Γ' *a typing context containing only type declarations of remote services, and T a type*

Assume $\Gamma \vdash \texttt{P} : \texttt{T}$. *Then* $\texttt{typecheck}(\texttt{P}, \Gamma', \emptyset, \emptyset) = (\texttt{T}', \Gamma'', \texttt{R}', \texttt{A}')$ *and there is a substitution* θ *such that* $\theta(\texttt{R}'(\texttt{T}')) = \texttt{T}$ *and* $\theta(\texttt{R}'(\Gamma'')) = \Gamma$ *and* $\theta(\texttt{R}'(\Gamma')) \subseteq \Gamma$

Decidability follows from the termination proof, which depends essentially on the termination of the unification algorithm, in turn based on standard well-founded orderings. In particular, we define a pair $< m, n >$ such that *n* is the number of variables unsolved in R and *m* the sum of the sizes of each term in R, then the lexicographic order of such pairs is a well-founded relation. We then prove that every

$$\frac{\Gamma,\ this : B;X \vdash P}{\Gamma,\ this : rec\ X.B;X \vdash \textbf{rec X}.\text{P}}$$

(a) Rec rule

$$\frac{}{\Gamma,\ this : X \vdash X}$$

(b) Var rule

Figure 6: Typing Rules for Recursive Behaviour Constructions.

transformation sequence terminates since each transformation results in a system where the pair $< m,n >$ is smaller under the lexicographic ordering.

4 Recursive Types

We have mainly focused on the finite part of conversation types, in this section we discuss how a simple system of iso-recursive is to be accommodated using standard techniques for unifying recursive equations on our constraint's language. Although we expect that general equi-recursive types may be also accommodated along the lines of [2], when dealing with recursive definitions in our language (such as a recursive functions), we don't focus on that issue in this paper. Instead, we develop here a simple solution, based on the simple interpretation of recursive types. Regarding recursive behaviour's constructions like CC's **rec X.P** (Figure 6), these would not introduce a recursive equation and therefore their treatment in our theory consists in adding two transformation rules to merge two recursive behavioural types, and to merge two recursive variables, respectively:

$$\{< x, \bowtie(B_1,\ldots,B_i;rec\ X.B;B'_i,\ldots,\ B_j;rec\ X.B';B'_j,\ldots,B_n) >\} \cup R \Longrightarrow^{A}_{merge\_rec}$$
$$\{< x, (B_i \mid B_j);rec\ X.y >\} \cup \{< y, \bowtie(B_1,\ldots,B;B'_i,\ \ldots,\ B';B'_j,\ldots,B_n) >\} \cup \sigma(R)$$

where $B_i \# B_j$ and for all B_k, $B_i \# B_k$ and $B_j \# B_k$ with $k \in \{1,\ldots,n\}$ and $k \neq i \neq j$, and $\sigma = [^{(B_i \mid B_j);rec\ X.y}/_{x}]$.

$$\{< x, \bowtie(B_1,\ldots,B_i;X;B'_i,\ldots,\ B_j;X';B'_j,\ldots,B_n) >\} \cup R \Longrightarrow^{A}_{merge\_recvar}$$
$$\{< x, (B_i \mid B_j);X;y >\} \cup \{< y, \bowtie(B_1,\ldots,B'_i,\ldots,B'_j,\ldots,B_n) >\} \cup \sigma(R)$$

where $B_i \# B_j$ and for all B_k, $B_i \# B_k$ and $B_j \# B_k$ with $k \in \{1,\ldots,n\}$ and $k \neq i \neq j$, and $\sigma = [^{(B_i \mid B_j);X;y}/_{x}]$.

We illustrate the application of these new rules on our previous example with a minor change, Figure 7. Notice that the **while** construction can be perceived as CC's **rec** X.P construction. In this case, in step 3 of the typechecking procedure, the unification algorithm is called with the following input:

R = { <x, ⋈(rec X.getReport!(String);X, rec X.getReport?(String);X;report!(String))> }
A = ∅

So **solve(R, A)** takes the following steps:

<x, ⋈(rec X.getReport!(String);X, rec X.getReport?(String);X;report!(String))>
$\Longrightarrow^{A}_{merge\_rec}$

<x, rec X.y> ∪ <y, ⋈(getReport!(String);X, getReport?(String);X;report!(String))>

```
remoteType WeatherStation: [weatherReport](rec X.getReport?(String);X;report!(String))
site WeatherSite {
  def forecastWeather as {
    val loc = receive(location);
    join weatherReport in
        http://localhost:8000/WeatherStation as {
          while(cond) do
            send(getReport);
        }
  }
};;
```

Figure 7: Weather Forecast Site Code Revisited.

$$\Longrightarrow^{A}_{merge\_sync}$$

$$<x, \text{rec } X.y> \cup <y, \text{getReport}\tau(\text{String});z> \cup <z, \bowtie(X, X;\text{report!(String)})>$$

$$\Longrightarrow^{A'}_{merge\_recvar}$$

with $A' = \{$ getReport#z$\}$

$$<x, \text{rec } X.y> \cup <y, \text{getReport}\tau(\text{String});z> \cup <z, X;w> \cup <w, \bowtie(\emptyset, \text{report!(String)})>$$

$$\Longrightarrow^{A'}_{merge\_inact}$$

$$<x, \text{rec } X.y> \cup <y, \text{getReport}\tau(\text{String});z> \cup <z, X;w> \cup <w, \bowtie(\text{report!(String)})>$$

$$\Longrightarrow^{A'}_{merge\_trivial}$$

$$<x, \text{rec } X.y> \cup <y, \text{getReport}\tau(\text{String});z> \cup <z,X;w> \cup <w, \text{report!(String)}>$$

We then obtain B = rec X.getReportτ(String);X;report!(String), A$'$ = { getReport#z }, and R = { <x, rec X.y>, <y, getReportτ(String);z>, <z, X;w>, <w, report!(String)> }.

Our basic results are not affected by our treatment of recursion. Namely, decidability is preserved since the new transformation rules preserves the well-founded ordering defined for the non-recursive case.

5 Concluding Remarks

We have presented a type inference algorithm for conversation types and proved it to be decidable, sound and complete. Our contributions essentially focus on the finite aspects, which are already challenging, due to the parallel-sequential behavioural algebra embedded in conversation types, and the need of coping with the behavioural merge of behaviours originating in multiple interaction partners, including dynamic conversation join and leave. We also showed how to accommodate recursive behaviour's constructions in our type inference algorithm and proved that our results are still preserved. For future work we wish to devise a subtyping algorithm for conversation types, and more general solution for handling recursion.

Acknowledgements. We thank Hugo Vieira and the anonymous referees for his insightful comments and suggestions regarding this paper. This work was supported by a grant from EU Project SENSORIA, and CMU-Portugal INTERFACES, funded by FCT/MCTES and ICTI.

References

[1] Luís Caires and Hugo Torres Vieira. Conversation types. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2009.

[2] Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983.

[3] Jean H. Gallier and Wayne Snyder. Complete sets of transformations for general e-unification. *Theor. Comput. Sci.*, 67(2&3):203–260, 1989.

[4] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[5] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pages 273–284. ACM, 2008.

[6] Leonardo Gaetano Mezzina. How to infer finite session types in a calculus of services and sessions. In Doug Lea anzd Gianluigi Zavattaro, editor, *Coordination Models and Languages, 10th International Conference, COORDINATION 2008. Proceedings*, volume 5052 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2008.

[7] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 2009.

[8] Hugo Torres Vieira, Luís Caires, and João Costa Seco. The conversation calculus: A model of service-oriented computation. In Sophia Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008. Proceedings*, volume 4960 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2008.

An Investigation on Types for X10 Clocks

Francisco Martins
LaSIGE & University of Lisbon, Portugal
fmartins@di.fc.ul.pt

Vasco T. Vasconcelos
LaSIGE & University of Lisbon, Portugal
vv@di.fc.ul.pt

Tiago Cogumbreiro
LaSIGE & University of Lisbon, Portugal
cogumbreiro@di.fc.ul.pt

Abstract

The X10 language provides the notion of clocks as a means of coordinating concurrent programs. In order to better understand the concept we study a type system for a stripped down version of X10. The main expected result is a progress property for typable programs. The study will open, we hope, doors to a more flexible utilisation of the clocks constructs in the X10 language.

Introduction New high-level concurrency primitives are needed more than ever, now that multicore machines lay on our desks and laps. One such primitive is *clocks*, a generalization of barriers, as present in the X10 programming language [1]. Clocks allow multiple concurrent activities to synchronise at a sequence of points in time.

Even though the language specification [3] provides a clear, plain English, description of the intended semantics (and properties) of the language, and a formalization of the semantics [4] allows to prove a deadlock freedom theorem, we decided to investigate a simpler setting in which similar results could be obtained. The aim is not only to obtain a progress property for typable programs based on simple type system, but also to hopefully provide for clock-safe extensions of the X10 language itself.

Towards this end, we have stripped X10 from most of its features, ending up with a simple concurrent language equipped with the full functionality of X10 clocks, which we call "X10 restricted to clocks," $\mathsf{X10}\,|_{\mathsf{clocks}}$ for short. For this language we have devised a simple operational semantics with thread (or activity as called in X10) local and global views of (heap allocated) clocks. We have also crafted a simple type system, based on singleton types, drawing expertise from previous work on low-level programming languages [7]. The type system enjoys subject-reduction. Typable programs are exempt from the clock related errors, as reported in the specification of the language [3]. We conjecture that typable programs enjoy a progress property.

There seems to be no formal account of clocks in the X10 language available on the literature. Saraswat and Jagadeesan study a bisimulation for X10 allowing to establish that programs do not deadlock, under certain conditions [4]. Lee and Palsberg present a core language for X10 suited to study the async-finish problem [2].

In summary, the contributions of this work are a) a simple operational semantics for clocks that allows to better understand the concept, b) a simple type systems allowing to prove safety and progress properties (alternative to the constraint-based system [4]), and c) the promise of a more flexible utilization of the clock constructs. The rest of this abstract presents the syntax, operational semantics, type system and main results of the language, and future work, in this order.

Syntax X10 is a modern language built from the ground up to handle future parallel systems: from multicore machines to cluster configurations [3]. Object-oriented and type-safe, X10 boasts support for concurrency, parallelism, and distribution. Of particular interest to us, the language provides a *clock* construct for synchronising multiple concurrent activities to wait for each other at certain points in time (or *phases*).

e ::=		*Expressions*
	v	value
	$\mid$ **async** $\vec{e}\ e$	fork
	$\mid$ **make**	new clock
	$\mid$ **drop** e	drop
	$\mid$ **resume** e	resume
	$\mid$ **next**	next
	$\mid$ **wait**	wait
	$\mid$ **clocked** e	clocked
	$\mid$ **let** $x = e$ **in** e	let
	$\mid$ **if** e **then** e **else** e	conditional
	$\mid \ldots$	

v ::=		*Values*
	x	variable/clock
	$\mid$ **true** $\mid$ **false**	boolean
	$\mid$ $()$	unit
V ::=	$\{c_1 : i_1, \ldots, c_n : i_n\}$	*Clocks local view*
a ::=	$\langle V, e \rangle$	*Activity*
A ::=	$\{l_1 : a_1, \ldots, l_n : a_n\}$	*Activities*
R, S ::=	$\{l_1, \ldots, l_n\}$	*Sets of activities*
h ::=	$\langle i, R, S \rangle$	*Clock values*
H ::=	$\{c_1 : h_1, \ldots, c_n : h_n\}$	*Heaps*
S ::=	$\langle H; A \rangle$	*States*

$\mathscr{E}$ ::= *Contexts*

$[\,] \mid$ **resume** $\mathscr{E} \mid$ **drop** $\mathscr{E} \mid$ **async** $(\vec{v}\mathscr{E}\vec{e})\ e \mid$ **clocked** $\mathscr{E} \mid$ **let** $x = \mathscr{E}$ **in** $e \mid$ **if** $\mathscr{E}$ **then** e **else** e

Figure 1: Syntax of $\mathsf{X10}|_{\mathsf{clocks}}$

We present a subset of the X10 language restricted to clocks, $\mathsf{X10}|_{\mathsf{clocks}}$, generated by the grammar in Figure 1, and relying on the following base sets: natural numbers ranged over by i, variables (also used for clock identifiers and heap addresses) ranged over by c or x, and activity identifiers ranged over by l.

Activities can create an arbitrary number of clocks with expression **make** and communicate them to spawned activities. There are only two forms for activities to be registered with clocks: (a) upon activity creation (**async** $\vec{e}\ e$) the new activity becomes registered with clock sequence $\vec{e}$, and (b) when creating a clock the activity is automatically registered with the new clock. Expression **drop** de-registers an activity from a clock. Activities are disallowed to manipulate clocks they are not registered with.

Clocks can be thought of as data structures holding a natural number representing its *global phase*, initially set to zero. A clock can be *advanced* to its next phase, which amounts to increment its global phase when every registered activity has *quiesced*; an activity is quiescent on a clock after performing a **resume** on that clock. An activity resumes all clocks it is registered with by executing **next** and suspends itself until these clocks become ready to advance to the next phase. Expressions **clocked** and **wait** are only available to the run-time syntax. **wait** describes an activity that has performed a **next** operation and awaits other activities; **clocked** describes the activity resulting from an **async** operation.

The state of a $\mathsf{X10}|_{\mathsf{clocks}}$ program consists of a heap H and a set of indexed activities A. The heap stores clock values that contain a natural number i representing its global phase, a set R describing the **r**egistered activities, and another set S for the re**s**umed activities. Each activity a is composed of the (registered) clocks local view V and the expression e under execution. The clocks local view V is a map from clock references to natural numbers describing the local phase. The global phase for a given clock, stored in the heap, is at most one phase ahead of each local view.

Operational Semantics Reduction rules for $\mathsf{X10}|_{\mathsf{clocks}}$ are presented in Figure 2. We focus on the semantics for clocks and activities and omit rules regarding control-flow, since they are fairly standard.

Reduction is defined for states via rules R-ASYNC and R-ACTIVITY. This last rule is a context rule

$$\frac{H(c) = \langle p,R,S\rangle \qquad V(c) = p' \qquad S' = \text{if } l \in S \text{ then } S \cup \{l'\} \text{ else } S \qquad l' \text{ is fresh}}{\langle H; A\{l\colon \langle V, \mathscr{E}[\mathsf{async}\ c\ e]\rangle\}\rangle \to_l \langle H\{c\colon \langle p, R\cup\{l'\}, S'\rangle\}; A\{l\colon \langle V, \mathscr{E}[()]\rangle\}\{l'\colon \langle\{c\colon p'\}, \mathsf{clocked}\ e\rangle\}\rangle} \quad \text{(R-ASYNC)}$$

$$\frac{H;V;e \to_l H';V';e'}{\langle H; A\{l\colon \langle V, \mathscr{E}[e]\rangle\}\rangle \to_l \langle H'; A\{l\colon \langle V', \mathscr{E}[e']\rangle\}\rangle} \quad \text{(R-ACTIVITY)}$$

$$\frac{c \text{ is fresh}}{H;V;\mathsf{make} \to_l H\{c\colon \langle 0,\{l\},\emptyset\rangle\}; V\{c\colon 0\}; c} \quad \text{(R-MAKE)}$$

$$\frac{H(c) = \langle p,R,S\rangle \qquad V(c) = p' \qquad S' = \text{if } p = p' \text{ then } S \cup \{l\} \text{ else } S}{H;V;\mathsf{resume}\ c \to_l H\{c\colon \langle p,R,S'\rangle\}; V; ()} \quad \text{(R-RESUME)}$$

$$\frac{C \text{ is the set } \{c \mid V(c) = p, H(c) = \langle p,R,S\rangle\}}{H;V;\mathsf{next} \to_l H\{c\colon \langle p,R,S\cup\{l\}\rangle\}_{c\in C}; V; \mathsf{wait}} \quad \text{(R-NEXT)}$$

$$\frac{\begin{array}{c} C_1 \text{ is the set } \{c \mid V(c) = p, H(c) = \langle p,R,R\rangle\} \\ C_2 \text{ is the set } \{c \mid V(c) = p, H(c) = \langle p+1,\_,\_\rangle\} \qquad C_1 \cup C_2 = \operatorname{dom} V \end{array}}{H;V;\mathsf{wait} \to_l H\{c\colon \langle p+1,R,\emptyset\rangle\}_{c\in C_1}; \{c\colon V(c)+1\}_{c\in V}; ()} \quad \text{(R-WAIT)}$$

$$\frac{H(c) = \langle p,R,S\rangle \qquad c \in \operatorname{dom} V}{H;V;\mathsf{drop}\ c \to_l H\{c\colon \langle p,R\setminus\{l\},S\setminus\{l\}\rangle\}; V\setminus\{c\}; ()} \quad \text{(R-DROP)}$$

$$\frac{H(c) = \langle p,R,S\rangle \qquad \forall c \in \operatorname{dom} V}{H;V;\mathsf{clocked}\ v \to_l H\{c\colon \langle p,R\setminus\{l\},S\setminus\{l\}\rangle\}_{c\in\operatorname{dom} V}; \emptyset; v} \quad \text{(R-CLOCKED)}$$

Figure 2: Reduction rules (clock related only)

for expressions; derivations of a reduction step are immediately preceded by one of the subsequent rules in the figure (from rule R-MAKE onward).

When creating a new activity (rule R-ASYNC) the programmer specifies the clock c on which the new activity is to be registered with.[1] The newly created activity identifier l' is added to the set of registered activities in clock c (in the heap), and a new activity is added to the set of activities. This new activity is composed of a clock view of clock c and of a clocked e expression that is responsible for making sure all registered clocks are dropped before e terminates. The new activity inherits clock c local view from activity l ($c\colon p'$), as well as the quiescence property of l with respect to c, i.e., if l is quiescent on clock c so is l' ($S' = \text{if } l \in S \text{ then } S \cup \{l'\}$ else S).

Expression make creates a new clock in the heap with phase 0, with l as the only registered activity, and with no resumed activities, $\langle 0,\{l\},\emptyset\rangle$. The activity creating the clock gains access to it though a local clock view $\{c\colon 0\}$ stored in the activity's clock local view V. Rule R-RESUME asserts that when the l-th activity issues a resume c, we record its index in the set of resumed activities R if the clock local phase is in sync with the clock global phase ($p = p'$); otherwise, the effect of the expression is discarded ($p \neq p'$), since the clock has already advance to the next phase. Expression next resumes all clocks held by the current activity and evaluates into wait (rule R-NEXT), which in turn blocks the activity until all clocks have been resumed (C_1) or have already advance their phases (C_2) (rule R-WAIT). Notice that when activities are waiting on a clock c, the clock can be in one of three states: (a) there are non-quiescent activities on the clock and c is neither a member of C_1 nor of C_2; (b) all registered activities are quiescent on the clock, and so c is a member of C_1; (c) the clock has advanced to the next phase thus becoming member of C_2. When an activity advances a clock global phase, it stops being a member of set

[1] For the sake of simplicity we define the operational semantics and the type system for async expressions on a single clock only; the extension to sequences of clocks of arbitrary length should be straightforward.

$$\langle \_; A\{l\colon \langle V, \mathscr{E}[\mathsf{async}\ c\ e]\rangle\}\rangle \quad \text{if } c \notin \operatorname{dom} V \qquad \text{(E-ASYNC-NO-CLOCK)}$$
$$\langle \_; A\{l\colon \langle V, \mathscr{E}[\mathsf{resume}\ c]\rangle\}\rangle \quad \text{if } c \notin \operatorname{dom} V \qquad \text{(E-RESUME-NO-CLOCK)}$$
$$\langle \_; A\{l\colon \langle V, \mathscr{E}[\mathsf{drop}\ c]\rangle\}\rangle \quad \text{if } c \notin \operatorname{dom} V \qquad \text{(E-DROP-NO-CLOCK)}$$

Figure 3: Run-time errors

$$\tau ::= \mathsf{bool} \mid \mathsf{unit} \mid \mathsf{clock}(\alpha)$$

Figure 4: Syntax of types

C_1 and becomes a member of set C_2 for the remaining activities waiting on that clock. Since rule R-WAIT only updates the clock phase of those belonging to C_1 ($H\{c\colon \langle p+1, R, \emptyset\rangle\}_{c \in C_1}$) it ensures that the global clock state is updated only once. With expression drop c, the l-th activity cedes its control over clock c (rule R-DROP): we remove c from the clock views V, and remove activity identifier l from the set of registered activities Q and from the set of resumed activities R. Two consequences of dropping a clock c: a) activities waiting on clock c are no longer blocked because of this activity; b) when executing a next expression, this activity no longer waits on clock c. Expression clocked drops all registered clocks after its body becomes a value (rule R-CLOCKED).

Run-time errors are described in Figure 3 and are consistent with some of the conditions that raise exception `ClockUseException`, as discussed in the X10 language specification report [3]. During an async operation, an activity cannot transmit clocks that is not registered with (rule E-ASYNC-NO-CLOCK). Similarly activities can only perform resume or drop operations on clocks they are registered with (rules E-RESUME-NO-CLOCK and E-DROP-NO-CLOCK). In particular, it constitutes an error if an activity drops a clock twice or if it resumes a clock after dropping it.

Our semantics represents clocks in the heap as triples $\langle p, R, S\rangle$ relying on two sets for recording the registered activities R and the quiesced activities S on a clock. Implementing operations that work with sets is costly; for instance Rule R-WAIT needs to compute sets C_1 and C_2, by checking if sets R and S are equal, and then verify if $C_1 \cup C_2 = \operatorname{dom} V$. Should we make a real life implementation of the proposed semantics, set operations would have a significant impact on performance. We sketch a much faster approach that chooses to represent clocks as triples $\langle p, r, s\rangle$ accounting the clock phase, as before, but taking r and s as the cardinal numbers of sets R and S. With this representation we lose information about the identification of activities registered with a clock and, in particular, we cannot determine if an activity has already resumed in the current phase (*vide* Rules R-ASYNC and R-RESUME). To overcome this problem we need to enrich the clock local view with an indicator of whether an activity has resumed in the current phase. Thus, a clock local view becomes a pair $\langle p, b\rangle$ containing the current clock phase p (as before) and the resume boolean indicator b, describing when the activity has resumed. With this information it is straightforward to adapt rules R-ASYNC, R-MAKE, R-RESUME, R-WAIT, R-DROP, and R-CLOCKED. For instance, rule R-RESUME only updates the clock global view ($s \leftarrow s+1$) whenever its local view indicator is false. Also, Rule R-WAIT needs to set r to zero when advancing the clock global phase, and to clear the indicator b upon advancing the clock local phase. Checking that all activities registered with a clock have quiesced amounts to compare two integer numbers ($r = s$), instead of two sets R and S as before. The main reasons for not adopting the semantics just sketched are that the chosen semantics needs fewer rules and is easier to read and understand.

$$\mathscr{R}, \alpha \vdash \mathsf{clock}(\alpha) \qquad \frac{\tau \neq \mathsf{clock}(\_)}{\mathscr{R} \vdash \tau} \qquad \text{(T-WF-CLOCK, T-WF)}$$

$$\frac{\mathscr{R} \vdash \tau}{\Gamma, x\colon \tau; \mathscr{R}; \mathscr{S} \vdash x\colon (\tau; \mathscr{R}; \mathscr{S})} \qquad \Gamma; \mathscr{R}; \mathscr{S} \vdash ()\colon (\mathsf{unit}; \mathscr{R}; \mathscr{S}) \qquad \text{(T-VAR, T-UNIT)}$$

$$\frac{\alpha \text{ is fresh}}{\Gamma; \mathscr{R}; \mathscr{S} \vdash \mathsf{make}\colon (\mathsf{clock}(\alpha); \mathscr{R} \cup \{\alpha\}; \mathscr{S})} \qquad \text{(T-MAKE)}$$

$$\Gamma; \mathscr{R}; \mathscr{S} \vdash \mathsf{next}\colon (\mathsf{unit}; \mathscr{R}; \mathscr{R}) \qquad \Gamma; \mathscr{R}; \mathscr{S} \vdash \mathsf{wait}\colon (\mathsf{unit}; \mathscr{R}; \emptyset) \qquad \text{(T-NEXT,T-WAIT)}$$

$$\frac{\Gamma; \mathscr{R}; \mathscr{S} \vdash e\colon (\mathsf{clock}(\alpha); \mathscr{R}'; \mathscr{S}') \qquad \alpha \in \mathscr{R}'}{\Gamma; \mathscr{R}; \mathscr{S} \vdash \mathsf{resume}\ e\colon (\mathsf{unit}; \mathscr{R}'; \mathscr{S}' \cup \{\alpha\})} \qquad \frac{\Gamma; \mathscr{R}; \mathscr{S} \vdash e\colon (\mathsf{clock}(\alpha); \mathscr{R}'; \mathscr{S}') \qquad \alpha \in \mathscr{R}'}{\Gamma; \mathscr{R}; \mathscr{S} \vdash \mathsf{drop}\ e\colon (\mathsf{unit}; \mathscr{R}' \setminus \{\alpha\}; \mathscr{S}' \setminus \{\alpha\})}$$

(T-RESUME, T-DROP)

$$\frac{\Gamma; \mathscr{R}; \mathscr{S} \vdash e\colon (\mathsf{clock}(\alpha); \mathscr{R}'; \mathscr{S}') \qquad \mathscr{S}'' = \text{if } \alpha \in \mathscr{S}' \text{ then } \{\alpha\} \text{ else } \emptyset \qquad \Gamma; \{\alpha\}; \mathscr{S}'' \vdash e'\colon \_}{\Gamma; \mathscr{R}; \mathscr{S} \vdash \mathsf{async}\ e\ e'\colon (\mathsf{unit}; \mathscr{R}'; \mathscr{S}')}$$

$$\frac{\Gamma; \mathscr{R}; \mathscr{S} \vdash e\colon (\tau; \_; \_)}{\Gamma; \mathscr{R}; \mathscr{S} \vdash \mathsf{clocked}\ e\colon (\tau; \emptyset; \emptyset)} \qquad \text{(T-ASYNC, T-CLOCKED)}$$

Figure 5: Typing rules for expressions (clock related only)

$$\frac{\{c_1, \ldots, c_n\} \subseteq \operatorname{dom}\Gamma}{\Gamma \vdash \{c_1\colon i_1, \ldots, c_n\colon i_n\}} \qquad \text{(T-VIEW)}$$

$$\frac{\Gamma \vdash a_1 \quad \ldots \quad \Gamma \vdash a_n}{\Gamma \vdash \{l_1\colon a_1, \ldots, l_n\colon a_n\}} \qquad \frac{\Gamma \vdash V \qquad \Gamma; \{\alpha \mid c \in \operatorname{dom} V, \Gamma(c) = \mathsf{clock}(\alpha)\}; \emptyset \vdash e\colon \_}{\Gamma \vdash \langle V, e\rangle}$$

(T-ACT-SET, T-ACT)

$$\frac{\{c_1, \ldots, c_n\} \subseteq \operatorname{dom}\Gamma \qquad \Gamma; \mathscr{A} \vdash h_1 \quad \ldots \quad \Gamma; \mathscr{A} \vdash h_n}{\Gamma; \mathscr{A} \vdash \{c_1\colon h_1, \ldots, c_n\colon h_n\}} \qquad \frac{S \subseteq R \subseteq \mathscr{A}}{\Gamma; \mathscr{A} \vdash \langle p, R, S\rangle} \qquad \text{(T-HEAP, T-CLOCK)}$$

$$\frac{\Gamma; \operatorname{dom} A \vdash H \qquad \Gamma \vdash A}{\Gamma \vdash \langle H; A\rangle} \qquad \text{(T-STATE)}$$

Figure 6: Typing rules for states and activities

Type System For types we rely on an additional base set of singleton types ranged over by α. The syntax of types is depicted in Figure 4 and it includes types for values, bool and unit, and the type for clocks, $\mathsf{clock}(\alpha)$. We assign a different type (singleton type α) to each clock in order to track clock usage throughout the program.

The type system for $\mathsf{X10}|_{\mathsf{clocks}}$ is defined in Figures 5 and 6. A typing Γ is a map from variables to types. We write $\operatorname{dom}\Gamma$ for the domain of Γ. When $x \notin \operatorname{dom}\Gamma$ we write $\Gamma, x\colon \tau$ for the typing Γ' such that $\operatorname{dom}\Gamma' = \operatorname{dom}\Gamma \cup \{x\}$, $\Gamma'(x) = \tau$, and $\Gamma'(y) = \Gamma(y)$ for $y \neq x$. The type system also uses sets of singleton types, ranged over by $\mathscr{R}$ (for **r**egistered clocks) and $\mathscr{S}$ (for re**s**umed clocks).

For typing expressions we use a type and effect system (Figure 5) that records the changes made to the set of registered clocks (either by creating or dropping clocks) and to the set of resumed clocks (using resume and next) of an expression. Typing judgements are of the form $\Gamma; \mathscr{R}; \mathscr{S} \vdash e\colon (\tau; \mathscr{R}'; \mathscr{S}')$ meaning that expression e is well typed assuming the types for the free variables in Γ, the registered clocks in $\mathscr{R}$, and the resumed clocks in $\mathscr{S}$. The type of the expression is a triple recording its type τ, the changes made to the set of clocks the expression is registered with ($\mathscr{R}'$) and to the set of clocks it has

resumed ($\mathscr{S}'$).

Most typing rules are straightforward. When creating a clock (rule T-MAKE) we associate a new singleton type α with the clock and include it in the registered clock set of the activity ($\mathscr{R} \cup \{\alpha\}$). Expression next resumes all clocks the activity is registered with (notice that set of resumed clocks is $\mathscr{R}$), while expression wait clears the set of resumed clocks. Rule T-RESUME states that an activity may only resume a clock α that is in the set of registered clocks $\mathscr{R}$. A drop e expression removes clock e from the set of registered clocks $\mathscr{R}$, thus the clock cannot be passed to new activities, be the target of a resume expression, or be dropped again via rule T-DROP. Rule T-ASYNC asserts that when an activity spawns another activity registered on a clock, the quiescent property is preserved by propagating the information about the registered clock α ($\mathscr{S}'' =$ if $\alpha \in \mathscr{S}'$ then $\{\alpha\}$ else $\emptyset$). A clocked expression clocked e has the type of expression e, but drops all clocks expression e is registered with (rule T-CLOCKED).

The typing rules for states and activities (Figure 6) are straightforward.

Examples Our first example concerns clock aliasing. The report on X10 [3] read until recently "All clock variables are implicitly final. The initializer for a local variable declaration of type Clock must be a new clock expression. Thus X10 does not permit aliasing of clocks." Clearly a type system with linear control like the one we present allows to relieve such a restriction. The following example is not typable in X10.

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{make}\ \mathsf{in}\\ \mathsf{let}\ y = x\ \mathsf{in}\\ \mathsf{async}\ y\ (\mathsf{resume}\ x)\end{array}$$

In our case the code is typable, assigning the same singleton type $\mathsf{clock}(\alpha)$ to both x and y.

Our second example deals with the so called *live clock condition*. Apart from the errors in Figure 3, X10 identifies another source of problems, prohibiting an activity to transmit, during an async operation, a clock that has been resumed. Our operational semantics allows the forked activity to inherit the "status" (resumed/not resumed) of the parent activity. The following example is not typable in X10.

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{make}\ \mathsf{in}\\ \mathsf{resume}\ x;\\ \mathsf{async}\ x\ (\mathsf{next})\end{array}$$

In our case the code is typable and the type system still guarantees progress. See discussion at the end of the paper on the incorporation of a finish construct.

Our last example deals with resuming after resuming, a pattern accepted in X10. Would one like to consider the following code an error (and this seems to be the case with phasers [6]),

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{make}\ \mathsf{in}\\ \mathsf{resume}\ x;\\ \mathsf{resume}\ x\end{array}$$

we can easily add another error situation,

$$\langle H\{c\colon \langle \_, \_, S\rangle\}; A\{l\colon \langle V, \mathscr{E}[\mathsf{resume}\ c]\rangle\}\rangle \quad \text{if } l \in S$$

to go with a more stringent reduction rule (notice the new precondition $l \notin S$)

$$\frac{H(c) = \langle p, R, S\rangle \qquad V(c) = p \qquad l \notin S}{H; V; \mathsf{resume}\ c \rightarrow_l H\{c\colon \langle p, R, S \cup \{l\}\rangle\}; V; ()}$$

and the corresponding typing rule (notice the new precondition $\alpha \notin \mathscr{S}'$).

$$\frac{\Gamma;\mathscr{R};\mathscr{S} \vdash e\colon (\mathsf{clock}(\alpha);\mathscr{R}';\mathscr{S}') \qquad \alpha \in \mathscr{R}' \qquad \alpha \notin \mathscr{S}'}{\Gamma;\mathscr{R};\mathscr{S} \vdash \mathsf{resume}\ e\colon (\mathsf{unit};\mathscr{R}';\mathscr{S}' \cup \{\alpha\})}$$

Main results The results of the paper are typing preservation and type safety for typable programs (predicate $\overset{\text{err}}{\longmapsto}$ is defined in Figure 3). The proof for the first result follows by induction on the derivation tree of the sequent and uses a standard substitution lemma. Type safety follows by assuming the state typable and reaching a contradiction.

Theorem 1 (Subject reduction)**.** *If $\Gamma \vdash S$ and $S \to S'$, then $\Gamma \vdash S'$.*

Theorem 2 (Type Safety)**.** *If $\Gamma \vdash S$, then $S \overset{err}{\not\longmapsto}$.*

We anticipate a *progress property* for typable processes. Typability ensures that processes do not get stuck when dropping a clock that is not in its clock set anymore (rule R-CLOCKED), or when otherwise trying to access a clock that it not allocated in the heap. The remaining case is wait where the activity waits for set C_1 (the set of resumed clocks the activity is registered with) to grow until becoming (together with C_2—the set of clocks that have already advance their phase) the clock set of the activity. And this is bound to happen for both next and drop in each activity both implicitly resume all clocks.

Further work Apart from studying the progress property, we intend to investigate other language constructs. For instance, the primitive finish e converts global termination of expression e into local termination, waiting for activities spawned in e to locally terminate. We need further investigation on X10 finish primitive in order to present an elegant operational semantics for the construct, together with typing rules that could (hopefully) relax the syntactic restrictions imposed by either X10 (live lock condition) and its phasers extension (immediate enclosing finish) on what concerns the interplay between finish and async constructs.

The language report also reads "X10 does not contain a register statement that would allow an activity to discover a clock in a data structure and register itself on it"; we would like to study type-safe extensions to the language that might alleviate this restriction in controlled situations.

Phasers are a coordination construct that unifies collective and point-to-point synchronisations with performance results competitive to existing barrier implementations [6]. Phasers can be seen as an extension over clocks that allow for more fine-grained control over synchronisation modes. *Phaser accumulators* are reduction constructs for dynamic parallelism that integrate with phasers [5]. Although further investigation is needed, we believe our work can be extended to accommodate phasers and phaser accumulators, specially with regards to the operational similarities between clocks and phasers.

References

[1] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA'05*, pages 519–538. ACM, 2005.

[2] Jonathan K. Lee and Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of PPoPP'10*, pages 25–36. ACM, 2010.

[3] Vijay Saraswat. Report on the programming language X10, version 2.01. Technical report, IBM Research, 2010.

[4] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *Proceedings of CONCUR'05*, volume 3653 of *LNCS*, pages 353–367. Springer-Verlag, 2005.

[5] J. Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[6] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS'08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.

[7] Vasco T. Vasconcelos, Francisco Martins, and Tiago Cogumbreiro. Type inference for deadlock detection in a multithreaded typed assembly language. In *Post-proceedings of PLACES'09*, EPTCS, 2010.

Author Index