Proceedings of the
Fourth Workshop on

Programming Language
Approaches to Concurrency and
Communication-cEntric Software

2nd April 2011
Saarbrücken, Germany
Preface

This document contains the proceedings of PLACES 2011, the fourth Workshop on Programming Language Approaches to Concurrency and Communication-centric Software, held in Saarbrücken, Germany, on April 2nd, 2011, co-located with ETAPS, the European Joint Conferences on Theory and Practice of Software.

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

The Program Committee, after a careful and thorough reviewing process, selected for inclusion in the programme 9 papers out of 11 submissions. Each submission was evaluated by two to four referees, and the accepted papers were selected during one week electronic discussion.

Places 2011 was made possible by the contribution and dedication of many people. First of all, we would like to thank all the authors who submitted papers for consideration. Secondly we would like to thank our invited speaker. We would also like to thank the members of the Program Committee for their careful reviews, and the balanced discussions during the selection process. Bruno Valente helped in setting up the workshop web page.

February 10th, 2011
Vivek Sarkar
Vasco T. Vasconcelos
Program Committee

Marco Carbone, IT University of Copenhagen
Swarat Chaudhuri, Pennsylvania State University
Alastair Donaldson, Oxford University
Tim Harris, Microsoft Research Cambridge
Alan Mycroft, University of Cambridge
Jens Palsberg, University of California, Los Angeles
Vijay A. Saraswat, IBM Research
Vivek Sarkar, Rice University (co-chair)
Vasco T. Vasconcelos, University of Lisbon (co chair)
Jan Vitek, Purdue University
Nobuko Yoshida, Imperial College London
# Table of Contents

## Invited talk

*Charting the course to a many core future: HW, SW and the parallel programming problem*, Timothy G. Mattson, Intel Corporation. ______________5

## Contributed papers

- **A G-Local Pi-Calculus**, Chiara Bodei, Viet Dung Dinh and Gian Luigi Ferrari ______________7
- **Reasoning about Explicit Resource Management**, Adrian Francalanza, Edsko De Vries and Matthew Hennessy ______________15
- **The Timed, Compensable Conversation Calculus**, Hugo A. López and Jorge A. Pérez ______________22
- **CPC: Programming with a massive number of lightweight threads**, Gabriel Kerneis and Juliusz Chroboczek ______________30
- **Static analysis and refactoring towards Erlang multicore programming**, Melinda Tóth, István Bozó, Zoltán Horváth and Atilla Erdödi ______________43
- **Resource Access with Variably Typed Return**, Gabrielle Anderson and Julian Rathke ______________51
- **Getting the Goods with Concurrent bondi**, Thomas Given-Wilson and Barry Jay ______________58
- **Tasking Event-B: An Extension to Event-B for Generating Concurrent Code**, Andrew Edmunds and Michael Butler ______________63
Invited Talk
Charting a course to our many-core future: HW, SW and the parallel programming problem

Timothy G. Mattson
Intel Corporation

The general trends that will dominate microprocessor design over the next decade are clear: the number of cores will steadily increase as will the level of heterogeneity. All of the major hardware vendors, from the embedded to laptop to HPC markets, are aggressively turning these trends into actual products. Unfortunately, the software industry is behind; creating a dangerous mismatch between hardware and software.

The academic community can choose to play a role in closing this gap. To do so, however, they must think more like software engineers and a bit less like isolated computer scientists. In particular, we need:

- A new foundation to build our software future upon, not new and elegant languages that few people actually use.
- To design hardware and software together so people can program the systems we build.
- To figure out how to support a world where domain experts with little or no training in computer science create applications.

These define the research imperatives that must be resolved to address the many-core challenge. In this talk, I will review work to address each of these imperatives, and hopefully launch a dialog exploring ideas on how to accelerate their resolution.

Biography
Tim Mattson is an old-fashioned parallel programmer starting in the mid-80’s with the Caltech Cosmic Cube and continuing up to the present. Along the way, he has worked with most classes of parallel computer (vector supercomputers, SMP, VLIW, NUMA, MPP, clusters, GPUs and many-core CPUs). Tim has published extensively including the books Patterns for Parallel Programming (with Beverly Sanders and Berna Massingill, Addison Wesley, 2004) and An Introduction to Concurrency in Programming Languages (with Matthew J. Sottile and Craig E Rasmussen, CRC Press, 2009). Tim has a Ph.D. degree in chemistry for his work on molecular scattering theory. He has been working at Intel since 1993.
A G-Local $\pi$-Calculus *

Chiara Bodei, Viet Dung Dinh and Gian Luigi Ferrari
Dipartimento di Informatica, Università di Pisa
Pisa, Italy

Abstract

The management of the operations acting over distributed and virtual resources plays an important role for achieving the success of modern distributed applications. In this paper, we advocate a notion of resources as entities with their own local policies and containing their global interactive properties. We introduce a variant of $\pi$-calculus with primitives to declare, acquire and release resource dynamically. A Control Flow Analysis predicting resource usage is also presented.

1 Introduction

Modern programming paradigms for distributed systems support the development of network applications by taking advantage of computational infrastructures, in which applications are executed interacting with the resources available in the network. Network applications are now become loosely-coupled entities. Moreover, resources are not created nor destroyed by applications but directly acquired on-the-fly when available. An important aspect of service acquisition concerns the agreement between client requirements and service guarantees (Service Level Agreement – SLA). Cloud computing provides the best illustrative example of this programming paradigm [7, 14, 1].

Devising expressive, flexible and efficient mechanisms to control resource usages is therefore a major issue in the design of resource-aware programming languages and the related software engineering methods. Security makes the problem even more crucial since network resources can be offered by (possibly untrusted) third parties and applications have little or no control about the security of the whole. In the last few years, the problem of providing the mathematical basis for the mechanisms that support resource control and usages has been tackled by several authors (see e.g. [2, 6, 10, 9, 13]).

In the programming model we present here, processes and resources are distinguished entities. Resources are computational entities with their own life-cycle. Processes acquire the resources needed for their internal purposes dynamically when available, but they cannot create any resource. This simple programming model abstracts the features of several interesting distributed systems, such as a system offering computing resources. The available resources are the CPU units of a given power and processes can only acquire the CPU time when available to run some specialized code. Similarly considerations apply to Web-based storage services, where client processes can only acquire slots of the available storage. Clearly, even though such distributed systems evolve by applying reconfiguration steps updating the structure of the available resources, reconfiguration is not under the control of client processes.

In this extended abstract, we introduce a formal model for resource usages focusing our attention on the way resources are acquired by processes on-the-fly when they are needed. We do not consider here the issues related to dynamic reconfiguration of resources. We start from the $\pi$-calculus [12] and we extend it with with primitives to represent resources and the operations to acquire and release resources on demand. Although, it may seem in principle simple to “add” resources to the $\pi$-calculus, the actual treatment of resources requires special care to avoid the development of awkward models with too many features. In our approach, resources are stateful entities available in the network environment where processes live. Moreover, resources are dynamically acquired and released by processes under certain conditions. All the operations processes perform over resources must obey the SLA established

*Research supported by the Italian PRIN Project “SOFT” and FET Project “ASCENS”.

1
at binding time. Specifically, a resource is described through the declaration of its interaction endpoint (the resource name), its local state and its global properties. The global property establishes and enforces the SLA agreement any interaction the resource engages with its client process has to satisfy. The interplay between local and global information occurring in the process-resource interactions motivates the adjective \textit{G-Local} given to our extension of the \(\pi\)-calculus. Here, we do not consider any specific model for defining the global interaction properties. They can be expressed as suitable resource-aware logic in the style of \cite{2}, or contract-based logic as in \cite{8,3}. To summarize, our approach combines the basic features of the \(\pi\)-calculus (i.e. dynamic communication topology of processes via name passing) with a suitable set of resource-aware constructs to declare, acquire and release resources by respecting the SLA constraints.

The main contribution of this paper is the introduction of the G-Local \(\pi\)-calculus. The calculus is a relatively simple variation of the \(\pi\)-calculus but sufficiently expressive to represent a variety of resource usages. The calculus is tailored to support the design of loosely-coupled processes and of the global SLAs governing resource usages. The interplay between local and global views is also one of the novel feature of our proposal. We adopted a declarative style that abstracts from the way resources are implemented to ease the development of reasoning techniques. In particular, we developed a Control Flow Analysis (CFA) for our calculus. The analysis computes an approximation of resource usages. Hence, it can be used to statically check whether or not the global properties of resources usages are respected by process interactions. Due to space limitation we only sketch the CFA and we discuss it analysis via a simple but illustrative example.

\textbf{Related Work.} In \cite{2} an extension of the \(\lambda\)-calculus is proposed to statically verify resource usages. Our notion of global usages is inspired by this work. The \(\pi\)-calculus dialect of \cite{10} provides a general framework for checking resource usages in distributed systems. In this approach private names are extended to resources, i.e. names with a set of traces to define control over resources. Also resource request and resource release are simulated through communicating private names and structural rules respectively. This gives shared semantics of resources, i.e. several processes can have a concurrent access to resources (by communicating private names). In our approach, when a process obtains a resource, it has an exclusive access to it. Furthermore, resource entities are costly and permanent, while this is not the case in \cite{10}. In \cite{9}, resources form a monoid and the evolution of processes and resources happens in a SCCS style. A logic of bunched implication serves as specification languages. In our approach, resources are independent stageful entities equipped with their own global interaction usage policy. A dialect of the \(\pi\)-calculus, where resources are abstractly represented via names and can be allocated or de-allocated has been introduced in \cite{13}. In this approach reconfigurations steps are internalized inside processes via the operations for allocating and de-allocating channels. A type system capturing safe reconfigurations over channels has been introduced. In our approach resources are more structured than channels and their reconfiguration steps are not under the control of processes. Finally, the work presented in \cite{6} mainly focuses on specifying SLA by describing resources as suitable constraints. Our approach can exploit constraints to express global resource usages as well.

\section{The calculus}

We consider the monadic version of \(\pi\)-calculus \cite{12} extended with suitable primitives to declare, access and dispose resources. The syntax is given in Fig. 1. Here, \(\mathcal{N}\) is a set of channel names (ranged over by \(x, y\)), \(\mathcal{R}\) is a set of resource names (ranged over by \(r, r'\)) and \(\mathcal{A}\) is a set of actions (ranged over by \(\alpha, \beta\)) for running over resources. We assume that these sets are pairwise disjoint. Furthermore, we distinguish among sequential processes (ranged over by \(Q, Q'\)) and processes (ranged over by \(P, P'\)).

Our calculus builds over the \(\pi\)-calculus. We do not explain the intuitive interpretation of its operators,
since it is standard \cite{12}. As usual in the \(\pi\)-calculus, the input prefix \(x(y).P\) binds the variable \(y\) in the process \(P\), while the output prefix process \(\bar{x}y.P\) sends \(y\) along \(x\) and continues as \(P\). Note that resource names cannot be communicated. As usual, input prefixes and restrictions act as bindings. The notions of names \(n()\), free names \(fn()\), bound names \(bn()\) and substitution \\{\(-/-\)\} are the standard ones and therefore we will freely use them without presenting.

Our extension introduces resource-aware constructs in the \(\pi\)-calculus. The action prefix \(\alpha(r;p)\) models an action \(\alpha\) operating over the resource \(r\) with some parameters \(p\). The parameters may be data or other kind of information. We do not introduce a precise syntax for parameters since it depends on the actual resources. We model resources as stateful entities, equipped with policies constraining their usages. More precisely, a resource is a triple \((r,\varphi,\eta)\), where \(r \in \mathcal{R}\) is a resource name, \(\varphi \in \Phi\) is the associated policy and \(\eta \in \mathcal{H}\) is a state (\(\varepsilon\) denotes the empty state). Policies specify the required properties on resource usages, which are usually defined by means of a resource-aware logic (\(\mathcal{H}\) \cite{2} \cite{8} \cite{3}), while states keep track of actions on resources, by means of execution traces. For the sake of simplicity, we model processes that perform their work upon resources as sequential processes (the right part of Fig.\cite{1} illustrates their syntax). Intuitively, a sequential process represents a single thread of execution in which one or more resources can be used.

To model the resource behaviour, we have two primitives managing resource boundaries: resource joint point \((r,\varphi,\eta)\{Q\}\) and resource request point \(\text{req}(r)\{Q\}\), where \(Q\) is a sequential process. Intuitively, process \(Q\) inside the resource boundary \((r,\varphi,\eta)\{Q\}\) can fire actions acting over the resource \(r\). The state \(\eta\) is updated at each action \(\alpha(r;p)\) according to the required policy \(\varphi\). A resource request point \(\text{req}(r)\{Q\}\) represents a process asking for the resource \(r\). Only if the request is fulfilled, i.e. the required resource is available, the process can enter the required resource boundary and can use the resource \(r\), provided that the policy is satisfied.

We introduce a notion of well-formedness for which actions on a resource \(r\) must occur inside the corresponding resource boundary. We assume that there are no multiple instances of the same resource, i.e. no more than one declaration \((r,\varphi,\eta)\) for the resource \(r\) is allowed. Furthermore, since resources have a cost and can be scarce, the replication operator \(!\) cannot be applied to resource joint points. Informally, a process is well-formed if: (i) every action \(\alpha(r;p)\) is underneath either a resource joint point \((r,\varphi,\eta)\) or a resource request point \(\text{req}(r)\); (ii) neither a resource join point nor a resource request point is included in a replication-subprocess.

---

**Figure 1:** Syntax of Processes.
To illustrate the main features of our calculus, consider a simple Web-based storage service, called $r$, that provides two different databases. Web-based applications working on it, i.e. that store and retrieve data, by exploiting the resources made available by the storage service $r$, are constrained to adhere to a certain SLA policy, when operating on $r$. Initially, an application is free to choose between the two databases. Then, it can proceed and operate only with the selected one. This kind of usage policy is known under the name of Chinese Wall Policy \[5\].

In our abstract setting we model the computational activities acting over the two databases with the following actions: $α(r;id)$ and $β(r;id)$, where $r$ is the name of the storage service, $α$ ($β$ resp.) is the abstract representation of the operation of opening on the database $α$ ($β$ resp.) and the tag $id$ is used to identify the application; $γ(r)$ denotes the other actions used to operate on the storage service $r$. The following process specifies a system consisting of two applications accessing the databases available in the storage service $r$:

$$
A_1 = \text{req}(r)\{ α(r;id_1).γ(r) \} || \text{req}(r)\{ β(r;id_1).γ(r) \}
$$

$$
A_2 = \text{req}(r)\{ β(r;id_2).γ(r) \}
$$

$$
S = (r,φ,ε)\{ 0 \} || A_1 || A_2,
$$

where the $φ$ is the specification of the Chinese Wall Policy intuitively described above (we do not enter into details, we refer to [11] for its complete specification).

We now introduce the operational semantics of our calculus. The operational semantics of processes is a late semantics, defined by the transition relation given in Fig.3. Labels $μ,μ'$ for transitions are $τ$ for silent actions, $x(y)$ for free input, $tx$ for free output, $x(y)$ for bound output, and $α(r;p)$ for actions. The effect of bound output is to extrude the sent name from the initial scope to the external environment.

We assume a notion of structural congruence, that we comment, but we do not report for space limitation. This includes the standard laws of $π$-calculus, such as alpha-conversion, the monoidal laws for the parallel composition and the choice operator. There are also new laws for managing the resource-aware constructs. If two processes $Q_1$ and $Q_2$ are equivalent, then also $Q_1$ and $Q_2$ when plugged inside the same resource boundaries are. Resource request and resource joint points can be swapped with restriction scopes. The only law that we show is crucial for managing the discharge of resources.

$$(r_1,φ_1,η_1)\{ (r_2,φ_2,η_2)\{ 0 \} || Q \} = (r_2,φ_2,η_2)\{ 0 \} || (r_1,φ_1,η_1)\{ Q \}$$

It allows an available resource, hold by the empty process $0$, to enter or escape a resource boundary. In this way, available resources can freely float everywhere. Notice that this law expresses a basic reconfiguration step making available un-used resources.

To manage resources, we need some auxiliary definitions. First, we need a way to check whether in a process a resource is still in use. Only “not-in-use” resources can be released. To specify which are the resources used by a process, we define the set of used resource names on a process, denoted by $@Res$, which is given in Fig.2. Intuitively, $@Res$ is a sort of dynamic inspection that collects the resources on which access actions are invoked.
The following trace illustrate the behaviour of the system. First application 1, the resource is now available, i.e. it encloses the empty process. That the rules check the updated state and the function update any longer. Recall that service and operates on the database Symmetrically, when a resource an "unfair" computation, otherwise available resources can perform this rule infinite number of times. To explain the operation semantics we consider our running example: the web based storage service. In addition, we use the predicate check we are now ready to comment on the semantic rules corresponding to the treatment of resources. The rule Policy checks whether an action , by appending . Notice that the global policy on the resource is always satisfied. Without loss of generality, we do not concretely define these functions.

We are now ready to comment on the semantic rules corresponding to the treatment of resources. The rules Act, Par, Res, Comm, Cong, Choice, Open and Close are the standard π-calculus ones. Note that the rules Comm, Choice and Close do not apply to sequential processes.

The rule Policy checks whether an action on the resource obeys the policy , i.e. whether the updated state , append the action to the current state , is consistent w.r.t. , by using the predicate check. If the policy is obeyed, then the updated state is recorded. According to the rule Discharge, can release resource , provided that is not in use in any longer. Recall that does not belong to the used names of , whether in there are no actions acting over . In the resulting process, the process escapes the resource boundary. Furthermore, the resource is now available, i.e. it encloses the empty process . The condition is needed to avoid an "unfair" computation, otherwise available resources can perform this rule infinite number of times. Symmetrically, when a resource has been made available, then it can be acquired by that enters the corresponding resource boundary , as stated by the rule Acquire. The rules Local1 and Local2 express that actions can bypass resource boundaries for only if they do not involve the bound resource .

With the above semantics, one can easily prove by structural induction that if is well-formed and , then is well-formed, i.e. well-formedness is preserved by computation.

To explain the operation semantics we consider our running example: the web based storage service. The following trace illustrate the behaviour of the system. First application gets access to the storage service and operates on the database . Then, the application proceeds elaborating data on the database by issuing the action . Notice that the global policy on the resource is always satisfied.
Again, the service storage is available. Let us assume that the first application gets again access to the resource storage. Then we have:

$$(r, \varphi, \varepsilon) \{ 0 \} \parallel A_1 \parallel A_2 \xrightarrow{\delta} (r, \varphi, \alpha(r,id_1) \gamma(r)) \{ 0 \} \parallel req(r) \{ \beta(r,id_1). \gamma(r) \} \parallel A_2$$

3 Control Flow Analysis

We have developed a CFA for our calculus, extending the one developed for \( \pi \)-calculus [4]. To this purpose, we associate labels \( \chi \in \mathcal{L} \) with resource boundaries as follows: \((r, \varphi, \eta)\{Q\}^\chi \) and \(req(r)\{Q\}^\chi\).

To record the traces of actions, we also introduce a predicate \( last \) to check whether or not an action \( \alpha(r,p) \) is the last action operating over \( r \) inside a sequential process. Note that these annotations can be performed in a pre-processing step and do not affect the semantics of the calculus. The analysis keeps track of the following information:

- An approximation \( \rho : \mathcal{N} \to \varphi(\mathcal{N}) \) of names bindings. If \( a \in \rho(x) \) then the variable \( x \) can assume the value \( a \).
- An approximation \( \kappa : \mathcal{N} \to \varphi(\mathcal{N}) \) of the values that can be sent on each channel. If \( b \in \kappa(a) \) then the value \( b \) can be output on the channel \( a \).
- An approximation \( \Gamma : \mathcal{R} \rightarrow \varphi(\{(\varphi, \eta), S|\varphi \in \Phi, S \in \mathcal{L}^*, \eta \in \mathcal{H}\}) \) of resource behavior. If \( (\varphi, \eta), S \in \Gamma(r) \) then \( \eta \) is one of the possible traces over \( r \) that is performed by a set of sub-processes labelled by \( S \).
- An approximation \( \psi : \mathcal{R} \rightarrow \varphi(\{(\varphi, \eta), S|\varphi \in \Phi, S \in \mathcal{L}^*, \eta \in \mathcal{H}\}) \) of errors, that can be possibly empty. If \( (\varphi, \eta), S \in \psi(r) \), then \( \eta \) is one of the possible traces over \( r \) that violates the policy associated with \( r \), and the fault is of the sub-process labelled by \( \chi \).

The judgments of the analysis are in the form \( (\rho, \kappa, \Gamma, \psi) \in^\delta P \), where \( \delta \) is a sequence of pairs \( ((r, \varphi, \eta), S) \), keeping trace of the resource scope nesting. This sequence is initially empty, denoted by \([\varepsilon, \varepsilon]\). The tuple \( (\rho, \kappa, \Gamma, \psi) \) is called an estimate for \( P \). The analysis correctly captures the behavior of \( P \), i.e. the estimate \( (\rho, \kappa, \Gamma, \psi) \) is valid for all the derivatives \( P' \) of \( P \). In particular, an approximation of resource behavior is provided. More precisely, \( \Gamma \) provides a set of traces of actions on each resource, while \( \psi \) captures a set of the violated traces.

Rules Due to space limitation, we do not describe the rules for the empty process, choice, parallel composition, restriction, replication, output and input that are based on the CFA for \( \Pi \)-calculus [4]. These rules essentially are used to obtain the \( \rho \) and \( \kappa \) components. The crucial judgments of the new CFA are given below.
\[(\rho, \kappa, \Gamma, \psi) =^\delta (r, \varphi, \eta)(Q)^X \quad \text{iff} \quad (\rho, \kappa, \Gamma, \psi) =^\delta [(r, \varphi, \eta), \chi] Q\]
\[(\rho, \kappa, \Gamma, \psi) =^\delta \text{req}(r)(Q)^X \quad \text{iff} \quad \forall [(\varphi, \eta), S] \in \Gamma(r) \wedge \chi \notin S \Rightarrow (\rho, \kappa, \Gamma, \psi) =^\delta [(r, \varphi, \eta), S, \chi] Q\]
\[(\rho, \kappa, \Gamma, \psi) =^\delta \alpha(r;p)^r(Q) \quad \text{iff} \quad [(r, \varphi, \eta), S^\chi] \text{ occurs in } \delta \wedge \eta \cdot \alpha(r;p) = \varphi \Rightarrow (\rho, \kappa, \Gamma, \psi) =^\delta Q\]
\[\wedge \text{ last}(\alpha(r;p)) \Rightarrow [(r, \varphi, \eta, \alpha(r;p)), S^\chi] \in \Gamma(r)\]
\[\wedge \eta \cdot \alpha(r;p) \neq \varphi \Rightarrow [(r, \varphi, \eta, \alpha(r;p)), S^\chi] \in \psi(r)\]

The rule for resource joint point updates \(\delta\) to record that immediate sub-process is inside the new resource scope and there it is analyzed. In the rule of resource request point, the analysis for \(Q\) is performed for every possible element \([[(\varphi, \eta), S]\) from the component \(\Gamma(r)\). This amounts to saying that the resource \(r\) can be used starting from any possible previous history \(\eta\). In order not to append the same history more than once, we have the condition that \(S\) does not contain \(\chi\).

According to the rule of action, if the \([(r, \varphi, \eta), S^\chi]\) occurs in \(\delta\) (i.e. if we are inside the resource scope of \(r\)) and the updated history \(\eta \cdot \alpha(r;p)\) obeys the policy \(\varphi\), then the analysis result also holds for the immediate sub-process and \(\delta\) is updated in \(\delta'\), as described below. If \(\alpha(r;p)\) is not the last action, then \([(r, \varphi, \eta), S^\chi]\) is replaced by \([(r, \varphi, \eta, \alpha(r;p)), S^\chi]\) in \(\delta\), therefore keeping trace of the resource accesses to \(r\) in the sub-process labelled by \(\chi\). Instead, if \(\alpha(r;p)\) is the last action then \([(r, \varphi, \eta), S^\chi]\) is removed from \(\delta\) and this reflects the fact that the process can release the resource \(r\) and exiting its scope. Moreover, the trace of actions \(\eta \cdot \alpha(r;p)\) over \(r\) at \(\chi\) is recorded in \(\Gamma(r)\). Other sub-processes can access the resource starting from the trace \(\eta \cdot \alpha(r;p)\). In case the action violates the policy associated with \(r\) (see the last conjunct), the component \(\psi\) is updated with the corresponding trace and the sequence of process labels \(S^\chi\), where \(\chi\) represents the sub-process that violates the policy.

We only state the main results. Our analysis respects the operational semantics of G-Local \(\pi\)-calculus. More precisely, the following subject reduction result holds.

**Theorem 3.1.** (Subject Reduction) \((\rho, \kappa, \Gamma, \psi) =^\delta P \text{ and } P \overset{\tau}{\rightarrow} P'\), then \((\rho, \kappa, \Gamma, \psi) =^\delta P'\).

Furthermore, there always exists a least choice of \((\rho, \kappa, \Gamma, \psi)\) that is acceptable for CFA rules.

**Definition 3.2.** A set of proposed solutions can be partially ordered by setting \((\rho, \kappa, \Gamma, \psi) \subseteq (\rho', \kappa', \Gamma', \psi')\) iff \(\forall a \in \mathcal{N}: \rho(a) \subseteq \rho'(a), \forall a \in \mathcal{N}: \kappa(a) \subseteq \kappa'(a), \forall r \in \mathcal{R}: \Gamma(r) \subseteq \Gamma'(r)\) and \(\psi(r) \subseteq \psi'(r)\).

**Theorem 3.3.** (Existence of solution) For all \(\delta, P\), the set \(\{ (\rho, \kappa, \Gamma, \psi) | (\rho, \kappa, \Gamma, \psi) =^\delta P \}\) is a Moore family.

To explain the features of our CFA, we consider our running example of the storage service, where we associate labels with resource boundaries in the following way.

\[
\begin{align*}
A_1 &= \text{req}(r) \{ \alpha(r;id_1).\gamma(r) \}^{X_{11}} \parallel \text{req}(r) \{ \beta(r;id_1).\gamma(r) \}^{X_{12}} \\
A_2 &= \text{req}(r) \{ \beta(r;id_2).\gamma(r) \}^{X_2} \\
S &= (r, \varphi, \epsilon) \{ 0 \}^{X_0} \parallel A_1 \parallel A_2
\end{align*}
\]

Our CFA computes a set of violated traces of actions on the resource \(r\):

\[
\begin{align*}
(\alpha(r;id_1).\gamma(r).\beta(r;id_1), \chi_0.\chi_{11}.\chi_{12}) \\
(\beta(r;id_1).\gamma(r).\alpha(r;id_1), \chi_0.\chi_{12}) \\
(\beta(r;id_2).\gamma(r).\alpha(r;id_1), \chi_0.\chi_{21}) \\
(\alpha(r;id_1).\gamma(r).\beta(r;id_2).\gamma(r).\beta(r;id_1), \chi_0.\chi_{11}.\chi_{12}) \\
(\alpha(r;id_1).\gamma(r).\beta(r;id_2).\gamma(r).\beta(r;id_1), \chi_0.\chi_{11}.\chi_{12}) \\
(\beta(r;id_2).\gamma(r).\alpha(r;id_1).\gamma(r).\alpha(r;id_1), \chi_0.\chi_{21}.\chi_{11}) \\
(\beta(r;id_1).\gamma(r).\beta(r;id_2).\gamma(r).\alpha(r;id_1), \chi_0.\chi_{12}.\chi_{11})
\end{align*}
\]
By looking at these traces, we can infer that violations occur when the first application $A_1$ try to access a database different from the one selected for the first access.

References


Reasoning about Explicit Resource Management* (Abstract)

Edsko de Vries
Trinity College Dublin, Ireland
devriee@cs.tcd.ie

Adrian Francalanza
University of Malta
afral@um.edu.mt

Matthew Hennessy
Trinity College Dublin, Ireland
Matthew.Hennessy@cs.tcd.ie

1 Introduction

We investigate the behaviour and efficiency of concurrent processes with explicit resource management. Our study is based on a $\pi$-calculus variant called $R\pi$ [4] where the only resources available are channels, which must be explicitly allocated before they can be used and can be deallocated when no longer required. A substructural type system guarantees the safe allocation and deallocation of channels, as well as safe channel reuse through strong updates. In this paper we use this type system to give compositional proof techniques for reasoning about the behaviour and efficiency of $R\pi$ processes.

Suppose two servers listen on channels srv$_1$ and srv$_2$ to receive a channel which they will use once to send a reply back to the client. Consider the following clients:

\[
C_0 \triangleq \text{rec}\,X.\text{alloc}\,x_1.\text{alloc}\,x_2.\text{srv}_1!x_1.x_1?y.\text{srv}_2!x_2.x_2?z.c!(y,z).X \\
C_1 \triangleq \text{rec}\,X.\text{alloc}\,x.\text{srv}_1!x.x?y.\text{srv}_2!x.x?z.c!(y,z).X \\
C_2 \triangleq \text{rec}\,X.\text{alloc}\,x.\text{srv}_1!x.x?y.\text{srv}_2!x.x?z.\text{free}\,x.c!(y,z).X
\]

$C_0$ is an idiomatic $\pi$-calculus client. It creates one private “reply” channel to communicate with srv$_1$ and another to communicate with srv$_2$. $C_1$ is more efficient and reuses the same channel for both servers. $C_2$ is more efficient still and deallocates the channel before recursing. Using our theory we can prove that these clients are equivalent, with $C_2$ being more efficient than $C_1$ and $C_1$ being more efficient than $C_0$.

We rely on type information to prove this. When $C_1$ allocates channel $x$, no other process knows $x$: from a typing perspective, $x$ is unique to $C_1$. $C_1$ then sends $x$ on srv$_1$ at an affine type, which (by definition) limits the server to use $x$ at most once. At this point $c$ is unique-after-1 to $C_1$: after one communication step, $C_1$ is once again the only process that knows about $x$ ($x$ once again becomes unique). This means that $C_1$ can reuse $x$, possibly for values of a different type (strong update), because the type system guarantees that a unique channel is indistinguishable from a freshly allocated channel.

Uniqueness thus records the “positive” information (the guarantee) at one end of a channel corresponding to the “negative” information of an affine permission (a restriction) at the other. Our theory tracks permission transfer: in the above example, this manifests itself both as the explicit transfer of an affine permission when $C_1$ sends its channel to the server, as well as the implicit transfer of this permission back from the server to the client after the communication, allowing us to recover channel uniqueness.

Our contributions are: (1) a compositional coinductive proof method for comparing the behaviour and efficiency of $R\pi$ processes that are well-behaved wrt. a uniqueness type system ensuring safe deallocations and strong updates. (2) the definition of a typed, costed, amortized efficiency preorder relation, which embodies the permission transfer discussed above, and a proof that the coinductive proof method is sound and complete with respect to this preorder.

2 Language

Fig. 1 shows the syntax and semantics of $R\pi$. It has the standard $\pi$-calculus constructs with the exception of scoping, which is replaced with primitives for explicit channel allocation and deallocation. The syntax

*Supported by SFI project SFI 06 IN.1 1898.
Reduction Rules

**Example 1.**

assumes two separate denumerable sets of channel names $c, d$ and variables $x, y$, and lets identifiers $u, v$ range over both. The input and channel allocation constructs are binders. The syntax also assumes a denumerable set of process variables $X,Y$, bound by the recursion construct; we use $k, l$ as metavariables to range over costs. $R\pi$ processes run in a resource environment $M$ represented as a function from resource names to \{⊤, ⊥\}, recording whether channels are allocated (⊤) or deallocated (⊥). We refer to a pair $M \vdash P$ of a resource environment and a process as a system.

In order to measure resource usage, we annotate the reduction semantics with a cost function $r$ as follows:

$$M, c: \top \vdash c!d.P \parallel c?x.Q \rightarrow_0 M, c: \top \vdash P[\tau/x] \quad \text{RCom}$$

$$M, c: \top \vdash \text{if } c = c \text{ then } P \text{ else } Q \rightarrow_0 M, c: \top \vdash P \quad \text{RThen}$$

$$M, c: \top, d: \top \vdash \text{if } c = d \text{ then } P \text{ else } Q \rightarrow_0 M, c: \top, d: \top \vdash Q \quad \text{RElse}$$

$$M \vdash \text{rec } X.P \rightarrow_0 M \vdash P[\text{rec } X.P/X] \quad \text{Rec}$$

$$M, c: \perp \vdash \text{alloc } x.P \rightarrow_1 M, c: \top \vdash P[\gamma/x] \quad \text{RAll}$$

**Figure 1: $R\pi$ Syntax and Reduction Semantics**

Example 1. Resource mismanagement in $R\pi$ may result in unexpected behaviour:

$$M, c: \top \vdash \text{free } c.(c!1 || c?x.P) \parallel \text{alloc } y.(y!42 || y?z.Q) \rightarrow_1 M, c: \perp \vdash c!1 || c?x.P \parallel \text{alloc } y.(y!42 || y?z.Q) \rightarrow_1 M, c: \top \vdash c!1 || c?x.P \parallel c!42 || c?z.Q$$

In this example premature deallocation of channel $c$ \[1\] allows $c$ to be reallocated by the right process \[2\]. This leads to unintended behaviour through interferences when communicating on $c$: these interferences are unintended because, intuitively, allocation should yield “fresh” channels.

In \[4\] we defined a type system with judgements $Γ \vdash P$ and $Γ \vdash M$ precluding such unwanted behaviour; the type syntax and the typing rules for processes are shown in Fig. \[2\]. Channel types are denoted as $[T]^T$, where type attributes a range over "1" for affine, "(\bullet,i)" for unique-after-$i$ ($i \in N$), and "ω" for unrestricted channels (no usage restrictions or guarantees). The type system is substructural,
Types and Type Attributes

\[
\begin{align*}
T & ::= [\bar{T}]^a \quad \text{(channel type)} \\
| & \quad \textproc \quad \text{(process)} \\
\end{align*}
\]

\[
a ::= 1 \quad \text{(affine)} \\
| \omega \quad \text{(unrestricted)} \\
| (\bullet, i) \quad \text{(unique after } i \text{ steps, } i \in \mathbb{N})
\]

Logical rules

\[
\begin{align*}
\frac{\Gamma, u : [\bar{T}]^a \vdash P}{\Gamma, u : [\bar{T}]^a + P} & \text{ tOut} \\
\frac{\Gamma, \Gamma \vdash P}{\Gamma \vdash \Gamma \times \Gamma'} & \text{ tIn} \\
\frac{\Gamma \vdash u = v \text{ then } P \text{ else } Q}{\Gamma \vdash \textproc X : \textproc P \vdash \Gamma \times \textproc P \vdash X} & \text{ tRec} \\
\frac{\Gamma \vdash \textproc X : \textproc P \vdash \Gamma \times \textproc X \vdash \Gamma}{\Gamma \vdash \textproc X \vdash \Gamma} & \text{ tVar} \\
\frac{\Gamma \vdash \textproc X : \textproc P \vdash \Gamma \times \textproc P \vdash \Gamma}{\Gamma \vdash \textproc X : \textproc P \vdash \Gamma} & \text{ tNil}
\end{align*}
\]

where \(\Gamma^\omega\) can only contain unrestricted assumptions and all bound variables are fresh.

Structural rules \((\triangleleft)\) is the least reflexive transitive relation satisfying

\[
\begin{align*}
\frac{T = T_1 \circ T_2}{\Gamma, u : T \vdash \Gamma, u : T_1, u : T_2} & \text{ tCon} \\
\frac{T = T_1 \circ T_2}{\Gamma, u : T \vdash \Gamma, u : T_1, u : T_2} & \text{ tJoin} \\
\frac{T_1 \triangleleft T_2}{\Gamma, u : T \vdash \Gamma, u : T_1} & \text{ tWeak} \\
\frac{T_1 \triangleleft T_2}{\Gamma, u : T \vdash \Gamma, u : T_1} & \text{ tSub} \\
\frac{T_1 \triangleleft T_2}{\Gamma, u : T \vdash \Gamma, u : T_1} & \text{ tRev}
\end{align*}
\]

Counting channel usage

\[
\begin{align*}
\Gamma, c : [\bar{T}]^a & \overset{\text{def}}{=} \begin{cases} 
\Gamma & \text{if } a = 1 \\
\Gamma, c : [\bar{T}]^\omega & \text{if } a = \omega \\
\Gamma, c : [\bar{T}]^{\bullet, i} & \text{if } a = (\bullet, i + 1)
\end{cases}
\end{align*}
\]

Type splitting

\[
\begin{align*}
[\bar{T}]^\omega & = [\bar{T}]^\omega \circ [\bar{T}]^a & \text{ pUnr} \\
\textproc & = \textproc \circ \textproc & \text{ tProc} \\
[\bar{T}]^{\bullet, i} & = [\bar{T}]^1 \circ [\bar{T}]^{\bullet, i+1} & \text{ pUnq}
\end{align*}
\]

Subtyping

\[
\begin{align*}
(\bullet, i) & \triangleleft_s (\bullet, i + 1) & \text{ sIndex} \\
(\bullet, i) & \triangleleft_s (\bullet, i + 1) & \text{ sUnq} \\
\omega & \triangleleft_s 1 & \text{ sAff} \\
a & \triangleleft_s a & \text{ sTyp}
\end{align*}
\]

Figure 2: Typing processes
so that typing assumptions can be interpreted as permissions. This is especially evident in the rule for parallel composition (\(\tau Par\)) where a permission can be used by either the left process or the right, but not by both. Some permissions can be split, however, using contraction (\(\tau Tr\), \(\tau Con\)). For example, an assumption \(c : [\tilde{T}]^{(*,0)}\) can be split as \(c : [\tilde{T}]\) and \(c : [\tilde{T}]^{(*,1)}\) (\(\tau Unq\)). Dually, \(c : [\tilde{T}]\) and \(c : [\tilde{T}]^{(*,1)}\) can be consolidated again into \(c : [\tilde{T}]^{(*,0)}\) (\(\tau Join\)). Permission splitting and merging plays a major role in the behaviour equivalences we study in Sections 3 and 4.

The type system guarantees that when a process is typed wrt a unique assumption for a channel, \([\tilde{T}]^{(*,0)}\), no other process has access to that channel. This means that deallocation and strong update (changing the object type of a channel) are safe for unique channels. In this paper we take advantage of this type system to provide a behavioural theory for \(R\pi\).

3 Labelled Transition System

We introduce a labelled transition system (LTS) dealing with permission ownership and transfer to enable compositional reasoning about resource management in \(R\pi\) processes. The LTS is defined over triples of the form \(\Gamma \prec M \triangleright P\), where \(\Gamma\) is the typing environment that types the observer. \(\Gamma \prec M \triangleright P\) is a configuration if there is a global set of permissions \(\Gamma_{global}\) which can be distributed as the permission \(\Gamma\) of the observer and some (existentially quantified) permissions \(\Delta\) of the process. Both the observer and the process can only have typing assumptions for channels which have been allocated.

The LTS is defined in Fig. 4. Consider rule lOut, describing process output to the observer, which captures the two forms of permission transfer discussed in the introduction. First we have the explicit transfer where the observer, characterized by the type environment \(\Gamma, c : [\tilde{T}]\), gains the permissions \(\hat{d} : \tilde{T}\) for the channels received from the process; the process loses these permissions, although this is implicit in the rule (it follows from the typing of the output process). Second, implicit transfer occurs in one of two ways. If the observer has a unique-after-\(i\) permission for \(c\), it gets one step closer to recovering full permission on \(c\) after the communication, since the process will have lost the corresponding (necessarily affine) permission. Dually, if the observer has an affine permission, this permission is transferred in the opposite direction. In the extreme case where this affine permission is the only one in \(\Gamma\), the observer loses all knowledge of channel \(c\). This is formalized by the operation \(\Gamma, c : [\tilde{T}]^{|\alpha| - 1} (\alpha \neq \bullet)\), given in Fig. 2.

Rule lStr allows the observer to apply structural operations to its permissions, such as splitting a unique permission into a unique-after-\(i\) and an affine permission; see Fig. 2 for the definition of (\(<\))

4 Costed Bisimulation and Characterization

We define an amortized bisimulation [6] to compare the efficiency and behaviour of systems \(M \triangleright P\) and \(N \triangleright Q\) at some credit \(n\), where \(M \triangleright P\) and \(N \triangleright Q\) exhibit the same behaviour but \(M \triangleright P\) is more efficient; the credit allows \(M \triangleright P\) to do a more expensive action than \(N \triangleright Q\) as long as the credit can make up for the difference. The amortized bisimulation is typed wrt. to an environment \(\Gamma\) characterizing the observer [5]. Novel in the definition of this bisimulation is that we explicitly allow local bijective renamings \(\sigma_T\)
Process moves

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, c : [\overline{T}]^k \vdash M \xrightarrow{c!d.P \cdot c^d_{\gamma_0}} \Gamma, c : [\overline{T}]^{k-1} \cdot d : \overline{T} \vdash M \xrightarrow{P} )</td>
<td>( \mathsf{LOUT} )</td>
</tr>
<tr>
<td>( \Gamma, c : [\overline{T}]^k \cdot d : \overline{T} \vdash M \xrightarrow{c?\tilde{c}.P \cdot c^d_{\gamma_0}} \Gamma, c : [\overline{T}]^{k-1} \cdot M \xrightarrow{P[d]} )</td>
<td>( \mathsf{LIN} )</td>
</tr>
<tr>
<td>( \Gamma_1 \vdash M \xrightarrow{c^d_{\gamma_0}} \Gamma'<em>1 \vdash M \xrightarrow{P'} \Gamma_2 \vdash Q \xrightarrow{c^d</em>{\gamma_0}} \Gamma'_2 \vdash M \xrightarrow{Q} )</td>
<td>( \mathsf{LCom-L} )</td>
</tr>
<tr>
<td>( \Gamma \vdash M \xrightarrow{\top} Q \xrightarrow{\top} \Gamma \vdash M \xrightarrow{P} )</td>
<td>( \mathsf{LPar-L} )</td>
</tr>
<tr>
<td>( \Gamma \vdash M \text{ if } c = c \text{ then } P \text{ else } Q \xrightarrow{\top} \Gamma \vdash M \xrightarrow{P} )</td>
<td>( \mathsf{LTHEN} )</td>
</tr>
<tr>
<td>( \Gamma \vdash M, c : \bot \vdash alloc \cdot x.P \xrightarrow{+1} \Gamma \vdash M, c : \top \vdash P[c/x] )</td>
<td>( \mathsf{LALL} )</td>
</tr>
<tr>
<td>( \Gamma \vdash M, c : \bot \vdash alloc \cdot x.P \xrightarrow{+1} \Gamma \vdash M, c : \top \vdash P[c/x] )</td>
<td>( \mathsf{LFree} )</td>
</tr>
</tbody>
</table>

Environment moves

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash M, c : \bot \vdash P \xrightarrow{alloc} \Gamma, c : [\overline{T}]^k \vdash M, c : \top \vdash P )</td>
<td>( \mathsf{LAlle} )</td>
</tr>
<tr>
<td>( \Gamma \vdash M, c : \bot \vdash P \xrightarrow{free} \Gamma \vdash M, c : \bot \vdash P )</td>
<td>( \mathsf{LFree} )</td>
</tr>
<tr>
<td>( \Gamma \vdash M \xrightarrow{\bot} \Gamma \vdash M \xrightarrow{\top} \Gamma \vdash M \xrightarrow{P} )</td>
<td>( \mathsf{LStr} )</td>
</tr>
</tbody>
</table>

Figure 4: LTS Process Moves

of names that are not known to the observer (that is, we must have \( c \in \text{dom}(\Gamma) \) implies \( c\sigma_\Gamma = c\sigma_\Gamma^{-1} = c \)). Bijective renamings are comparable to alpha-renaming of scoped bound names in the standard \( \pi \)-calculus. In our calculus however, processes may regain uniqueness of a channel after “extruding” it, as illustrated in the next example.

**Example 2.** Consider clients \( C_0 \) and \( C_1 \) from the introduction and an observer characterized by \( \Gamma = \mathsf{srv}_1 : [\mathbb{[\mathit{Bool}]}^*]_{\mu}, \mathsf{srv}_2 : [\mathbb{[\mathit{Int}]}^*]_{\mu} \). Since these two clients have different memory behaviour, they need to be typed under different typing environments. The bisimulation relation handles this by relating configurations, which existentially quantify over process type environments.

When the clients interact with \( \Gamma \) on \( \mathsf{srv}_1 \), they both send channels that are not known to \( \Gamma \) (the channel allocations for \( x_1 \) and \( x \) respectively) and bijective renamings allow us to match these actions. More importantly however, the same situation repeats itself when the clients interact with the observer on \( \mathsf{srv}_2 \). \( C_0 \) sends the yet unused channel allocated for \( x_2 \) whereas \( C_1 \) reuses the channel allocated for \( x \). From the point of view of the observer, however, the situation is indistinguishable from the first interaction on \( \mathsf{srv}_1 \), as the server will have lost all permissions for the channel it received from the client \( (x \text{ or } x_1) \) after using it to send a reply. Our LTS allows us to track these permission consumptions and, once again, bijective renamings allow us to match these actions.

It is important that these renamings are \emph{locally} bijective: we rename \( x \) to \( x_1 \) during the first interaction, and \( x \) to \( x_2 \) in the second. \( \square \)

**Definition 1 (Amortised Typed Bisimulation).** An amortized type-indexed relation over processes \( \mathcal{R} \) is a bisimulation at \( \Gamma \) with credit \( n \) if, whenever \( \Gamma \vdash (M \xrightarrow{P} N \xrightarrow{Q}) \),

- If \( \Gamma \vdash M \xrightarrow{\mu} \Gamma' \vdash M' \xrightarrow{P'} \), then there exist \( \sigma_\Gamma, N' \) and \( Q' \) such that
For an observer characterized by Example 3.

\[
\Gamma \vdash N \triangleright Q \quad \text{where} \quad \Gamma \vdash (M' \triangleright P') R^{n+l-k} (N' \triangleright Q')
\]

- If \( \Gamma \vdash N \triangleright Q \quad \mu_\Gamma \vdash_{\kappa} \Gamma' \vdash N' \triangleright Q' \) then there exist \( \sigma_\Gamma, M' \) and \( P' \) such that
  \[
  \Gamma \vdash (M \triangleright P) R^{n} (N \triangleright Q)
  \]

where \( \mu_\Gamma \) is \( \tau_\Gamma \) if \( \mu = \tau \) and \( \tau_\Gamma \) otherwise. Bisimilarity at \( \Gamma \) with credit \( n \), denoted \( \Gamma \vdash M \triangleright P \Rsim_{\text{bis}} N \triangleright Q \), is the largest amortized typed bisimulation at \( \Gamma \) with credit \( n \).

**Example 3.** For an observer characterized by \( \Gamma = \text{srv}_1 : [[\text{Bool}^1]]^n, \text{srv}_2 : [[\text{Int}^1]]^n \), we can prove \( \Gamma \vdash (M \triangleright C_1) \Rsim_{\text{bis}}^{n} (N \triangleright C_0) \) and \( \Gamma \vdash (M \triangleright C_2) \Rsim_{\text{bis}}^{n} (N \triangleright C_1) \). We however cannot prove that \( \Gamma \vdash (M \triangleright C_0) \Rsim_{\text{bis}}^{n} (N \triangleright C_1) \) nor that \( \Gamma \vdash (M \triangleright C_1) \Rsim_{\text{bis}}^{n} (N \triangleright C_2) \) for any \( n \).

An important theoretical result is that our amortized bisimulation admits compositional analysis.

**Theorem 1** (Compositionality). If \( \Gamma, \Gamma' \vdash (M \triangleright P) \Rsim_{\text{bis}}^{n} (N \triangleright Q) \) and \( \Gamma' \vdash R \) then

\[
\Gamma \vdash (M \triangleright P \parallel R) \Rsim_{\text{bis}}^{n} (N \triangleright Q \parallel R) \quad \text{and} \quad \Gamma \vdash (M \triangleright R \parallel P) \Rsim_{\text{bis}}^{n} (N \triangleright R \parallel Q)
\]

This theorem allows us to abstract away from common code when exhibiting bisimulations by extending the observer environment with the permission environment that characterizes this code.

We also give a sound and complete characterization of our amortized bisimulation in terms of our reduction semantics through a costed version of reduction closed barbed congruence which takes into account the transfer of permissions. This congruence relies on a novel definition of contextuality, which tracks permission transfer in a similar way to Theorem 1.

**Definition 2** (Contextuality). An amortized type-indexed relation \( R \) is contextual at environment \( \Gamma \) iff whenever \( \Gamma \vdash (M \triangleright P_1) R^n (N \triangleright P_2) \):

1. If \( M = M', c : \bot \) and \( N = N', c : \bot \) then \( \Gamma, c : [\hat{T}]^* \vdash (M', c : \top \triangleright P_1) R^n (N', c : \top \triangleright P_2) \)
2. If \( \Gamma \vdash \Gamma_1, \Gamma_2 \) where \( \Gamma_2 \vdash Q \) then \( \Gamma_1 \vdash (M \triangleright P_1 \parallel Q) R^n (N \triangleright P_2 \parallel Q) \)

**Definition 3** (Cost Improving). An amortized type-indexed relation \( R \) is cost improving at credit \( n \) iff whenever \( \Gamma \vdash (M \triangleright P) R^n (N \triangleright Q) \) and

1. if \( M \triangleright P \rightarrow_k M' \triangleright P' \) then \( N \triangleright Q \rightarrow_{k} N' \triangleright Q' \) such that \( \Gamma \vdash (M' \triangleright P') R^{n+i-l} (N' \triangleright Q') \);
2. if \( N \triangleright Q \rightarrow_{i} N' \triangleright Q' \) then \( M \triangleright P \rightarrow_{l} M' \triangleright P' \) such that \( \Gamma \vdash (M' \triangleright P') R^{n+i-l} (N' \triangleright Q') \).

**Definition 4** (Barb). \( (\Gamma \vdash M \triangleright P) \uparrow_{c} \text{barb} \overset{\text{def}}{=} c \in \text{dom}(\Gamma) \) and \( (M \triangleright P) \rightarrow_{k} (M' \triangleright P') \).

**Definition 5** (Barb Preservation). A typed relation \( R \) is barb preserving if and only if

\[
\Gamma \vdash M \triangleright P \text{ R N \triangleright Q implies } (\Gamma \vdash M \triangleright P \uparrow_{c} \text{barb} \iff \Gamma \vdash N \triangleright Q \downarrow_{c} \text{barb})).
\]

**Definition 6** (Behavioral Contextual Preorder). \( \Gamma_{ beh}^{n} \) is the largest family of amortized typed relations that is barb preserving, cost improving at credit \( n \) and contextual at environment \( \Gamma \).

**Theorem 2** (Full Abstraction). \( \Gamma \vdash (M \triangleright P) \Rsim_{\text{bis}}^{n} (N \triangleright Q) \) iff \( \Gamma \vdash (M \triangleright P) \Rsim_{\text{beh}}^{n} (N \triangleright Q) \).

**Example 4.** For any observer characterized by \( \Gamma = \text{srv}_1 : [[\text{Bool}^1]]^n, \text{srv}_2 : [[\text{Int}^1]]^n \), we can choose an appropriate context to show that, for any \( n \), we cannot have \( \Gamma \vdash (M \triangleright C_0) \Rsim_{\text{beh}}^{n} (N \triangleright C_1) \) or \( \Gamma \vdash (M \triangleright C_1) \Rsim_{\text{beh}}^{n} (N \triangleright C_2) \). Despite the quantification over all possible contexts in Definition 6, we can also show that \( \Gamma \vdash (M \triangleright C_1) \Rsim_{\text{beh}}^{n} (N \triangleright C_0) \) and \( \Gamma \vdash (M \triangleright C_2) \Rsim_{\text{beh}}^{n} (N \triangleright C_1) \) following Example 3 and Theorem 2.
5 Conclusions and Related Work

We outlined how to take advantage of uniqueness information to construct compositional proof techniques for comparing behaviour and amortized resource usage efficiency in Rπ. We gave a sound and complete characterization of the proof method in terms of a costed preorder based on the reduction semantics, which relies essentially on a novel definition of contextuality that takes care of permission transfer.

Kobayashi et al. [7] introduce an affine type system for the π-calculus and a definition of reduction closed barbed congruence, but no compositional proof methods. Yoshida et al. [9] define a linear type system for a calculus based on πI in which dynamic sharing is controlled dynamically, limiting channel reuse. They give compositional proof techniques for their behavioural equivalence, but no complete characterization.

Our unique-after-i type is related to fractional permissions, introduced in [3] and used in settings such as separation logic for shared-state concurrency [2]. A detailed survey of this field is however beyond the scope of this paper.

Pym and Tofts [8] give a behavioural theory for SCCS with resources based on separation of permissions. They define a bisimulation relation, and show that it can be characterized by a modal logic. A comparison between their untyped approach and our typed approach would be worthwhile.

The use of substitutions in our bisimulation is reminiscent of the name-bijections used in spi [1]. In the spi calculus however these bijections are carried through the bisimulation, whereas we use local renamings (one per action). This is essential to enable more channel reuse.

References

The Timed, Compensable Conversation Calculus

Hugo A. López  
PLS, IT University of Copenhagen  
Jorge A. Pérez  
CITI - DI, FCT - New University of Lisbon

Abstract

We are interested in the interplay of timing issues and exceptional behavior in the realm of structured communications. As an initial step, we have defined $C^3$, a variant of the Conversation Calculus in which conversation contexts are enriched with a time duration, a compensating activity, and a designated signal for conversation abortion. In this note, we present $C^3$ and its basic semantics, and discuss ongoing work for equipping the language with logic-based reasoning techniques.

1 Introduction

This paper investigates the rôle of time in languages for structured communication. In particular, we are interested in how time and constructs for exceptional behavior can be jointly captured in models of structured communications and lead to more effective reasoning techniques.

Time plays an increasingly crucial rôle in practical scenarios of structured communication. Consider, for instance, a web banking application: interaction between a user and her bank generally takes place by means of a secure session, which is meant to expire after a certain period of inactivity of the user. When that occurs, the user must exhibit again her authentication credentials, so as to initiate another session or to continue with the expired session. In some cases, the session (or parts of it) have a predetermined duration and so interactions may also be bounded in time. The user must then reinitiate the session if, for instance, she is taking too long in providing some input or if the network connection is too slow. Crucially, the different incarnations of time in interactions (session durations, timeouts, delays) can be seen to be closely related to the behavior of the whole system in exceptional circumstances. In the previous example: a specification of the web banking application would appear incomplete unless one specifies how the system should behave when, e.g., the session has expired and the user attempts to reinitiate it, or when an interaction within the session is taking longer than anticipated.

It then appears that in the specification of practical instances of structured communications timed behavior goes hand in hand with exceptional behavior. This observation acquires more relevance in the realm of healthcare scenarios [14]—a central source of motivation for our work. In healthcare applications, structured communications often involve strict time bounds, such as, e.g., “monitor the patient’s signs every two hours, for the following 48 hours”. They may also include behavioral patterns which may be defined as both a default behavior and an alternative behavior to be executed in case of unexpected conditions, including time-related ones: “contact the substitute doctor if the responsible doctor cannot be reached within 15 minutes”. Also, structured communications involve critical tasks that may be indefinitely suspended or aborted, such as, e.g., “stop the administering the medicine if the patient reacts badly to it”. We find that this kind of requirements are all hard to express appropriately in known formalisms for structured communication.

Here we report on our initial steps towards a language for structured communications with an explicit treatment of time and exceptional behavior. It arises as an extension of the Conversation Calculus (CC) [16, 15] in which conversations have durations and are sensible to compensations. The CC is an interesting base language for our study. First, it is a simple model: the CC corresponds to a $\pi$-calculus extended with conversation contexts. Hence, we may draw inspiration from previous work on variants of the $\pi$-calculus with time and constructs for exceptional behavior. Second, the CC allows for the compact specification of multiparty interactions, which are ubiquitous in healthcare scenarios. Third, the CC counts with a number of reasoning techniques to build upon, which include a behavioral theory and so-called conversation types [3] which, by means of suitable projections, allow for a unified treatment of the global and local views of a system (also known as choreographies and orchestrations, respectively).
We illustrate a few basic notions of the CC by means of an example—the purchasing scenario proposed in, e.g., [7, 15]. This scenario describes the interaction between a buyer and a seller for buying a certain product; the seller later involves a shipper who is in charge of delivering the product. In the CC, this scenario can be given as follows:

\[
\begin{align*}
\text{Buyer} & \triangleright [\text{newSeller}.\text{BuyService} \Leftarrow \text{buy}^{!}(\text{prod}).\text{price}^{?}(p).\text{details}^{?}(d))] \\
| \text{Seller} & \triangleright [\text{PriceDB}] \quad \text{def} \text{BuyService} \Rightarrow \text{buy}^{!}(\text{prod}).\text{askPrice}^{!}(\text{prod}).\text{priceVal}^{?}(p).\text{price}^{?}(p), \\
\quad & \text{join} \text{Shipper}.\text{DeliveryService} \Leftarrow \text{product}^{!}(\text{prod})] \\
| \text{Shipper} & \triangleright [\text{def} \text{DeliveryService} \Rightarrow \text{product}^{?}(p).\text{details}^{?}(\text{data})]
\end{align*}
\]

In the CC, a conversation context represents a distributed communication medium where partners may interact. We write \( n \triangleright [ P ] \) for the conversation context with behavior \( P \) and identity \( n \); process \( P \) may seamlessly interact with processes inside any other conversation contexts with identity \( n \). The model above considers three participants: Buyer, Seller, and Shipper. Buyer invokes a new instance of the BuyService service, which is defined by Seller. As a result, a conversation on a fresh name is established between them; this name can then be used to exchange information on the desired product and its price (the latter is retrieved by Seller from PriceDB, which represents a database). When the transaction has been agreed, Shipper joins in the conversation, and receives product information from Seller and delivery details from Buyer. Notice how def, new, and join—the three idioms available in the CC—are used to define services, creating a new instance of a predefined service, and joining a running service instance, respectively. Remarkably, these idioms can be derived from the basic syntax of the CC:

\[
\begin{align*}
\text{def} \ s & \Rightarrow P \triangleq s^{?}(x).x \triangleright [P] & \text{new} n \cdot s & \Leftarrow Q \triangleq (vc)(n \triangleright [s^{!}(c)] | c \triangleright [Q]) \\
\text{join} n \cdot s & \Rightarrow Q \triangleq \text{this}(x).(n \triangleright [s^{!}(x)] | Q) & * \text{def} \ s & \Rightarrow P \triangleq \mu X. \ s^{?}(x). (s \triangleright [P] | X)
\end{align*}
\]

where \( s^{?}(x) \) and \( s^{!}(c) \) are input and output prefixes (as in the \( \pi \)-calculus), and \( \text{this}(x) \) is a prefix that binds the enclosing conversation context to name \( x \). * def \( s \Rightarrow \) is the persistent version of def \( s \Rightarrow \).

We would like simple yet general constructs capturing both timing issues and exceptional behavior, and the relationship between the two. Constructs such as try/catch appear inadequate for our purposes. In fact, the kind of relationship between time and exceptional behavior we would like to address is more a specification issue than a runtime issue: we find that the “runtime character” of constructs such as try/catch cannot be lifted to specifications in a natural way. In order to include both time and exceptional behavior into specifications, we have opted to include these issues directly into conversation contexts. We propose \( C^{3} \), a variant of the CC in which the simple conversation contexts \( n \triangleright [P] \) are replaced by timed, compensable conversation contexts, denoted as \( n \triangleright [P; Q]_{\kappa}^{\tau} \). As before, \( n \) is the identity of the conversation context. Process \( P \) describes the default behavior for \( n \), which is performed while the duration \( t \) is greater than 0. Observable actions from \( P \) witness the time passage in \( n \); as soon as \( t = 0 \), the default behavior is dynamically replaced by \( Q \), the compensating behavior. The duration \( t \) can also be \( \infty \). Name \( \kappa \) represents an abort mechanism: the interaction of \( n \triangleright [P; Q]_{\kappa}^{\tau} \) with the kill signal \( \kappa^{!} \) immediately sets \( t \) to 0. Notice that the simple conversation context \( n \triangleright [P] \) in the CC would correspond in \( C^{3} \) to the extended context \( n \triangleright [P; Q]_{\kappa}^{\tau} \), for some \( \kappa \) unknown to \( P \) and to the rest of the system.

An immediate and pleasant consequence of the timed, compensable conversation contexts in \( C^{3} \) is that the signature of the service idioms can be extended too. In this way, service specifications in \( C^{3} \) can express richer information on compensation signals, timeouts, and exceptional behavior. This achieves the above-mentioned “lifting” of timed and exceptional behavior to (service) specifications. Note that it suffices to extend def and new, the idioms representing service invocation and service definition:

\[
\begin{align*}
\text{def} \ s \text{ with } (\kappa, t) & \Rightarrow \{P; Q\} \triangleq s^{?}(y).y \triangleright [P; Q]_{\kappa}^{\tau} \\
\text{new} n \cdot s \text{ with } (\kappa, t) & \Leftarrow \{P; Q\} \triangleq (vc)(n \triangleright [s^{!}(c); 0]_{\kappa}^{\tau} | c \triangleright [P; Q]_{\kappa}^{\tau})
\end{align*}
\]
where \( y \) and \( c \) are assumed to be fresh in \( P \) and \( Q \), and different from \( \kappa, t, n \). The definition of a service \( s \) with (main, default) protocol \( P \) is extended with a compensation signal \( \kappa \), a timeout value \( t \), and an alternative protocol definition \( Q \), to be used in case of failure or unexpected behavior of \( P \). Similarly, the extended idiom for invoking an new instance of service \( s \) residing at \( n \) considers a main protocol \( P \), an alternative protocol \( Q \), a duration, and a compensation signal. This way, for instance, processes

\[
\text{Client } \triangleleft [ \text{new Provider } \cdot \text{Service with } (\kappa, t, n) \equiv \{ P; Q \} ] \quad \text{Provider } \triangleright [ \text{def Service with } (\kappa_p, t_p) \Rightarrow \{ R; T \} ]
\]

may interact and evolve into \((v_s) (\text{Client } \triangleleft [ s \triangleright [ P; Q ] ] \triangleright [ P; Q ] )\).

We illustrate the expressiveness of \( C^3 \) by extending the purchase scenario. Suppose a buyer who is willing to interact with a specific provider only for a finite amount of time. She engages in conversations with several providers at the same time, picks the one which presents the best offer, and finally abandons the conversations with the other providers. This refined scenario can be modeled in \( C^3 \) as follows:

\[
\text{NewBuyer } \triangleright \prod_{i \in \{1,2,3\}} \text{new Buyer}_i \cdot \text{BuyService with } (e_i, w_i) \equiv \{ P; Q_i \} | \text{Control} \cdot \text{CancelOrder}^{|t_{\text{max}} |}
\]

\[
\mid \prod_{i \in \{1,2,3\}} \text{Seller}_i \triangleright \text{BuyService with } (h_i, w_i) \Rightarrow \{ \text{offer}^?!(\text{prod}), \text{askPrice}^!?(\text{prod}), \text{priceVal}^?!(p), \text{price}^!?(p) \}, \text{join Shipper} \cdot \text{DeliveryService} \equiv \text{product}^!?(\text{prod}); R_i; \text{CancelSell}_i^{[d_i]}
\]

\[
\text{Shipper } \triangleleft [ \text{def DeliveryService with } (d, t) \Rightarrow \{ \text{product}^!?(p), \text{details}^!?(\text{data}); t \}; 0 \}^{[d]}
\]

where process \( P_i \triangleq \text{offer}^!?(\text{prod}), \text{price}^!?(p), \text{com}^!?!(p), \text{details}^!?(\text{data}); t \rangle \) represents the default behavior of the buyer when interacting with \( \text{Seller}_i \) (the compensating behavior \( Q_i \) is left unspecified) and \( \text{Control} \triangleq \ast(\text{com}^!?!(p), (c_2^? \mid c_3^?) + \text{com}^!?!(p), (c_1^? \mid c_2^?) + \text{com}^!?!(p), (c_3^? \mid c_4^?) \). With a little abuse of notation, we use \( \ast P \) for denoting the replication of \( P \), with the usual semantics. We consider one buyer and three sellers. \( \text{NewBuyer} \) creates three instances of the \( \text{BuyService} \) service, one from each seller. The part of each such instances residing at \( \text{NewBuyer} \) can be aborted by suitable messages on \( c_i \), which we assume fresh to the rest of the process. The part of the protocol for \( \text{BuyService} \) that resides at \( \text{NewBuyer} \) (denoted \( P_i \)), is similar as before, and is extended with an output signal \( \text{com} \), which allows to commit the selection of seller \( i \). The commitment to one particular seller (and the discard of the rest) is implemented by process \( \text{Control} \) using an exclusive (guarded) choice. The duration of \( \text{NewBuyer} \) is given by \( t_{\text{max}} \): its compensation activity (\( \text{CancelOrder} \)) is left unspecified. As for \( \text{Seller}_i \), it follows the lines of the basic scenario, extended with compensation signals \( y_i \) which trigger the compensation process \( \text{CancelSell}_i \).

This extended example illustrates two of the features of \( C^3 \): explicit conversation abortion and conversations bounded in time. The first one can be appreciated in the selection implemented by \( \text{Control} \), which ensures that only one provider will be able to interact with \( \text{NewBuyer} \), by explicitly aborting the conversations at \( \text{NewBuyer} \) with the other two providers. However, \( \text{Control} \) only takes care of the interactions at the buyer side; there are also conversation pieces residing at each \( \text{Seller}_i \), which are not under the influence of \( \text{Control} \) (we assume \( c_i \neq w_i \)). The “garbage-collection” of such pieces is captured by the second feature: since such conversations are explicitly defined with the time bound \( w_i \), we are sure that after a fixed amount of time they will be automatically collected (i.e. aborted). That is, the passing of time avoids “dangling” conversation pieces. This simple example reveals the complementarity between the explicit conversation abortion (achieved via abortion signals) and the more implicit conversation abortion associated to the passing of time.

\(^1\)We could have assumed \( c_i = w_i \) so as to allow \( \text{Control} \) to abort conversations at both buyer and seller sides. However, it is somewhat more realistic to assume that \( \text{Control} \) restricts to services in \( \text{NewBuyer} \).
Even if the study of C^3 is in an early stage, our preliminary results are promising. Up to now we have defined a basic transition semantics for C^3, which extends that of the CC. We have also developed a few compelling examples, which reveal the usefulness of C^3 in the realm of healthcare scenarios. Unsurprisingly, the increased expressiveness of C^3 with respect to the CC comes at the price of a slightly more involved semantics. Preliminary investigations suggest that the definition of, e.g., behavioral equivalences and associated equalities, will require similar considerations and extra care.

2 Syntax and Semantics

Syntax. As discussed before, the syntax of C^3 extends the CC (as introduced in [3] with timed, compensable conversation contexts and a process for aborting running conversations. Let \( \mathcal{N} \) be an infinite set of names. Also, let \( \mathcal{V} \) and \( \mathcal{L} \) be infinite sets of variables and labels, respectively. Using \( d \) to range over \( \uparrow \) and \( \downarrow \), the set of actions \( \alpha \) and processes \( P \) is given as follows:

\[
\alpha ::= 1^d!(\bar{n}) | 1^d?(\bar{x}) | \text{this}(x) \quad P, Q ::= P \mid Q \mid \Sigma_{i \in I} \alpha_i \cdot P_i \mid (\nu n) P \mid \mu X. P \mid X \mid n \leftarrow | P \mid Q \right|_{\kappa} \mid \kappa^i
\]

Given \( \bar{n} \in \mathcal{N} \) and \( \bar{x} \in \mathcal{X} \), actions can be the output \( 1^d!(\bar{n}) \) or the input \( 1^d?(\bar{x}) \), as in the \( \pi \)-calculus, assuming \( 1 \in \mathcal{L} \) in both cases. The message direction \( \downarrow \) (read “here”) decrees that the action should take place in the current conversation, while \( \uparrow \) (read “up”) decrees that the action should take place in the enclosing conversation. The context-aware prefix \text{this}(x) \) binds the name of the enclosing conversation context to \( x \). Processes include parallel composition, a guarded choice construct, name restriction, and recursive processes, all with their standard \( \pi \)-calculus interpretation. Notions of free and bound names of a process \( P \), denoted \( fn(P) \) and \( bn(P) \), are also as expected. Given \( \Sigma_{i \in I} \alpha_i \cdot P_i \), we write \( 0 \) when \( |I| = 0 \), and \( P_1 + P_2 \) when \( |I| = 2 \). Moreover, the set of processes includes the conversation context \( n \leftarrow [P ; Q]_{\kappa} \mid \kappa \) and the abortion process \( \kappa^i \), where \( n, \kappa \in \mathcal{N} \) and \( t \in \mathbb{N}_0 \cup \{\infty\} \). As in the CC, notice that labels in \( \mathcal{L} \) are not subject to restriction or binding.

Transition Semantics. We give the operational semantics of C^3 in terms of the labelled transition system (LTS) in Figure 1. It is an extension of the LTS proposed for the CC [15, 3]. The LTS features transitions of the form \( P \xrightarrow{\lambda} P' \), with the usual meaning: \( P \) is able to perform transition label \( \lambda \) and then evolve to \( P' \). Transition labels are defined in terms of actions \( \sigma \), as defined by the following grammar:

\[
\sigma ::= \tau \mid 1^d?(\bar{x}) \mid 1^d!(\bar{n}) \mid \text{this} \mid \kappa^i \quad \lambda ::= \sigma \mid e \sigma \mid (\nu n) \lambda
\]

Action \( \tau \) denotes internal communication, while \( 1^d?(\bar{x}) \) and \( 1^d!(\bar{n}) \) represent an input and output to the environment, respectively. Action \text{this} \) represents a conversation identity access. The new action \( \kappa^i \) represents the throwing of an abortion message. A transition label \( \lambda \) can be either the (unlocated) action \( \sigma \), an action \( \sigma \) located at conversation \( e \) (written \( e \sigma \), or a transition label in which \( n \) is bound with scope \( \lambda \). This is the case of bounded output actions. We therefore have notions of free/bound names in a transition label, denoted \( fn(\lambda) \), \( bn(\lambda) \), and defined as expected. Also, we use \( out(\lambda) \) to denote the names produced by a transition, so \( out(\lambda) = n \) if \( \lambda = 1^d!(\bar{n}) \) or if \( \lambda = e1^d!(\bar{n}) \) and \( e \neq n \). A transition label \( \lambda \) denoting communication, such as \( 1^d?(\bar{x}) \) or \( 1^d!(\bar{n}) \) is subject to duality \( \overline{\lambda} \). We write \( \overline{1^d?(\bar{x}) = 1^d!(\bar{n})} \) and \( \overline{1^d!(\bar{n}) = 1^d?(\bar{x})} \).

The LTS relies on the following definition, which formalizes time passing for conversation contexts:

**Definition 2.1** (Time-elapsing function). \( \phi(P) \) denotes the function from processes to processes decreasing the time durations in \( P \) by \( 1 \), given by:

\[
\phi((\nu n) P) = (\nu n) \phi(P) \quad \phi(Q \mid R) = \phi(Q) \mid \phi(R) \quad \phi(n \leftarrow [Q ; R]_{\kappa}^{i+1}) = n \leftarrow [\phi(Q) ; R]_{\kappa}^i \\
\phi(n \leftarrow [Q ; R]_{\kappa}^0) = n \leftarrow [Q ; \phi(R)]_{\kappa}^0 \quad \overline{\phi(P)} = P \quad \text{Otherwise.}
\]
The Timed, Compensable Conversation Calculus

H. A. López & J. A. Pérez

(IN) \( \lambda(x). P \xrightarrow{\mu(x)} P[\mu/x] \)

(OUT) \( \lambda(n). P \xrightarrow{\mu(n)} P \)

(THIS) \( \text{this}(x) \xrightarrow{\text{this}} P[c/x] \)

(ABORT) \( k \xrightarrow{\ell} 0 \)

(RES) \( P \xrightarrow{\ell} Q \eta \neq \eta(n) \)

(SUM) \( \sum_{i \in I \eta_i, \eta_i \xrightarrow{\ell} Q} \)

(OPEN) \( P \xrightarrow{\ell} Q \eta \neq \eta(\lambda) \)

(CLOSE) \( P \xrightarrow{\ell} Q \eta \neq \eta(\lambda) \)

(REC) \( P \xrightarrow{\ell} Q \eta \neq \eta(\lambda) \)

(Fail) \( P \xrightarrow{\ell} P' \)

(LOC) \( c \xleftarrow{\ell} Q; R \xleftarrow{\ell} P \eta \neq \eta(\lambda) \)

(HERE) \( P \xrightarrow{\ell} P' \eta \neq \eta(\lambda) \)

(ThisClose) \( P \xrightarrow{\ell} P' \)

(ThisComm) \( P \xrightarrow{\ell} P' \)

(Thru) \( c \xrightarrow{\ell} Q; R \xrightarrow{\ell} P \eta \neq \eta(\lambda) \)

(ThisLclo) \( c \xrightarrow{\ell} Q; R \xrightarrow{\ell} P \)

Figure 1: An LTS for \( C^3 \).

We now give intuitions on the rules in Figure [1]. As a convention, rules ending in “1” have a symmetric counterpart which is elided. Similarly, for each one of the all the left rules—which describe evolution in the default behavior of conversation contexts, and have names ending in “L”—, there is a set of right rules describing evolution in the compensation behavior. Apart from rules (ABORT), (THIS), and (TAUPAR1), the first four rows in Table [1] collect basic transition rules for a \( \pi \)-calculus with recursion. The distinction between (TAUPAR1) and (PAR1) captures the fact that time passes only as a result of visible actions. While rule (FAIL) formalizes the abortion of a conversation context, rule (COMP) represents the evolution of the compensated part of the conversation context. Rule (LOC) locates an action to a particular conversation context, and rule (HERE) changes the direction of an action occurring inside a context. Rules (THISCLOSE) and (THISCOMM) are located versions of (CLOSE) and (COMM), respectively. Rules (Thru) and (TAU) formalize the way actions change when they “cross” a conversation context. Rule (THISLclo) hides an action occurring inside a conversation context.

Let us illustrate the LTS of \( C^3 \) by revisiting the extended purchase scenario discussed in the introduction. We describe the evolution of the system where the buyer invokes \( Seller_1 \) with an expected response time of two time units. We recall the definition of the complete system:

\[
\begin{align*}
\text{NewBuyer} & \xrightarrow{\ell} [\prod_{i \in \{1, 2, 3\}} \text{new Buyer}_i; \text{BuyService with } (e_i, v_i) \equiv \{P_i; Q_i\}] \mid \text{Control; CancelOrder}^{\text{max}}_x \\
\text{NewBuyer} & \xrightarrow{\ell} [\prod_{i \in \{1, 2, 3\}} \text{Seller}_i \xrightarrow{\ell} \text{PriceDB} \mid \text{def BuyService with } (b_i, v_i) \Rightarrow \{\text{offer}^i!(\text{prod}), \text{askPrice}^i!(\text{prod}), \text{priceVal}^i!(p), \text{priceVal}^i!(p) \} \\
\text{CancelSale}_{1} & \xrightarrow{\ell} \text{join Shipper; DeliveryService} \leftarrow \text{product}^i!(\text{prod}); R_i \\
\text{Delivery} & \xrightarrow{\ell} \text{def DeliveryService with } (d, t) \Rightarrow \{\text{product}^i!(p), \text{details}^i!(\text{data}); T\} ; 0 \end{align*}
\]
where \( P \triangleq \text{offer}^1 \text{!(prod). price}^1 ?(p). \text{com}^1 1!(p). \text{details}^1 ?(d) \). By expanding the definition of \text{def} and \text{new}, we have:

\[
\begin{align*}
\text{NewBuyer} & \triangleright ((v) \text{ Seller}_1 \triangleright \text{ BuyService}^1(c); 0)_w \triangleright c \triangleleft P_1 Q_2 1 \triangleright S_b \triangleright \text{ Control ; CancelOrder}^1 \text{!max} \\
\text{ Seller}_1 & \triangleleft \text{ PriceDB } \triangleright \text{ BuyService}^1 ?(y); y \triangleleft \text{ offer}^1 ?(prod). (...) ; R_1 x y ; \text{ CancelSell}^1 1 \text{!max} \triangleright S_c \triangleright \text{ Shipper}
\end{align*}
\]

where \( S_b \) abbreviates the definitions of \text{Seller}_2 and \text{Seller}_3 at the buyer side, and \( S_c \) is the analogous process at the shippers side. Focusing on \text{NewBuyer}, we can infer the following transition, using rules (\text{OUT}), (\text{RES}), (\text{LOCL}), and (\text{PAR}1):

\[
\begin{align*}
\text{NewBuyer} & \triangleright ((v) \text{ Seller}_1 \triangleright \text{ BuyService}^1(c); 0)_w \triangleright c \triangleleft P_1 Q_2 1 \triangleright S_b \triangleright \text{ CancelOrder}^1 \text{!max} \\
\text{ Seller}_1 & \triangleright \text{ PriceDB } \triangleright \text{ BuyService}^1 ?(y); y \triangleleft \text{ offer}^1 ?(prod). (...) ; R_1 x y ; \text{ CancelSell}^1 1 \text{!max} \triangleright S_c \triangleright \text{ Control}
\end{align*}
\]

which decreases the time bound for conversation context \( c \) to 1. The behavior of \text{Seller}_1 \text{ is completely complementary to the above output, as inferred by using (IN), (LOCL), and (PAR}1):

\[
\begin{align*}
\text{Seller}_1 & \triangleright \text{ PriceDB } \triangleright \text{ BuyService}^1 ?(y); y \triangleleft \text{ offer}^1 ?(prod). (...) ; R_1 x y ; \text{ CancelSell}^1 1 \text{!max} \triangleright S_c \triangleright \text{ Control}
\end{align*}
\]

Given these two transitions, a synchronization can be inferred using rules (\text{CLOSEL}) and (\text{PAR}1), using \( c \) as shared name for \text{NewBuyer} and \text{Seller}_1 to communicate. The state of the system is then:

\[
\begin{align*}
((v) \text{ NewBuyer} \triangleright \text{ [Seller}_1 \triangleright \text{ [BuyService}^1(c); 0)_w \triangleright c \triangleleft P_1 Q_2 1 \triangleright S_b \triangleright \text{ CancelOrder}^1 \text{!max} \\
\text{ Seller}_1 & \triangleright \text{ PriceDB } \triangleright \text{ BuyService}^1 ?(y); y \triangleleft \text{ offer}^1 ?(prod). (...) ; R_1 x y ; \text{ CancelSell}^1 1 \text{!max}) \triangleright \text{ Will }
\end{align*}
\]

where \( W \) represents the rest of the system. At this point, a communication on \text{offer} between \text{NewBuyer} and \text{Seller}_1 becomes possible. Omitting process \text{Seller}_1 \triangleright [0; 0]_w, at the buyer side we have:

\[
\begin{align*}
((v) \text{ NewBuyer} & \triangleright c \triangleleft \text{ offer}^1 ?(prod). \text{ price}^1 ?(p). (...) ; Q_1 1 \triangleright S_b \triangleright \text{ CancelOrder}^1 \text{!max} \\
\text{ Seller}_1 & \triangleright \text{ PriceDB } \triangleright c \triangleleft \text{ price}^1 ?(p). (...) ; Q_1 1 \triangleright S_b \triangleright \text{ CancelOrder}^1 \text{!max}) \triangleright \text{ Will }
\end{align*}
\]

while \text{Seller}_1 makes an input transition located at \( c \), inferred using (\text{OUT}) and (\text{LOCL}):

\[
\begin{align*}
((v) & \text{ Seller}_1 \triangleright \text{ PriceDB } \triangleright c \triangleleft \text{ offer}^1 ?(prod). \text{ askPrice}^1 ?(prod). (...) ; R_1 x y ; \text{ CancelSell}^1 1 \text{!max}) \triangleright \text{ Will }
\end{align*}
\]

Again, these complementary transitions can synchronize, thus firing an unobservable transition inferred using rule (\text{COMP}). The system then evolves to

\[
((v) \text{ NewBuyer} \triangleright \text{ [[price}^1 ?(p). (...) ; Q_1 0 y ; \phi(S_b)) \triangleright ; \text{ CancelOrder}^1 \text{!max} \\
\text{ Seller}_1 & \triangleright \text{ PriceDB } \triangleright \text{ askPrice}^1 ?(prod). (...) ; R_1 x y ; \text{ CancelSell}^1 1 \text{!max}) \triangleright \text{ Will }
\]

At this point, the default behavior of the system establishes that \text{Seller}_1 contacts \text{PriceDB} in order to communicate the price of the selected product to \text{NewBuyer}. However, we notice that conversation context \( c \) inside \text{NewBuyer} has reached a timeout. As a consequence, the only possible way for progressing is by engaging into the compensating behavior, represented by process \( Q_1 \). Assuming \( Q_1 \xrightarrow{\lambda} Q_1' \), the evolution of the compensating behavior can be inferred using rule (\text{COMP}). We then have:

\[
\begin{align*}
((v) & \text{ NewBuyer} \triangleright \text{ [price}^1 ?(p). (...) ; Q_1 0 y ; \phi(S_b)) \triangleright ; \text{ CancelOrder}^1 \text{!max} \\
\text{ Seller}_1 & \triangleright \text{ PriceDB } \triangleright \text{ askPrice}^1 ?(prod). (...) ; R_1 x y ; \text{ CancelSell}^1 1 \text{!max}) \triangleright \text{ Will }
\end{align*}
\]
3 Related Work

Time and its interplay with forms of exceptional behavior do not seem to have been jointly studied in the context of models for structured communication. In our previous work [13] we have studied an LTL interpretation of the session language in [9] and proposed a extension of it with time, declarative information, and a construct for session abortion. The language in [9], however, does not support multiparty interactions. The original presentation of the CC [16] features a try/catch construct which handles compensations in the expected way; this aspect is investigated in detail in [2]. (Subsequent presentations of the CC, in particular those defining type disciplines [3], leave this construct out). The language in [16] appears to be as expressive as the sub-language of $C^3$ in which durations are all $\infty$. However, because of the interplay of time and compensations, encoding full $C^3$ into the CC in [16] appears quite difficult.

Time and exceptional behavior have been considered only separately in orchestrations and choreographies. As for time, Timed Orc [17] introduced real-time observations for orchestrations by introducing a delay operator. Timed COWS [12] extends COWS (the Calculus for Orchestration of Web Services [11]) with operators of delimitation, abortion, and delays; we are not aware of reasoning techniques for Timed COWS. As for exceptional behavior, [8, 4] propose languages for interactional exceptions, in which exceptions in a protocol generate coordinated actions between all peers involved. Associated type systems ensure communication safety and termination among protocols with normal and compensating executions. In [4], the language is enriched further with multiparty session and global escape primitives, allowing nested exceptions to occur at any point in an orchestration. Concerning choreographies, [5] introduced an extension of a language of choreographies with try/catch blocks, guaranteeing that embedded compensating parts in a choreography are not arbitrarily killed as a result of an abortion signal.

Our work has been influenced by extensions to the (asynchronous) $\pi$-calculus, notably [10, 1]. In particular, the role of the time-elapsing behavior for conversation contexts used in $C^3$ draws inspiration from the behavior of long transactions in web$\pi$ [10], and from the $\pi$-calculus with timers in [11]. Notice however that the nature of these languages and $C^3$ is very different. First, the communication model is different: $C^3$ is synchronous, while the calculi in [10, 11] are asynchronous. Second, web$\pi$ is a language tailored to study long-running transactions, and therefore exceptions in web$\pi$ and compensations in $C^3$ have a completely different meaning, even if both constructs look similar.

4 Concluding Remarks

We have described initial steps towards a joint study of time and exceptional behavior in the context of structured communications. We have presented $C^3$, a generalization of the Conversation Calculus in which conversation contexts have an explicit duration, a compensation activity, and can be explicitly aborted. Ongoing work concerns equipping $C^3$ with behavioral equivalences and associated properties in the lines of that proposed for the CC. In particular, we would like to obtain a set of behavioral equations (such as those defined for the CC in [16, Prop. 4.3]) as a basic reasoning technique for $C^3$. Our main interest is in logic-based reasoning techniques for $C^3$. One option is to exploit linear-temporal logic (LTL) as in our previous work [13]. Examples of the LTL reasoning we have in mind include properties such as “every process failed is followed by a compensation activity”, “the specification always compensates in case of failure” or, in the realm of healthcare scenarios, properties such as “when the doctor is unavailable, an urgency treatment will be covered by the head nurse, within 20 minutes”. In fact, for $C^3$ one could follow pretty much the approach we detailed in [13], namely to provide encodings relating languages for structured communication ($C^3$, in our case) to models or languages with a connection with logic. However, one drawback of this (indirect) approach is that encodings are typically untyped, and so one would have to define them carefully in order to avoid the loss of important information about possible
projections in $C^3$. We are also considering the definition of a modal logic describing behaviors between conversations, as the one proposed for the choreography language in [6]. A parallel line of future work concerns studying how already proposed type disciplines for the CC can be adapted/extended so as to exploit the additional information on time and exceptional behavior available in $C^3$ descriptions.

Acknowledgements. We are grateful to Thomas Hildebrandt and Hugo T. Vieira for their useful comments and suggestions. This research has been supported by the Trustworthy Pervasive Healthcare Services (TrustCare) project - Danish Research Agency, Grants # 2106-07-0019 (www.TrustCare.eu) as well as by FCT / MCTES (CMU-PT/NGN44-2009-12) - INTERFACES.

References

1 Introduction

Threads are a convenient and modular abstraction for writing concurrent programs. Unfortunately, threads, as they are usually implemented, are fairly expensive, which often forces the programmer to use a somewhat coarser concurrency structure than he would want to. The standard alternative to threads, event-loop programming, allows much lighter units of concurrency; however, event-loop programming splits the flow of control of a program into small pieces, which leads to code that is difficult to write and even harder to understand [1, 8].

Continuation Passing C (CPC) [4, 6] is a translator that converts a program written in threaded style into a program written with events and native system threads, at the programmer’s choice. Threads in CPC, when compiled to events, are extremely cheap, roughly two orders of magnitude cheaper than in traditional programming systems; this encourages a somewhat unusual programming style.

Together with two undergraduate students [2], we taught ourselves how to program in CPC by writing Hekate, a BitTorrent seeder, a massively concurrent network server designed to efficiently handle tens of thousands of simultaneously connected peers. In this paper, we describe a number of programming idioms that we learnt while writing Hekate; while some of these idioms are specific to CPC, many should be applicable to other programming systems with sufficiently cheap threads.

The CPC translation process itself is described in detail elsewhere [6].

2 Cooperative CPC threads

The extremely lightweight, cooperative threads of CPC lead to a “threads are everywhere” feeling that encourages a somewhat unusual programming style.

Lightweight threads Contrary to the common model of using one thread per client, Hekate spawns at least three threads for every connecting peer: a reader, a writer, and a timeout thread. Spawning several CPC threads per client is not an issue, especially when only a few of them are active at any time, because idle CPC threads carry virtually no overhead.

The first thread reads incoming requests and manages the state of the client. The BitTorrent protocol defines two states for interested peers: “unchoked,” i.e. currently served, and “choked.” Hekate maintains 90% of its peers in choked state, and unchokes them in a round-robin fashion.

The second thread is in charge of actually sending the chunks of data requested by the peer. It usually sleeps on a condition variable, and is woken up by the first thread when needed. Because these threads are scheduled cooperatively, the list of pending chunks is manipulated by the two threads without need for a lock.

Each read on a network interface is guarded by a timeout, and a peer that has not been involved in any activity for a period of time is disconnected. Earlier versions of Hekate which did not include this protection would end up clogged by idle peers, which prevented new peers from connecting.

In order to simplify the protocol-related code, timeouts are implemented in the buffered read function, which spawns a new timeout thread on each invocation. This temporary third thread sleeps for the duration
cps void
listening(hashtable * table) {
    /* ... */
    while(1) {
        cpc_io_wait(socket_fd, CPC_IO_IN);
        client_fd = accept(socket_fd, ...);
        cpc_spawn client(table, client_fd);
    }
}

Figure 1: Accepting connections and spawning threads

of the timeout, and aborts the I/O if it is still pending. Because most timeouts do not expire, this solution
relies on the efficiency of spawning and context-switching short-lived CPC threads [4, 5].

Native and cps functions CPC threads might execute two kinds of code: native functions and cps
functions (annotated with the cps keyword). Intuitively, cps functions are interruptible and native functions
are not. From a more technical point of view, cps functions are compiled by performing a transformation
to Continuation Passing Style (CPS), while native functions execute on the native stack [6].

There is a global constraint on the call graph of a CPC program: a cps function may only be called by
a cps function; equivalently, a native function can only call native functions — but a cps function can call
a native function. This means that at any point in time, the dynamic chain consists of a “cps stack” of
cooperating functions followed by a “native stack” of regular C functions. Since context switches are
forbidden in native functions, only the former needs to be saved and restored when a thread cooperates.

Figure 1 shows an example of a cps function: listening calls the primitive cpc_io_wait to wait for
the file descriptor socket_fd to be ready, before accepting incoming connections with the native function
accept and spawning a new thread for each of them.

3 Comparison with event-driven programming

Code readability Hekate’s code is much more readable than its event-driven equivalents. Consider for
instance the BitTorrent handshake, a message exchange occurring just after a connection is established.
In Transmission, a popular and efficient BitTorrent client written in (mostly) event-driven style, the
handshake is a complex piece of code, spanning over a thousand lines in a dedicated file. By contrast,
Hekate’s handshake is a single function of less than fifty lines including error handling.

While some of Transmission’s complexity is explained by its support for encrypted connections,
Transmission’s code is intrinsically much more messy due to the use of callbacks and a state machine
to keep track of the progress of the handshake. This results in an obfuscated flow of control, scattered
through a dozen of functions (excluding encryption-related functions), typical of event-driven code [1].

Expressivity Surprisingly enough, CPC threads turn out to be more expressive than native threads, and
allow some idioms that are more typical of event-driven style.

A case in point: buffer allocation for reading data from the network. When a native thread performs a
blocking read, it needs to allocate the buffer before the read system call; when many threads are blocked
waiting for a read, these buffers add up to a significant amount of storage. In an event-driven program,
it is possible to delay allocating the buffer until after an event indicating that data is available has been received.

The same technique is not only possible, but actually natural in CPC: buffers in Hekate are only allocated after `cpc_io_wait` has successfully returned. This provides the reduced storage requirements of an event-driven program while retaining the linear flow of control of threads.

4 Detached threads

While cooperative, deterministically scheduled threads are less error-prone and easier to reason about than preemptive threads, there are circumstances in which native operating system threads are necessary. In traditional systems, this implies either converting the whole program to use native threads, or manually managing both kinds of threads.

A CPC thread can switch from cooperative to preemptive mode at any time by using the the `cpc_attach` primitive (inspired by FairThreads’ `ft_thread_link`[3]). A cooperative thread is said to be attached to the default scheduler, while a preemptive one is detached.

The `cpc_attach` primitive takes a single argument, a scheduler, either the default event loop (for cooperative scheduling) or a thread pool (for preemptive scheduling). It returns the previous scheduler, which makes it possible to eventually restore the thread to its original state. Syntactic sugar is provided to execute a block of code in attached or detached mode (`cpc_attached`, `cpc_detached`).

Hekate is written in mostly non-blocking cooperative style; hence, Hekate’s threads remain attached most of the time. There are a few situations, however, where the ability to detach a thread is needed.

Blocking OS interfaces Some operating system interfaces, like the `getaddrinfo` DNS resolver interface, may block for a long time (up to several seconds). Although there exist several libraries which implement equivalent functionality in a non-blocking manner, in CPC we simply enclose the call to the blocking interface in a `cpc_detached` block (see Figure 2a).

Figure 2 shows how `cpc_detached` is expanded by the compiler into two calls to `cpc_attach`. Note that CPC takes care to attach the thread before returning to the caller function, even though the return statement is inside the `cpc_detached` block.

![Figure 2](attachment:figure2.png)

Figure 2: Expansion of `cpc_detached` in terms of `cpc_attach`

Blocking library interfaces Hekate uses the `curl` library[2] to contact BitTorrent trackers over HTTP. Curl offers both a simple, blocking interface and a complex, non-blocking one. We decided to use the one interface that we actually understand, and therefore call the blocking interface from a detached thread.

Parallelism Detached threads make it possible to run on multiple processors or processor cores. Hekate does not use this feature, but a CPU-bound program would detach computationally intensive tasks and let the kernel schedule them on several processing units.

prefetch(source, length);  /* (1) */
cpc_yield();  /* (2) */
if(!incore(source, length)) {  /* (3) */
cpc_yield();  /* (4) */
if(!incore(source, length)) {  /* (5) */
cpc_detached {
    rc = cpc_write(fd, source, length);
}
goto done;
}
rc = cpc_write(fd, source, length);  /* (7) */
done:
...

The functions prefetch and incore are thin wrappers around the posix madvise and mincore system calls.

Figure 3: An example of hybrid programming (non-blocking read)

5 Hybrid programming

Most realistic event-driven programs are actually hybrid programs [7, 9]: they consist of a large event loop, and a number of threads (this is the case, by the way, of the Transmission BitTorrent client mentioned above). Such blending of native threads with event-driven code is made very easy by CPC, where switching from one style to the other is a simple matter of using the cpc_attach primitive.

This ability is used in Hekate for dealing with disk reads. Reading from disk might block if the data is not in cache; however, if the data is already in cache, it would be wasteful to pay the cost of a detached thread. This is a significant concern for a BitTorrent seeder because the protocol allows requesting chunks in random order, making kernel readahead heuristics useless.

The actual code is shown in Figure 3: it sends a chunk of data from a memory-mapped disk file over a network socket. In this code, we first trigger an asynchronous read of the on-disk data (1), and immediately yield to threads servicing other clients (2) in order to give the kernel a chance to perform the read. When we are scheduled again, we check whether the read has completed (3); if it has, we perform a non-blocking write (7); if it hasn’t, we yield one more time (4) and, if that fails again (5), delegate the work to a native thread which can block (6).

Note that this code contains a race condition: the prefetched block of data could have been swapped out before the call to cpc_write, which would stall Hekate until the write completes. However, our measurements show that the write never lasted more than 10 ms, which clearly indicates that the race does not happen. Note further that the call to cpc_write in the cpc_detached block (6) could be replaced by a call to write: we are in a native thread here, so the non-blocking wrapper is not needed. However, the CPC runtime is smart enough to detect this case, and cpc_write simply behaves as write when invoked in detached mode; for simplicity, we choose to use the CPC wrappers throughout our code.

6 Experimental results

Benchmarking a BitTorrent seeder is a difficult task because it relies either on a real-world load, which is hard to control and only provides seeder-side information, or on an artificial testbed, which might fail to accurately reproduce real-world behaviour. Our experience with Hekate in both kinds of setup shows that CPC generates efficient code, lightweight enough to run Hekate on embedded hardware. This confirms our earlier results [5], where we measured the performance of toy web servers.
Real-world workload  To benchmark the ability of Hekate to sustain a real-world load, we need popular torrents with many requesting peers over a long period of time. Updates for Blizzard’s game World of Warcraft (WoW), distributed over BitTorrent, meet those conditions: each of the millions of WoW players around the world runs a hidden BitTorrent client, and at any time many of them are looking for the latest update.

We have run an instance of Hekate seeding WoW updates without interruption for weeks. We saw up to 1,000 connected peers (800 on average) and a throughput of up to 10 MB/s (around 5 MB/s on average). Hekate never used more than 10 % of the 3.16 GHz dual core CPU of our benchmarking machine.

Stress-test on embedded hardware  We have ported Hekate to OpenWrt\(^3\), a Linux distribution for embedded devices. Hekate runs flawlessly on a MIPS-based router with a 266 MHz CPU, 32 MB of RAM and a 100 Mbps network card. The torrent files were kept on a USB key.

Because Hekate maps every file it serves in memory, and the MIPS routers running OpenWrt are 32-bit machines, we are restricted to no more than 2 GB of content. Our stress-test consists in 1,000 clients, requesting random chunks of a 1.2 GB torrent from a computer directly connected to the device. Hekate sustained a throughput of 2.9 MB/s. The CPU was saturated, mostly with software interrupt requests (60 % sirq, the usb-storage kernel module using up to 25 % of CPU).

7 Conclusions

Hekate has shown that CPC is a tool that is able to produce efficient network servers, even when used by people who do not fully understand its internals and are not specialists of network programming. While writing Hekate, we had a lot of fun exploring the somewhat unusual programming style that CPC’s lightweight, hybrid threads encourage.

We have no doubt that CPC, possibly with some improvements, will turn out to be applicable to a wider range of applications than just network servers, and are looking forward to experimenting with CPU-bound distributed programs.

References


\(^3\)http://openwrt.org
Lightweight Dynamic Task Creation and Scheduling on the Intel® Single Chip Cloud (SCC) Processor

Deepak Majeti
Rice University
Houston, Texas, USA
Deepak.Majeti@rice.edu

Abstract

In this paper, we introduce a runtime framework to support and schedule dynamic task parallelism on the Intel® Single Chip Cloud processor. We follow the Habanero approach and implement the async and hierarchical place constructs. We implement a help first work stealing scheduling policy and extend it to also support a work sharing policy.

1 Introduction

A major challenge in many-core processor research is to deliver high-performance while being power-efficient. The next generations of multi-core architectures are expected to have even more processing units on a chip. The Single Chip Cloud (SCC) processor is an example. SCC offers message-based programming support and fine-grain power management.

Dynamic task parallelism is needed by many applications. Some programming paradigms which support dynamic task parallelism include OpenMP 3.0, Java Concurrent Utilities, Microsoft Task Parallel Library, Intel Thread Building Blocks, Cilk, X10 [1] and Habanero Java [2]. Implementations of these models are based on work stealing scheduling algorithms that have gained in popularity [3]. We believe that high levels of parallelism can be achieved using the async, finish, isolated and hierarchical place constructs in the Habanero- Java and Habanero-C language [2] which are derived from v1.5 of the X10 language [1].

Two well-known approaches for scheduling are work stealing and work sharing [3]. In work sharing, all tasks are stored in a common queue from which the processors compete to pull tasks. New tasks generated are inserted in the queue. In work stealing, each processor has its queue of tasks. Idle processors now try to steal work from other processors. Migration of tasks occurs less frequently in work-stealing than work-sharing. The choice of policy will depend on the work load and also the architecture. If the system load is less, work stealing could possibly incur higher overhead(lower power efficiency) compared to work-sharing as idle processors waste cycles searching for processors to steal work from.

In this paper, we introduce a runtime framework to support the Habanero constructs, async and hierarchical place on the SCC. The framework supports work stealing and work sharing scheduling policies. The rest of the paper is organized as follows. Section 2 provides a brief overview of the SCC architecture. Section 3 describes our implementation. Section 4 discusses some of the design trade-offs in our design. Section 5 contains preliminary experimental results, and we conclude in Section 6.

2 SCC - Architecture Overview

The SCC has the following configuration (taken from the document [4]):

- There are 24 tiles in a chip. Each tile has 2 P54C cores(processors) resulting in a total of 48 cores.
- Each tile has a local memory buffer (Message Passing Buffer) of size 16KB. Any core or the system interface can write or read from these 24 MPBs. This gives the notion of a NUMA.
• Four memory controllers provide a maximum capacity of 64GB of DDR3 memory which is accessible by an SCC processor.
• There is a shared memory region on DDR3 with a default size of 64MB. This can be extended up to 512MB.
• A LUT (Lookup Table) maps a core’s physical address to the extended memory map of the system.
• Each core has a test and set register.

3 Implementation

In this section, we discuss the implementation of the constructs, async and hierarchical place. We also describe the help-first work stealing scheduling policy and the work sharing scheduling policy.

3.1 Async

The role of the async construct is to create an asynchronous lightweight task. Here task means a sequence of instructions. Table 1 describes the attributes of a task.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>int function</td>
<td>The function to be executed.</td>
</tr>
<tr>
<td>int label</td>
<td>Program point in the function where the execution must start.</td>
</tr>
<tr>
<td>void *input</td>
<td>The address location of the input data.</td>
</tr>
<tr>
<td>void *output</td>
<td>The address location where the values computed must be written to.</td>
</tr>
</tbody>
</table>

Table 1: Description of async node attributes

An integer ID is used to represent a function instead of a function pointer. This is because in the SCC processor, each core has a different virtual address space. The address location for input or output could be an MPB or on the off-chip DRAM.

3.2 Hierarchical Places

A place is a collection of resident (non-migrating) mutable data objects and the activities that operate on data [5]. Places help in scheduling work. They also help in achieving locality. Places are virtual and are mapped to physical locations. Objects and tasks do not migrate across places. The deployment is free to migrate places across physical locations based on affinity and load balance considerations. Message passing buffers (MPBs) on the SCC give a notion of non-uniform memory access. A place on the SCC could be a tile or the off-chip DRAM. The async is used along with the place construct.

3.3 Work Stealing

The runtime supports a help-first [6] work stealing scheduling policy. The help-first scheduling policy dictates that a worker executes the continuation and leaves the spawned task to be stolen. Also, the control strategy is receiver-initiated which means that idle processors search for congested processors from which work can be transferred. A concurrent queue has been implemented on the MPBs. We choose the MPB for the following reasons.

• Accessing the MPB is faster compared to the off-chip DRAM.
• All the MPBs are accessible by all the processors.

The processors enqueue tasks onto this queue and other idle processors can steal from these queues.
3.4 Work Sharing

The runtime also supports work sharing among the processors. The MPB queue can now be used to share work among the worker threads. Apart from this, a concurrent queue has been implemented in shared memory regions available on the off-chip DRAM. If the MPB queues are full, the processors can either wait or insert the task into the work sharing queue on the off-chip DRAM. Synchronization on the off-chip DRAM can be achieved by using the locks (refer section 4.1) of one of the tile.

4 Design Trade-offs

Our runtime models each tile as a place with two workers, one for each core. Some of the main issues involved in the design are,

- Shared memory is available in the form of MPB and off-chip DRAM.
- Synchronization can be achieved only via the test-and-set register available on each core.

4.1 Locks

Since there are no atomic operations, one can achieve synchronization only via locks. With the help of a test-and-set register, we designed a lock. A successful read to a test-and-set register returns a 1. The processor which reads a 1 gets the lock. A simple write to the test-and-set register resets the register. A value 0 is usually written. Each processor must now specify the tile and the corresponding core whose test-and-set register needs to be locked.

4.2 Queue

In order to facilitate the help-first work stealing policy and also work sharing, we require a double ended deque. The design has been inspired from [7]. The queue has been implemented on top of the Message Passing Buffer (MPB) and shared memory region on the off-chip DDR3. Two fields top and bottom are used to indicate the two ends of the queue. These fields are located on the MPBs at a fixed address location. Every tile has a MPB of size 16KB. Table 2 describes the operations of the queue.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>readTop( place )</td>
<td>Reads the top of the queue on the specified 'place'.</td>
</tr>
<tr>
<td>setTop( place, val )</td>
<td>Sets the top of the queue on the specified 'place' to 'val'.</td>
</tr>
<tr>
<td>readBottom( place )</td>
<td>Reads the bottom of the queue on the specified 'place'.</td>
</tr>
<tr>
<td>setBottom( place, val)</td>
<td>Sets the bottom of the queue on the specified 'place' to 'val'.</td>
</tr>
<tr>
<td>enqueue( place, tile, node )</td>
<td>Inserts the 'node' at the top of the queue on the specified 'place'. The 'tile' specifies the lock needed for synchronization.</td>
</tr>
<tr>
<td>dequeue( place, tile, node )</td>
<td>Removes the 'node' at the bottom of the queue on the specified 'place'. The 'tile' specifies the lock needed for synchronization.</td>
</tr>
</tbody>
</table>

Table 2: Description of queue operations

The enqueue and dequeue methods use locks implemented using the test-and-set registers for synchronization. Note that there are 2 locks on each tile. The enqueue and dequeue methods use different locks for their operations. This gives way to some parallelism as the enqueue and dequeue operations can now occur concurrently.

The attribute place could be either a tile or the off-chip shared DRAM. Whenever a worker needs to insert a task on a particular queue, it will call the enqueue operation on that queue. Whenever a worker wants to steal or take a new task, it will call the dequeue operation on that queue. Synchronization of the queue located on a tile is achieved by using the locks on the same tile.
Synchronization of the queue located on the off-chip DRAM is achieved by using a lock on one of the tile.

The C-code for the above operations is given at the Appendix.

5 Experimental Results

5.1 Work Stealing Vs. Work Sharing

In this section we measure the performance of work stealing and work sharing in our implementation. A simple test application has been written using the async and place constructs in a for loop. Table 3 shows the pseudo code for the master and worker.

```c
// Master code
// enqueues the "work" onto the
// queue at the given "place"
for (int i = 0; i < ITERATIONS; i++) {
    async_place(place, 1);
    // place = i%24 in work stealing
    // = 8 in work sharing
}
// implicit finish

void async_place(int place, int function) {
    async_node work;
    ..................
    work.func = function;
    ..................
    enqueue(place, work);
}

// Worker code
// Under work stealing:
// Dequeues "work" from its own queue.
// if its queue is empty, it tries to
// grab "work" from a random queue.
// Under work sharing:
// Dequeues "work" from the master's queue.
while (true) {
    work = dequeue();
    if (work.func == 1) {
        func_1();
    }
    void func_1() {
        for (int i = 0; i < rand() % RANGE; i++) {
            for (int j = 0; j < WORK; j++) {
                a = a + j;
            }
        }
    }
}
```

Table 3: Master and Worker pseudo code

Under the work stealing approach, a master thread enqueues an iteration(task) onto each tile in a round-robin fashion via the `async` and `place` construct. If a local queue is full, the master thread waits till it is not-full. Under the work sharing approach, a central work-sharing queue has been setup on the MPB of the master thread. Now the master thread enqueues tasks onto the MPB. Other worker processors try to dequeue work from the queue of the master thread. The master thread in both the cases is on tile 8. The choice has been made based on the geographical location of the tile. The 2 cores within a tile compete for the tasks enqueued. Under the work stealing policy, if the worker’s local queue becomes empty, it tries to dequeue from a random queue(place).

Table 4 gives the time taken under a high variation in work per task and a low variation in work per task. The work load per task has been varied with the help of the value `RANGE`. The value of `WORK` has been set to 5000 in both the cases.

For `RANGE = 100,000` in Table 4 we can see the time taken by work stealing and work sharing under a high variation of work per task. The work sharing policy performs better than the work stealing policy in this case. This is because the work sharing policy balances out the variation as the work is shared in a single location. Under the work stealing policy the idle processors must now waste cycles looking for tasks.
Lightweight Dynamic Task Creation and Scheduling on the SCC

Table 4: Work stealing and Work sharing under varied work per task

<table>
<thead>
<tr>
<th>RANGE = 100,000</th>
<th>Total Time(sec)</th>
<th>RANGE = 1,000</th>
<th>Total Time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITERATIONS</td>
<td>Work Sharing</td>
<td>Work Stealing</td>
<td>ITERATIONS</td>
</tr>
<tr>
<td>100</td>
<td>3.4</td>
<td>4.4</td>
<td>10,000</td>
</tr>
<tr>
<td>200</td>
<td>5.12</td>
<td>7.9</td>
<td>20,000</td>
</tr>
<tr>
<td>400</td>
<td>9.17</td>
<td>15.3</td>
<td>40,000</td>
</tr>
<tr>
<td>800</td>
<td>17.2</td>
<td>20</td>
<td>80,000</td>
</tr>
<tr>
<td>1,600</td>
<td>33</td>
<td>36</td>
<td>160,000</td>
</tr>
<tr>
<td>3,200</td>
<td>64.2</td>
<td>69.7</td>
<td>320,000</td>
</tr>
</tbody>
</table>

For RANGE = 1,000 in Table 4, we can see the time taken by work stealing and work sharing under a lower variation of work per task. The work sharing policy performs better than the work stealing policy. The difference is more prominent as the number of iterations increase. This is because the overhead due to work sharing (contention at a shared location) increases as the number of iterations increase.

6 Conclusion

Extracting and scheduling dynamic task parallelism is important for future multicore systems. Programming languages like Habanero Java support dynamic task parallelism [2]. It is important to build a runtime framework to support these languages on new architectures like the SCC. The uniqueness of SCC is that, it is a hybrid of a distributed architecture and a shared memory system. One can now gain the advantages of both the architectures. SCC also offers fine-grain power management. These features make SCC an interesting platform for dynamic task parallelism.

7 Future Work

- Automatic place inference. The idea is to allow the compiler/runtime to automatically infer the place. An interesting runtime implementation is to start from a single work sharing queue. Based on the work load, the number of work sharing queues increases dynamically and the system moves from a work sharing mode to a work stealing mode.

- Dynamic power management. If the system load is low, it would be useful to switch off some of the cores or reduce the frequency. This saves power. An interesting research direction would be to explore power management using both hardware and software support.

References


Appendix

The following are the functions related to a non-blocking lock.

```c
int read_TAS(int tile, int core) {
    /*
    Reads the value of a test-and-set register of the 'core' on the given 'tile'.
    Every tile has 2 cores (core−0, core−1). Each core has a test-and-set register.
    */
    if (core==0)
        return *(unsigned int*)(CRBMappedAddr[tile]+CORE0_TESTSET);
    else if (core==1)
        return *(unsigned int*)(CRBMappedAddr[tile]+CORE1_TESTSET);
    else // invalid core value
        return 0;
}

void write_TAS(int tile, int core, int value) {
    // Writes a 'value' to the test-and-set register of the 'core' on the given 'tile'.
    if (core==0)
        *(unsigned int*)(CRBMappedAddr[tile]+CORE0_TESTSET)=value;
    else
        *(unsigned int*)(CRBMappedAddr[tile]+CORE1_TESTSET)=value;
}

int lock(int tile, int core) {
    /*
    A processor on a successful read to a test-and-set register reads a 1, else a 0 is read.
    Only one processor succeeds in reading the register.
    */
    if (read_TAS(tile, core)==1) {
        return 1; // acquired a lock successfully
    }
    return 0; // failed to acquire a lock
}

void unlock(int tile, int core) {
    // A value 0 is usually written to reset the test-and-set register.
    write_TAS(tile, core, 0);
}
```
The following are the queue operations.

```c
void set_top(int tile, int value){
    // sets the value of 'top' on the given 'tile' to 'value'
    *(unsigned int*)(MPBMappedAddr[tile]+TOP_OFFSET)=value;
}

int read_top(int tile){
    // reads the value of 'top' on the given 'tile'.
    return *(unsigned int*)(MPBMappedAddr[tile]+TOP_OFFSET);
}

void set_bottom(int tile, int value){
    // sets the value of 'bottom' on the given 'tile' to 'value'
    *(unsigned int*)(MPBMappedAddr[tile]+BOTTOM_OFFSET)=value;
}

int read_bottom(int tile){
    // reads the value of 'bottom' on the given 'tile'
    return *(unsigned int*)(MPBMappedAddr[tile]+BOTTOM_OFFSET);
}

int enqueue_place(int place, int tile, async_node *node){
    // use core-0 task to compete
    int top;
    int bottom;
    int result;

    result=lock(tile,0);
    if(result == 0){ // lock acquire failed
        return 0;
    }

    top=read_top(place);
    bottom=read_bottom(place);

    if(top >= bottom+SIZE_QUEUE){ // queue is full. cannot enqueue
        unlock(tile,0);
        return 1;
    }

    // insert node
    if(IS TILE(place)) // if the place is a tile
        memcpy(((async_node*)(MPBMappedAddr[place]))+(top)%SIZE_QUEUE, node, sizeof(async_node));
    else // enqueue on shared queue place on the off-chip DRAM
        memcpy(((async_node*)(MPBMappedAddr[place]))+(top)%SHM_SIZE_QUEUE, node, sizeof(async_node));

    set_top(place, top+1); // increment top;
    unlock(tile,0);
    return 2;
}
```
The enqueue/dequeue operation uses the `place` parameter to locate the queue and the `tile` parameter to locate the lock needed for synchronization. The parameter `node` has the task details.

```c
int dequeue(int place, int tile, async_node *node){
    // use core-1 task to compete
    int top;
    int bottom;
    int result;
    result=lock(tile,1);
    if(result==0){ // lock acquire failed
        return 0;
    }
    top=read_top(place);
    bottom=read_bottom(place);
    if(top==bottom){ // queue is empty: cannot dequeue
        unlock(tile,1);
        return 1;
    }
    // delete node
    if(IS_TILE(place))
        memcpy(node,(async_node *)(MPBMappingAddr[place])+(bottom)%SIZE_QUEUE,sizeof(async_node));
    else
        memcpy(node,(async_node *)(MPBMappingAddr[place])+(bottom)%SHM_SIZE_QUEUE,sizeof(async_node));
    set_bottom(place,bottom+1); // increment bottom
    unlock(tile,1);
    return 2;
}
```

The async node definition is as follows

```c
typedef struct async_n{
    int function;
    int label;
    void *input;
    void *output;
}async_node;
```
Static analysis and refactoring towards Erlang multicore programming

Melinda Tóth
Eötvös Loránd University
Budapest, Hungary
toth_m@inf.elte.hu

István Bozó
Eötvös Loránd University
Budapest, Hungary
bozo_i@inf.elte.hu

Zoltán Horváth
Eötvös Loránd University
Budapest, Hungary
hz@inf.elte.hu

Atilla Erdődi
Eötvös Loránd University
Budapest, Hungary
erdodi@elte.hu

Abstract

Most of the processors deploy multiple cores to achieve power efficiency and high performance, but even in case of functional languages supporting parallel and distributed evaluation, program need to be structured in an appropriate way to take the advantages of multicore processors. Our research focuses on determining possible parallelisation of programs written in a dynamically typed functional programming language, Erlang, which has special language elements, library support and scheduler logic for parallel execution. A high abstraction level of program representation can be used to determine those program parts, that can be computed in parallel in an efficient way and to identify bottlenecks. There are some factors that may influence the parallel evaluation of expressions or sequences of expressions. The main factors are the control and data dependencies. For this purpose we use dependency graphs calculated statically from the control and data dependencies of Erlang programs.

1 Introduction

A number of legacy program have not been structured to run in parallel, thus we do not achieve any performance improvement by evaluating them on multicore architectures. Various approaches have been proposed to improve performance for sequential programs on multiple core systems, ours among them focuses on Erlang programs. The Erlang virtual machine (VM) has had a symmetric multiprocessing (SMP) support since 2006, so we try to determine those program parts that can be execute in parallel efficiently in the Erlang VM.

The first industrial success with using SMP in Erlang systems was Ericsson’s Telephony Gateway Controller [22]. Migrating the project into a dual-core processor produced an impressive 1.7x increase in performance. The migration took less than one man-year including the testing, which is a very short time for a project of this complexity.

Our goal is to provide tool support for developers to further reduce the time necessary for migrating legacy applications to multicore systems.

To identify the various parallelisable components in the program we have to identify both control and data dependencies between the expressions of the program. That can be easily represented using a directed dependency graph (DG), where the Erlang expressions are represented as nodes and dependencies among the expressions as directed edges. Using the DG we can identify the parallelisable components by computing strongly connected components [17] in the graph.

The paper is structured as follows. Section 2 provides a description of Erlang multicore. Section 3 introduces the dependency graphs of Erlang programs and describes special dependencies among Erlang

*Supported by TECH_08_A2-SZOMIN08, ELTE IKKK and Ericsson Hungary
expressions. Section 4 shows an example to demonstrate parallelisation. Finally, Section 5 describes related work, and Section 6 is a conclusion.

2 Introduction to Erlang multicore

Erlang is a general purpose functional programming language originally developed by Ericsson to develop fault tolerant telecom applications. Its dynamic type system allows rapid prototyping and hot code swapping. Processes in Erlang are lightweight and handled by the Virtual Machine.

The fist prototype for a multi-threaded Erlang/VM implementation was presented by Pekka Hedquist in his Master’s thesis in 1998 [9]. Since there was not demand for multicore support in the industry at the time, the work for multicore implementation was only restored in 2005. Erlang/OTP R11B with official SMP support was released in May 2006 [12].

In the current implementation (R14B) each scheduler runs on one operating system level thread with an own run queue. Load balancing among the schedulers is done using a migration logic [13].

Due to Erlang’s lightweight processes and message passing concurrency model, software developed in Erlang for single core processors generally scale well on multicore processors. However there are several factors to consider on migration.

There might be bottleneck processes in the system that should be identified and removed. When a single process provides some sort of functionality used by many other processes, the increasing number of cores also increases the workload of the singleton process. This is not just causing a performance hit on the system but the whole system can crash after a while when a process cannot handle the requests and its mailbox become flooded.

Erlang comes with a built-in term storage called Erlang Term Storage (ETS). Arbitrary Erlang terms can be stored in ETS tables implemented as hash tables. Tables can be set to public and therefore be accessed simultaneously by multiple processes. In a sense this is a form of shared memory and introduces the known related problems. ETS tables are heavily used by most Erlang applications.

All shared data structure is protected by locks. If lock-conflicts occur too often during the operation of the system, it causes performance degradation. Recent improvements in the Erlang VM reduced this overhead by fine grade locking, but the programmer still needs to take the effect of lock-conflicts into account.

When splitting a process into multiple processes, race conditions can also be introduced, which might be hidden on a single core machine due to deterministic scheduling, but occurs on true multiprocess systems [22]. Dialyzer, a lightweight static analyser tool for Erlang, can detect some of the introduced race conditions [2].

3 Dependency graphs for Erlang

To calculate the dependency and the necessary data-flow and control-flow information we use the Semantic Program Graph (SPG) of RefactorErl [10, 11]. RefactorErl is a source code transformer and static analyser tool for Erlang. The latest release of the tool contains 24 refactoring transformations and different facilities to support program maintenance tasks. RefactorErl introduces the SPG to represent the source code. The SPG is a rooted, directed graph and stores the lexical, syntactic and semantic information in three layers. The information retrieval is efficient from the graph, therefore using the SPG to calculate the dependency and flow information is more efficient than calculating them from the source code.

We have defined both Data-Flow Graph (DFG) [21] and Control-Flow Graph (CFG) [20] for Erlang that consist of compositional graph building rules based on the syntax and semantics of Erlang language.
The data- and control-flow graph build upon the SPG. The DFG is part of the SPG. The direct data-flow edges are added to the SPG as additional semantic information (data-flow information is also needed for data related refactorings, such as introducing records instead of tuple function arguments) and it is updated incrementally after a transformation performed on the source code. The control-flow edges are not added to the SPG (except of the parallelising analysis RefactorErl does not include refactorings that require control-flow information) and it is built only on request. We use the same graph identifiers, thus the mapping between the CFG and the SPG is straightforward.

The Dependency Graph (DG) contains data and control dependencies calculated from the DFG and CFG. To build the control dependency graph we calculate the CFG at first. From the CFG we can build a Post Dominator Tree (PDT) \[16\]. The Control Dependency Graph (CDG) can be built from the CFG and the PDT. The CDG eliminates the unnecessary sequencing from the control flow graph, thus only the relevant control dependencies are stored in the CDG. That makes possible to efficiently find the parallelisable component of the graph. If we do not consider that data dependencies may occur among the selected components, then we have to synchronise them during the parallelisation. Thus we introduce a communication overhead and lose efficiency.

Therefore we build the DG containing both data and control dependency information. According to the nature of the multicore Erlang, we have to consider some kinds of hidden data dependencies. This dependency information is calculated form the DFG and SPG. For instance, when two or more expressions read data from the same ETS table, we can run them in parallel, but it could became a bottleneck of the system. The number of parallel processes allowed to access the same table could be a design rule and a parameter of the parallelisation.

There are expressions or expression sequences in Erlang programs which can not be evaluated in parallel (or the necessary synchronisation among them is too costly). Among these statements are data or control dependencies. Some of them are easily detectable, for example the head of a case expression can not be evaluated in parallel with the body of the case expression, because the executed branch of the case depends on the return value of the head. A more Erlang specific example is that we can not run in parallel two expressions which can read and write the same ETS table. These kinds of data dependencies are detectable in case of named tables, otherwise we need process identifier (Pid) analysis.

The candidates for parallel execution are the strongly connected components in the dependency graph, but further analysis is needed to decide whether the result is appropriate and at the same time efficient. If the components become large, an iterative splitting of the component could produce a more applicable result.

### 3.1 Detecting special dependencies

The usage of ETS tables is a sort of hidden dependency that have to be considered. We should group those statements from the program into one parallel component that use data from the same ETS table or tables. That can reduce the number of possibly lock-conflict in the Erlang VM. Therefore, when two expressions use the same ETS table we should list them into one parallel component (see an example in Section 4). This kind of dependency is reachable from the semantic program graph using a data-flow reaching. Figure 1 shows the draft of that query. After querying the used ETS table names from an expression the result should contain variables or other expressions that stores the value of the table name. Thus we have to calculate the original value of them using a data-flow backward reaching and then we have to compare the values from the result. If there are no matching values in the resulted lists we assume that the used ETS tables are different.

An other type of dependency related to ETS tables was mentioned above, in the previous section. When program parts sequentially read and write the same ETS table we can not execute that parts in parallel – that could change the result of the program.
ets-dependency(Expr_1, Expr_2) ->
    ETS_1 = find_ets(Expr_1),
    ETS_2 = find_ets(Expr_2),
    case match(origin(ETS_1), origin(ETS_2)) of
        [] -> ok;
        _  -> make_dep(Expr_1, Expr_2)
    end.

match(List1, List2) ->
    [true || E1 <- List1, E2 <- List2,
        (E1 == E2) orelse (value(E1) == value(E2))].

Figure 1: Finding “ETS dependent” components

The difference between the two mentioned ETS related dependencies is:

- the latter mentioned dependency – reading and writing data from the same ETS table – is strict, we can not execute the dependent parts in parallel since we could introduce race conditions to our program.

- the former one – reading data from the same ETS table – is not very rigid. However we could execute them in parallel, but it could became a bottleneck of the system. The number of parallel processes allowed to use the same table can be a some kind of design rule and a parameter of the parallelisation.

Using the same Mnesia database tables or using the same file to store some kind of shared data results the same situation as we have with ETS tables. Fortunately they can be handled in a similar way, with the generalisation of the mentioned algorithm (Figure 1).

We can generalise the listed problems to the problem of side effects in the program to be parallelised. Side effect is one of the hidden data dependency we have to consider. A very strict rule could be: do not run those parts of the program in parallel that may have side effect, because of the hidden dependency between them. Hence reading some data from an ETS table has side effect we can not run these parts of the program in parallel, but we argued before that this is possible – maybe sometimes it is not efficient. This kind of side effect is a “good” side effect and should not indicate dependency among expressions. Therefore we introduce hidden dependency between two expressions if both have side effects and at least one of them is not “good” – and could introduce a race condition to the system.

An other type of race condition is the usage of the process registry in Erlang([2,3]). The mentioned problem is also detectable with the generalisation of our algorithm.

4 An example to demonstrate parallelisation

Consider the code of a singleton process with dummy function calls shown on Figure 2.

For each request the reply is generated by the functions, do_computation/0, do_computation_ets_1/1 and do_computation_ets_2/1. Lets assume that there is only one side effect present in do_computation_ets_1/1 and do_computation_ets_2/1, that is reading data form the ets table EtsTable. Besides do_computation/0 either does not have a side effect or have a side effect that is not related to EtsTable. In this case, the first two function calls and the third function call can be evaluated in parallel. Of course, this is only meaningful if the delegated functions are complex.
Parallelising Erlang Programs

Tóth, M., Bozó, I., Horváth, Z., Erdődi, A.

```erlang
loop(EtsTable) ->
    receive
        {Pid, Request} ->
            Reply = handle_request(Request),
            Pid ! {reply, handle_request(Request, EtsTable)}
    end,
    loop(EtsTable).

handle_request(Request, EtsTable) ->
    update_ets_table(Request, EtsTable),
    Data1 = do_computation_ets_1(EtsTable),
    Data2 = do_computation_ets_2(EtsTable)
    Data3 = do_computation(),
    ...
    {{Data1, Data2}, Data3}.

Figure 2: Example code of a simple server process

handle_request(Request, EtsTable) ->
    update_ets_table(Request, EtsTable),
    Data1_key = rpc:async_call(node(), ?MODULE, do_computation_ets, [EtsTable]),
    Data2_key = rpc:async_call(node(), ?MODULE, do_computation, []),
    ...
    Data12 = rpc:yield(Data1_key),
    Data3 = rpc:yield(Data2_key),
    {Data12, Data3}.

do_computation_ets(EtsTable) ->
    Data1 = do_computation_ets_1(EtsTable),
    Data2 = do_computation_ets_2(EtsTable),
    {Data1, Data2}.

Figure 3: Parallelised code of the request handling part

enough, that the gains by the parallel evaluations are higher than the overhead caused by process spawn
and message passing.

Using the Erlang module `rpc` for delegating evaluation to separate processes, the transformed code
is shown at Figure 3.

The scratch of the DG of the `handle_request/2` function is shown on Figure 4. The dotted edges
represent the first component, the dashed edge represents the second component.

Exceptions thrown by the delegated functions are caught by the helper functions and returned by
`rpc:yield/1`. Therefore the semantics of the code two code snippets are not identical, which should be
considered by the developer [22].

5
5 Related work

Dependency graphs are originally designed and used in compilers to prevent statement execution in wrong order, i.e. the order that changes the meaning of the program [16].

Nowadays the usage of different dependency graphs is a commonly used technology in different software engineering tasks (in program understanding, maintenance, debugging, testing, differencing, specialisation, re-engineering, optimisation, parallelisation, anomaly detection, etc).

The parallelisation problem is a special usage of the dependency graphs. This kind of optimisation is studied in different point of view. The paper [5] presents a dependency representation useful in an optimising compiler for a vector or parallel machine. Methods for parallelisation of Prolog programs and related problems are presented in [19] and [8].


In case of functional languages, most of the researches concentrated on explicit parallelism and its efficient implementation and execution on multicore systems [14, 6, 1].

The paper [7] introduces a data parallel, functional array processing language SAC and its concept of generic, compositional array programming. The paper presents some tasks in compiling the SAC code into efficient code for modern multicore processors.

Efficiency in case of running Erlang programs on multicore systems also studied [23]. The paper [3] describes race condition detection according to the nature of the Erlang programs running on multicore systems.

6 Conclusions and future work

A number of legacy Erlang program have not been structured to run in parallel, thus they can not run efficiently on multicore architectures. Migrating an application manually is a time consuming process. Our goal is to provide a tool that helps the developers to determine the parallelisable components of the system, thus they can migrate legacy Erlang applications to efficiently executable programs on multicore systems.

In our dependency graph representation we try to consider the special language elements and library support of Erlang. While we select the parallelisable parts of the program we also try to consider the scheduler logic for parallel execution of the Erlang runtime system.

The mentioned static analysis allows the programmers to transform the sequential source code to an explicitly parallel code in a way, that it should be performed by a compiler in a language that supports
implicit parallelism. With such an analyser tool, the programmer can supervise the suggested interpretation of the sequential source code into a parallel code, thus make the debugging easier and avoid arbitrary decisions of the compiler that can result in a less optimal code.

As RefactorErl is also a source code transformer, we want to implement automatic transformation of the source code according to the result of the studied static analysis. Total semantic equivalence however will not be possible due to the limitations of static analysis. Only a limited subset of side effects could be handled and the transformation will not preserve exception semantics. Therefore interaction with the developer will be necessary to decide the applicability of the refactoring.

The transformation also have to take into account the available configuration of the system (how many cores are available) and the possible number of parallel components. Based on this, we should transform the source code adaptively to the different configuration of the systems.

References


Abstract

Shared resources are a feature of many concurrent distributed systems. Access to these resources often involves using data of different types at each access. We consider the use of static analysis to guarantee type safety in such such systems and provide a view of generalised resource usage which subsumes that of session typing systems.

1 Introduction

In many real world systems we often need to be able to obtain, and use type-safely, values from shared resources; such systems include, for example, multithreaded message passing systems, where the shared resources are channels with associated queues, or those where mobile code is transmitted between locations. When the type or the effect of these values is variable, say for untyped channels in a message passing system, it is possible to send a value which is not of the type that the receiver is expecting. If the receiver does not dynamically check the type of the received value itself it could then use the value inappropriately and cause a run time type error.

Traditionally many systems solve this problem by performing runtime checks on all values received. This is true, for example, for deserialisation of objects in Java. This approach has obvious runtime costs and in general one might seek to mitigate these costs through static analysis. There are different static approaches which can be applied to this problem; we can use model checking to examine the types of all possible values returned, and in some cases we can use structural analyses of the impure actions such as sending messages. In the specific case of message passing systems a number of static analyses known as session typing [2–6] have been developed to address this issue. These approaches leverage the semantics of the message passing system, such as blocking receives and FIFO queues for sent messages, to observe that if two threads act on a channel and their actions are complementary [4] then there will be no run time type errors. We argue that the session typing analysis exemplifies a more general analysis of type and effects for resource accesses with variable types. The specific details of message passing systems can be factored out into a semantics of resource access so that the underlying mechanics of the type safety proof are independent of the particular resources being accessed. Doing this has the advantages of making the proof of type safety robust with respect to changes in the resource model being used, thereby allowing one to obtain the guarantees of type safety without having to reprove the result.

In this paper, following [7], we present a generalised notion of resource with variably typed access, discuss static analysis for these and state the key properties of subject reduction and fidelity for this analysis. We show how this general model can be instantiated to yield session type analyses for message-passing systems. Crucially we separate the aspects of the safety proof which can be done for general systems from those which are dependent on the semantics of accessing the resources (Section 4). The general solution to proving safety is a model checking approach (for which we do not consider the algorithms). We show how to prove safety in various examples, and how the additional knowledge allows us to simplify and reduce the cost of the proofs.
2 Message Passing Systems

In this section we consider message passing systems as a means to discussing our generalised notion of resource. Message passing systems consist of multiple threads of code that uses shared channels. Accesses to shared channels can add and remove values of varying types from channel queues.

In message passing systems type usage errors can occur when the value obtained from a receive action is not of the expected type, for example consider the abstracted code effects in Figure 1 which uses a suggestive notation $c \mapsto T_1, \ldots, T_k$ for the type of a channel queue and $c!T$ for code which sends a value of type $T$ on channel $c$ and $c?T$ for code which expects to receive a value of type $T$.

Session typing analyses are static analyses that can be used to determine whether a program in a message passing system with untyped channels will have any type errors due to the values sent and received on the channels. In essence this is done as follows: perform a type and effect analysis on the code to obtain an effect which represents the behaviour of the system on resources and then define some predicate over these effects that exploits knowledge of the semantics of resources to ensure that no type errors occur. For example, for systems in which it is known that only two threads use each channel; one thread for sending only and one for receiving only, then we could define a predicate that asserts that for each receive action in a given thread there is an entry at the front of queue of the correct type or the queue is empty and there is a complementary send of the same type in the other thread. This will then catch the error in Figure 1 as the value sent is not of the type expected by the receiver. Note that whether or not a type error occurs may depends on the starting state of the resources. For example, with the same code and different starting resources, as in Figure 2, no error occurs. The predicate described above accounts for this. If we could not rely on channels only being used by two threads, or that the threads use the channel to send or receive only, then the effects being complementary would not be a sufficient condition to prevent errors; in Figure 3 the effect of each of the senders is complementary to that of the receiver, but as there are multiple senders the sends can interleave resulting in a value of an unexpected type being received.

What we see from the example above is that in checking type safety of code that accesses resources the relevant information is

• the code’s effect on the resources,

• the initial state of the resources, and

• semantics of the resources within the system

The first of these can be calculated using a standard type and effect system [8]. The latter two inform our generalised analysis.


\section{Generalised System}

We now present our generalised resource system and describe the type and effect analysis used to guarantee type safety. We will work with a simply-typed lambda calculus with recursion and primitives \texttt{acc} to describe resource access. The labels \( l \) are of the form \( \alpha(v_1, \ldots, v_n) \) for some \textit{action} \( \alpha \) and values \( v_i \). We will consider systems \( P \) of parallel threads of these lambda terms. We parameterise our work on a \textit{resource model}, which is a collection of \textit{states} ranged over by \( \sigma \) and a function that maps a state and a label to a return value and another state: we write \( \sigma(l) = v \) and \( \sigma \xrightarrow{l} \sigma' \) to denote this map. The semantics of systems is given with respect to a resource state: \( [\sigma]P \rightarrow [\sigma']P' \) in a mostly obvious way such that a resource access \( \texttt{acc}^l \) respects the semantics of the resource model:

\[
\begin{align*}
\sigma & \xrightarrow{l} \sigma' \\
[\sigma]\texttt{acc}^l & \rightarrow [\sigma'][\sigma(l)]
\end{align*}
\]

The resource model is a generalised model, and hence can be instantiated with either finite or infinite models. In order to present our analysis we make use of an \textit{abstract resource model} which abstracts away from particular values associated with resource states and provides a representation of the resource types. An abstract resource model is a collection of abstract states ranged over by \( \Sigma \) and a function that maps a state and an abstract label to a type and another abstract state: we write \( \Sigma(L) = T \) and \( \Sigma \xrightarrow{L} \Sigma' \) as above. Abstract labels \( L \) are of the form \( \alpha(T_1, \ldots, T_n) \) so that a label \( l \) corresponds to a unique abstract label \( L \) via simple typing of values. We relate the resource model and abstract resource model via an abstraction map \( \Lambda \) with the property that \( \sigma \xrightarrow{l} \sigma' \) implies \( \Lambda(\sigma) \xrightarrow{L} \Lambda(\sigma') \) and \( \Gamma \vdash \sigma(l) : \Lambda(\sigma)(L) \), under type variable assumptions \( \Gamma \).

We would now like to define our type and effect system in a standard way following \cite{8} where the judgements are of the form \( \varphi; \Gamma \vdash t : T \) for effects \( \varphi \) and simple type \( T \). However, as we are working in a system in which resource accesses may return values of different types depending on the state of the resource it is not generally possible to locally determine a type for the resource access primitive \texttt{acc}^l. Indeed, the type of this expression will depend on the state in which it is accessed. What we can do locally is determine the \textit{expected} type of the \texttt{acc}^l expression with respect to the remainder of the thread in which the value returned is used. This generates a constraint on the effect for that thread which will be solved globally when all threads are placed together in parallel. In order to do this, the effect of a primitive access \texttt{acc}^l takes the form \((L, T)\) where \( L \) is the corresponding abstract label and \( T \) records the expected type of the value to be returned.

\[
\Gamma \vdash \texttt{acc}^l : T
\]

Our language of effects is

\[
\varphi ::= \varepsilon \mid x \mid (L, T) \mid \varphi; \varphi \mid \mu x. \varphi \quad \Phi ::= \varphi \mid \Phi \parallel \Phi
\]

where we allow sequencing of effects and recursive effects (infinitely unfolded). Given an effect \( \varphi \), it is easy to see how to lift the reduction relation over states and threads to abstract states and effects \( \Sigma(\varphi) \rightarrow \Sigma'(\varphi') \) by making use of the \( \Sigma \xrightarrow{L} \Sigma' \) relation. We add error transitions to this as follows

\[
\Sigma(L) = T \quad T \neq T' \rightarrow [\Sigma][(L, T')] \varphi \rightarrow \text{error}
\]

We define the judgement \( \vdash P : \Phi \) using a type and effect system in the usual way modulo the rule for \texttt{acc}^l expressions as discussed above. This judgement says that the system consists of well-typed threads
whose effects make up $\Phi$. Our global check on the consistency of the local constraints in the effects is expressed in the rule:

$$
\vdash P : \Phi \quad \exists \mathcal{C} \in (\Lambda(\sigma), \Phi) \text{ and } (\mathcal{C} \implies \text{compGen})
$$

$$
\vdash [\sigma]P : \Phi
$$

where

$$
\text{int}(\varphi_1 \parallel \ldots \parallel \varphi_n) \overset{\text{def}}{=} \bigcup_{i=1}^{n} \{(l, T); \varphi'_1 = (l, T); \varphi'_i \land \varphi' \in \text{int}(\varphi_1 \parallel \ldots \parallel \varphi'_i \parallel \ldots \parallel \varphi_n, )\}
$$

$$
\text{compGen}(\varphi_1 \parallel \ldots \parallel \varphi_n, \Sigma) \overset{\text{def}}{=} \forall \varphi \in \text{int}(\varphi_1 \parallel \ldots \parallel \varphi_n). [\Sigma] \varphi \not\rightarrow^* \{] \text{error}
$$

In order to guarantee type safety we need to verify that each resource access returns a value of the expected type annotated on the effect of that resource access. This must be true irrespective of the interleaving of accesses which may occur before it. In the worst case we must look at all possible interleavings of a parallel effect and ensure that none of these reduce to an error under the resource reduction semantics. However, it may be that we can find a predicate over abstract states and effects that implies this property. We discuss this point further in the next section but for now we state our main properties of the generalised analysis:

**Theorem 3.1.** (Subject Reduction and Fidelity) If $\vdash [\sigma]P : \Phi$ and $[\sigma]P \rightarrow [\sigma']P'$ then there exists some $\Phi'$ such that $\vdash [\sigma']P : \Phi'$ and $[\Lambda(\sigma)]\Phi \rightarrow [\Lambda(\sigma')]\Phi'$

### 4 Invariability Proofs

The $\text{compGen}(\Sigma, \Phi)$ predicate is a formal statement of the safety of possible interleavings of the effects on the resources $\Sigma$. Establishing $\text{compGen}$ amounts to model checking the space of all traces on $[\Sigma]\Phi$. Whilst this approach has exponential complexity, it is a general solution where we have no additional knowledge about the structure of the resources, the permissible accesses, or the reduction semantics of the resources. Given more information about the resource’s reduction relation then we may be able to define some other check which is much easier to establish but still implies $\text{compGen}$ in that system. We illustrate this with the following examples.

**Blocking Message Passing.** In order to define a less costly check than model checking we consider a resource model of shared channels and channel queues with resource accesses $c!()$ and $c?()$. It is easy to define an appropriate resource model and a corresponding abstract resource model for these: the states represent the current state of the channel queues and the actions move values in and out of these queues. The receive action has a blocking semantics so it has no action on an empty queue but will return the value at the head of the queue otherwise. The send action always returns a unit value. Suppose we know that each channel is shared between at most two threads and that threads use channels uni-directionally. Let us also suppose that systems always start with empty queues. Using this information we can define a complementary relation $[4]$:

**Definition 4.1.** Complementary relation for blocking message passing systems:

$$
\text{compl}((c!(), T'); \varphi_1, (c?(), T''); \varphi_2) \overset{\text{def}}{=} T' = \text{Unit} \land T = T'' \land \text{compl}(\varphi_1, \varphi_2)
$$

$$
\text{compl}(\varphi_1, \varphi_2) = \varphi_1 = \varepsilon \lor \varphi_2 = \varepsilon
$$

Given this we define compatibility as:
Definition 4.2. Compatibility for blocking message passing systems:

\[
\text{compatible}(\Sigma, \Phi) \overset{\text{def}}{=} \forall c. \text{compl}(\varphi_1 \mid c, \varphi_2 \mid c) \lor \text{compl}(\varphi_2 \mid c, \varphi_1 \mid c)
\]

where \( \varphi \mid c \) projects the effect \( \varphi \) to just the actions using channel \( c \) and \( \varphi_1 \) and \( \varphi_2 \) are the effects of the two threads that use \( c \).

Given this definition it is not too hard to show that \( \text{compatible}(\Sigma_e, \Phi) \) implies \( \text{compGen}(\Sigma_e, \Phi) \) where \( \Sigma_e \) is the abstract state representing empty channel queues. We can use this predicate in establishing that blocking message passing systems are well-typed, and hence are type safe, from the initial resource state. Subject reduction tells us that well-typed systems stay safe under reduction. This check is linear in the size of the effects which use the channel.

We sketch the proof that \( \text{compatible}(\Sigma_e, \Phi) \) implies \( \text{compGen}(\Sigma_e, \Phi) \) as follows. We know that only two threads perform accesses using that channel. In combination with our knowledge from the semantics that only actions using a channel can modify that channel’s queue we can consider only the possible interweavings of the actions of these two threads which make use of a given channel (\( \varphi_1, c \) and \( \varphi_2, c \) respectively). If the usage is uni-directional, then irrespective of the interweavings the latter can only perform receive actions after the former performs a send action, as receive is a blocking action. In combination with FIFO access semantics then the receiver will receive values in the order that they are sent by the sender. This is true irrespective of whether all the sends are performed first and then all the receives, whether the sends and receives interchange, or any other interweaving. Hence we preserve compatibility.

Non-Blocking Message Passing. A slightly different example is a message passing system where the receive actions are non-blocking. This uses the same resource and abstract models as blocking message passing, with one exception; the receive action has a non-blocking semantics where it returns a default (unit) value when performed on an empty queue and the head of the queue otherwise. We again suppose that two threads use a given channel uni-directionally and that we start with empty queues. Then the complementary behaviour to sending a value is polling a channel until a non-default value is received. We can again show that \( \text{compatible}(\Sigma_e, \Phi) \) implies \( \text{compGen}(\Sigma_e, \Phi) \). As before we can only need to consider the effect of the two threads which are communicating uni-directionally using the channel. In order to guarantee that in the case of receiving a unit value the receiver doesn’t go on to performing the next receive in its sequence we require it to perform polling and wait to receive the non-default value.
Hence the receiver will receive all the values in the order sent. This may seem like a re-engineering of blocking receive, but recall that we are working with effects projected onto a specific channel; the receiver could go off and do some other behaviour on other channels before looping round and this would be invisible when performing the complementary check for this channel.

**Bounded message passing** To demonstrate the robustness of our technique we consider a resource model with a different semantics. In this case, the $c!\langle v \rangle$ action will block if the channel queue is full with $K$ messages waiting in it. The semantics for this are easy to define. The changes we need to make to our analysis lie in the $\text{compl}(\phi_1, \phi_2)$ predicate given above. In this case we simply introduce a counter in the definition clause for $\text{compl}((c!\langle T \rangle, T'); \phi_1, \varepsilon)$ so that the predicate becomes true when the counter reaches $K$. Again, it is easy to show that this predicate implies $\text{compGen}$ and therefore may be used in establishing type-safety of such systems. In particular, we do not need to reprove subject reduction in this case.

## 5 Conclusion

In this paper we present a general resource model for resource accesses that return values of varying type. We provide a type and effect system for a multithreaded simply typed lambda calculus that features locally inferred return types as constraints in the effect annotations. These constraints are resolved globally between the multiple threads of the system in worst case by considering all possible interleavings of effects upon the resource state. This is tantamount to model checking and we provide examples in which we show that, given specific knowledge of the resource semantics, it is possible to define predicates that are easier to establish than the full model check yet nonetheless are sound. We believe that this approach clarifies the link between the semantics of the resources and the more routine aspects of the type safety proof and moreover allows reuse of the subject reduction and fidelity results for the general system.

The technical details of this work are included in our technical report [1] where we also include details of how to handle internal and external choice. In future work we intend to explore dynamic software updating for such systems and how the semantics of specific systems can inform safety proofs in the same way that the semantics informs compatibility definitions.

## References


1 Introduction

When computing in a collaborative environment the first challenge is to discover another process with which meaningful communication is possible. In concurrent bondi computation, discovery, and communication are driven by patterns. Computation is achieved through pattern matching described by pattern calculus [JK09, Jay09], while discovery and communication are achieved through pattern unification described by concurrent pattern calculus [GGJ10]. This abstract illustrates the approach through code [GWJ11] that addresses the problem of shopping for a term deposit (the goods) from a variety of different banks. These banks not only use different shapes of data, but completely different paradigms for programming. We hope that the novelty of the approach will be appreciated but have no space to consider other approaches here.

In the sequential setting, pattern matching is defined by the rule

\[(\theta p \rightarrow s) u \rightarrow \{u/\theta p\} s\]

where the pattern \(p\) is matched against the argument \(u\) of the function \(\theta p \rightarrow s\). When successful, the resulting substitution on the binders in \(\theta p\) is applied to the body \(s\). In the concurrent setting this is generalised to pattern unification with interaction defined by

\[(p \rightarrow P | q \rightarrow Q) \rightarrow (\sigma P) | (\rho Q) \quad \{p\|q\} = (\sigma, \rho)\]

if the patterns \(p\) and \(q\) can be unified to create substitutions \(\{p\|q\} = (\sigma, \rho)\) then these are applied to the respective processes \(P\) and \(Q\). In both settings the patterns describe the shape of the structure required for reduction or interaction.

This approach to computation and communication is able to exploit diverse and distributed systems, where parallel processing over various shapes of data is desirable, and communication is expensive.

The key point about this approach is that there is no need for a global shape to represent term deposits. It is enough that each bank write two lines of code: one to describe the shape of its term deposit data

\[\text{lin} \ \text{shapeNSW} \ b \ m \ r \ p = \text{TDADT} \ b \_ \ m \ r \ p \_\]

and one to create the distributed computing service

\[\text{cpc} \ \text{descNSW} \ tdDesc \ q \rightarrow \text{security} \ q \ \text{shapeNSW} \ nsw\]

Details of these code fragments will be explained in context.

The remainder of this abstract is structured as follows. Section 2 develops two banks with disparate programming paradigms and data representations. Section 3 considers the shopping problem in a sequential setting with known banks. Section 4 generalises this to the concurrent setting with the additional challenge of service discovery. Section 5 draws conclusions.
2 Term Deposits

The example is of banks that offer a similar product, namely term deposits. Two banks shall be introduced throughout the example: the NSW bank represents its data using algebraic data types (ADTs); and the Vic bank uses object-oriented classes. The development of the example will be given in concurrent bondi, a multi-paradigm language that supports imperative and functional programming in the style of OCaml [Cam11] and object-orientation in the style of Java [GJS05].

Bank accounts at NSW bank of type Account are of the form Acct n x where n is the name of the account (a string) and x is the balance of the account (a float):

datatype Account = Acct of String and Float

in the familiar functional programming style. Term deposits are declared similarly, with the bank name, product name, minimum deposit, rate, period and government guarantee.

datatype TermDepositADT =
    TDADT of String and String and Int and Float and Int and Bool

The NSW bank is represented by Bank with lists of accounts and term deposits.

datatype BankADT = BankADT of String and List Account and List TermDepositADT

with toString += | BankADT n _ _ -> n

A novel feature of bondi is the ability to dynamically add cases to existing functions. The default print function would print out all of the accounts and term deposits, so a special case is added (+=) which prints only the bank name.

The NSW bank containing two customer accounts and two term deposits is given by:

let acct1 = Acct "John Citizen" 2222.00;;
let acct2 = Acct "Jane Doe" 2736.30;;
let tdnsw1 = TDADT "NSW" "Standard TD" 1000 4.7 12 True;;
let tdnsw2 = TDADT "NSW" "Short and cheap TD" 500 3.3 1 False;;
let nsw = BankADT "NSW" [acct1,acct2] [tdnsw1,tdnsw2];;

The double semi-colon is used to end a declaration.

The Vic bank is implemented using mostly object-oriented classes, which are modelled upon those of Java. The term deposits are defined below, with a minimum deposit, rate and period.

class TermDepositOO {
    minDep: Int; (* Minimum deposit *)
    rate: Float; (* Rate *)
    period: Int; (* Period *)
    (* Get and set methods. *)
    with toString += | (x:TermDepositOO) -> "" }

Observe that in addition to using a different format, the Vic bank does not store the bank name within its term deposits. The Vic bank is also represented by a class:

class BankOO {
    name: String; (* Bank name *)
    accts: List Account; (* List of accounts, same ADT as NSW *)
    tds: List TermDepositOO; (* List of term deposit classes. *)
    (* Get and set methods. *)
    with toString += | (x:BankOO) -> x.getName() }

2
The Vic bank is created as follows.

```ocaml
let tdvic = new TermDepositOO;;
tdvic.setMinDep(2000);
tdvic.setRate(4.6);
tdvic.setPeriod(3);;
let vic = new BankOO;;
vic.setName("Vic");
vic.setAccts([acct2]);
vic.setTDs([tdvic]);;
```

## 3 Dynamic Patterns

The relevant attributes of term deposits are chosen to be the name of the bank, the minimum amount, rate of return, and period. Routine techniques are used to display this information:

```ocaml
let display b m r p = println ("Term deposit from " ^ b ^ 
    " with minimum $" ^ (toString m) ^ ", rate " ^ (toString r) ^ 
    "%", and period " ^ (toString p) ^ " months.")
```

Harder is to recognise a term deposit as such within a larger, arbitrary data structure, and to extract the relevant attributes. As the pattern (or shape) for a term deposit varies between banks this needs to be given by a *shape parameter* `shape` to the function.

```ocaml
let findTDs f = fun (shape: lin (String -> Int -> Float -> Int -> a)) ->
  iter (| {b,m,r,p} shape b m r p -> f b m r p
        | _ -> ()
```

The first argument `f` of `findTDs` is some program to accept the desired attributes of any term deposits found. The second argument `shape` is used to create the pattern `shape b m r p` which will match against term deposits. Note that `shape` accepts the binding symbols `b, m, r, p` as arguments and must be evaluated before matching can occur, i.e. the pattern is *dynamic*. Pattern parameters such as `shape` must be *linear*, to avoid duplication or elimination of pattern binders, as indicated by the keyword `lin` applied to its type. The `iter` function uses path polymorphism [Jay09] to traverse any data structure applying the anonymous function that matches term deposits and applies `f`.

Finding and displaying term deposits is now as simple as providing the shape and bank as arguments to `findTDs display`. Create an ADT for banks that contains the shape and data.

```ocaml
datatype Bank = Bank of (lin (String -> Int -> Float -> Int -> a)) and b
```

Note that the types for the result of the pattern and for the data are existential here. Now

```ocaml
let allTDs = iter (| Bank shape bank -> findTDs display shape bank | _ -> ()
```

will display all term deposits found within any bank within any data structure containing, say, businesses. Observe that all this has been written before any actual shapes have been defined.

Now define the shape for term deposits in the NSW bank by

```ocaml
lin shapeNSW b m r p = TDADT b _ m r p _
```

Here `lin` indicates that a linear term is being declared, and the wildcards `_` are used to ignore the information that is not required. The arguments to the shape are the binding symbols that will be bound in the matching process. The shape for the Vic bank uses a helper function and fills in the missing bank name with an explicit value.
let getTDVicData (x:TermDepositOO[a]) =
    ("Vic",x.getMinDep(),x.getRate(),x.getPeriod());
lin shapeVic b m r p = view(getTDVicData,(b,m,r,p)) as (_:TermDepositOO);;

The pattern view(getTDVicData,(b,m,r,p)) as (_:TermDepositOO) successfully matches any
object of the class TermDepositOO and then matches (b,m,r,p) against getTDVicData of that object.

Now combine the NSW and Vic banks into a single data structure

let banks = [Bank shapeNSW nsw, Bank shapeVic vic]

-taking advantage of the existential types to remain type safe. Then evaluating allTDs banks yields

"Term deposit from NSW with minimum $1000, rate 4.7%, and period 12 months."
"Term deposit from NSW with minimum $500, rate 3.3%, and period 1 months."
"Term deposit from Vic with minimum $2000, rate 4.6%, and period 3 months."

showing that it finds and displays all the term deposits as desired.

4 Pattern unification

In the concurrent setting the main challenge is for the bank and customer to find each other and com-
municate. Discovering each other requires that they have some small amount of shared data, namely a
parameter that describes term deposits. Here tdDesc is a constant pattern to describe shared term deposit
data that is understood by both parties. For the rest, each party is free to create its own processes.

For example, the NSW bank process for term deposits will involve a pattern descNSW that provides a
bank description, and a function security that sandboxes the query, as well as the parameters shapeNSW
and nsw employed in Section 3. These are combined to form the process

cpc descNSW ~tdDesc \q -> security q shapeNSW nsw

As indicated by the keyword cpc, this declares a process with pattern descNSW ~tdDesc \q and body
security q shapeNSW nsw. Each component of the pattern behaves differently in unification. The
term deposit description tdDesc is protected by ~, in the sense that it must be matched exactly, i.e. the
other process must have the same pattern. The query binder \q inputs a customer query which is bound
to q in the body. The bank description is available for a mix of matching and output, that is the other
process may match the description exactly (to identify a specific bank), or input the bank name (to know
which bank they have interacted with). Note that the query is passed to the bank, whose internals remain
hidden from the customer. A similar process can describe term deposits at the Vic bank.

To interact with these services the customer must create a process that will unify with a bank’s term
deposit pattern. This requires accepting the bank’s description, being interested in term deposits, as
described, and providing a query, that consists of a query in the sense of Section 3 plus the machinery
to communicate its results back to the customer along a protected (i.e. private) channel session. For
example,

let query session = findTDs (fun b m r p -> cpc ~session b m r p -> ())

executes the familiar findTDs query on the banks data, communicates this back to the customer along
session and terminates. Note how the higher-order function findTDs has been applied to a function
that returns a process. The results are displayed using the process returned by the function

let displayFromSession session = !(cpc ~session \b \m \r \p -> display b m r p)
using replication $! \text{of the process}$ to accept any number of results along a private channel $\text{session}$ from the bank. Then


$! \text{rest session in}$

\begin{verbatim}
cpc \bank tdDesc (query session) -> (  
    println ("Querying " ^ \bank);  
    displayFromSession session)
\end{verbatim}

uses $\text{rest session}$ to restrict access to the channel $\text{session}$ (just as $(\nu \text{session})$ would in $\pi$-calculus), and then replicates the whole, so as to be able to interact with many banks. Summarising, when a bank is discovered by unification the service reports the bank that accepted the query and creates another process that collects all responses from that bank to display them.

Now running this process in parallel with the banks NSW and Vic yields

"Querying Vic Bank"
"Querying NSW Bank"

"Term deposit from NSW with minimum $500, rate 3.3\%, and period 1 months."
"Term deposit from Vic with minimum $2000, rate 4.6\%, and period 3 months."
"Term deposit from NSW with minimum $1000, rate 4.7\%, and period 12 months."

Observe that the queries were computed in parallel and the results displayed as they arrived with no relation to the structure of the banks’ data.

5 Conclusions & Future Work

This solution to the problem of finding term deposits in a collaborative environment shows how patterns can be used to drive computation and communication. Dynamic patterns support queries that adapt to the shape of heterogeneous data and programming styles. Further, pattern unification supports decentralised discovery of compatible processes with exchange of information, all within an atomic interaction.

Future work will continue the development of concurrent pattern calculus, and the integration of concurrent and sequential features in concurrent bondi. Further efforts will be required to apply the mechanisms illustrated here to large data sets, and in an open environment such as the Internet.

References


Abstract

The Event-B method is a formal approach for modelling systems in safety- and business-critical, domains. Initially, system specification takes place at a high level of abstraction; detail is added in refinement steps as the development proceeds toward implementation. Our aim has been to develop a novel approach for generating code, for concurrent programs, from Event-B. We formulated the approach so that it integrates well with the existing Event-B methodology and tools. In this paper we introduce a tasking extension for Event-B, with Tasking and Shared Machines. We make use of refinement, decomposition, and the extension, to structure projects for code generation for multi-tasking implementations. During the modelling phase decomposition is performed; decomposition reduces modelling complexity and makes proof more tractable. The decomposed models are then extended with sufficient information to enable generation of code. A task body describes a task’s behaviour, mainly using imperative, programming-like constructs. Task priority and life-cycle (periodic, triggered, etc.) are also specified, but timing aspects are not modelled formally. We provide tool support in order to validate the practical aspects of the approach.

1 Introduction

Event-B [3] can be used to model both single and multi-tasking software systems. The approach that we present here is greatly influenced by our previous experience [9, 10, 11] where we link Event-B with Java. We continue this section discussing our motivation. Section 2 provides an overview of Event-B. Section 3 describes the Tasking Event-B extension. Section 4 describes the preparation of a model for code generation. Section 5 describes the translation. In Section 6 we discuss our results and future work.

The work reported here has been undertaken as part of the EU DEPLOY [2] project, where our target domain is multi-tasking, real-time embedded systems. Our more general interest is in modelling concurrency in the application domain, with a view to automatic code generation. In previous work [9, 10, 11] we identified a problem with the large semantic gap between Event-B and the language that we used for specifying the implementation. We also encountered problems working with large models; our language introduced too much fine-grained atomicity, giving rise to a large number of proof obligations.

The undertaking, described here, was to develop an approach which integrates well with the existing Event-B methodology. One of our aims was to have just a few additions to the Event-B language; to have a small semantic gap between Event-B and the tasking specification. In essence we extend Event-B with just enough information to be able to derive an implementation. To address the problem of large model size we make use of Event-B’s decomposition approach [6, 19]. After decomposition a machine can be further refined, and decomposed again if necessary. At a suitable point we introduce implementation specific constructs to the development and generate the code. We also use the extended Event-B model to generate a model of the implementation. We have developed a demonstrator tool [22] to validate the approach; the tool integrates with the existing Rodin platform [23].
2 Event-B

The Event-B method \cite{abrial02} was developed by J.R. Abrial, and uses set-theory, predicate logic and refinement to model discrete systems. The basic structural features of Event-B are contexts and machines. Contexts are used to describe the static features of a system using sets and constants, and the relationships between them are specified in the axioms. Machines are used to describe the variable features of a system in the form of state variables and guarded events which update state; system properties are specified using the invariants clause. Machines are able to see Contexts; the contents of a Context is visible and accessible to a machine. The invariants give rise to proof obligations, which are generated automatically by the tool; a large number of the proof obligations may be discharged without user intervention by the provers. Where auto-provers fail to discharge proof obligations the user guides the interactive prover. They proceed by suggesting strategies, and sub-goals in the form of hypotheses, in the endeavour to complete the proof.

2.1 An Event-B Model

A fragment of an Event-B specification is shown in Fig. 1. The specification has a number of variable declarations which are typed in the invariant clause. Additional predicates are added to the invariants clause to describe the desired safety properties; the invariants clause consists of a conjunction of predicates. The write event has a single guard clause (but may have more) following the where keyword, and a number of actions following the then keyword. The guard is a predicate, over the sets, constants, and variables of the system, describing a condition under which an event may occur. Each event guard may have a number of guard clauses which are conjoined. Action clauses consist of assignments to variables, and may be non-deterministic. The event action is a parallel composition of the action clauses.

3 Tasking Event-B

3.1 Extending Event-B Machines

Tasking Event-B introduces a number of new concepts to facilitate code generation. The most significant is the extension of Event-B with two new types of machine, namely Tasking and Shared Machines. Tasking Machines are related to the concept of an Ada \cite{ada02} task (but we are not restricted to Ada implementations). Shared Machines are related to the concept of a protected resource, such as a monitor \cite{monitor02}.

Event-B machines without any additional additional features may be implemented; but we wish to implement the machines in a particular way which necessitates the use of information in addition to...
that available in a standard Event-B machine. We introduce specific tasking constructs to facilitate code generation for multi-tasking systems where the implemented tasks may have interleaving executions, but are restricted to communicate only with some protected resource in a mutually exclusive manner. The tasking features are an extension of standard Event-B machines, and machines to be implemented are characterised by one of the following attributes:

- **Auto Task Machine or Shared Machine**

Auto Tasks are tasks that will be declared and defined in the Main procedure of the implementation. The effect of this is that the Auto Tasks are created when the program first loads, and then activated (made ready to run) before the Main procedure body runs.

### 3.2 Tasking Specifics

#### 3.2.1 Task Scheduling

The following extensions relate only to Tasking Machines, and provide implementation details; but note that timing aspects of periodic tasks, and scheduling, is not modelled formally.

- **TaskType** and **Priority**

the TaskType construct is used to define the scheduling, cycle and lifetime of a task. i.e. one-shot, periodic or triggered. The period of a task is specified in milliseconds. Priority is an integer value; the task with the highest value priority takes precedence when being scheduled.

#### 3.2.2 Flow Control

Each Tasking Machine has a task body which contains the following flow control (algorithmic) constructs.

- **Sequence, Branch, Loop, EventSynch, Event**

The Sequence construct is used for imposing an order on events. Branch is choice between a number of mutually exclusive events. Loop specifies event repetition while its guard remains true. EventSynch is used to synchronize an event in a Tasking Machine with an event in a Shared Machine. The EventSync construct allows the updates in a Tasking Machine and Shared Machine to be viewed as an atomic update. Synchronization is implemented as a subroutine call, with atomic (with respect to an external viewer) updates. The updates in the protected resource are implemented by a procedure call to a protected object. It is important to note that tasks communicate through shared resources, and not directly with each other. The EventSync construct facilitates subroutine parameter declarations, and substitution in calls, by pairing ordered Event-B parameter declarations. The EventSync construct may also be used with a single event, in which case it is implemented as a subroutine call with no parameters. The event may belong to the Tasking Machine (local), a Shared Machine (remote), or both.

### 3.3 Events

Events can play one of several roles in the mapping to the implementation as follows,

- **ProcedureSynch, ProcedureDef, Branch, Loop**

Events with the ProcedureSynch extension can take part in event synchronization, whilst ProcedureDef indicates implementation as a parameterless subroutine call. The remainder are self explanatory.
3.4 Tasking Constructs

Events that are local to a Tasking Machine only update the Tasking Machine state; conversely events that are remote only update the state of the Shared Machine. Synchronised events share parameters to facilitate communication between Tasking and Shared Machines.

To represent the combined updates on local and remote machines we introduce synchronized event composition. The synchronization of the two events is equivalent to a single atomic event, with the guards and actions of the individual events merged. We can write the guards and actions of the events as guarded commands [8]. The general case of event synchronization is shown in Equation (1) where a local event is written as \( g_l \rightarrow a_l \), and \( g_l \) and \( a_l \) are local guards and actions. The remote event is \( g_r \rightarrow a_r \), where \( g_r \) and \( a_r \) are remote guards and actions. The synchronization of one local and one remote event uses the event composition operator \( || \). The actions describing state updates are composed with the parallel update operator \( || \).

\[
g_l \rightarrow a_l || e g_r \rightarrow a_r \triangleq g_l \land g_r \rightarrow a_l \parallel a_r
\]  

The merged guard is the conjunction of the guards of the local and remote events, and the merged action is the parallel composition of the actions of the local and remote events.

The general case of event synchronisation may lead to undesirable behaviour if our implementation followed Event-B semantics in an unrestricted manner. For example, an event guard prevents an update of state until the guard is true. This is implemented as a subroutine with a conditional critical region [13]. In an implementation the calling task should not block itself, so the local guard \( g_l \) is redundant; the task state is not visible externally so the task would remain blocked. Therefore the definition of our synchronized event \( s \) omits the local guard, as shown in Equation (2).

\[
s = a_l || e g_r \rightarrow a_r
\]  

In the current version of our language we define the Branch and Loop constructs with the restriction that guards are defined for the local event only. The definition of a simple branch \( b \) is shown in Equation (3), which makes use of the alternative choice operator \( [] \). The same restrictions are applied to the loop construct thus when we define a loop it has the same form as the simple branch shown.

\[
b = g_l \rightarrow a_l || e a_r []
\]

4 Preparing a Model for Implementation

Fig. 2 illustrates key features of the following section, and we begin with the AbstractBuffer write event of Fig. 1. The buffer \( \text{buff} \) is used to transmit natural numbers, so a reader sets \( \text{buff} \) to -1 when it has read and removed the value; this signals that the writer can write and the reader is blocked. The write event is enabled when \( \text{buff} \) < 0; the action writes \( \text{wVal} \) to the buffer. We also keep track of the number of updates to the buffer with \( \text{sCount} \). \( \text{wCount} \) will be the writer’s copy of \( \text{sCount} \). We keep track of the count of the writer’s writes with \( \text{wCount2} \). The counts \( \text{wCount} \) and \( \text{wCount2} \) may differ if further writer tasks are added. Using the existing decomposition tools [6, 19] we separate the variables into separate components during decomposition. The ReadWriteBuffer is the first refinement, where parameters are added in preparation for decomposition. The abstract and refined write events are shown in Appendix A.1 for reference.

We now focus on the first refinement of the write event, see Appendix A.1. We introduce shared parameters \( p1 \) and \( p2 \), define \( p1 = \text{wVal} \) in the guard, and make the assignment to \( \text{buff} \) using \( p1 \). We do the same for \( \text{wCount} \) and \( p2 \). The use of shared parameters is necessary for decomposition, since \( \text{buff} \) and \( \text{wVal} \) will reside in different machines, as will \( \text{wCount} \) and \( \text{sCount} \).
We decompose into a writer, reader and shared machine using shared-event decomposition \cite{6, 19}. We segregate the variables; \texttt{buff} and \texttt{sCount} become part of the shared machine, \texttt{wVal} and \texttt{wCount} become part of the writer machine. See Appendix \ref{sec:appendix}. Following decomposition, the write events in the writer and shared machines are not structurally linked in Event-B, but may synchronize in Tasking Event-B. The development, shown below the dashed line in Fig. \ref{fig:linking_event-b_to_code} continues. We specify the task body, where we define the flow of control (see Appendix \ref{sec:appendix}). We have the \texttt{Sequence}, \texttt{Branch}, \texttt{Loop} and \texttt{EventSynch} constructs available to use; in our example we just use \texttt{Sequence} and \texttt{EventSynch}. Clause \texttt{w1} specifies that the writer’s write event should synchronize with the shared machine’s write event. This is sequentially composed with clause \texttt{w2}. The write event parameters are marked as \texttt{actualIn} and \texttt{actualOut} to indicate the type and direction of parameters in the implementation. The shared machine’s write event is the same as its abstraction (thus not shown) except for the specification of formal in and out parameters.

\section{Translation}

We have developed tool support to translate the Tasking Event-B to Ada source, and a model of the implementation. The writer task body can be seen in Appendix \ref{sec:appendix).

In the translation of the task body we map the tasking constructs to a Common Language Model. The Common Language Model is an abstraction of commonly used programming constructs, such sequence, loop, branch, subroutine call. It is useful to provide such an abstraction to simplify the task of translating to a number of different implementations. The tasking extension has both an Event-B semantics, and an operational semantics which makes use of the Common Language. The translator is then an implementation of the rule definitions. For instance a branch specification may be written

\begin{verbatim}
if \texttt{evt}_1 \parallel \texttt{evt}_2
  \texttt{else} \texttt{evt}_3 \parallel \texttt{evt}_4
end
\end{verbatim}

This is syntactic sugar for the following, where \texttt{evt}_i = \texttt{g}_i \rightarrow \texttt{a}_i and \texttt{g}_3 = \neg \texttt{g}_1.

\begin{align*}
\texttt{g}_1 \rightarrow \texttt{a}_1 \parallel \texttt{a}_2
\end{align*}
Then the definition of a branch translation to Common Language Model is as follows,

<table>
<thead>
<tr>
<th>Event</th>
<th>Common Language Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g_1 \rightarrow a_1 \parallel e \ a_2 )</td>
<td>( \text{if } g_1 ) \text{ then } a_1 ; a_2 ) ( \text{else } a_3 ; a_4 ) ( \text{end} )</td>
</tr>
</tbody>
</table>

### 6 Discussion

#### 6.1 Correctness of the Translation

We have successfully used the tool to specify tasking developments, and generate code in several small case studies. The question of correctness of the generated code has not been formally addressed at present. However, we do have rules defining the operational semantics, and these are embodied in the translation to the Common Language Model. The tasking constructs are relatively simple, so we are able to check that the translator implementation corresponds to the rules. As an example we consider the atomic event which appears in a task body. This atomic event maps to a atomic subroutine call in the Common Language Model. In an Ada implementation we translate the atomic subroutine call to protected object call. The protected object acts as a monitor to enforce mutually exclusive access, and we assume that the Ada protected object correctly enforces this. The other constructs can be reasoned about in a similar manner, however, further work is being undertaken to address the issue of correctness of the translation.

Confidence in the correctness of the resulting code can be increased by the use of SPARKAda [1]. SPARKAda is an approach where Ada code is augmented with pre-, post, and assert conditions, and a verifier is used to prove that the code satisfies the conditions. Additional restrictions are placed on multi-tasking developments in accordance with the Ravenscar profile [5]. We would expect to be able to derive the SPARKAda annotations from the Tasking Event-B model, but this remains a task for the future.

#### 6.2 Related Work

The motivation for this work was to facilitate the link between Event-B and multi-tasking implementations; with the specific aim of overcoming the shortcomings discovered in our previous work [11]. To our knowledge no other work has been undertaken to facilitate this. The closest comparable work is that of providing implementations for Classical-B [4] using the B0 implementation notation described in [7]. B0 is similar to a programming language, and consists only of concrete programming constructs that map to programming constructs in programming languages. B0 forms part of the Classical-B refinement chain, so the implementation level specification is shown to refine an abstract development. B0 is similar to our current work in that the final step between B0 specification and actual implementation code is verified by inspection rather than formal proof. It should be noted that, although B0 can be translated to various multi-tasking programming languages, there is no support for concurrency in B0.

In previous investigations we considered using a combined CSP [15] and Event-B approach to specify the order in which events occur and synchronize. In such a development, the specifications are combined so that Event-B events synchronize with CSP events with the same name. We considered this approach to be more complex than the approach we ultimately adopted, and we prefer a streamlined approach which uses only Event-B in combination with tasking extensions.

Another code generation approach for multi-tasking involves the use of CSP and Java, and is called JCSP [24, 17] and JCSProB [27]. JCSP links the OCCAM [18] subset of the CSP process algebra and the
Java programming language. The result is the ability to specify process behaviour in CSP, and translate to those to Java threads. The resulting Java is a message passing style implementation of communication between processes. This differs from the shared memory approach described in our work. JCSProB combines the CSP and classical-B formal methods, the ProB tool \[16\] can be used to provide a unified approach for specification and model checking. The most obvious difference between the JCSProB approach and Tasking Event-B is that our work is aimed at the more recent Event-B approach.

Other work involving JCSP is that of Circus [25], which is a combined approach using CSP and $Z$ notation [20]. In a Circus specification the $Z$ and CSP constructs are used to build a specification that is amenable to model checking using [26]. In this respect, Circus has more in common with JCSProB than Tasking Event-B since it is a combined approach using model-checking technology. Circus can be translated to Java as described in [12], making use of the JCSP library code.

There is also code generation for VDM++ which can be used to specify and implement multi-tasking systems. VDM++ is an object-oriented extension to VDM-SL formal specification language. Models can be described textually; or using a graphical interface using UML diagrams. The VDM++ Toolbox can be used to generate multi-tasking C++ and Java code. Conditional waiting can be specified using permission predicates; this corresponds to the Shared Machine guards used to specify blocking behaviour in Tasking Event-B.

7 Conclusions

We have developed an approach for generating source code from Event-B, specifically targeting multi-tasking, embedded, real-time systems. We have succeeded in achieving a small semantic gap between Event-B and the tasking extension through the use of a minimal set of constructs; this is achieved by restricting the use of event guards so that we can implement them using sequences, loops, branches, or procedure calls (with parameter passing). We keep models manageable using the decomposition approach; and allocating variables to machines during shared event decomposition automatically generates the parameters. We anticipate that the approach will be scalable because the decompositions can be performed repeatedly, whenever required, prior to application of the tasking extension. Development of each of the decomposed machines can then continue in isolation from the other components.

We extend the machine with a task body to define the control flow and generate source code from this, and an Event-B model of the implementation. Tasking Machines are implemented as Tasks, Shared Machines as protected objects, and event synchronizations as procedure calls. The approach has been applied to a multi-tasking read/write buffer, and a controller for a heating system.

In the current tool we are limited to integer and Boolean types; the tool is not yet fully integrated with the Rodin platform; and we have not yet implemented triggered tasks. The tool support will be developed further to add new types, and special sensing variables that allow sensing in tasks. We will investigate when is the best time in the development to apply the tasking extensions, and further formalize the approach.

References


7


[23] The RODIN Project. at http://rodin.cs.ncl.ac.uk


A Appendix

A.1 Abstract and First refinement

event write
where
  buff < 0
then
  buff := wVal
  sCount := sCount + 1
  wCount := wCount + 1
  wCount2 := wCount2 + 1
end

event write refines write
any p1 p2
where
  p1 = wVal
  p2 = sCount + 1
  buff < 0
then
  buff := p1
  sCount := sCount + 1
  wCount := p2
  wCount2 := wCount2 + 1
end

A.2 Decomposed Machines

machine Writer
variables wVal wCount . . .
invariants . . .
events
  event write
  any p1 p2
  where
    p1 = wVal
  then
    wCount := p2
    wCount2 := wCount2 + 1
end

machine Shared
variables buff sCount . . .
invariants . . .
events
  event write
  any p1 p2
  where
    p2 = sCount + 1
    buff < 0
then
  buff := p1
  sCount := sCount + 1
end
  . . .

A.3 Tasking Machine Extension

machine WriterTsk refines Writer

tasktype periodic(250)
priority 5

taskbody is
  w1: WriterTsk.write || Shared.write;
  w2: . . .

event write
  is procedureSynch refines write
  any
    actualOut p1
    actualIn p2
  where
    p1 = wVal
    p1 ∈ Z
    p2 ∈ Z
then
  wCount := p2
end
A.4 Ada Task Code

task body WriterTsk is . . .
loop
  t := clock;
  write;
  shared.write(wVal, wCount);
  . . .
  delay until t + period;
end loop;

protected body Shared is
  entry write(p1: in Integer; p2: out Integer)
  when buff < 0 is
  begin
    p2 := sCount + 1;
    buff := p1;
    sCount := sCount + 1;
  end;