EPTCS 246

Proceedings of the

Tenth Workshop on

Programming Language Approaches to Concurrency- and Communication-cEntric Software

Uppsala, Sweden, 29th April 2017

Edited by: Vasco T. Vasconcelos and Philipp Haller

Published: 8th April 2017
DOI: 10.4204/EPTCS.246
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

Vasco T. Vasconcelos and Philipp Haller

This volume contains the proceedings of PLACES 2017, the 10th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software. The workshop was held in Uppsala, Sweden, on April 29th 2017, co-located with ETAPS, the European Joint Conferences on Theory and Practice of Software. The PLACES workshop series aims to offer a forum where researchers from different fields exchange new ideas on one of the central challenges for programming in the near future: the development of programming languages, methodologies and infrastructures where concurrency and distribution are the norm rather than a marginal concern. Previous editions of PLACES were held in Eindhoven (2016), London (2015), Grenoble (2014), Rome (2013), Tallin (2012), Saarbrcken (2011), Paphos (2010) and York (2009), all co-located with ETAPS, and the first PLACES was held in Oslo and co-located with DisCoTec (2008).

The Program Committee of PLACES 2017 consisted of:

- Sebastian Burckhardt, Microsoft Research, USA
- Ilaria Castellani, INRIA Sophia Antipolis, FR
- Marco Carbone, IT University of Copenhagen, DK
- Silvia Crafa, University of Padua, IT
- Patrick Eugster, TU Darmstadt, DE
- Ganesh L. Gopalakrishnan, University of Utah, USA
- Philipp Haller, KTH Royal Institute of Technology, SE (co-chair)
- Dimitrios Kouzapas, University of Glasgow, UK
- Sam Lindley, University of Edinburgh, UK
- Luca Padovani, University of Turin, IT
- Aleksandar Prokopec, Oracle Labs, CH
- Peter Thiemann, University of Freiburg, DE
- Vasco T. Vasconcelos, University of Lisbon, PT (co-chair)

The Program Committee, after a careful and thorough reviewing process, selected 8 papers out of 9 submissions for presentation at the workshop. Each submission was evaluated by at least three referees, and the accepted papers were selected during a week-long electronic discussion. Revised versions of all the accepted papers appear in these proceedings.

In addition to the contributed papers, the workshop featured two invited talks: first, a talk by Dave Clarke, Uppsala University, entitled The Encore Programming Language: Actors, Capabilities, Garbage, ...; second, a talk by Edwin Brady, University of St Andrews, entitled Dependent Types for Correct Concurrent Programming.

PLACES 2017 was made possible by the contribution and dedication of many people. We thank all the authors who submitted papers for consideration. Thanks also to our invited speakers, Dave Clarke and Edwin Brady. We are extremely grateful to the members of the Program Committee and additional

V.T. Vasconcelos and P. Haller (Eds.): Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES'17)
EPTCS 246, 2017, pp. ii–iii, doi:10.4204/EPTCS.246.0

experts for their careful reviews, and the balanced discussions during the selection process. The EasyChair system was instrumental in supporting the submission and review process; the EPTCS website was similarly useful in production of these proceedings.

March 30th, 2017
Vasco T. Vasconcelos
Philipp Haller

The Encore Programming Language: Actors, Capabilities, Garbage, ... (Invited Talk)

Dave Clarke
Uppsala University
Sweden
dave.clarke@it.uu.se

Encore is an actor-based programming language developed in the context of the FP7 EU Project UPSCALE. Encore is aimed at general purpose parallel programming, and shuns multithreading to avoid the lack of scalability associated with it. Encore shares a run-time with the pure actor language Pony, which, in contrast to more common run-times, offers local, per-actor garbage collection. Encore includes various abstractions for parallelism and concurrency whose correct behaviour depends, surprisingly, on correct interaction with the garbage collector. To facilitate correct behaviour and allow safe communication between actors, Encore offers a novel capability language. This talk will describe Encore, how it uses and abuses the Pony run-time, the garbage-collection related problems that ensue, and the capability language that resolves them.

V.T. Vasconcelos and P. Haller (Eds.): Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES'17)
EPTCS 246, 2017, pp. 1–1, doi:10.4204/EPTCS.246.1

Dependent Types for Correct Concurrent Programming (Invited Talk)

Edwin Brady
University of St Andrews
Scotland, UK
ecb10@st-andrews.ac.uk

Modern software systems rely on communication, for example mobile applications communicating with a central server, distributed systems coordinating a telecommunications network, or concurrent systems handling events and processes in a desktop application. However, reasoning about concurrent programs is hard, since we must reason about each process and the order in which communication might happen between processes. In this talk, I will describe an approach to implementing communicating concurrent programs, inspired by Session Types, using the dependently typed programming language Idris.

I will introduce Idris, and show how its type system can be used to describe resource access protocols (such as controlling access to a file handle or managing the state transitions for logging in to a secure data store) and verify that programs correctly follow those protocols. I will then show how to use this type-driven approach to resource tracking to reason about the order of communication between concurrent processes, ensuring that each end of a communication channel follows a defined protocol.

By type-driven, I mean that the approach involves writing an explicit type describing the pattern of communication, and verifying that processes follow that pattern by type-checking. Communication channels are explicitly parameterised by their state; operations on a channel require a channel in the correct state, and update the channel's state. As a result, a well-typed program working with a communication channel is guaranteed to follow the correct protocol for that channel.

V.T. Vasconcelos and P. Haller (Eds.): Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES'17)
EPTCS 246, 2017, pp. 2–2, doi:10.4204/EPTCS.246.2

Towards an Empirical Study of Affine Types for Isolated Actors in Scala

Philipp Haller
KTH Royal Institute of Technology
Stockholm, Sweden
phaller@kth.se

Fredrik Sommar
KTH Royal Institute of Technology
Stockholm, Sweden
fsommar@kth.se

LaCasa is a type system and programming model to enforce the object capability discipline in Scala, and to provide affine types. One important application of LaCasa's type system is software isolation of concurrent processes. Isolation is important for several reasons including security and data-race freedom. Moreover, LaCasa's affine references enable efficient, by-reference message passing while guaranteeing a "deep-copy" semantics. This deep-copy semantics enables programmers to seamlessly port concurrent programs running on a single machine to distributed programs running on large-scale clusters of machines.

This paper presents an integration of LaCasa with actors in Scala, specifically, the Akka actor-based middleware, one of the most widely-used actor systems in industry. The goal of this integration is to statically ensure the isolation of Akka actors. Importantly, we present the results of an empirical study investigating the effort required to use LaCasa's type system in existing open-source Akka-based systems and applications.

1 Introduction

The desire for languages to catch more errors at compile time seems to have increased in the last couple of years. Recent languages, like Rust [15], show that a language does not have to sacrifice a lot, if any, convenience to gain access to safer workable environments. Entire classes of memory-related bugs can be eliminated, statically, through the use of affine types. In the context of this paper it is important that affine types can also enforce isolation of concurrent processes.

LACASA [5] shows that affine types do not necessarily need to be constrained to new languages: it introduces affine types for Scala, an existing, widely-used language. LACASA is implemented as a compiler plugin for Scala 2.11.[1] However, so far it has been unclear how big the effort is to apply LACASA in practice. This paper is a first step to investigate this question empirically on open-source Scala programs using the Akka actor framework [8].

Contributions This paper presents an integration of LACASA and Akka. Thus, our integration enforces isolation for an existing actor library. Furthermore, we present the results of an empirical study evaluating the effort to use isolation types in real applications. To our knowledge it is the first empirical study evaluating isolation types for actors in a widely-used language.

Selected Related Work Active ownership [3] is a minimal variant of ownership types providing race freedom for active objects while enabling by-reference data transfer between active objects. The system is realized as an extended subset of Java. Kilim [14] combines static analysis and type checking to

[1] See https://github.com/phaller/lacasa

V.T. Vasconcelos and P. Haller (Eds.): Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES'17)
EPTCS 246, 2017, pp. 3–9, doi:10.4204/EPTCS.246.3

provide isolated actors in Java. For neither of the two above systems, active ownership and Kilim, have the authors reported any empirical results on the syntactic overhead of the respective systems, unlike the present paper. SOTER [12] is a static analysis tool which infers if the content of a message is compatible with an ownership transfer semantics. This approach is complementary to a type system which enables developers to require ownership transfer semantics. Pony [4] and Rust [15] are new language designs with type systems to ensure data-race freedom in the presence of zero-copy transfer between actors/concurrent processes. It is unclear how to obtain empirical results on the syntactic overhead of the type systems of Pony or Rust. In contrast, LACASA extends an existing, widely-used language, enabling empirical studies.

2 Background

In this paper we study affine types as provided by LACASA [5], an extension of the Scala programming language. LACASA is implemented as a combination of a compiler plugin for the Scala 2.11.x compiler and a small runtime library. LACASA provides affine references which may be consumed at most once. In LACASA an affine reference to a value of type `T` has type `Box[T]`. The name of type constructor `Box` indicates that access to an affine reference is restricted. Accessing the wrapped value of type `T` requires the use of a special `open` construct:

```
1  val box: Box[T] = ...
2  box open { x => /* use ‘x‘ */ }
```

`open` is implemented as a method which takes the closure `{ x => /* use ‘x‘ */ }` as an argument. The closure body then provides access to the object `x` wrapped by the `box` of type `Box[T]`. However, LACASA restricts the environment (*i.e.*, the captured variables) of the argument closure in order to ensure affinity: mutable variables may not be captured. Without this restriction it would be simple to duplicate the wrapped value, violating affinity:

```
1  val box: Box[T] = ...
2  var leaked: Option[T] = None
3  box open { x =>
4    leaked = Some(x) // illegal
5  }
6  val copy: T = leaked.get
```

LACASA also protects against leaking wrapped values to global state:

```
1  object Global { var cnt: LeakyCounter = null }
2  class LeakyCounter {
3    var state: Int = 0
4    def increment(): Unit = { state += 1 }
5    def leak(): Unit = { Global.cnt = this }
6    ...
7  }
8  val box: Box[LeakyCounter] = ... // illegal
9  box open { cnt =>
10   cnt.leak()
11 }
12 val copy: LeakyCounter = Global.cnt
```

The above `LeakyCounter` class is illegal to be wrapped in a box. The reason is that even without capturing a mutable variable within `open`, it is possible to create a copy of the counter, because the `leak` method leaks a reference to the counter to global mutable state (the `Global` singleton object). To prevent

```
1   def m[T](b: Box[T])(implicit p: CanAccess { type C = b.C }): Unit = {
2     b open { x => /* use `x` */ }
3   }
```

Figure 1: Boxes and permissions in LaCasa.

```
1  class Box[T] { self =>
2    type C
3    def open(fun: T => Unit)
4            (implicit p: CanAccess { type C = self.C }): Box[T] = {
5      ...
6    }
7  }
```

Figure 2: Type signature of the `open` method.

this kind of affinity violation, LaCasa restricts the creation of boxes of type `Box[A]` to types `A` which conform to the object capability discipline [11]. According to the object capability discipline, a method `m` may only use object references that have been passed explicitly to `m`, or `this`. Concretely, accessing `Global` on line 5 is illegal, since `Global` was not passed explicitly to method `leak`.

In previous work [5] we have formalized the object capability discipline as a type system and we have shown that in combination with LaCasa's type system, affinity of box-typed references is ensured.

Affine references, *i.e.*, references of type `Box[T]`, may be consumed, causing them to become unaccessible. Consumption is expressed using *permissions* which control access to box-typed references. Consuming an affine reference consumes its corresponding permission.

Ensuring at-most-once consumption of affine references thus requires each permission to be linked to a specific box, and this link must be checked statically. In LaCasa permissions are linked to boxes using path-dependent types [2]. For example, Figure 1 shows a method `m` which has two formal parameters: a box `b` and a permission `p` (its `implicit` modifier may be ignored for now). The type `CanAccess` of permissions has a *type member* `C` which is used to establish a static link to box `b` by requiring the equality `type C = b.C` to hold. The type `b.C` is a path-dependent type with the property that there is only a single runtime object, namely `b`, whose type member `C` is equal to type `b.C`. In order to prevent forging permissions, permissions are only created when creating boxes; it is impossible to create permissions for existing boxes.

Since permissions may be consumed (as shown below), it is important that opening a box requires its permission to be available. Figure 2 shows how this is ensured using an *implicit parameter* [13] of the `open` method (line 5). Note that the shown type signature is simplified; the actual signature uses a *spore* type [10] instead of a function type on line 4 to ensure that the types of captured variables are immutable.

Consuming Permissions Permissions in LaCasa are just Scala implicit values. This means their availability is flow-insensitive. Therefore, changing the set of available permissions requires changing scope. In LaCasa, calling a permission-consuming method requires passing an explicit continuation closure. The LaCasa type checker enforces that the consumed permission is then no longer available in the scope of this continuation closure. Figure 3 shows an example. LaCasa enforces that such continuation-passing methods do not return (see [5]), indicated by Scala's bottom type, `Nothing`.

```
1   def m[T](b: Box[T])(cont: () => Unit)(implicit p: CanAccess { type C = b.C }): Nothing = {
2     b open { x => /* use ‘x‘ */ }
3     consume(b) {
4       // explicit continuation closure
5       // permission ‘p‘ unavailable
6       ...
7       cont() // invoke outer continuation closure
8     }
9   }
```

Figure 3: Consuming permissions in LACASA.

```
1  class ExampleActor extends Actor {
2    def receive = {
3      case msgpat_1 =>
4      ...
5      case msgpat_n =>
6    }
7  }
```

Figure 4: Defining actor behavior in Akka.

2.1 Akka

Akka [8] is an implementation of the actor model [6, 1] for Scala. Actors are concurrent processes communicating via asynchronous messages. Each actor buffers received messages in a local “mailbox” – a queue of incoming messages. An Akka actor processes at most one incoming message at a time. Figure 4 shows the definition of an actor’s behavior in Akka. The behavior of each actor is defined by a subclass of a predefined `Actor` trait. The `ExampleActor` subclass implements the `receive` method which is abstract in trait `Actor`. The `receive` method returns a message handler defined as a block of pattern-matching cases. This message handler is used to process each message in the actor’s mailbox. The `Actor` subclass is then used to create a new actor as follows:

```
1   val ref: ActorRef = system.actorOf(Props[ExampleActor], "example-actor")
2   ref ! msg
```

The result of creating a new actor is a reference object (`ref`) of type `ActorRef`. An `ActorRef` is a handle that can be used to send asynchronous messages to the actor using the `!` operator (line 2).

3 Integrating LACASA and Akka

The Adapter The LaCasa-Akka adapter[2] is an extension on top of Akka. During its design, an important constraint was to keep it separate from Akka’s internals – primarily to limit the effect of internal changes as Akka evolves.

The adapter consists of two parts: `SafeActor[T]` and `SafeActorRef[T]`, both with the same responsibilities as their counterparts in the Akka API. However, note that in contrast to the latter, they are generic over the message type. Akka instead relies on pattern matching to discern the types of received

[2] See `https://github.com/fsommar/lacasa/tree/akka`

```
1 trait SafeActor[T] extends Actor {
2   def receive(msg: Box[T])(implicit acc: CanAccess { type C = msg.C }): Unit
3 }
```

Figure 5: Usage of LACASA's boxes and permissions in `SafeActor`.

Program	LOC (Scala/Akka)	LOC (LACASA/Akka)	Changes	Changes (%)
ThreadRing	130	153	27 add./10 del.	28.5%
Chameneos	143	165	26 add./7 del.	23.1%
Banking	118	135	27 add./12 del.	33.1%
Average	130	151		28.2%

Table 1: Results of the empirical study.

messages (see Section 2.1). For the LaCasa-Akka adapter, however, it is necessary to know the types of messages at compile time, to prevent the exchange of unsafe message types.

SafeActor A subclass of Akka's `Actor`, `SafeActor` provides a different `receive` method signature, which is the primary difference between the two. Instead of receiving an untyped message, of type `Any`, `SafeActor[T]` receives a boxed message of type `T`, and an access permission for the contents of the box (see Figure 5).

SafeActorRef The API for `SafeActorRef` is a wrapper of Akka's `ActorRef`, and contains a subset of the latter's methods and functionality. It uses the same method names, but method signatures are different, to include necessary safety measures. For every method accepting a box, there is a dual method accepting a box and a continuation closure. Recall that it is the only way to enforce that boxes are consumed (see Section 2). The dual methods use the `AndThen` suffix to indicate that they accept a continuation closure.

For message types that are immutable, the API can be significantly simplified, resembling that of a regular Akka `ActorRef`. Meanwhile, internally, the message is still boxed up and forwarded for handling by the `SafeActor`. Importantly, though, the box does not have to be consumed, enabling the method to return and continue execution – removing the need for the `AndThen` family of methods.

4 Empirical Study

We converted several Scala/Akka programs to use the LaCasa-Akka adapter described in Section 3. The goal of this conversion is to evaluate the effort required to use LACASA's type system in practice. The converted programs are part of the Savina actor benchmark suite [7]. Concretely, we converted the following programs: (1) In `ThreadRing`, an integer token message is passed around a ring of N connected actors. This benchmark is adopted from Theron [9]; (2) `Chameneos` is a micro-benchmark measuring the effects of contention on shared resources while processing messages; (3) `Banking` is a bank transaction micro-benchmark measuring synchronous request-response with interfering transactions.

Table 1 shows the results. On average 28.2% of the lines of code of each program needed to be changed (we exclude changes to imports). It is important to note that we expect this number to be significantly lower for larger applications where sequential, non-actor-based code dominates the code

base. The most important reasons for code changes are (a) the declaration of safe message classes and (b) the insertion and removal of messages into/from boxes. For example, in `ThreadRing` 33.3% of added lines are due to declaring message classes as safe.

5 Conclusion

LACASA extends Scala's type system with affine types, with applications to race-free concurrent programming and safe off-heap memory management. This paper shows how LACASA can ensure the isolation of actors in Akka, a widely-used actor framework for Scala, while providing safe and efficient ownership transfer of asynchronous messages. According to our empirical study, adjusting existing Akka-based Scala programs requires changing 28.2% of the lines of code on average. However, this initial result represents a worst-case scenario, since the study only considered micro-benchmarks where actor-related code dominates, unlike larger real-world applications. An empirical study extending our results to medium-to-large open-source code bases is ongoing.

References

[1] Gul A. Agha (1986): *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Series in Artificial Intelligence, The MIT Press, Cambridge, Massachusetts.

[2] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf & Sandro Stucki (2016): *The Essence of Dependent Object Types*. In: *A List of Successes That Can Change the World*, Springer, pp. 249–272, doi:10.1007/978-3-319-30936-1_14.

[3] Dave Clarke, Tobias Wrigstad, Johan Östlund & Einar Broch Johnsen (2008): *Minimal Ownership for Active Objects*. In: *APLAS*, Springer, pp. 139–154, doi:10.1007/978-3-540-89330-1_11.

[4] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing & Andy McNeil (2015): *Deny capabilities for safe, fast actors*. In: *AGERE!@SPLASH*, ACM, pp. 1–12, doi:10.1145/2824815.2824816.

[5] Philipp Haller & Alex Loiko (2016): *LaCasa: Lightweight affinity and object capabilities in Scala*. In: *OOPSLA*, ACM, pp. 272–291, doi:10.1145/2983990.2984042.

[6] Carl Hewitt, Peter Bishop & Richard Steiger (1973): *A Universal Modular ACTOR Formalism for Artificial Intelligence*. In: *IJCAI*, William Kaufmann, pp. 235–245.

[7] Shams Mahmood Imam & Vivek Sarkar (2014): *Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries*. In: *AGERE!@SPLASH*, ACM, pp. 67–80, doi:10.1145/2687357.2687368.

[8] Lightbend, Inc. (2009): *Akka*. `http://akka.io/`.

[9] Ashton Mason (2012): *The ThreadRing benchmark*. `http://www.theron-library.com/index.php?t=page&p=threadring`.

[10] Heather Miller, Philipp Haller & Martin Odersky (2014): *Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution*. In: *ECOOP*, Springer, pp. 308–333, doi:10.1007/978-3-662-44202-9_13.

[11] Mark Samuel Miller (2006): *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. thesis, Johns Hopkins University, Baltimore, Maryland, USA.

[12] Stas Negara, Rajesh K. Karmani & Gul A. Agha (2011): *Inferring ownership transfer for efficient message passing*. In: *PPOPP*, ACM, pp. 81–90, doi:10.1145/1941553.1941566.

[13] Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee & Kwangkeun Yi (2012): *The implicit calculus: a new foundation for generic programming*. In: *PLDI*, ACM, pp. 35–44, doi:10.1145/2254064.2254070.

[14] Sriram Srinivasan & Alan Mycroft (2008): *Kilim: Isolation-Typed Actors for Java*. In: *ECOOP*, Springer, pp. 104–128, doi:10.1007/978-3-540-70592-5_6.

[15] Aaron Turon (2017): *Rust: from POPL to practice (keynote)*. In: *POPL*, ACM, p. 2, doi:10.1145/3009837.3011999.

Actors without Borders: Amnesty for Imprisoned State

Elias Castegren Tobias Wrigstad
Uppsala University, Sweden

In concurrent systems, some form of synchronisation is typically needed to achieve data-race freedom, which is important for correctness and safety. In actor-based systems, messages are exchanged concurrently but executed sequentially by the receiving actor. By relying on isolation and non-sharing, an actor can access its own state without fear of data-races, and the internal behavior of an actor can be reasoned about sequentially.

However, actor isolation is sometimes too strong to express useful patterns. For example, letting the iterator of a data-collection alias the internal structure of the collection allows a more efficient implementation than if each access requires going through the interface of the collection. With full isolation, in order to maintain sequential reasoning the iterator must be made part of the collection, which bloats the interface of the collection and means that a client must have access to the whole data-collection in order to use the iterator.

In this paper, we propose a programming language construct that enables a relaxation of isolation but without sacrificing sequential reasoning. We formalise the mechanism in a simple lambda calculus with actors and passive objects, and show how an actor may leak parts of its internal state while ensuring that any interaction with this data is still synchronised.

1 Introduction

Synchronisation is a key aspect of concurrent programs and different concurrency models handle synchronisation differently. Pessimistic models, like locks or the actor model [1] serialise computation *within certain encapsulated units*, allowing sequential reasoning about internal behavior.

In the case of the actor model, for brevity including also active objects (which carry state, which actor's traditionally do not), if a reference to an actor A's internal state is accessible outside of A, operations inside of A are subject to data-races and sequential reasoning is lost. The same holds true for operations on an aggregate object behind a lock, if a subobject is leaked and becomes accessible where the appropriate lock is not held.

In previous work, we designed Kappa [4], a type system in which the boundary of a unit of encapsulation can be statically identified. An entire encapsulated unit can be wrapped inside some synchronisation mechanism, *e.g.,* a lock or an asynchronous actor interface, and consequently all operations inside the boundary are guaranteed to be data-race free. An important goal of this work is facilitating object-oriented reuse in concurrent programming: internal objects are oblivious to how their data-race freedom is guaranteed, and the building blocks can be reused without change regardless of their external synchronisation.

This extended abstract explores two extensions to this system, which we explain in the context of the actor model (although they are equally applicable to a system using locks). Rather than rejecting programs where actors leak internal objects, we allow an actor to *bestow* its synchronisation mechanism upon the exposed objects. This allows multiple objects to effectively construct an actor's interface. Exposing internal operations externally makes concurrency more fine-grained. To allow external control of the possible interleaving of these operations, we introduce an *atomic block* that groups them together. The following section motivates these extensions.

V.T. Vasconcelos and P. Haller (Eds.): Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES'17)
EPTCS 246, 2017, pp. 10–20, doi:10.4204/EPTCS.246.4

```
class Node[t]
  var next : Node[t]
  var elem : t
  // getters and setters omitted

actor List[t]
  var first : Node[t]
  def getFirst() : Node[t]
    return this.first

  def get(i : int) : t
    var current = this.first
    while i > 0 do
      current = current.next
      i = i - 1
    return current.elem
```

(a)

```
class Iterator[t]
  var current : Node[t]
  def init(first : Node[t]) : void
    this.current = first

  def getNext() : t
    val elem = this.current.elem
    this.current = this.current.next
    return elem

  def hasNext() : bool
    return this.current != null

actor List[t]
  def getIterator() : Iterator[t]
    val iter = new Iterator[t]
    iter.init(this.first)
    return iter
```

(b)

Figure 1: (a) A list implemented as an actor. (b) An iterator for that list.

2 Breaking Isolation: Motivating Example

We motivate breaking isolation in the context of an object-oriented actor language, with actors serving as the units of encapsulation, encapsulating zero or more passive objects. Figure 1a shows a Kappa program with a linked list in the style of an actor with an asynchronous external interface. For simplicity we allow asynchronous calls to return values and omit the details of how this is accomplished (*e.g.,* by using futures, promises, or by passing continuations).

Clients can interact with the list for example by sending the message `get` with a specified index. With this implementation, each time `get` is called, the corresponding element is calculated from the head of the list, giving linear time complexity for each access. Iterating over all the elements of the list has quadratic time complexity.

To allow more efficient element access, the list can provide an iterator which holds a pointer to the current node (Figure 1b). This allows constant-time access to the *current* element, and linear iteration, but also breaks encapsulation by providing direct access to nodes and elements without going through the list interface. *List operations are now subject to data-races.*

A middle ground providing linear time iteration without data-races can be implemented by moving the iterator logic into the list actor, so that the calls to `getNext` and `hasNext` are synchronised in the message queue of the actor. This requires a more advanced scheme to map different clients to different concurrent iterators, clutters the list interface, creates unnecessary coupling between `List` and `Iterator`, and complicates support of *e.g.,* several kinds of iterators.

Another issue with concurrent programs is that interleaving interaction with an actor makes it hard to reason about operations that are built up from several smaller operations. For example, a client might want to access two adjacent nodes in the list and combine their elements somehow. When sending two `get` messages, there is nothing that prevents other messages from being processed by the list actor after the first one, possibly removing or changing one of the values.

```
actor List[t]
  ...
  def getIterator() : B(Iterator[t])
    val iter = new Iterator[t]
    iter.init(this.first)
    return bestow iter
```

```
val iter = list!getIterator()
while iter!hasNext() do
  val elem = iter!getNext()
  ...
```

Figure 2: A list actor returning a bestowed iterator, and the code for a client using it

Again, unless the list actor explicitly provides an operation for getting adjacent values, there is no way for a client to safely express this operation.

3 Bestowing and Grouping Activity

Encapsulating state behind a synchronisation mechanism allows reasoning sequentially about operations on that state. However, since Kappa lets us identify the encapsulation boundary of the data structure [4], it is possible to *bestow* objects that are leaked across this boundary with a synchronisation wrapper. Statically, this means changing the type of the returned reference to reflect that operations on it may block. Dynamically it means identifying with what and how the leaked object shall synchronise.

For clarity, we explicate this pattern with a **bestow** operation. In the case of actors, an actor `a` that performs **bestow** on some reference r creates a wrapper around r that makes it appear like an actor with the same interface as r, *but asynchronous*. Operations on the bestowed reference will be relayed to `a` so that the actor `a` is the one actually performing the operation. If r was leaked from an enclosure protected by a lock `l`, r's wrapper would instead acquire and release `l` around each operation.

Figure 2 shows the minimal changes needed to the code in Figure 1b, as well as the code for a client using the iterator. The only change to the list is that `getIterator()` returns a bestowed iterator (denoted by wrapping the return type in **B**`(...)`[1]), rather than a passive one. In the client code, synchronous calls to `hasNext()` and `getNext()` become asynchronous message sends. These messages are handled by the list actor, even though they are not part of its interface. This means that any concurrent usages of iterators are still free from data-races.

It is interesting to ponder the difference between creating an iterator *inside* the list and bestowing it, or creating an iterator *outside* the list, and bestowing each individual list node it traverses. In the former case, `getNext()` is performed without interleaved activities in the same actor. In the latter case, it is possible that the internal operations are interleaved with other operations on list. The smaller the object returned, the more fine-grained is the concurrency.

Sometimes it is desirable that multiple operations on an object are carried out in a non-interleaved fashion. For this purpose, we use an **atomic** block construct that operates on a an actor or a bestowed object, *cf.* Figure 3. In the case of operations on an actor, message sends inside an atomic block are *batched* and sent as a single message to the receiver. In the case of operations on an object guarded by a lock, we replace each individual lock–release by a single lock–release wrapping the block. It is possible to synchronise across multiple locked objects in a single block.

[1] If desired, this type change can be implicit through view-point adaptation [9].

```
class Iterator[t]
  var current : B(Node[t])
  def getNext() : t
    val elem = this.current ! elem()
    // Possible interleaving of other messages
    this.current = this.current ! next()
    return elem
```

```
class Iterator[t]
  var current : B(Node[t])
  def getNext() : t
    atomic c <- this.current
      val elem = c ! elem()
      this.current = c ! next()
      return elem
```

Figure 3: Fine-grained (left) and coarse-grained (right) concurrency control.

An **atomic** block allows a client to express new operations by composing smaller ones. The situation sketched in § 2, where a client wants to access two adjacent nodes in the list actor without interleaving operations from other clients is easily resolved by wrapping the two calls to `get` (or `getNext`, if the iterator is used) inside an **atomic** block. This will batch the messages and ensure that they are processed back to back:

```
atomic it <- list ! getIterator()
  val e1 <- it.getNext()
  val e2 <- it.getNext()
```

$\Longrightarrow$

```
(e1, e2) =
  list ! λ this .
    {val it = this.getIterator();
     val e1 = it.getNext();
     val e2 = it.getNext();
     return (e1, e2)}
```

4 Formalism

To explain **bestow** and **atomic** we use a simple lambda calculus with actors and passive objects. We abstract away most details that are unimportant when describing the behavior of bestowed objects. For example, we leave out classes and actor interfaces and simply allow arbitrary operations on values. By disallowing sharing of (non-bestowed) passive objects, we show that our language is free from data-races (*cf.* § 4.4).

The syntax of our calculus is shown in Figure 4. An expression e is a variable x, a function application $e\ e'$ or a message send $e!v$. Messages are sent as anonymous functions, which are executed by the receiving actor. We abstract updates to passive objects as $e.\texttt{mutate}()$, which has no actual effect in the formalism, but is reasoned about in § 4.4. A new object or actor is created with **new** τ and a passive object can be bestowed by the current actor with **bestow** e. We don't need a special **atomic** construct in the formalism as this can be modeled by composing operations in a single message as sketched in the end of the previous section.

Statically, values are anonymous functions or the unit value (). Dynamically, id is the identifier of an actor, ι is the memory location of a passive object, and ι_{id} is a passive object ι bestowed by the actor id. A type is an active type α, a passive type p, a function type $\tau \to \tau$, or the `Unit` type. An active type is either an actor type c or a bestowed type $\mathbf{B}(\mathsf{p})$. Note that for simplicity,

$$
\begin{array}{lcl}
e & ::= & x \mid e\ e \mid e!v \mid e.\texttt{mutate}() \mid \textbf{new}\ \tau \mid \textbf{bestow}\ e \mid v \\
v & ::= & \lambda x : \tau . e \mid () \mid id \mid \iota \mid \iota_{id}
\end{array}
\qquad
\begin{array}{lcl}
\tau & ::= & \alpha \mid \mathsf{p} \mid \tau \to \tau \mid \texttt{Unit} \\
\alpha & ::= & \mathsf{c} \mid \mathbf{B}(\mathsf{p})
\end{array}
$$

Figure 4: The syntax of a simple lambda calculus with actors, **bestow** and **atomic**.

p and c are not meta-syntactic variables; every passive object has type p, every actor has type c, and every bestowed object has type $\mathbf{B}(\mathsf{p})$.

$\boxed{\Gamma \vdash e : \tau}$ *(Expressions)*

$$\textsc{e-var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{e-apply}\ \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\, e' : \tau} \qquad \textsc{e-new-passive}\ \frac{}{\Gamma \vdash \mathbf{new}\,\mathsf{p} : \mathsf{p}} \qquad \textsc{e-new-actor}\ \frac{}{\Gamma \vdash \mathbf{new}\,\mathsf{c} : \mathsf{c}}$$

$$\textsc{e-mutate}\ \frac{\Gamma \vdash e : \mathsf{p}}{\Gamma \vdash e.\texttt{mutate()} : \texttt{Unit}} \qquad \textsc{e-bestow}\ \frac{\Gamma \vdash e : \mathsf{p}}{\Gamma \vdash \mathbf{bestow}\, e : \mathbf{B}(p)} \qquad \textsc{e-send}\ \frac{\Gamma \vdash e : \alpha \quad \Gamma_\alpha, x : \mathsf{p} \vdash e' : \tau' \quad \nexists\, \iota\ .\ \iota \in e'}{\Gamma \vdash e\,!\,\lambda x : \mathsf{p}.e' : \texttt{Unit}}$$

$$\textsc{e-fn}\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau.e) : \tau \to \tau'} \qquad \textsc{e-unit}\ \frac{}{\Gamma \vdash () : \texttt{Unit}} \qquad \textsc{e-loc}\ \frac{}{\Gamma \vdash \iota : \mathsf{p}} \qquad \textsc{e-id}\ \frac{}{\Gamma \vdash id : \mathsf{c}} \qquad \textsc{e-bestowed}\ \frac{}{\Gamma \vdash \iota_{id} : \mathbf{B}(p)}$$

Figure 5: Static semantics. Γ maps variables to types. Γ_α contains only the active types α of Γ.

4.1 Static Semantics

The typing rules for our formal language can be found in Figure 5. The typing context Γ maps variables to types. The "normal" lambda calculus rules E-VAR and E-APPLY are straightforward. The **new** keyword can create new passive objects or actors (E-NEW-*). Passive objects may be mutated (E-MUTATE), and may be bestowed activity (E-BESTOW).

Message sends are modeled by sending anonymous functions which are run by the receiver (E-SEND). The receiver must be of active type (*i.e.,* be an actor or a bestowed object), and the argument of the anonymous function must be of passive type p (this can be thought of as the **this** of the receiver). Finally, all free variables in the body of the message must have active type to make sure that passive objects are not leaked from their owning actors. This is captured by Γ_α which contains only the active mappings $\_ : \alpha$ of Γ. Dynamically, the body may not contain passive objects ι. Typing values is straightforward.

4.2 Dynamic Semantics

Figure 6 shows the small-step operational semantics for our language. A running program is a heap H, which maps actor identifiers id to actors (ι, L, Q, e), where ι is the **this** of the actor, L is the local heap of the actor (a set containing the passive objects created by the actor), Q is the message queue (a list of lambdas to be run), and e is the current expression being evaluated.

An actor whose current expression is a value may pop a message from its message queue and apply it to its **this** (EVAL-ACTOR-MSG). Any actor in H may step its current expression, possibly also causing some effect on the heap (EVAL-ACTOR-RUN). The relation $id \vdash \langle H, e\rangle \hookrightarrow \langle H', e'\rangle$ denotes actor id evaluating heap H and expression e one step.

$\boxed{H \hookrightarrow H'}$ *(Evaluation)*

$$\textsc{eval-actor-msg}\ \frac{H(id) = (\iota, L, Q\,v', v) \qquad H' = H[id \mapsto (\iota, L, Q, v'\,\iota)]}{H \hookrightarrow H'}$$

$$\textsc{eval-actor-run}\ \frac{H(id) = (\iota, L, Q, e) \qquad id \vdash \langle H, e\rangle \hookrightarrow \langle H', e'\rangle \qquad H'(id) = (\iota, L', Q', e) \qquad H'' = H'[id \mapsto (\iota, L', Q', e')]}{H \hookrightarrow H''}$$

$\boxed{id \vdash \langle H, e\rangle \hookrightarrow \langle H', e'\rangle}$ *(Evaluation of expressions)*

$$\textsc{eval-send-actor}\ \frac{H(id') = (\iota, L, Q, e) \qquad H' = H[id' \mapsto (\iota, L, v\,Q, e)]}{id \vdash \langle H, id' \,!\, v\rangle \hookrightarrow \langle H', ()\rangle}$$

$$\textsc{eval-send-bestowed}\ \frac{H(id') = (\iota', L, Q, e) \qquad H' = H[id' \mapsto (\iota', L, (\lambda x : \mathsf{p}.v\,\iota)\,Q, e)]}{id \vdash \langle H, \iota_{id'} \,!\, v\rangle \hookrightarrow \langle H', ()\rangle}$$

$$\textsc{eval-apply}\ \frac{e' = e[x \mapsto v]}{id \vdash \langle H, (\lambda x : \tau.e)\,v\rangle \hookrightarrow \langle H, e'\rangle}$$

$$\textsc{eval-mutate}\ \frac{}{id \vdash \langle H, \iota.\texttt{mutate()}\rangle \hookrightarrow \langle H, ()\rangle}$$

$$\textsc{eval-bestow}\ \frac{}{id \vdash \langle H, \mathbf{bestow}\,\iota\rangle \hookrightarrow \langle H, \iota_{id}\rangle}$$

$$\textsc{eval-new-passive}\ \frac{H(id) = (\iota, L, Q, e) \qquad \iota'\,\mathbf{fresh} \qquad H' = H[id \mapsto (\iota, L \cup \{\iota'\}, Q, e)]}{id \vdash \langle H, \mathbf{new}\,\mathsf{p}\rangle \hookrightarrow \langle H', \iota'\rangle}$$

$$\textsc{eval-new-actor}\ \frac{id'\,\mathbf{fresh} \qquad \iota'\,\mathbf{fresh} \qquad H' = H[id' \mapsto (\iota', \{\iota'\}, \epsilon, ())]}{id \vdash \langle H, \mathbf{new}\,\alpha\rangle \hookrightarrow \langle H', id'\rangle}$$

$$\textsc{eval-context}\ \frac{id \vdash \langle H, e\rangle \hookrightarrow \langle H', e'\rangle}{id \vdash \langle H, E[e]\rangle \hookrightarrow \langle H', E[e']\rangle}$$

$E[\bullet] ::= \bullet\ e \mid v\ \bullet \mid \bullet\,!v \mid \bullet.\text{mutate()} \mid \mathbf{bestow}\ \bullet$

Figure 6: Dynamic semantics.

Sending a lambda to an actor prepends this lambda to the receiver's message queue and results in the unit value (EVAL-SEND-ACTOR). Sending a lambda v to a bestowed value instead prepends a new lambda to the queue of the actor that bestowed it, which simply applies v to the underlying passive object (EVAL-SEND-BESTOWED).

Function application replaces all occurrences of the parameter x in its body by the argument v (EVAL-APPLY). Mutation is a no-op in practice (EVAL-MUTATE). Bestowing a passive value ι in actor id creates the bestowed value ι_{id} (EVAL-BESTOW).

Creating a new object in actor id adds a fresh location ι' to the set of the actors passive objects L and results in this value (EVAL-NEW-PASSIVE). Creating a new actor adds a new actor with a fresh identifier to the heap. Its local heap contains only the fresh **this**, its queue is empty, and its current expression is the unit value (EVAL-NEW-ACTOR).

We handle evaluation order by using an evaluation context E (EVAL-CONTEXT).

4.3 Well-formedness

A heap H is well-formed if all its actors are well-formed with respect to H, and the local heaps L_i and L_j of any two different actors are disjoint (WF-HEAP). We use $\mathcal{LH}(H(id))$ to denote the local heap of actor id. An actor is well-formed if its **this** is in its local heap L and its message

$\boxed{\vdash H \qquad H \vdash (\iota, L, Q, e) \qquad H \vdash Q}$ *(Well-formedness)*

$$\textsc{wf-heap}\ \frac{\forall id_1 \neq id_2\ .\ \mathcal{LH}(H(id_1)) \cap \mathcal{LH}(H(id_2)) = \emptyset \qquad \forall id \in \mathbf{dom}(H)\ .\ H \vdash H(id)}{\vdash H}$$

$$\textsc{wf-actor}\ \frac{\iota \in L \qquad H; L \vdash Q \qquad \epsilon \vdash e : \tau \qquad \forall \iota \in e\ .\ \iota \in L \qquad \forall id \in e\ .\ id \in \mathbf{dom}(H) \qquad \forall \iota_{id} \in e\ .\ \iota \in \mathcal{LH}(H(id))}{H \vdash (\iota, L, Q, e)}$$

$$\textsc{wf-queue-message}\ \frac{H; L \vdash Q \qquad x : \mathsf{p} \vdash e : \tau \qquad \forall \iota \in e\ .\ \iota \in L \qquad \forall id \in e\ .\ id \in \mathbf{dom}(H) \qquad \forall \iota_{id} \in e\ .\ \iota \in \mathcal{LH}(H(id))}{H; L \vdash (\lambda x : \mathsf{p}.e)\ Q}$$

$$\textsc{wf-queue-empty}\ \frac{}{H; L \vdash \epsilon}$$

Figure 7: Well-formedness rules. $\mathcal{LH}$ gets the local heap from an actor: $\mathcal{LH}((\iota, L, Q, e)) = L$

queue Q is well-formed. The current expression e must be typable in the empty environment, and all passive objects ι that are subexpressions of e must be in the local heap L. Similarly, all actor identifiers in e must be actors in the system, and all bestowed objects must belong to the local heap of the actor that bestowed it (WF-ACTOR).

A message queue is well-formed if all its messages are well-formed (WF-QUEUE-*). A message is well-formed if it is a well-formed anonymous function taking a passive argument, and has a body e with the same restrictions on values as the current expression in an actor.

4.4 Meta Theory

We prove soundness of our language by proving progress and preservation in the standard fashion:

> **Progress**: A well-formed heap H can either be evaluated one step, or only has actors with empty message queues and fully reduced expressions:
>
> $$\vdash H \implies (\exists H'\ .\ H \hookrightarrow H')\ \vee\ (\forall id \in \mathbf{dom}(H)\ .\ H(id) = (\iota, L, \epsilon, v))$$
>
> **Preservation**: Evaluation preserves well-formedness of heaps: $\vdash H\ \wedge\ H \hookrightarrow H' \implies \vdash H'$

Both properties can be proven to hold with straightforward induction.

The main property that we are interested in for our language is data-race freedom. As we don't have any actual effects on passive objects, we show this by proving that if an actor is about to execute ι.`mutate()`, no other actor will be about to execute `mutate` on the same object:

> **Data-race freedom**: Two actors will never mutate the same active object
>
> $$\left(\begin{array}{c} id_1 \neq id_2 \\ \wedge\ H(id_1) = (\iota_1, L_1, Q_1, \iota.\texttt{mutate()}) \\ \wedge\ H(id_2) = (\iota_2, L_2, Q_2, \iota'.\texttt{mutate()}) \end{array}\right) \implies \iota \neq \iota'$$

This property is simple to prove using two observations on what makes a well-formed heap:

1. An actor will only ever access passive objects that are in its local heap (WF-ACTOR).
2. The local heaps of all actors are disjoint (WF-HEAP).

The key to showing preservation of the first property is in the premise of rule E-SEND which states that all free variables and values must be active objects ($\Gamma_\alpha, x : \mathsf{p} \vdash e' : \tau'$ and $\nexists \iota \,.\, \iota \in e'$). This prevents sending passive objects between actors without bestowing them first. Sending a message to a bestowed object will always relay it to the actor that owns the underlying passive object (by the premise of WF-ACTOR: $\forall \iota_{id} \in e \,.\, \iota \in \mathcal{LH}(H(id))$). Preservation of the second property is simple to show since local heaps grow monotonically, and are only ever extended with fresh locations (EVAL-NEW-PASSIVE).

Having made these observations, it is trivial to see that an actor in a well-formed heap H that is about to execute ι.`mutate()` must have ι in its own local heap. If another actor is about to execute ι'.`mutate()`, ι' must be in the local heap of this actor. As the local heaps are disjoint, ι and ι' must be different. Since well-formedness of heaps are preserved by evaluation, all programs are free from data-races.

5 Related Work

An important property of many actor-based systems is that a single actor can be reasoned about sequentially; messages are exchanged concurrently but executed sequentially by the receiving actor. For this property to hold, actors often rely on *actor isolation* [10], *i.e.,* that the state of one actor cannot be accessed by another. If this was the not the case, concurrent updates to shared state could lead to data-races, breaking sequential reasoning.

Existing techniques for achieving actor isolation are often based on restricting aliasing, for example copying all data passed between actors [2], or relying on linear types to transfer ownership of data [3, 5, 6, 10]. Bestowed objects offer an alternative technique which relaxes actor isolation and allows sharing of data without sacrificing sequential reasoning. Combining bestowed objects with linear types is straightforwand and allows for both ownership transfer and bestowed sharing between actors in the same system.

Miller *et al.*propose a programming model based on function passing, where rather than passing data between concurrent actors, functions are sent to collections of stationary and immutable data called *silos* [7]. Bestowed objects are related in the sense that sharing them doesn't actually move data between actors. In the function passing model, they could be used to provide an interface to some internal part of a silo, but implicitly relay all functions passed to it to its owning silo. While the formalism in § 4 also works by passing functions around, this is to abstract away from unimportant details, and not a proposed programming model.

References to bestowed objects are close in spirit to remote references in distributed programming or eventual references in E [8]. In the latter case, the unit of encapsulation, *e.g.,* an actor or an aggregate object protected by a lock, acts similar to a Vat in E, but with an identifiable boundary and an identity with an associated interface. By bestowing and exposing sub-objects, a unit of encapsulation can safely delegate parts of its interface to its inner objects, which in turn need not be internally aware of the kind of concurrency control offered by their bestower.

6 Discussion

Although our formal description and all our examples focus on actors, **bestow** also works with threads and locks. An object protected by a lock can share one of its internal objects while requiring that any interaction with this object also goes via this lock. We believe there is also a straightforward extension to software transactional memory. In the future, we would like to study combinations of these.

Bestowed objects lets an actor expose internal details about its implementation. Breaking encapsulation should always be done with care as leaking abstractions leads to increased coupling between modules and can lead to clients observing internal data in an inconsistent state. The latter is not a problem for bestowed objects however; interactions with bestowed objects will be synchronised in the owning actor's message queue, so as long as data is always consistent *between* messages, we can never access data in an inconsistent state (if your data is inconsistent between messages, you have a problem with or without bestowed objects).

Sharing bestowed objects may increase contention on the owner's message queue as messages to a bestowed object are sent to its owner. Similarly, since a bestowed object is protected by the same lock as its owner, sharing bestowed objects may lead to this lock being polled more often. As always when using locks there is a risk of introducing deadlocks, but we do not believe that bestowed objects exacerbate this problem. Deadlocks caused by passing a bestowed object back to its owner can be easily avoided by using reentrant locks (as accessing them both would require taking the same lock twice).

When using locks, **atomic** blocks are very similar to Java's **synchronized**-blocks. With actors, an **atomic** block groups messages into a single message. For fairness, it may make sense to only allow **atomic** blocks that send a limited number of messages.

It is possible to synchronise on several locked objects by simply grabbing several locks. Synchronising on several actors is more involved, as it requires actors to wait for each other and communicate their progress so that no actor starts or finishes before the others. The canonical example of this is atomically withdrawing and depositing the same amount from the accounts of two different actors. Interestingly, if the accounts are bestowed objects from the same actor (*e.g.,* some bank actor), this atomic transaction can be implemented with the message batching approach suggested in this paper. We leave this for future work.

6.1 Implementation

We are currently working on implementing bestowed objects and **atomic** blocks in the context of Encore [3], which uses active objects for concurrency. In Encore, each object (passive or active) has an interface defined by its class, and only the methods defined therein may be invoked. Thus it does not follow the formal model from § 4, where message passing is implemented by sending anonymous functions. It does however use the same approach for the implementation of bestowed objects and **atomic** blocks.

We extend each active class with an implicit method `perform` which takes a function, applies it to the **this** of the receiver, and returns the result wrapped in a future. A bestowed object is logically implemented as an object with two fields `owner` and `object`. A message send `x ! foo()` to a bestowed object is translated into the message send `x.owner ! perform((λ _ . x.object.foo()))`.

The **atomic** block can be implemented as sketched in the end of § 3, where messages are batched and sent as a single message:

```
atomic x <- e
  x ! foo(42)   ⟹  e ! perform(λ this . {this.foo(42); this.bar(-42)})
  x ! bar(-42)
```

This implementation works for the use-cases discussed here, but is somewhat limiting as it doesn't allow the caller to react to intermediate values. We are therefore exploring an alternative approach where we temporarily switch the message queue of an active object to one that only the caller can submit messages to. Other messages passed to the active object will end up in the original message queue, and will be processed first when the **atomic** block finishes.

Each active object would implicitly be extended with two methods `override`, which switches the current message queue to a new one, and `resume`, which discards the temporary queue and resumes execution with the original queue. Logically, the translation could look like this:

```
atomic x <- e                          val q = new MessageQueue()
  val v1 = x ! foo(42)                 e ! override(q) // 1
  val v2 = this.bar(v1)          ⟹    val v1 = q.enqueue(("foo", [42]))
  x ! baz(v2)                          val v2 = this.bar(v1)
                                       q.enqueue(("baz", [v2]))
                                       q.enqueue(("resume", [])) // 2
```

When the message at `1` is processed by receiver, it stops reading from its regular message queue and instead starts using the queue provided by the caller. Rather than sending messages normally, the caller interacts with `x` through this queue (waiting for responses if necessary). When the message at `2` has been processed by the receiver, it goes back to reading messages normally.

6.2 Abstracting Over Synchronisation Methods

Finally, we note the connection to the **safe** type qualifier introduced by the Kappa type system [4], which ranges over both actors and locks (and immutables etc.). A value with a **safe** type can be accessed concurrently without risk of data-races, but how this is achieved depends on the type of the value at runtime. Let `x` have the type **safe** τ. Now, `z = x.foo()` is equivalent to `z = x!foo().get()` when `x` is an actor returning a future value, and `get()` is a blocking read on the future. When `x` is protected by a lock `l`, the same access is equivalent to `lock(l); z = x.foo(); unlock(l);`. When `x` is immutable, no special synchronisation is needed.

Consequently, the **safe** qualifier can be used to express operations on objects with concurrency control abstracted out, without losing safety. An **atomic** block can be used to atomically compose operations on a **safe** object, and the choice of concurrency control mechanism can be relegated to the runtime. Similarly, bestowed objects internally has no knowledge about their own concurrency control. Thus, when a bestowed object is used as a **safe** object, neither the object itself nor its client needs knows how the interaction is made safe.

7 Conclusion

Actor isolation is important to maintain sequential reasoning about actors' behavior. By bestowing activity on its internal objects, an actor can share its representation without losing sequential reasoning and without bloating its own interface. With **atomic** blocks, a client can create new behavior by composing smaller operations. The bestowed objects themselves do not need to know why access to them is safe. They can just trust the safety of living in a world where actors have no borders.

References

[1] G. Agha (1986): *Actors: a Model of Concurrent Computation in Distributed Systems, Series in Artificial Intelligence. MIT Press* 11.

[2] J. Armstrong (2007): *A History of Erlang.* In: *HOPL III*, doi:10.1145/1238844.1238850.

[3] S Brandauer et al. (2015): *Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore.* In: *Formal Methods for Multicore Programming*, doi:10.1007/978-3-319-18941-3_1.

[4] E. Castegren & T. Wrigstad (2016): *Reference Capabilities for Concurrency Control.* In: *ECOOP*, doi:10.4230/LIPIcs.ECOOP.2016.5.

[5] S. Clebsch, S. Drossopoulou, S. Blessing & A. McNeil (2015): *Deny Capabilities for Safe, Fast Actors.* In: *AGERE*, doi:10.1145/b2824815.2824816.

[6] P. Haller & M. Odersky (2010): *Capabilities for Uniqueness and Borrowing.* In: *ECOOP*, doi:10.1007/978-3-642-14107-2_17.

[7] Heather Miller, Philipp Haller, Normen Müller & Jocelyn Boullier (2016): *Function Passing: A Model for Typed, Distributed Functional Programming.* In: *Onward!*, doi:10.1145/2986012.2986014.

[8] M. Miller (2006): *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. thesis, Johns Hopkins University, USA.

[9] P. Müller (2002): *Modular Specification and Verification of Object-oriented Programs.* Springer-Verlag, Berlin, Heidelberg, doi:10.1007/3-540-45651-1.

[10] S. Srinivasan & A. Mycroft (2008): *Kilim: Isolation-Typed Actors for Java.* In: *ECOOP*, doi:10.1007/978-3-540-70592-5_6.

Quantifying and Explaining Immutability in Scala

Philipp Haller
KTH Royal Institute of Technology
Stockholm, Sweden
phaller@kth.se

Ludvig Axelsson
KTH Royal Institute of Technology
Stockholm, Sweden
ludvigax@kth.se

Functional programming typically emphasizes programming with first-class functions and immutable data. Immutable data types enable fault tolerance in distributed systems, and ensure process isolation in message-passing concurrency, among other applications. However, beyond the distinction between reassignable and non-reassignable fields, Scala's type system does not have a built-in notion of immutability for type definitions. As a result, immutability is "by-convention" in Scala, and statistics about the use of immutability in real-world Scala code are non-existent.

This paper reports on the results of an empirical study on the use of immutability in several medium-to-large Scala open-source code bases, including Scala's standard library and the Akka actor framework. The study investigates both shallow and deep immutability, two widely-used forms of immutability in Scala. Perhaps most interestingly, for type definitions determined to be mutable, explanations are provided for why neither the shallow nor the deep immutability property holds; in turn, these explanations are aggregated into statistics in order to determine the most common reasons for why type definitions are mutable rather than immutable.

1 Introduction

Immutability is an important property of data types, especially in the context of concurrent and distributed programming. For example, objects of immutable type may be safely shared by concurrent processes without the possibility of data races. In message-passing concurrency, sending immutable messages helps ensure process isolation. Finally, in distributed systems immutability enables efficient techniques for providing fault tolerance.

Scala's type system does not have a built-in notion of immutability for type definitions. Instead, immutability is "by-convention" in Scala. In addition, statistics about the use of immutability in real-world Scala code are non-existent. This is problematic, since such statistics could inform extensions of Scala's type system for enforcing immutability properties.

Contributions This paper presents the first empirical results evaluating the prevalence of immutability in medium-to-large open-source Scala code bases, including the Scala standard library and the Akka actor framework [7]. We considered three different immutability properties, all of which occur frequently in all our case studies. In addition, we provide empirical results, evaluating causes for mutability of type definitions.

2 Immutability Analysis

This paper uses a notion of immutability that applies to *type definitions* rather than object references as in other work [11, 3]. For example, the definition of an immutable class implies that all its instances

V.T. Vasconcelos and P. Haller (Eds.): Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES'17)
EPTCS 246, 2017, pp. 21–27, doi:10.4204/EPTCS.246.5

are immutable. We refer to class, trait, and object definitions collectively as *templates*, following the terminology of the Scala language specification [9].

We distinguish three different immutability properties: (a) deep immutability, (b) shallow immutability, and (c) conditional deep immutability. Deep immutability is the strongest property; it requires that none of the declared or inherited fields is reassignable, and that the types of all declared or inherited fields are deeply immutable. Shallow immutability requires that none of the parents is mutable and that none of the declared or inherited fields is reassignable. Conditional deep immutability requires that none of the declared or inherited fields is reassignable, and that the types of all declared or inherited fields are deeply immutable, unless they are abstract types. For example, the type parameter `T` of the generic class `Option[T]` is abstract; type `T` is unknown within the definition of type `Option[T]`. Similarly, a Scala abstract type member [1] is treated as an abstract type. Finally, a class that declares or inherits a reassignable field (a Scala `var`) is *mutable*.

2.1 Implementation

We implement our analysis as a compiler plugin for Scala 2.11.x.[1] The plugin can be enabled when building Scala projects using the sbt or Maven build tools. The immutability analysis is implemented using Reactive Async [4] which extends LVars [6], lattice-based variables, with cyclic dependency resolution. For each template definition we maintain a "cell" that keeps track of the immutability property of the template. The value of the cell is taken from an immutability lattice; the analysis may update cell values monotonically according to the immutability lattice, based on evidence found during the analysis. For example, the cell value of a subclass is updated to `Mutable` when the analysis detects that one of the superclasses is mutable. Initially, all templates are assumed to be deeply immutable; this assumption is then updated incrementally based on evidence found by the analysis.

3 Empirical Study

We evaluate the prevalence of the immutability properties defined in Section 2 in four medium-to-large Scala open-source projects: Scala's standard library (version 2.11.8), Akka's actor package (version 2.4.17), ScalaTest (version 3.0.1), and Signal/Collect (version 8.0.2).

The Scala standard library consists of 33107 source lines of code (excluding blank lines and comments).[2] The library includes an extensive collection package [8] with both mutable and immutable collection types, as well as math, I/O, and concurrency packages such as futures [5]. Certain packages are designed to only define immutable types, including package `scala.collection.immutable` and package `scala.collection.parallel.immutable`. Other packages are designed to define mutable types, including packages `scala.collection.mutable`, `scala.collection.concurrent`, and `scala.collection.parallel.mutable`.

Akka's actor package is the standard actor implementation for Scala. ScalaTest [2] is the most widely-used testing framework for Scala. Signal/Collect [10] is a distributed graph processing framework based on Akka.

Our empirical study aims to answer the following two main research questions:

RQ1 How frequent is each immutability property for classes, traits, and objects?

[1] See `https://github.com/luax/scala-immutability-plugin`

[2] Measured using cloc v1.70, see `https://github.com/AlDanial/cloc`

Template	Occurrences	Mutable	Shallow	Deep	Cond. Deep
Class	626 (33,5%)	330 (52,7%)	54 (8,6%)	124 (19,8%)	118 (18,8%)
Case class	75 (4,0%)	19 (25,3%)	7 (9,3%)	9 (12,0%)	40 (53,3%)
Anon. class	330 (17,7%)	209 (63,3%)	26 (7,9%)	95 (28,8%)	0 (0%)
Trait	466 (25,0%)	224 (48,1%)	15 (3,2%)	93 (20,0%)	134 (28,8%)
Object	358 (19,2%)	106 (29,6%)	29 (8,1%)	223 (62,3%)	0 (0%)
Case object	12 (0,6%)	3 (25,0%)	0 (0%)	9 (75,0%)	0 (0%)
Total	**1867 (100,0%)**	**891 (47,7%)**	**131 (7,0%)**	**553 (29,6%)**	**292 (15,6%)**

Table 1: Immutability statistics for Scala standard library.

Template	Occurrences	Mutable	Shallow	Deep	Cond. Deep
Class	299 (26,8%)	115 (38,5%)	93 (31,1%)	82 (27,4%)	9 (3,0%)
Case class	206 (18,4%)	23 (11,2%)	64 (31,1%)	90 (43,7%)	29 (14,1%)
Anon. class	77 (6,9%)	33 (42,9%)	8 (10,4%)	36 (46,8%)	0 (0%)
Trait	239 (21,4%)	22 (9,2%)	17 (7,1%)	140 (58,6%)	60 (25,1%)
Object	220 (19,7%)	9 (4,1%)	47 (21,4%)	164 (74,5%)	0 (0%)
Case object	76 (6,8%)	2 (2,6%)	0 (0%)	74 (97,4%)	0 (0%)
Total	**1117 (100,0%)**	**204 (18,3%)**	**229 (20,5%)**	**586 (52,5%)**	**98 (8,8%)**

Table 2: Immutability statistics for Akka (`akka-actor` package).

RQ2 For classes/traits/objects that are not deeply immutable: what are the most common reasons why stronger immutability properties are not satisfied?

3.1 Research Question 1

Tables 1 shows the immutability statistics for Scala's standard library. One of the most important results is that *the majority of classes/traits/objects in Scala's standard library satisfy one of the immutability properties.* This confirms the intuition that functional programming with immutable types is an important programming style in Scala. Interestingly, the most common immutability property for case classes and traits is conditional deep immutability. Thus, *whether a case class or trait is deeply immutable in most cases depends on the instantiation of type parameters or abstract types.* In contrast, the majority of classes that are not case classes is mutable. Note that objects and anonymous classes cannot be conditionally deeply immutable, since these templates cannot have type parameters or abstract type members.

Table 2 shows the immutability statistics for Akka. The percentage of mutable classes/traits/objects is significantly lower compared to Scala's standard library (18.3% for Akka versus 47.7% for the Scala library).

Table 4 shows the immutability statistics for Signal/Collect. Unique to Signal/Collect is the high percentage of mutable singleton objects (46.3%), which ranges between 4.1% (Akka) and 29.6% (Scala library). However, also in Signal/Collect is the percentage of mutable case classes low compared to other kinds of templates.

Summary In our case studies, the majority of classes/traits/objects satisfy one of our immutability properties. The prevalence of mutability is especially low for case classes (with structural equality)

Template	Occurrences	Mutable	Shallow	Deep	Cond. Deep
Class	791 (36,1%)	216 (27,3%)	249 (31,5%)	288 (36,4%)	38 (4,8%)
Case class	153 (7,0%)	15 (9,8%)	81 (52,9%)	54 (35,3%)	3 (2,0%)
Anon. class	688 (31,4%)	200 (29,1%)	293 (42,6%)	195 (28,3%)	0 (0%)
Trait	227 (10,3%)	61 (26,9%)	45 (19,8%)	91 (40,1%)	30 (13,2%)
Object	254 (11,6%)	19 (7,5%)	18 (7,1%)	217 (85,4%)	0 (0%)
Case object	81 (3,7%)	2 (2,5%)	0 (0%)	79 (97,5%)	0 (0%)
Total	**2194 (100,0%)**	**513 (23,4%)**	**686 (31,3%)**	**924 (42,1%)**	**71 (3,2%)**

Table 3: Immutability statistics for ScalaTest.

Template	Occurrences	Mutable	Shallow	Deep	Cond. Deep
Class	160 (58,0%)	78 (48,8%)	24 (15,0%)	14 (8,8%)	44 (27,5%)
Case class	42 (15,2%)	4 (9,5%)	11 (26,2%)	15 (35,7%)	12 (28,6%)
Anon. class	4 (1,4%)	4 (100,0%)	0 (0%)	0 (0%)	0 (0%)
Trait	24 (8,7%)	6 (25,0%)	1 (4,2%)	3 (12,5%)	14 (58,3%)
Object	41 (14,9%)	19 (46,3%)	5 (12,2%)	17 (41,5%)	0 (0%)
Case object	5 (1,8%)	0 (0%)	0 (0%)	5 (100,0%)	0 (0%)
Total	**276 (100,0%)**	**111 (40,2%)**	**41 (14,9%)**	**54 (19,6%)**	**70 (25,4%)**

Table 4: Immutability statistics for Signal/Collect.

Reason	Immutability Property	Attribute Key
Parent type mutable (assumption)	Mutable	A
Parent type mutable	Mutable	B
Reassignable field (public)	Mutable	C
Reassignable field (private)	Mutable	D
Parent type unknown	Mutable	E
Parent type shallow immutable	Shallow immutable	F
`val` field with unknown type	Shallow immutable	G
`val` field with mutable type	Shallow immutable	H
`val` field with mutable type (assumption)	Shallow immutable	I

Table 5: Template attributes and their influence on immutability properties.

and singleton objects. Except for Signal/Collect, which is unique in this case, the majority of singleton objects are deeply immutable, ranging between 62.3% and 85.4% in our case studies. The percentage of deeply immutable *case objects* is even higher, ranging between 75% and 100%, including Signal/Collect.

In order to answer RQ2, we identified nine template *attributes*, shown in Table 5, which explain why certain immutability properties cannot be satisfied. The presence of the first five attributes forces the corresponding template to be classified as mutable. For example, a template is classified as mutable if it declares a reassignable field (attributes C and D). The last four attributes prevent the corresponding template from satisfying either deep of conditionally deep immutability. For example, if a parent class or trait is only shallow immutable (but not deeply immutable), then the corresponding template cannot be deeply immutable or conditionally deeply immutable either (attribute F).

Attribute(s)	Occurrences
B	609 (68,4%)
B C	71 (8,0%)
B C D	1 (0,1%)
B D	19 (2,1%)
B E	7 (0,8%)
C	26 (2,9%)
C D	1 (0,1%)
D	87 (9,8%)
D E	4 (0,4%)
E	66 (7,4%)

Table 6: Scala library: attributes causing mutability.

Attribute(s)	Occurrences
F	28 (21,4%)
F G	5 (3,8%)
F G H	1 (0,8%)
F H	4 (3,1%)
F J	6 (4,6%)
G	22 (16,8%)
G H	4 (3,1%)
G H J	3 (2,3%)
G J	2 (1,5%)
H	40 (30,5%)
H J	3 (2,3%)
J	7 (5,3%)

Table 7: Scala library: attributes causing shallow immutability (instead of deep immutability).

3.2 Research Question 2

Tables 6 and 7 show the causes for mutability and shallow immutability, respectively, for the Scala library. The main cause for a template to be classified as mutable is the existence of a parent which is mutable. Important causes for templates to be classified as shallow immutable rather than deeply immutable are (a) the existence of a non-reassignable field with a mutable type (attribute H), and (b) the existence of a parent which is shallow immutable (attribute F).

Tables 8 and 9 show the causes for mutability and shallow immutability, respectively, for Akka actors. The main cause for a template to be classified as mutable is the existence of a parent which is mutable; this matches the statistics of the Scala library. Other important causes are (a) parent types whose immutability is unknown (*e.g.*, due to third-party libraries for which no analysis results are available) and (b) private reassignable fields. Unlike the Scala library, the most important cause for shallow immutability (rather than deep immutability) in Akka are non-reassignable fields of a type whose immutability is unknown; this suggests that the absence of analysis results for third-party libraries has a significant impact on the classification of a type as shallow immutable rather than deeply immutable. On the other hand, this means that the actual percentage of deeply immutable templates may be even higher. Therefore, an important avenue for future work is to enable the analysis of third-party libraries. The second most important cause is the existence of a parent which is shallow immutable (attribute F).

4 Conclusion

Immutability is an important property of data types, especially in the context of concurrent and distributed programming. For example, objects of immutable type may be safely shared by concurrent processes without the possibility of data races. In message-passing concurrency, sending immutable messages helps ensure process isolation. In this paper we presented the first empirical results evaluating the prevalence of immutability in medium-to-large open-source Scala code bases, including the Scala standard library and the Akka actor framework. We considered three different immutability properties,

Attribute(s)	Occurrences
A	3 (1,5%)
A B D	1 (0,5%)
A E	1 (0,5%)
B	76 (37,3%)
B C	3 (1,5%)
B C D	1 (0,5%)
B D	6 (2,9%)
B E	6 (2,9%)
C	7 (3,4%)
C D	2 (1,0%)
C D E	1 (0,5%)
D	24 (11,8%)
D E	1 (0,5%)
E	72 (35,3%)

Table 8: Akka: attributes causing mutability.

Attribute(s)	Occurrences
F	38 (16,6%)
F G	9 (3,9%)
F G H	2 (0,9%)
F G J	3 (1,3%)
F H	3 (1,3%)
F J	3 (1,3%)
G	94 (41,0%)
G H	8 (3,5%)
G H I	1 (0,4%)
G H J	1 (0,4%)
G J	16 (7,0%)
H	22 (9,6%)
H J	4 (1,7%)
J	25 (10,9%)

Table 9: Akka: attributes causing shallow immutability (instead of deep immutability).

all of which occur frequently in all our case studies. In our case studies, the majority of classes/traits/objects satisfy one of our immutability properties. The prevalence of mutability is especially low for case classes (classes with structural equality) and singleton objects. The most important causes for mutability are mutable parent classes and private reassignable fields. To our knowledge we presented the first empirical study of its kind. We believe our insights are valuable both for informing the further evolution of the Scala language, and for designers of new wide-spectrum languages, combining functional and imperative features.

References

[1] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf & Sandro Stucki (2016): *The Essence of Dependent Object Types*. In: *A List of Successes That Can Change the World*, Springer, pp. 249–272, doi:10.1007/978-3-319-30936-1_14.

[2] Artima, Inc. (2009): *ScalaTest*. http://www.scalatest.org.

[3] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield & Joe Duffy (2012): *Uniqueness and reference immutability for safe parallelism*. In: *OOPSLA*, ACM, pp. 21–40, doi:10.1145/2384616.2384619.

[4] Philipp Haller, Simon Geries, Michael Eichberg & Guido Salvaneschi (2016): *Reactive Async: Expressive Deterministic Concurrency*. In: *ACM SIGPLAN Scala Symposium*, ACM, pp. 11–20, doi:10.1145/2998392.2998396.

[5] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn & Vojin Jovanovic (2012): *Futures and promises*. http://docs.scala-lang.org/overviews/core/futures.html.

[6] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami & Ryan R. Newton (2014): *Freeze after writing: quasi-deterministic parallel programming with LVars*. In: *POPL*, ACM, pp. 257–270, doi:10.1145/2535838.2535842.

[7] Lightbend, Inc. (2009): *Akka*. http://akka.io/.

[8] Martin Odersky & Adriaan Moors (2009): *Fighting bit Rot with Types (Experience Report: Scala Collections)*. In: *FSTTCS, LIPIcs* 4, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 427–451, doi:10.4230/LIPIcs.FSTTCS.2009.2338.

[9] Martin Odersky et al. (2014): *The Scala Language Specification Version 2.11*. Available at `http://www.scala-lang.org/files/archive/spec/2.11/`.

[10] Philip Stutz, Abraham Bernstein & William W. Cohen (2010): *Signal/Collect: Graph Algorithms for the (Semantic) Web*. In: *ISWC*, Springer, pp. 764–780, doi:10.1007/978-3-642-17746-0_48.

[11] Matthew S. Tschantz & Michael D. Ernst (2005): *Javari: adding reference immutability to Java*. In: *OOPSLA*, ACM, pp. 211–230, doi:10.1145/1094811.1094828.

Inferring Types for Parallel Programs

Francisco Martins
LaSIGE, Faculty of Sciences, University of Lisbon

Vasco Thudichum Vasconcelos
LaSIGE, Faculty of Sciences, University of Lisbon

Hans Hüttel
Aalborg Universitet

The Message Passing Interface (MPI) framework is widely used in implementing imperative programs that exhibit a high degree of parallelism. The PARTYPES approach proposes a behavioural type discipline for MPI-like programs in which a type describes the communication protocol followed by the entire program. Well-typed programs are guaranteed to be exempt from deadlocks. In this paper we describe a type inference algorithm for a subset of the original system; the algorithm allows to statically extract a type for an MPI program from its source code.

1 Introduction

Message Passing Interface (MPI) has become generally accepted as the standard for implementing massively parallel programs. An MPI program is composed of a fixed number of processes running in parallel, each of which bears a distinct identifier—a *rank*—and an independent memory. Process behaviour may depend on the value of the rank. Processes call MPI primitives in order to communicate. Different forms of communication are available to processes, including *point-to-point* message exchanges and *collective* operators such as broadcast.

Parallel programs use the primitives provided by MPI by issuing calls to a dedicated application program interface. As such the level of verification that can be performed at compile time is limited to that supported by the host language. Programs that compile flawlessly can easily stumble into different sorts of errors, that may or not may be caught at runtime. Errors include processes that exchange data of unexpected types or lengths, and processes that enter deadlocked situations. The state of the art on the verification of MPI programs can only address this challenge partially: techniques based on runtime verification are as good as the data the programs are run with; strategies based on model checking are effective only in verifying programs with a very limited number of processes. We refer the reader to Gopalakrishnan et al. [2] for a discussion on the existing approaches to the verification of MPI programs.

PARTYPES is a type-based methodology for the analysis of C programs that use MPI primitives [7, 9]. Under this approach, a type describes the protocol to be followed by some program. Types include constructors for point-to-point messages, e.g. `message from to float[]`, and constructors for collective operations, e.g. `allreduce min integer`. Types can be further composed via sequential composition and primitive recursion, an example being `foreach i: 1..9 message 0 i`. *Datatypes* describe values exchanged in messages and in collective operations, and include `integer` and `float`, as well as support for arrays `float[]` and for refinement types that equip types with refinement conditions, an example being `{v:integer|v>0}`. Index-dependent types allow for protocols to depend on values exchanged in messages; an example of this is `allreduce min x:{v:integer|1<=v<=9}.message 0 x`. Our notion of refinement types is inspired by Xi and Pfenning [11], where datatypes are restricted by indices drawn from a decidable domain.

V.T. Vasconcelos and P. Haller (Eds.): Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES'17)
EPTCS 246, 2017, pp. 28–36, doi:10.4204/EPTCS.246.6

The idea of describing a protocol by means of a type is inspired by multiparty session types (MPST), introduced by Honda et al. [5]. MPST feature a notion of global types describing, from a all-inclusive point of view, the interactions all processes engage upon. A projection operation extracts from a global type the local type of each individual participant. ParTypes departs from MPST in that it does not distinguish between local and global types. Instead the notion of types is equipped with a flexible equivalence relation. Projection can be recovered by type equivalence in the presence of knowledge about process ranks, e.g., `rank:{x:integer|x=2}` $\vdash$ **`message 0 1 integer`** $\equiv$ `skip`, where `skip` describes the empty interaction.

The type equivalence relation is at the basis of our strategy for type reconstruction:

- We analyse the source code for each individual process, extracting (inferring) for each process a type that governs that individual process;
- We then gradually merge the thus obtained types, while maintaining type equivalence.

This approach is related to that of Carbone and Montesi [1], where several choreographies are merged into a single choreography, and to the work of Lange and Scalas [6] where a global type is constructed from a collection of contracts.

Typable programs are assured to behave as prescribed by the type, exchanging messages and engaging in collective operations as detailed in the type. Moreover, programs that can be typed are assured to be deadlock free [7]. As such, programs that would otherwise deadlock cannot be typed, implying that the inference procedure will fail in such cases, rendering the program untypable.

2 The *n*-body pipeline and its type

We base our presentation on a classical problem on parallel programming. The *n*-body pipeline computes the trajectories of *n* bodies that influence each other through gravitational forces. The algorithm computes the forces between all pairs of bodies, applying a pipeline technique to distribute and balance the work on a parallel architecture. It then determines the bodies' positions [4].

The program in Figure 1 implements this algorithm. Each body (henceforth called particle) is represented by a quadruple of `floats` consisting of a 3D position and a mass. The program starts by connecting to the MPI middleware (line 15), and then obtains the number of available processes and its own process number, which it stores in variables `size` and `rank` (lines 16–17). The overall idea of the program is as follows: (a) each process starts by obtaining a portion of the total number of particles, `MAX_PARTICLES`, and computes the trajectories (line 19). Then, (b) each process enters a loop that computes `NUM_ITER` discrete steps. In each iteration (c) the algorithm computes the forces between all pairs of particles. It accomplishes this in two phases: (c.1) compute the forces among its own particles (lines 22–23), and (c.2) compute the forces between its particles and those from the neighbour processes (lines 25–36). Towards this end, each process passes particles to the right process and receives new particles from the left (lines 26–32). Then it compute the forces against the particles received (line 33–34). After `size-1` steps all processes have visited all particles. Then, (d) each process computes the position of its particles (line 37), which results in the computation of a local time differential (`dt_local`), and (e) updates the simulation time (`sim_t`).

The simulation time is incremented by the minimum of the local time differentials of all processes. In order to obtain this value, each process calls an `MPI_Allreduce` operation (line 38). This collective operation takes the contribution of each individual process (`dt_local`), computes its minimum (`MPI_MIN`), and distributes it to all processes (`dt`). The minimum is then added to the simulation time (line 39). The program terminates by disconnecting from the MPI middleware (line 41).

```
1  #define MAX_PARTICLES 10000
2  #define NUM_ITER      5000000
3
4  void InitParticles(float* part, float* vel, int npart);
5  float ComputeForces(float* part,  float* other_part, float* vel, int npart);
6  float ComputeNewPos(float* part, float* pv, int npart, float);
7
8  int main(int argc,char** argv) {
9    int  rank, size, iter, pipe, i;
10   float sim_t, dt, dt_local, max_f, max_f_seg;
11   float particles[MAX_PARTICLES * 4];   /* Particles on all nodes */
12   float pv[MAX_PARTICLES * 6];   /* Particle velocity */
13   float send_parts[MAX_PARTICLES * 4], recv_parts[MAX_PARTICLES * 4]; /* Particles from other processes */
14
15   MPI_Init(&argc, &argv);
16   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
17   MPI_Comm_size(MPI_COMM_WORLD, &size);
18
19   InitParticles(particles, pv, MAX_PARTICLES / size);
20   sim_t = 0.0f;
21   for (iter = 1; iter <= NUM_ITER; iter++) {
22     max_f_seg = ComputeForces(particles, particles, pv, MAX_PARTICLES / size);
23     memcpy(send_parts, particles, MAX_PARTICLES / size * 4);
24     if (max_f_seg > max_f) max_f = max_f_seg;
25     for (pipe = 0; pipe < size - 1; pipe++) {
26       if (rank == 0) {
27         MPI_Send(send_parts, MAX_PARTICLES / size * 4, MPI_FLOAT, rank == size - 1 ?  0 : rank + 1, ...);
28         MPI_Recv(recv_parts, MAX_PARTICLES / size * 4, MPI_FLOAT, rank == 0 ? size - 1 : rank -1,  ...);
29       } else {
30         MPI_Recv(recv_parts, MAX_PARTICLES / size * 4, MPI_FLOAT, rank == 0 ? size - 1 : rank -1,  ...);
31         MPI_Send(send_parts, MAX_PARTICLES / size * 4, MPI_FLOAT, rank == size - 1 ?  0 : rank + 1, ...);
32       }
33       max_f_seg = ComputeForces(particles, recv_parts, pv, MAX_PARTICLES / size);
34       if (max_f_seg > max_f) max_f = max_f_seg;
35       memcpy(send_parts, recv_parts, MAX_PARTICLES / size * 4);
36     }
37     dt_local = ComputeNewPos(particles, pv, MAX_PARTICLES / size, max_f);
38     MPI_Allreduce(&dt, &dt_local, 1, MPI_FLOAT, MPI_MIN, ...);
39     sim_t += dt;
40   }
41   MPI_Finalize();
42   return 0;
43 }
```

Figure 1: Excerpt of an MPI program for the n-body pipeline problem (adapted from [3])

Communication is performed on a ring communication topology. The conditional statement within the loop (lines 26–32) breaks the communication circularity. Because operations `MPI_Send` and `MPI_Recv` implement *synchronous* message passing, a completely symmetrical solution would lead to a deadlock with all processes trying to send messages and no process ready to receive.

From this discussion it should be easy to see that the communication behaviour of 3-body pipeline can be described by the protocol (or type) in Figure 2. The rest of this abstract describes a method to infer the type in Figure 2 from the source code in Figure 1.

3 The problem of type inference

Given a parallel program P composed of n processes (or expressions) $e_0, \ldots, e_{n-1}$, we would like to find a common type that types each process e_i, or else to decide there is no such type. We assume that size is the only free variable in processes, so that the typing context only needs an entry for this variable. We are then interested in a context where size is equal to n, which we write as $\mathsf{size}\colon \{x\colon \mathsf{int} \mid x = n\}$ and abbreviate to Γ^n. Our type inference problem is then to find a type T such that $\Gamma^n \vdash e_i : T$, or else decide that there is no such type.

```
1    foreach iter: 1..5000000
2      foreach pipe: 1..2
3          message 0 1 float[1000000 / 3 * 4];
4          message 1 2 float[1000000 / 3 * 4];
5          message 2 0 float[1000000 / 3 * 4]
6     allreduce min float
```

Figure 2: Protocol for the parallel n-body algorithm with three processes

We propose approaching the problem in two steps:

1. From the source code e_i of each individual process extract a type T_i such that $\Gamma^n \vdash e_i : T_i$;
2. From types $T_0, \ldots, T_{n-1}$ look for a type T that is equal to all such types, that is, $\Gamma^n \vdash T_i \equiv T$.

Then, from these two results, we conclude that $\Gamma^n \vdash e_i : T$, hence that $\Gamma^n \vdash P : T$, as required.
We approach the *first step* in a fairly standard way:

- Given an expression e_i, collect a system of equations $\mathscr{D}_i$ over datatypes and a type U_i;
- Solve $\mathscr{D}_i$ to obtain a substitution σ_i. We then have $\Gamma^n \vdash e_i : U_i\sigma_i$, as required for the first phase. If there is no such substitution, then e_i is not typable.

For this step we introduce variables over datatypes. Then we visit the syntax tree of each process and, guided by the typing rules [7], collect restrictions (in the form of a set of equations over datatypes) and a type for the expression. We need rules for expressions, index terms (the arithmetic in types), and propositions. We omit the rules for extracting a system of equations and a type from a given expression. Based on the works by Vazou et al. [10] and Rondon et al. [8], we expect the problem of solving a system of datatype equations to be decidable.

We address the second step in more detail. The goal is to build a type T from types $T_0, \ldots, T_{n-1}$. We start by selecting some type T_i and merge it with some other type T_j (for $i \neq j$) to obtain a new type. The thus obtained type is then merged with another type T_k ($k \neq j, i$), and so forth. The result of merging all the types is the sought type T. The original inference problem has no solution if one of the merge operations fail.

4 Merging types

We give an intuitive overview of the merge operation, discuss its rules and apply them to our running example. The intuition behind the merge operator is the following:

- messages must be matched exactly once by the sender and the receiver processes (the two end-points of the communication);
- collective operations (**`allreduce`**, for example) establish horizontal synchronisation lines among all processes, meaning that all processes must perform all communications (collective or not) before the synchronisation line, carry out the collective operation, and then proceed with the remainder of the protocol.

Having this in mind, the merge rules make sure that collective operations match each other and that messages are paired together before and after each collective operation.

The merge operation receives a typing context Γ, the type merged so far T, the type to be merged U and its rank k, to yield a new type V. We write all this as follows $\Gamma \vdash T \parallel_k U \rightsquigarrow V$. The typing context

$$\Gamma \vdash \mathsf{skip} \parallel_k \mathsf{skip} \rightsquigarrow \mathsf{skip} \qquad \text{(skip-skip)}$$

$$\frac{\Gamma \vdash i_3, i_4 \neq k \ \mathbf{true}}{\Gamma \vdash \mathsf{skip} \parallel_k \mathsf{message}\ i_3\ i_4\ D \rightsquigarrow \mathsf{skip}} \qquad \text{(skip-msgS)}$$

$$\frac{\Gamma \vdash i_1, i_2 \neq \mathsf{rank} \wedge i_3, i_4 \neq k \ \mathbf{true}}{\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D_1 \parallel_k \mathsf{message}\ i_3\ i_4\ D_2 \rightsquigarrow \mathsf{skip}} \qquad \text{(msgS-msgS)}$$

$$\frac{\Gamma \vdash (i_1 = \mathsf{rank} \vee i_2 = \mathsf{rank}) \wedge i_1, i_2 \neq k \ \mathbf{true}}{\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D_1 \parallel_k \mathsf{skip} \rightsquigarrow \mathsf{message}\ i_1\ i_2\ D_1} \qquad \text{(msg-skip)}$$

$$\frac{\Gamma \vdash i_3, i_4 \neq \mathsf{rank} \wedge (i_3 = k \vee i_4 = k) \ \mathbf{true}}{\Gamma \vdash \mathsf{skip} \parallel_k \mathsf{message}\ i_3\ i_4\ D_2 \rightsquigarrow \mathsf{message}\ i_3\ i_4\ D_2} \qquad \text{(skip-msg)}$$

$$\frac{\Gamma \vdash (i_1 = \mathsf{rank} \vee i_2 = \mathsf{rank}) \wedge (i_3 = k \vee i_4 = k) \wedge i_1 = i_3 \wedge i_2 = i_4 \ \mathbf{true} \qquad \Gamma \vdash D_1 \equiv D_2 : \mathbf{dtype}}{\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D_1 \parallel_k \mathsf{message}\ i_3\ i_4\ D_2 \rightsquigarrow \mathsf{message}\ i_1\ i_2\ D_1} \qquad \text{(msg-msg-eq)}$$

$$\frac{\Gamma \vdash (i_1 = \mathsf{rank} \vee i_2 = \mathsf{rank}) \wedge (i_3 = k \vee i_4 = k) \wedge i_1 \neq i_4 \wedge i_2 \neq i_3 \ \mathbf{true}}{\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D_1 \parallel_k \mathsf{message}\ i_3\ i_4\ D_2 \rightsquigarrow \mathsf{message}\ i_3\ i_4\ D_2; \mathsf{message}\ i_1\ i_2\ D_1} \qquad \text{(msg-msg-right)}$$

$$\frac{\Gamma \vdash D_1 \equiv D_2 : \mathbf{dtype} \qquad \Gamma, x\colon D_1 \vdash T_1 \parallel_k T_2 \rightsquigarrow T_3}{\Gamma \vdash \mathsf{allreduce}\ x\colon D_1.\ T_1 \parallel_k \mathsf{allreduce}\ x\colon D_2.\ T_2 \rightsquigarrow \mathsf{allreduce}\ x\colon D_1.\ T_3} \qquad \text{(allred-allred)}$$

$$\frac{\Gamma \vdash i_1 = i_2 \wedge i_1' = i_2' \ \mathbf{true} \qquad \Gamma, x\colon \{y\colon \mathsf{int} \mid i_1 \leq y \leq i_1'\} \vdash T_1 \parallel_k T_2 \rightsquigarrow T_3}{\Gamma \vdash \mathsf{foreach}\ x\colon i_1..i_1'.T_1 \parallel_k \mathsf{foreach}\ x\colon i_2..i_2'.T_2 \rightsquigarrow \mathsf{foreach}\ x\colon i_1..i_1'.T_3} \qquad \text{(foreach-foreach)}$$

$$\frac{\Gamma \vdash T_1 \parallel_k T_3 \rightsquigarrow T_5 \qquad \Gamma \vdash T_2 \parallel_k T_4 \rightsquigarrow T_6}{\Gamma \vdash T_1; T_2 \parallel_k T_3; T_4 \rightsquigarrow T_5; T_6} \qquad \text{(seq-seq)}$$

$$\frac{\Gamma \vdash (i_1 = \mathsf{rank} \vee i_2 = \mathsf{rank}) \wedge (i_3 = k \vee i_4 = k) \wedge i_1 \neq i_4 \wedge i_2 \neq i_3 \ \mathbf{true} \quad \Gamma \vdash T_1 \parallel_k \mathsf{message}\ i_3\ i_4\ D_2; T_2 \rightsquigarrow T_3}{\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D_1; T_1 \parallel_k \mathsf{message}\ i_3\ i_4\ D_2; T_2 \rightsquigarrow \mathsf{message}\ i_1\ i_2\ D_1; T_3} \qquad \text{(msgT-msgT-left)}$$

$$\frac{\Gamma \vdash \mathsf{skip} \parallel_k \mathsf{message}\ i_3\ i_4\ D \rightsquigarrow T_2 \quad \Gamma \vdash \mathsf{skip} \parallel_k T_1 \rightsquigarrow T_3}{\Gamma \vdash \mathsf{skip} \parallel_k \mathsf{message}\ i_4\ i_4\ D; T_1 \rightsquigarrow T_2; T_3} \qquad \text{(skip-msgT)}$$

Figure 3: Rules defining the merge partial function (excerpt)

contains entries for variables size and rank, the latter recording the ranks whose types have been merged. This context will then be updated with new entries arising from collective (dependently typed) operations, such as allreduce. An excerpt of rules defining the merge operation is in Figure 3.

We first discuss merging skip and message types. There are ten different cases that we group into the five categories detailed below. Notice that a message $i_1\ i_2\ D_1$ appearing as the left operand of a merge is equivalent to skip when both i_1 and i_2 are different from all ranks merged so far, which we write as $i_1, i_2 \neq \mathsf{rank}$. Otherwise, when $i_1 = \mathsf{rank}$ or $i_2 = \mathsf{rank}$, the message is the endpoint of a communication between ranks i_1 and i_2 that are already merged. When message $i_3\ i_4\ D_2$ appears as the right operand of a merge at rank k it is equivalent to skip when both i_3 or i_4 are not k, which we abbreviate as $i_3, i_4 \neq k$. Otherwise, when $i_3 = k$ or $i_4 = k$, the message is the endpoint of a communication with rank k. Rule names try to capture these concepts. For instance, rule skip-msgS merges skip (left operand) with a

message (right operand) that is semantically equivalent to skip, whereas rule skip-msg designates the merging of skip with a message that is not equivalent to skip. We proceed by analysing each category.

merge yields skip. In this case both operands are semantically equivalent to skip. This category comprises rules skip-skip, skip-msgS, msgS-skip (not shown), and msgS-msgS. We include the appropriate premises for enforcing that one or both parameters are equivalent to skip, depending on the message being the left or the right operand. For instance, rule skip-skip has no premises, while rule msgS-msgS includes two premises to make sure that both messages are equivalent to skip.

merge yields the left operand. In this category the left operand is not equivalent to skip, whereas the right operand is. It encompasses rules msg-skip and msg-msgS (not shown). Apart from the condition enforcing that the left message is not equivalent to skip ($i_1 = \mathsf{rank} \vee i_2 = \mathit{rank}$), rank k being merged must not be the source or the target of the message. Would this be the case and the program has a deadlock, since the messages on the left talk about rank k (either as a source or a target) and the type at rank k is skip (or equivalent to it), meaning that the merged messages will never be matched.

merge yields the right operand. In this case the left operand is semantically equivalent to skip, and the right operand is not. The category includes rules skip-msg and msgS-msg (not shown). The message is from or targeted at rank k ($i_3 = k \vee i_4 = k$). We also need to check that the other rank of the message (the source or target that is different from k) is still to be merged ($i_3, i_4 \neq \mathsf{rank}$). Why? Because otherwise the type of the other endpoint is already merged and is skip (the left operand), therefore the message at rank k (the right operand, which is not skip) is never going to be matched, indicating the program has a deadlock.

messages are the endpoints of the same communication. In this category (rule msg-msg-eq) the messages correspond to the two endpoints of a communication. The result of the merge is the left operand, which is semantically equivalent to the right one. No message is semantically equivalent to skip as witnessed by the premises. Additionally we need to check that the source and the target ranks, as well as the payload, of the two messages coincide.

messages are the endpoints of different communications. This last category includes messages that are the endpoints of two different communications. The result of the merge is an interleaving of the messages. The messages are semantically different from skip and are unrelated. The category includes rules msg-msg-left (not shown) and msg-msg-right. As in the previous category we check that no message is semantically equivalent to skip. Additionally, we check that the messages do not interfere, that is, that their ranks are not related. These two rules can be non-deterministically applied in an appropriate way to match the types.

There are no rules to merge messages against collective operations, since this is not admissible; the merging of messages against foreach loops is left for future work. Collective operations can only be merged against each other (cf. rule allred-allred). We omit the rules for other MPI collective operations for they follow a similar schema. In this paper we only merge foreach loops against foreach loops. Refer to the next section for a discussion about the challenges on this subject.

The last three rules apply to the sequential composition of types: rule seq-seq allows for types to be split at the sequential operator (;) and merged separately; rules msgT-msgT-left and msgT-msgT-right (not shown) allow for the non-deterministic ordering of unrelated messages, as described for rules msg-msg-left and msg-msg-right, but here at the level of the sequential composition of types. The last rule allows for messages after the last collective communication (if any) to be merged. For the sake of brevity, we also omit rules for the sequential composition of skip types.

We now outline how merging works on our running example. Fix **size** = 3. From the program in Figure 1 extract **size** programs, one per rank, in such a way that programs do not mention variable rank. We leave this to the reader.

Run the first step of our procedure on each program to obtain the three types below, where D is the datatype **float**[MAX_PARTICLES / **size** ∗ 4].

For rank 0:

```
foreach iter: 1..5000000
  foreach pipe: 1..2
    message 0 1 D;
    message 2 0 D
allreduce min float
```

For rank 1:

```
foreach iter: 1..5000000
  foreach pipe: 1..2
    message 0 1 D;
    message 1 2 D
allreduce min float
```

For rank 2:

```
foreach iter: 1..5000000
  foreach pipe: 1..2
    message 1 2 D;
    message 2 0 D
allreduce min float
```

Run the second step as follows. We only show the merging of the various messages; the cases of **foreach** and **allreduce** are of simple application.

We start by taking the type for the process at rank 0 and merge it with that of rank 1. The initial typing context Δ_1 says that the type on the left corresponds to rank 0 in a total of 3, which we write as $\mathsf{size}\colon \{x\colon \mathsf{int} \mid x = 3\}, \mathsf{rank}\colon \{x\colon \mathsf{int} \mid x = 0\}$. Using rules seq-seq, msg-msg-eq, and msg-msg-right we have:

$$\Delta_1 \vdash \begin{array}{l}\texttt{message 0 1 D;}\\ \texttt{message 2 0 D}\end{array} \;\texttt{||}_1\; \texttt{message 0 1 D;}\quad \rightsquigarrow \quad \begin{array}{l}\texttt{message 0 1 D;}\\ \texttt{message 1 2 D;}\\ \texttt{message 2 0 D}\end{array}$$

Then we merge the resulting type with that of rank 2. This time we need a typing context Δ_2 that records the fact that the type on the left corresponds to ranks 0 and 1. We write it as $\mathsf{size}\colon \{x\colon \mathsf{int} \mid x = 3\}, \mathsf{rank}\colon \{x\colon \mathsf{int} \mid x = 0 \vee x = 1\}$. Using rules msgT-msgT-left, seq-seq, msg-msg-eq (x2), we get:

$$\Delta_2 \vdash \begin{array}{l}\texttt{message 0 1 D;}\\ \texttt{message 1 2 D;}\\ \texttt{message 2 0 D}\end{array} \;\texttt{||}_2\; \texttt{message 1 2 D;}\quad \rightsquigarrow \quad \begin{array}{l}\texttt{message 0 1 D;}\\ \texttt{message 1 2 D;}\\ \texttt{message 2 0 D}\end{array}$$

The type obtained is that of Figure 2.

5 Discussion

The procedure outlined in this paper is not complete with respect to the ParTypes type system [7]. We discuss some of its shortcomings.

Variables in MPI primitives In order to increase legibility, code that sends messages to the left or to the right process in a ring topology often declares variables for the effect. The original source code [3] declares a variable right with value rank == size - 1 ? 0 : rank + 1. The MPI_Send operation in line 27 is then written as follows:

```
MPI_Send(sendbuf, MAX_PARTICLES / size * 4, MPI_FLOAT, right, ...);
```

In this particular case the value of right is computed from the two distinguished ParTypes variables—size and rank—and it may not be too difficult to replace right by rank == size - 1 ? 0 : rank + 1 in the type. In general, however, the value of variables such as right may be the result of arbitrarily

complex computations, thus complicating type inference in step one of our approach. In addition, indices present in types can only rely on variables whose value is guaranteed to be uniform across all processes. It may not be simple to decide whether an index falls in this category or not.

Parametric types The type in Figure 2 fixes the number of bodies in the simulation (line 1). The original source code, however, reads this value from the command line using `atoi(argv[1])`. The ParTypes language includes a dependent product constructor **`val`** that allows to describe exactly this sort of behaviour:

```
val n: natural.
foreach iter: 1..5000000
  foreach pipe: 1..2
    message 0 1 float[n / 3 * 4]
    ...
```

The ParTypes verification procedure seeks the help of the user in order to link the value of expression `atoi(argv[1])` in the source code to variable `n` in the type [7, 9]. When we think of type inference, it may not be obvious how to resolve this connection during the first step of our proposal.

Type inference and type equivalence ParTypes comes equipped with a rich type theory, allowing in particular to write the three messages in the protocol (Figure 2, lines 3–5) in a more compact form:

```
foreach i: 0..2
  message i (i == 2 ? 0 : i + 1) float[n / size * 4]
```

It is not clear how to compute the more common **`foreach`** protocol from the three messages, but this intensional type is not only more compact but also conductive of further generalisations of the procedure, as outlined in the next example.

The number of processes is in general not fixed A distinctive feature of ParTypes—one that takes it apart from all other approaches to verify MPI-like code—is that verification does not depend on the number of processes. The approach proposed in this paper, however, requires a fixed number of processes, each running a different source code (all of which can nevertheless be obtained from a common source code, such as that in Figure 1). Then, the first step computes one type per process, and the second step merges all these types into a single type. The ParTypes verification procedure allows to check the program in Figure 1 against a protocol for an arbitrary number of processes (greater than 1), where the internal loop (lines 2–5) can be written as

```
foreach pipe: 1..size-1
  foreach i: 0..size-1
    message i (i + 1 < size ? i + 1 : 0) float[n / size * 4]
```

The merge algorithm outlined in this paper crucially relies on a fixed number of types, one per process, and is not clear to us how to relieve this constraint.

One-to-all loops The type presented in the paragraph above contains two **`foreach`** loops: the former corresponds to an actual loop in the source code (lines 23–33), the latter to a conditional (lines 26–32). By expanding the source code in Figure 1 for each different process rank, the first step of our proposal extracts types of the same "shape" for all processes, as we have seen in Section 4. Now consider the following code snippet, where process 0 sends a message to all other processes:

```
if (rank == 0)
  for(i = 1; i < size; i++)
    MPI_Send(sendbuf, n / size * 4, MPI_FLOAT, i, ...);
else
  MPI_Recv(recvbuf, n / size * 4, MPI_FLOAT, 0,  ...);
```

Fixing `size == 3` as before, the first phase yields the following types:

`foreach i: 1..2 message 0 i float[n * 4]`	for rank 0,
`message 0 1 float[n * 4]`	for rank 1, and
`message 0 2 float[n * 4]`	for rank 2.

leaving for phase two the difficult problem of merging one **foreach** type against a series of **message** types. When the limits of the **foreach** loop are constant, we can unfold it and merge the thus obtained sequence of messages as in Section 4, but this is, in general, not the case.

References

[1] Marco Carbone & Fabrizio Montesi (2012): *Merging Multiparty Protocols in Multiparty Choreographies*. In: *PLACES*, *EPTCS* 109, pp. 21–27, doi:10.4204/EPTCS.109.4.

[2] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen F. Siegel, Rajeev Thakur, William Gropp, Ewing L. Lusk, Bronis R. de Supinski, Martin Schulz & Greg Bronevetsky (2011): *Formal Analysis of MPI-based Parallel Programs*. *Communications of the ACM* 54(12), pp. 82–91, doi:10.1145/2043174.2043194.

[3] William Gropp, Ewing Lusk & Anthony Skjellum (1999): *Using MPI (2nd Ed.): Portable Parallel Programming with the Message-passing Interface*. MIT Press.

[4] Per Brinch Hansen (1991): *The N-Body Pipeline*. Electrical Engineering and Computer Science Technical Reports Paper 120, College of Engineering and Computer Science, Syracuse University.

[5] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. *J. ACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695.

[6] Julien Lange & Alceste Scalas (2013): *Choreography Synthesis as Contract Agreement*. In: *ICE*, *EPTCS* 131, pp. 52–67, doi:10.4204/EPTCS.131.6.

[7] Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos & Nobuko Yoshida (2015): *Protocol-based Verification of Message-passing Parallel Programs*. In: *OOPSLA*, ACM, pp. 280–298, doi:10.1145/2814270.2814302.

[8] Patrick Maxim Rondon, Ming Kawaguchi & Ranjit Jhala (2008): *Liquid Types*. In: *POPL*, ACM, pp. 159–169, doi:10.1145/1375581.1375602.

[9] Vasco Thudichum Vasconcelos, Francisco Martins, Eduardo R. B. Marques, Nobuko Yoshida & Nicholas Ng (2017): *Behavioural Types: From Theory to Practice*, chapter Deductive Verification of MPI Protocols. River Publishers.

[10] Niki Vazou, Patrick Maxim Rondon & Ranjit Jhala (2013): *Abstract Refinement Types*. In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, *LNCS* 7792, Springer, pp. 209–228, doi:10.1007/978-3-642-37036-6_13.

[11] Hongwei Xi & Frank Pfenning (1999): *Dependent Types in Practical Programming*. In: *POPL*, ACM, pp. 214–227, doi:10.1145/292540.292560.

Multiparty Session Types, Beyond Duality (Abstract)

Alceste Scalas
Imperial College London
alceste.scalas@imperial.ac.uk

Nobuko Yoshida
Imperial College London
n.yoshida@imperial.ac.uk

Multiparty Session Types (MPST) are a well-established typing discipline for message-passing processes interacting on *sessions* involving two or more participants. Session typing can ensure desirable properties: absence of communication errors and deadlocks, and protocol conformance. However, existing MPST works provide a *subject reduction* result that is arguably (and sometimes, surprisingly) restrictive: it only holds for typing contexts with strong *duality* constraints on the interactions between pairs of participants. Consequently, many "intuitively correct" examples cannot be typed and/or cannot be proved type-safe. We illustrate some of these examples, and discuss the reason for these limitations. Then, we outline a novel MPST typing system that removes these restrictions.

MPST in a Nutshell In the MPST framework [4], *global types* (describing interactions among *roles*) are projected to *local types* used to type-check *processes*. E.g., the global type G involves roles $\mathtt{p}$, $\mathtt{q}$, $\mathtt{r}$:

$$G \;=\; \mathtt{p}\to\mathtt{q}:\left\{\begin{array}{l}\mathtt{m1}(\mathrm{Int}).\,\mathtt{q}\to\mathtt{r}:\mathtt{m2}(\mathrm{Str}).\,\mathtt{r}\to\mathtt{p}:\mathtt{m3}(\mathrm{Bool}).\,\mathbf{end},\\ \mathtt{stop}.\,\mathtt{q}\to\mathtt{r}:\mathtt{quit}.\,\mathbf{end}\end{array}\right\}$$

G says that $\mathtt{p}$ sends to $\mathtt{q}$ *either* a message $\mathtt{m1}$ (carrying an Int) *or* $\mathtt{stop}$; in the first case, $\mathtt{q}$ sends $\mathtt{m2}$ to $\mathtt{r}$ (carrying a Str), then $\mathtt{r}$ sends $\mathtt{m3}$ to $\mathtt{p}$ (carrying a Bool), and the session **end**s; otherwise, in the second case, $\mathtt{q}$ sends $\mathtt{quit}$ to $\mathtt{r}$, and the session **end**s. The *projections of* G are the I/O actions of each role in G:

$$S_{\mathtt{p}} = \mathtt{q}\oplus\left\{\begin{array}{l}\mathtt{m1}(\mathrm{Int}).\,\mathtt{r}\&\mathtt{m3}(\mathrm{Bool}),\\ \mathtt{stop}\end{array}\right\}\quad S_{\mathtt{q}} = \mathtt{p}\,\&\left\{\begin{array}{l}\mathtt{m1}(\mathrm{Int}).\,\mathtt{r}\oplus\mathtt{m2}(\mathrm{Str}),\\ \mathtt{stop}.\,\mathtt{r}\oplus\mathtt{quit}\end{array}\right\}\quad S_{\mathtt{r}} = \mathtt{q}\,\&\left\{\begin{array}{l}\mathtt{m2}(\mathrm{Str}).\,\mathtt{p}\oplus\mathtt{m3}(\mathrm{Bool}),\\ \mathtt{quit}\end{array}\right\}$$

Here, $S_{\mathtt{p}}$, $S_{\mathtt{q}}$, $S_{\mathtt{r}}$ are the projections of G resp. onto $\mathtt{p}$, $\mathtt{q}$, $\mathtt{r}$. E.g., $S_{\mathtt{p}}$ is a session type that represents the behaviour of $\mathtt{p}$ in G: it must send ($\oplus$) to $\mathtt{q}$ either $\mathtt{m1}(\mathrm{Int})$ or $\mathtt{stop}$; in the first case, the channel is then used to receive (&) message $\mathtt{m3}(\mathrm{Bool})$ from $\mathtt{r}$, and the session ends; otherwise, in the second case, the session ends. Now, a *typing context* Γ can assign types $S_{\mathtt{p}}$, $S_{\mathtt{q}}$ and $S_{\mathtt{r}}$ to *multiparty channels* $s[\mathtt{p}]$, $s[\mathtt{q}]$ and $s[\mathtt{r}]$, used to play roles $\mathtt{p}$, $\mathtt{q}$ and $\mathtt{r}$ on *session* s. Then, if e.g. some parallel processes $P_{\mathtt{p}}$, $P_{\mathtt{q}}$ and $P_{\mathtt{r}}$ type-check w.r.t. Γ, then we know that such processes use the channels abiding by their types.

Subject Reduction, or Lack Thereof We would expect that typed processes reduce type-safely, e.g.:

$$\vdash P \triangleright \Gamma \text{ and } P \to^* P' \quad \text{implies} \quad \exists \Gamma' : \vdash P' \triangleright \Gamma' \quad (\text{where } P = P_{\mathtt{p}} \mid P_{\mathtt{q}} \mid P_{\mathtt{r}} \text{ and } \Gamma = s[\mathtt{p}]{:}S_{\mathtt{p}}, s[\mathtt{q}]{:}S_{\mathtt{q}}, s[\mathtt{r}]{:}S_{\mathtt{r}}) \quad (1)$$

But surprisingly, *this is not the case!* In MPST works (e.g., [1]), the subject reduction statement reads:

$$\vdash P \triangleright \Gamma \ \underline{\text{with } \Gamma \text{ consistent}} \ \text{and} \ P \to^* P' \quad \text{implies} \quad \exists \Gamma' \ \underline{\text{consistent}} \ \text{such that} \ \vdash P' \triangleright \Gamma' \qquad (2)$$

Intuitively, Γ is consistent if all its potential interactions between pairs of roles are *dual*: e.g., all potential outputs of $S_{\mathtt{p}}$ towards $\mathtt{r}$ are matched by compatible input capabilities of $S_{\mathtt{r}}$ from $\mathtt{p}$. Consistency

V.T. Vasconcelos and P. Haller (Eds.): Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES'17)
EPTCS 246, 2017, pp. 37–38, doi:10.4204/EPTCS.246.7

is quite restrictive, due to its (rather intricate) *syntactic* nature—and does *not* hold in our example. This is due to *inter-role dependencies*: $S_{\mathtt{p}}$ allows to decide what to send to $\mathtt{q}$ — and depending on such a choice, whether to input $\mathtt{m3}$ from $\mathtt{r}$, or not. This breaks the definition of consistency between $S_{\mathtt{p}}$ and $S_{\mathtt{r}}$; hence, Γ in (1) is not consistent, and we cannot apply (2) to ensure that $P_{\mathtt{p}}$, $P_{\mathtt{q}}$, $P_{\mathtt{r}}$ reduce type-safely.

Our Proposal In "standard" MPST works, consistency cannot be lifted without breaking subject reduction [1, p.163]. Hence, to prove that our example is type-safe, we need to revise the MPST foundations. We propose a *novel MPST typing system* that safely lifts the consistency requirement, by introducing:

1. a new MPST typing judgement with the form $\Theta \vdash P \triangleright \Gamma_g \triangleleft \Gamma_r$ —where Γ_g and Γ_r are respectively the *guarantee* and *rely typing contexts*. Intuitively, Γ_g describes how P uses its channels, while Γ_r describes how other processes (possibly interacting with P) are expected to use their channels;
2. a *semantic* notion of typing context safety, called *liveness*, based on MPST context reductions [1]. In our typing judgement, the pair Γ_g, Γ_r must be live: this ensures that each output can synchronise with a compatible input (and *vice versa*). Unlike consistency, liveness supports complex inter-role dependencies, and ensures that the typing context cannot deadlock.

Related Work A technical report with more examples and discussion is available in [6]. Our novel typing system allows to prove type safety of processes implementing global types with complex inter-role dependencies and delegations. To the best of our knowledge, the only work with a similar capability is [3]; however, its process calculus only supports *one* session, and this restriction is crucially exploited to type parallel compositions without "splitting" them (cf. Table 8, rule [T-SESS]). Hence, unlike our work, [3] does not support multiple sessions and delegation—and extending it seems challenging. Further, unlike [3], our typing rules do *not* depend on global types and projections: by removing this orthogonal concern, we simplify the theory. If needed, a set of local types can be related to a global type via "top-down" projection or "bottom-up" synthesis [5]. Similarly to most MPST papers, our work ensures that a typed process $(\nu s)(\big|_{\mathtt{p}\in I} P_{\mathtt{p}})$, with each $P_{\mathtt{p}}$ only interacting on $s[\mathtt{p}]$, is deadlock-free—but does not guarantee deadlock freedom for multiple interleaved sessions [2]: we leave this topic as future work.

Thanks to the reviewers for their suggestions, and to R. Hu, J. Lange, B. Toninho for the fruitful discussion. Work supported by: EPSRC (EP/K011715/1, EP/K034413/1, EP/L00058X/1), EU (COST Action IC1201, FP7-612985).

References

[1] M. Coppo, M. Dezani-Ciancaglini, L. Padovani & N. Yoshida (2015): *A Gentle Introduction to Multiparty Asynchronous Session Types*. doi:10.1007/978-3-319-18941-3_4.

[2] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida & L. Padovani (2016): *Global Progress for Dynamically Interleaved Multiparty Sessions*. *MSCS* 26(2), doi:10.1017/S0960129514000188.

[3] M. Dezani-Ciancaglini, S. Ghilezan, S. Jakšić, J. Pantović & N. Yoshida (2016): *Precise subtyping for synchronous multiparty sessions*. In: *PLACES 2015*, doi:10.4204/EPTCS.203.3.

[4] K. Honda, N. Yoshida & M. Carbone (2008): *Multiparty asynchronous session types*. In: *POPL*, doi:10.1145/1328438.1328472. Full version: Volume 63, Issue 1, March 2016 (9), pages 1-67, *JACM*.

[5] J. Lange, E. Tuosto & N. Yoshida (2015): *From Communicating Machines to Graphical Choreographies*. In: *POPL*, doi:10.1145/2676726.2676964.

[6] A. Scalas & N. Yoshida (2017): *Multiparty Session Types, Beyond Duality*. Technical Report, Imperial College London. Available at `https://www.doc.ic.ac.uk/research/technicalreports/2017/`.

Generating Representative Executions

Extended Abstract

Hendrik Maarand Tarmo Uustalu

Dept. of Software Science, Tallinn University of Technology

Analyzing the behaviour of a concurrent program is made difficult by the number of possible executions. This problem can be alleviated by applying the theory of Mazurkiewicz traces to focus only on the canonical representatives of the equivalence classes of the possible executions of the program. This paper presents a generic framework that allows to specify the possible behaviours of the execution environment, and generate all Foata-normal executions of a program, for that environment, by discarding abnormal executions during the generation phase. The key ingredient of Mazurkiewicz trace theory, the dependency relation, is used in the framework in two roles: first, as part of the specification of which executions are allowed at all, and then as part of the normality checking algorithm, which is used to discard the abnormal executions. The framework is instantiated to the relaxed memory models of the SPARC hierarchy.

1 Introduction

Let us consider a fragment from Dekker's mutual exclusion algorithm as an example.

Init: `x = 0; y = 0;`

P_1	P_2
(a) `[x] := 1`	(c) `[y] := 1`
(b) `r1 := [y]`	(d) `r2 := [x]`

Observed? `r1 = 0; r2 = 0;`

This is a concurrent program for two processors, P_1 and P_2, where x is the flag variable for P_1 that is used to communicate that P_1 wants to enter the critical section and y is for P_2. A processor may enter the critical section, if it has notified the other processor by setting its flag variable to 1, reading the flag variable of the other processor and checking that it is 0. We are interested in whether it is possible, starting from an initial state where both x and y are 0, that both processors see each others' flag variables as 0, meaning that both processors enter the critical section. Here we are interested in the mutual exclusion property, that at most one processor can enter the critical section.

In the interleaving semantics of Sequential Consistency (SC), the above program can have the following executions: *abcd*, *cdab*, *acbd*, *cabd*, *acdb*, *cadb*. Out of these six, the four last executions are actually equivalent (in the sense that from the same initial state they will reach the same final state) and for our purposes it is enough to check the final state of only one of them. We can observe that the mutual exclusion property is satisfied. The situation is different, if we consider the possible executions on a real-world processor, like x86, which follows the Total Store Order (TSO) model [8]. Under TSO, it is possible for writes to be reordered with later reads from the same processor, resulting in an execution that is observable as *bdac*. This does not satisfy the mutual exclusion property.

In this paper, we seek to alleviate the difficulty analyzing the large numbers of executions concurrent programs, especially on relaxed memories, generate, by applying the theory of Mazurkiewicz traces to focus only on some type of canonical representatives of the equivalence classes of the possible executions of the program. We present a generic framework for interpreting concurrent programs under different

V.T. Vasconcelos and P. Haller (Eds.): Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES'17)
EPTCS 246, 2017, pp. 39–48, doi:10.4204/EPTCS.246.8

semantics, so that only executions in the Foata normal form (corresponding to maximal parallelism) are generated. We instantiate the framework to the relaxed memory models of the SPARC hierarchy. This work is in the vein of partial order reduction techniques for analysis of systems, which are widely used especially in model checking and have also been applied to relaxed memories, e.g., by Zhang et al. [13]. The novelties here are that the different memory models are modelled uniformly based on a flexible notion of a backlog of shadow events, using a standard normal form from trace theory, and using generalized traces (with a dynamic independency relation) to be able to define execution equivalence more finely, resulting in bigger and fewer equivalence classes. The framework has been prototyped in Haskell where one can easily separate the phases of generating the tree of symbolic executions of a program, discarding the abnormal executions, and running the tree of symbolic executions from an initial state. This separation can be made without a performance penalty thanks to lazy evaluation.

2 Mazurkiewicz Traces

An execution (or a run) of a sequential program can be represented as a sequence of symbols that record the events caused by the program in the order that they occurred. Such a sequence is a string over some (finite) alphabet Σ. An execution of a concurrent program can be represented as an interleaving of the executions on the processors involved, thereby reducing concurrency to non-deterministic choice. Mazurkiewicz traces [7] (or just traces) are a generalization of strings, where some of the letters in the string are allowed to commute. This allows representation of non-sequential behaviour. In other words, traces are equivalence classes of strings with respect to a congruence relation that allows to commute certain pairs of letters.

A dependency relation $D \subseteq \Sigma \times \Sigma$ is a reflexive and symmetric binary relation. $a\ D\ b$ if and only if the events a and b can be causally related, meaning that the two events cannot happen concurrently. The complement of the dependency relation, $I = (\Sigma \times \Sigma) \setminus D$, is called the independency relation. If $a\ I\ b$, then the strings *sabt* and *sbat* represent the same non-sequential behaviour. Two strings $s,t \in \Sigma^*$ are said to be Mazurkiewicz equivalent, $s \equiv_D t$, if and only if s can be transformed to t by a finite number of exchanges of adjacent, independent events. For example, if $\Sigma = \{a,b,c,d\}$ and $a\ I\ c$ and $b\ I\ d$ then the trace *acbd* represents the strings *acbd*, *cabd*, *acdb* and *cadb*.

For our purposes, standard Mazurkiewicz traces are not enough and therefore we turn to the generalized Mazurkiewicz traces of Sassone et al. [10]. In generalized Mazurkiewicz traces, the dependency relation is dynamic, it depends on the current context, which is the partial execution that has been performed so far. The dependency relation for a prefix s will be denoted by D_s and the subscript is omitted, if the relation is static. Besides D_s having to be reflexive and symmetric for any s, D must satisfy some sanity conditions. Most importantly, if $s \equiv_D t$, then it must be the case that $D_s = D_t$. In this setting, the strings *sabt* and *sbat* are considered equivalent, if $a\ I_s\ b$.

Normal Forms As traces are equivalence classes, it is reasonable to ask what the canonical representative or normal form of a trace is. There are two well-known normal forms for traces, the lexicographic and Foata [4] normal forms. We are going to look at Foata normal forms for our purposes.

A step is a subset $s \subseteq \Sigma$ of pairwise independent letters. The Foata normal form of a trace is a sequence $s_1 \ldots s_k$ of steps such that the individual steps $s_1, \ldots, s_k$ are chosen from the left to the right with maximal cardinality. Since each step consists of independent letters, a step can be executed in parallel, meaning that the Foata normal form encodes a maximal parallel execution. For example, if $\Sigma = \{a,b,c,d\}$ and $a\ I\ c$ and $b\ I\ d$, then the Foata normal form of *acbd* is *(ac)(bd)*.

We are interested in checking whether a given string is in normal form according to a given depen-

dency relation. As a convenience, we also assume to have an ordering $\prec$ on Σ that is total on events that are independent. A string is in Foata normal form, if it can be split into a sequence of steps $s_1, \ldots, s_k$ so that concatenation of the steps gives the original string and the following conditions are satisfied:

1. for every $a, b \in s_i$, if $a \neq b$ then $a\ I_i\ b$;
2. for every $b \in s_{i+1}$, there is an $a \in s_i$ such that $a\ D_i\ b$;
3. for every step s_i, the letters in it are in increasing order wrt. $\prec$.

In these definitions, we consider D_i to be the dependency relation for the context $s_0 \ldots s_{i-1}$ and similarly for I_i. The first condition ensures that the events in a step can be executed in parallel. The second condition ensures that every event appears in the earliest possible step, i.e., maximal parallelism. The third condition picks a permutation of a step as a representative of the step. Notice that if a string is not in normal form, then neither is any string with that string as a prefix in normal form. This means that when checking a string for normality by scanning it from the left to the right, we can discard it as soon as we discover an abnormal prefix.

3 Framework

We now proceed to describing our framework for generating representative executions of a program and its instantiations to different memory models.

We are going to look at programs executing on a machine that consists of processors and a shared memory. Each processor also has access to a local memory (registers). The executions that we investigate are symbolic, in the sense that we do not look at the actual values propagating in the memory, but just the abstract actions being performed. Still, our goal is to find the possible final states of a program from a given initial state. The idea is that once the symbolic executions have been computed, the canonical executions can be picked and the final state needs to be computed only for those. This can be done lazily, meaning that the evaluation of a particular execution for the given initial state is cancelled immediately, if it is discovered that the execution is not normal.

The language for our system consists of arithmetic and boolean expressions and commands. An arithmetic expression is either a numeral value, a register, or an arithmetic operation. A boolean expression is either boolean constant, a boolean operation, or a comparison of arithmetic expressions. Commands consist of assignments to registers, loads and stores to shared memory, and *if* and *while* constructs.

Our framework is defined on top of the events generated by the system. We think of events as occurrences of (the phases of) the actions that executing the program can trigger. An event can be thought of as a record *(pid, eid, kind, act)* where *pid* is the identifier of the processor that generated the event, *eid* is the processor-local identifier of the event, *kind* defines whether it is a main or a shadow event, and *act* is the action performed in this event. An action can be an operation between registers, a load from or a store to a variable, or an assertion on registers. An assertion is used to record a decision made in the unfolding of a control structure of the program, for example, that a particular execution is one where the *true* branch of a conditional was taken. If an assertion fails when an execution is evaluated from a given initial state, then this execution is not valid for that initial state.

Since we are interested in modelling different memory models, our framework is parameterized by an architecture, which characterizes the behavioural aspects of the system. An architecture consists of four components. A predicate *shadows* describes whether an action is executed in a single stage or two stages, generating just one (main) event or two events (a main and a shadow event). An irreflexive-antisymmetric relation *sameDep* describes which events from a processor must happen before which other events from the same processor: it plays a role in determining the possible next events from this processor, but also

defines which events from it are dependent. A relation *diffDep* describes when two events from different processors are dependent. Finally, a relation $\prec$ orders independent events. The relations *sameDep* (its reflexive-symmetric closure) and *diffDep* together determine the dependency relation in the sense of Mazurkiewicz traces and $\prec$ is the relation used to totally order the events within a step.

In the previous paragraph, we mentioned shadow events. These are the key ingredients of this framework for modelling more intricate behaviours, for example, when some actions are non-atomic and this fact needs to be reflected in the executions by two events, a main event and a shadow event. TSO, for example, can be described as a model where writes to memory first enter the processor's write-buffer and are later flushed from the write-buffer to memory. We consider the write to buffer to be the main event of the write action and the flush event to be the shadow event of the write action. Of these two events, the shadow event is globally observable.

Generating Normal Forms The process of generating normal-form executions of a program can be divided into two stages: lazily generating all executions of the program and then discarding those not in normal form.

The executions are generated as follows: if all processors have completed, then we have a complete execution and we are done, otherwise we pick a processor that has not yet completed and allow it to make a small step, then repeat the process. The local configuration of a processor consists of its residual program, backlog, and the value of a counter to provide identifiers for the generated events. The small step can either correspond to beginning the action of the next instruction according to the program—in which case a new main event is generated and added to the execution—, or to completing an already started action—in this case, a shadow event is removed from the processor's backlog and added to the execution. If the step is to start a new action, then the *shadows* predicate is used to check whether a new shadow event should be added to the backlog (if not, the action is completed by the main event). A side-condition for adding a new main event is that there are no shadow events in the backlog that are dependent with it. An event can be removed from the backlog, if it is independent (according to *sameDep*) of all of the older events in the backlog. Conditionals like *if* and *while* are expanded to a choice between two programs, where the choices correspond to the branches of the conditional together with an assertion of the condition. The generation of executions is described by the small step rules in Appendix A.

The second stage of the procedure is to single out the normal forms among the generated executions. This is done by checking the normality of the executions according to the three conditions given in Section 2 for Foata normal forms. The rules for checking the normality of an execution by scanning it from the left to the right are given in Appendix A.

Instead of generating a flat set of executions in the first stage, we actually generate a tree of executions, so that the prefixes of executions are shared. Since the process of selecting the canonical executions (more precisely, discarding the non-canonical ones) according to the conditions of Foata normal forms can be fused into the generation stage, we can discard a whole set of executions when we discover that the current path down the tree violates the normality conditions. More precisely, walking down the tree, we keep track of the current prefix (which must be in normal form) and at each node we check whether the event associated with the node would violate the normality conditions when added to the prefix. Only if the normality condition is not violated does the subtree starting from that node need to be computed actually.

We require *sameDep a b* to hold at least when *a* and *b* are main events and $eid\ a < eid\ b$ or when they are a main event and its shadow event (in which case they have the same *eid*). We also require that *sameDep a b* can only hold when $eid\ a < eid\ b$ or when $eid\ a = eid\ b$ and *a* is a main event and *b*

the corresponding shadow event. Under these assumptions, we can prove that the total set of executions captured in the generated tree is closed under equivalence. As the normality checking stage keeps all normal forms and discards all non-normal forms, it follows that the pruned set of executions contains exactly one representative for every execution of the program.

In the introduction, we noted that our example program has six executions under interleaving semantics, of which four are equivalent. The executions are depicted in Figure 1 and the four equivalent executions *acbd*, *acdb*, *cabd* and *cadb* are the ones in the middle. For this program we have that $a\ I\ c$ and $b\ I\ d$. Our framework would only generate *acbd* out of these four, as this corresponds to the Foata normal form *(ac)(bd)* and the other three would be discarded. More precisely, *(ac)(d)* is in normal form, but it cannot be extended by *b*, as neither *(ac)(db)* nor *(ac)(d)(b)* is in normal form: the first one fails due to condition 3 and the second one fails due to condition 2. The node *b* of this path is shaded in the picture to highlight the place where the normality condition is violated. For *cabd*, we start checking normality from *(c)*, which is valid, but neither *(ca)* nor *(c)(a)* is in normal form and we can discard all executions that start with *ca*, which includes both *cabd* and *cadb*. The subtree at node *a* is shaded to highlight this fact.

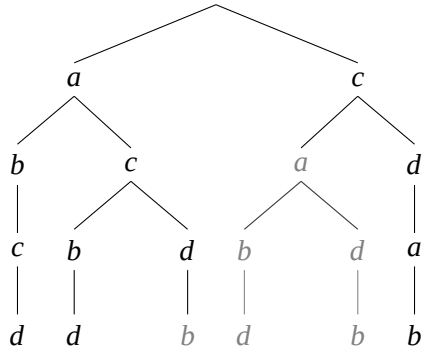


Figure 1: SC executions of the example program.

4 Instantiation to Relaxed Memory Models

Sequential Consistency In the Sequential Consistency (SC) model [6], any execution of a concurrent program is an interleaving of the program order executions of its component threads. SC can be specified as an architecture in the following way:

$$\begin{aligned} \mathit{shadows}\ a &= \mathit{false} \\ \mathit{sameDep}\ a\ b &= \mathit{eid}\ a < \mathit{eid}\ b \\ \mathit{diffDep}\ x\ y\ a\ b &= \mathit{crxw}\ a\ b \\ a \prec b &= \mathit{pid}\ a < \mathit{pid}\ b \end{aligned}$$

crxw a b represents the concurrent-read-exclusive-write property, which returns *true*, if events *a* and *b* access the same location and at least one of them is a write. *diffDep* also takes two arguments that are ignored here, which represent the backlogs of the two processors from which the events *a* and *b* originate from. This information can be recovered from the prefix of the execution and it is as much information

as we need about the prefix of the execution in the memory models we consider. We could also just take the prefix of the execution itself and compute the necessary information. Setting *shadows* to be always *false* means that all instructions execute atomically. Setting *sameDep a b* to require $eid\ a < eid\ b$ means that the events from the same processor must be generated in program order and cannot be reordered, which reflects the definition of SC.

Total Store Order In the Total Store Order (TSO) model [11], it is possible for a write action to be reordered with later reads, meaning that writes happen asynchronously, but at the same time the order of write actions is preserved. TSO can be specified in the following way:

$$\begin{aligned} shadows\ a &= isWrite\ a \\ sameDep\ a\ b &= isMain\ a \wedge isMain\ b \wedge eid\ a < eid\ b \\ &\quad \vee\ isMain\ a \wedge isShadow\ b \wedge eid\ a == eid\ b \\ &\quad \vee\ isShadow\ a \wedge isShadow\ b \wedge eid\ a < eid\ b \\ diffDep\ x\ y\ a\ b &= crxw'\ x\ y\ a\ b \\ a \prec b &= pid\ a < pid\ b \vee pid\ a == pid\ b \wedge eid\ a < eid\ b \end{aligned}$$

crxw' is like *crxw*, except that it considers shadow write events instead of main write events as the global write events, and read events as global only if they access the memory. This is where we need generalized Mazurkiewicz traces, since if there is a pending write to the location of the read, then the read action would not read its value from memory and thus could not be dependent with events from other processors.

We consider the main event of a write instruction to be the write to buffer and the shadow event to be the flushing of the write from buffer to memory. TSO can be thought of as a model where every processor has a shadow processor and all events on every main processor are in program order, all of the events on the associated shadow processor are in program order and an event on the shadow processor must happen after the corresponding event on the main processor. Our example from introduction has the following traces in Foata normal form under TSO: *(ac)(a'c')(bd)*, *(ac)(a'b)(c'd)*, *(ac)(c'd)(a'b)* and *(ac)(bd)(a'c')* where *a'* stands for the shadow event of *a*. The last of these is the one rejected by SC.

Partial Store Order The Partial Store Order (PSO) model [11] allows the reorderings of TSO, but it is also possible for a write to be reordered with a later write to a different location. This can be thought of as having a separate write buffer for every variable. PSO can be specified as TSO with the exception of the *sameDep* relation:

$$\begin{aligned} sameDep\ a\ b &= isMain\ a \wedge isMain\ b \wedge eid\ a < eid\ b \\ &\quad \vee\ isMain\ a \wedge isShadow\ b \wedge eid\ a == eid\ b \\ &\quad \vee\ isShadow\ a \wedge isShadow\ b \wedge eid\ a < eid\ b \wedge var\ a == var\ b \end{aligned}$$

Intuitively, this corresponds to PSO, since it is like TSO except for the dependency relation on events from the same processor, where the shadow events are dependent only if they are to the same location, which allows one to reorder writes to different locations.

Relaxed Memory Order The Relaxed Memory Order [11] (RMO) model only enforces program order on write-write and read-write instruction pairs to the same variable and on instruction pairs in dependency, where the first instruction is a read. Dependency on instruction pairs here means that there is

data- or control-dependency between the instructions. We can specify RMO in the following way:

$$
\begin{aligned}
\mathit{shadows}\ a &= \mathit{true} \\
\mathit{sameDep}\ a\ b &= \mathit{isMain}\ a \wedge \mathit{isMain}\ b \wedge \mathit{eid}\ a < \mathit{eid}\ b \\
&\quad \vee\ \mathit{isMain}\ a \wedge \mathit{isShadow}\ b \wedge \mathit{eid}\ a == \mathit{eid}\ b \\
&\quad \vee\ \mathit{isShadow}\ a \wedge \mathit{isShadow}\ b \wedge \mathit{eid}\ a < \mathit{eid}\ b \\
&\qquad \wedge\ (\mathit{var}\ a == \mathit{var}\ b \wedge (\mathit{isWrite}\ a \vee \mathit{isRead}\ a) \wedge \mathit{isWrite}\ b \\
&\qquad\quad \vee\ \mathit{dataDep}\ a\ b \vee \mathit{controlDep}\ a\ b) \\
\mathit{diffDep}\ x\ y\ a\ b &= \mathit{crxw}''\ x\ y\ a\ b \\
a \prec b &= \mathit{pid}\ a < \mathit{pid}\ b \vee \mathit{pid}\ a == \mathit{pid}\ b \wedge \mathit{eid}\ a < \mathit{eid}\ b
\end{aligned}
$$

crxw" is like *crxw'* except that it considers shadow reads and shadow writes as the global read and write events. As for TSO and PSO, a shadow read is considered global, if it actually reads its value from memory, which in this model happens, if there is no older shadow write to the same location in the backlog. We consider events a and b to be in data-dependency, if a reads a register that is written by b. We consider two events to be in control-dependency, if the older one is a conditional and the newer one is a write.

4.1 Fences

In models like TSO, PSO and RMO that allow the reordering of some events, it becomes necessary to be able to forbid these reorderings in certain situations, to rule out relaxed behaviour. Our example from introduction does not behave correctly on TSO, where it is possible for both processors to read the value 0. To avoid this situation, it is necessary to make sure that both processors first perform the write and when the effects of the write operation have become globally visible they may perform the read. With this restriction the program behaves correctly on TSO and the way to achieve this is to insert a fence between the write and read instructions.

In our framework, fences are described by two parameters that can take the values *store* or *load*, which indicate between which events the ordering is enforced. Under SC, the fence instructions can be ignored since no reorderings are possible. To be able to restore sequentially consistent behaviour, TSO requires store-load fences, PSO requires also store-store fences, and RMO requires all four kinds of fences. For TSO, PSO, and RMO, the idea is that fences have shadow events and the *sameDep* relation is modified to disallow unwanted reorderings. Our example program requires a store-load fence, so that the read operations appearing after the fence cannot be performed before the write operations appearing before the fence have completed. This means that *sameDep* must be modified to consider a shadow store-load fence to be dependent with all older shadow write events and all newer read events. Dependence with a shadow event prevents the fence event from being removed from the backlog until the older dependent events have been removed and it also prevents removing the newer dependent events until the fence has been removed from the backlog. Likewise, a new main read event cannot be added to the execution, if there is a store-load fence event in the backlog. The idea is similar for the other types of fences.

5 Related Work

Relaxed memory consistency models and their specification and verification tasks have been an extensive research topic. Owens et al. [8] showed that x86 adheres to TSO model and they gave both operational

and axiomatic models. Alglave [2] defined a framework in an axiomatic style for working with relaxed memory models, which is also generic in the sense that different memory models can be represented by specifying which relations are considered global. Generating the possible executions in our framework turns out to be quite similar to an executable specification for RMO given by Park and Dill [9], more precisely, our notion of backlog seems to correspond to the reordering box used there. Boudol et al. [3] defined a generic operational semantics that captures TSO, PSO and RMO and uses temporary stores that again are similar to our backlogs; they did not however consider any partial order reduction of the set of executions of a program. As mentioned before, due to the interest in exploring the full set of executions by constructing it explicitly and the use of trace theory, which is the foundation for partial order reduction [5], this work is also close to methods based on model checking, like Zhang et al.'s [13] and Abdulla et al.'s [1]. An executable specification was also given by Yang et al. [12]. Their approach is based on axiomatic specifications and an execution is found by searching for an instantiation that satisfies all of the constraints, either by Prolog or a SAT solver.

6 Conclusion

We have presented a generic framework for finding canonical representatives of equivalence classes of the possible executions of a program. The framework proceeds by lazily generating all executions of the given program and discards all those that are not in Foata normal form. The framework allows to uniformly represent the semantics of a certain class of relaxed memory models, which we have illustrated by encoding the models from the SPARC hierarchy in terms of our framework. An instantiation of the framework to a particular model specifies which executions can occur at all for the given program and which of those are equivalent, i.e., correspond to one generalized Mazurkiewicz trace, representable by its normal form.

We plan to continue this work by elaborating on the formal aspects of the framework. We have formalized soundness and completeness of Foata normalization of (standard) traces in the dependently typed functional language Agda—any string is equivalent to its normal form, and if a string is equivalent to a normal form, it is that string's normal form. This development can be scaled for generalized traces, adapted to prove that the tree filtering algorithm keeps exactly one representative of each equivalence class of executions, to then move on to formalization of specifications of memory models.

Acknowledgments This research was supported by the Estonian Ministry of Education and Research institutional research grant no. IUT33-13 and the ERDF funded CoE project EXCITE (2014-2020.4.01.15-0018).

References

[1] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson & K. Sagonas (2015): *Stateless Model Checking for TSO and PSO*. In: C. Baier & C. Tinelli, editors: *Proc. of 21st Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2015, Lect. Notes in Comput. Sci.* 9035, Springer, pp. 353–367, doi:10.1007/978-3-662-46681-0_28.

[2] J. Alglave (2010): *A Shared Memory Poetics*. Ph.D. thesis, Université Paris 7. Available at http://www0.cs.ucl.ac.uk/staff/J.Alglave/these.pdf.

[3] G. Boudol, G. Petri & Serpette G. (2012): *Relaxed Operational Semantics of Concurrent Programming Languages*. In B. Luttik & M. A. Reniers, editors: *Proc. of Combined 19th Wksh. on Expressiveness in Concurrency and 9th Wksh. on Structural Operational Semantics, EXPRESS/SOS 2012, Electron. Proc. in Theor. Comput. Sci.* 89, pp. 19–33, doi:10.4204/eptcs.89.3.

[4] P. Cartier & D. Foata (1969): *Problemes combinatoires de commutation et réarrangements*. *Lect. Notes in Math.* 85, Springer, doi:10.1007/bfb0079468.

[5] P. Godefroid (1996): *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, doi:10.1007/3-540-60761-7.

[6] L. Lamport (1979): *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*. *IEEE Trans. on Comput.* 28(9), pp. 690–691, doi:10.1109/tc.1979.1675439.

[7] A. Mazurkiewicz (1995): *Introduction to Trace Theory*. *The Book of Traces*, pp. 3–41, doi:10.1142/9789814261456_0001.

[8] S. Owens, S. Sarkar & P. Sewell (2009): *A Better x86 Memory Model: x86-TSO*. In S. Berghofer, T. Nipkow, C. Urban & M. Wenzel, editors: *Proc. of 22nd Int. Conf. on Theorem Proving in Higher Order Logics, TPHOLs 2009*, *Lect. Notes in Comput. Sci.* 5674, Springer, pp. 391–407, doi:10.1007/978-3-642-03359-9_27.

[9] S. Park & D. L. Dill (1995): *An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order)*. In: *Proc. of 7th Ann. ACM Symp. on Parallel Algorithms and Architectures, SPAA '95*, ACM, pp. 34–41, doi:10.1145/215399.215413.

[10] V. Sassone, M. Nielsen & G. Winskel (1993): *Deterministic Behavioural Models for Concurrency*. In A. M. Borzyszkowski & S. Sokolowski, editors: *Proc. of 18th Int. Symp. on Mathematical Foundations of Computer Science, MFCS '93*, *Lect. Notes in Comput. Sci.* 711, Springer, pp. 682–692, doi:10.1007/3-540-57182-5_59.

[11] SPARC International Inc. & David L. Weaver (1994): *The SPARC Architecture Manual*. Prentice-Hall.

[12] Y. Yang, G. Gopalakrishnan, G. Lindstrom & K. Slind (2004): *Nemos: A Framework for Axiomatic and Executable Specifications of Memory Consistency Models*. In: *Proc. of 18th Int. Parallel and Distributed Processing Symposium, IPDPS 2004*, IEEE, pp. 31–40, doi:10.1109/ipdps.2004.1302944.

[13] N. Zhang, M. Kusano & C. Wang (2015): *Dynamic Partial Order Reduction for Relaxed Memory Models*. In: *Proc. of 36th ACM SIGPLAN Conf. on Principles of Language Design and Implementation, PLDI 2015*, ACM, pp. 250–259, doi:10.1145/2737924.2737956.

A Semantic Rules

Small steps of a processor

$$\frac{}{[]\ I^{same}\ e'} \qquad \frac{e\ I^{same}\ e' \quad bklg\ I^{same}\ e'}{e : bklg\ I^{same}\ e'}$$

$$\frac{shadows(act) \quad bklg\ I^{same}\ (eid, \circ, act)}{(act : prg, bklg, eid) \xrightarrow{(eid,\circ,act)} (prg, (eid, \bullet, act) : bklg, eid + 1)}$$

$$\frac{\neg shadows(act) \quad bklg\ I^{same}\ (eid, \circ, act)}{(act : prg, bklg, eid) \xrightarrow{(eid,\circ,act)} (prg, bklg, eid + 1)}$$

$$\frac{older\ I^{same}\ le}{(prg, newer +\!\!+ (le : older), eid) \xrightarrow{le} (prg, newer +\!\!+ older, eid)}$$

$$\frac{(prg_i, bklg, eid) \xrightarrow{le} c}{(prg_0 + prg_1, bklg, eid) \xrightarrow{le} c}$$

Small steps of the system

$$\frac{c(pid) = lc \quad lc \xrightarrow{le} lc'}{c \xrightarrow{(pid,le)} c[pid \mapsto lc']}$$

Executions

$$\frac{\forall pid.\, c(pid) = ([],[],\_)}{c \overset{[]}{\Rightarrow} c} \qquad \frac{c \xrightarrow{e} c'' \quad c'' \overset{es}{\Rightarrow} c'}{c \overset{e:es}{\Longrightarrow} c'}$$

Normal executions

$$\frac{le\, I^{same}\, le'}{(pid,le)\, I_{ss}\, (pid,le')} \qquad \frac{pid \neq pid' \quad le\, I_{ss}^{diff}\, le'}{(pid,le)\, I_{ss}\, (pid',le')}$$

$$\frac{e \prec e'}{[e] \prec e'} \qquad \frac{e \prec e'}{s : e \prec e'} \qquad \frac{e\, I_{ss}\, e'}{[e]\, I_{ss}\, e'} \qquad \frac{s\, I_{ss}\, e' \quad e\, I_{ss}\, e'}{s : e\, I_{ss}\, e'}$$

$$\frac{}{ss \vdash []}$$

$$\frac{[] : [e] \vdash es}{[] \vdash e : es} \qquad \frac{s\, I_{[]}\, e \quad s \prec e \quad [] : (s : e) \vdash es}{[] : s \vdash e : es}$$

$$\frac{\neg(s\, I_{ss}\, e) \quad ss : s : [e] \vdash es}{ss : s \vdash e : es} \qquad \frac{\neg(s\, I_{ss}\, e) \quad s'\, I_{ss:s}\, e \quad s' \prec e \quad ss : s : (s' : e) \vdash es}{ss : s : s' \vdash e : es}$$

Towards a Categorical Representation of Reversible Event Structures

Eva Graversen Iain Phillips Nobuko Yoshida
Imperial College London, UK

We study categories for reversible computing, focussing on reversible forms of event structures. Event structures are a well-established model of true concurrency. There exist a number of forms of event structures, including prime event structures, asymmetric event structures, and general event structures. More recently, reversible forms of these types of event structures have been defined. We formulate corresponding categories and functors between them. We show that products and coproducts exist in many cases. In most work on reversible computing, including reversible process calculi, a cause-respecting condition is posited, meaning that the cause of an event may not be reversed before the event itself. Since reversible event structures are not assumed to be cause-respecting in general, we also define cause-respecting subcategories of these event structures. Our longer-term aim is to formulate event structure semantics for reversible process calculi.

1 Introduction

Event structures [10], a well-known model of true concurrency, consist of events and relations between them, describing the causes of events and conflict between events. Winskel [18] defined a category of event structures, and used this to define event structure semantics of CCS.

Reversible process calculi are a well-studied field [3, 5, 6, 8, 9, 11]. When considering the semantics of reversible processes, the ability to reverse events leads to finer distinctions of a true concurrency character [12]; for example the CCS processes $a \mid b$ and $a.b + b.a$ can easily be distinguished by whether both a and b can be reversed at the end of the computation. This motivates the study of *reversible event structures*. So far, no event structure semantics have been defined for reversible variants of CCS [5, 6, 11] (though the reversible π-calculus has been modelled using rigid families [4]); we intend this work to be one of the first steps towards doing so.

Reversible versions of various kinds of event structures were introduced in [13, 15]. Our aim here is to interpret these as objects in appropriate categories and study functors between them. So far few reversible frameworks have been defined categorically, though [7] used category theory to describe the relationship between RCCS processes and their histories, and [2] used dagger categories to define a reversible process calculus called Π.

We define categories for the reversible event structures from [13, 15], defining morphisms for each category and functors, and in some cases adjunctions, between them, along with coproducts, and, in the case of general reversible event structures, products.

With a few exceptions [14, 16], reversible process calculi have always adopted *causal* reversibility. The reversible event structures of [13, 15] allow non-causal reversibility, inspired by bonding in biochemical processes. We here define subcategories of the reversible event structures of [15] which are (1) *stable*, meaning that the causes of an event cannot be ambiguous, which is clearly important for reversibility, and (2) *cause-respecting*, meaning that no action can be reversed unless all the actions caused by it have been reversed first [13], which can be seen as a safety property for causal reversibility. We show that under these conditions any reachable configuration is forwards reachable (Theorem 6.9).

V.T. Vasconcelos and P. Haller (Eds.): Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES'17)
EPTCS 246, 2017, pp. 49–60, doi:10.4204/EPTCS.246.9

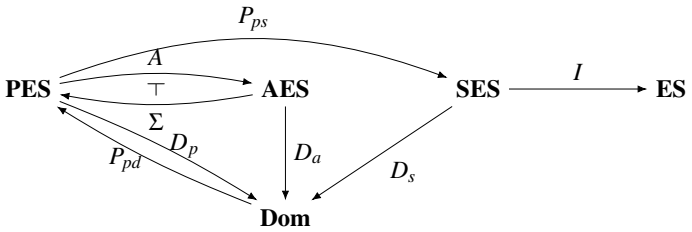


Figure 1: Categories of forward-only event structures and functors between them: PES were introduced in [10], and defined categorically along with **Dom**, **SES**, **ES**, D_p, P_{pd}, P_{ps}, and D_s in [19], **AES**, A, and D_a were introduced in [1], and Σ in [13]. The adjunction between A and Σ, denoted by $\dashv$, is new.

We also consider configuration systems [13], a model of concurrency intended to serve a similar purpose as domains do for the forward-only event structures, letting the various kinds of reversible event structures be translated into one formalism. We show that, just as stable domains can be modelled as event structures, so finitely enabled configuration systems can be modelled as general reversible event structures, giving a tight correspondence in the stable setting (Theorem 6.8).

Structure of the Paper. Section 2 reviews forwards-only event structures; Section 3 looks at reversible prime and asymmetric event structures, while Section 4 covers reversible general event structures. Section 5 describes the category of configuration systems, and Section 6 describes stable and cause-respecting reversible event structures and configuration systems.

2 Forwards-Only Event Structures

Before describing the different categories of reversible event structures, we recall the categories of forward-only event structures and functors between them, as seen in Figure 1.

A *prime event structure* consists of a set of events, and causality and conflict relations describing when these events can occur. If $e < e'$ then e' cannot happen unless e has already happened. And if $e \sharp e'$ then e and e' each prevent each other from occurring.

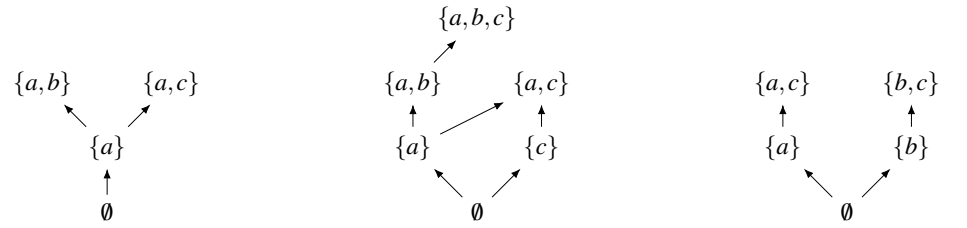
Definition 2.1 (Prime Event Structure [10])**.** *A prime event structure (PES) is a triple $\mathscr{E} = (E, <, \sharp)$, where E is the set of events and* causality*, $<$, and* conflict*, $\sharp$, are binary relations on E such that $\sharp$ is irreflexive and symmetric, $<$ is an irreflexive partial order such that for every $e \in E$, $\{e' \mid e' < e\}$ is finite, and $\sharp$ is hereditary with respect to $<$, i.e. for all $e, e', e'' \in E$, if $e \sharp e'$ and $e < e''$ then $e'' \sharp e'$.*

For any PES $\mathscr{E} = (E, <, \sharp)$, we say that $X \subseteq E$ is a configuration of $\mathscr{E}$ if X is left-closed under $<$ and conflict-free, meaning no $e, e' \in X$ exist, such that $e \sharp e'$. Configurations can be ordered by inclusion to form stable domains (coherent, prime algebraic, finitary partial orders) [19], as seen in Example 2.2.

Example 2.2. *The PES $\mathscr{E}_1$ with events a, b, c where $a < b$, $a < c$, and $c \sharp b$, has configurations $\emptyset$, $\{a\}$, $\{a, b\}$, and $\{a, c\}$, forming the domain seen in Figure 2a.*

Morphisms are defined on PESs in Definition 2.3, yielding the category **PES**. Morphisms on event structures act as a sort of synchronisation between the two structures, where if X is a configuration then $f(X)$ is too, and two events, e, e' can only synchronise with the same $f(e) = f(e')$ if they are in conflict.

Definition 2.3 (PES morphism [19])**.** *Let $\mathscr{E}_0 = (E_0, <_0, \sharp_0)$ and $\mathscr{E}_1 = (E_1, <_1, \sharp_1)$ be PESs. A morphism $f : \mathscr{E}_0 \to \mathscr{E}_1$ is a partial function $f : E_0 \to E_1$ such that for all $e \in E_0$, if $f(e) \neq \bot$ then $\{e_1 \mid e_1 <_1 f(e)\} \subseteq$*



(a) Domain of PES $\mathscr{E}_1$ (b) Domain of AES $\mathscr{E}_2$ (c) Domain of SES $\mathscr{E}_3$

Figure 2: Examples of domains representing event structures.

$\{f(e') \mid e' <_0 e\}$, *and for all* $e,e' \in E_0$, *if* $f(e) \neq \bot \neq f(e')$ *and* $f(e) \sharp_1 f(e')$ *or* $f(e) = f(e')$ *then* $e \sharp_0 e'$ *or* $e = e'$.

Asymmetric event structures [1] resemble prime event structures, with the difference being that the conflict relation $e \rhd e'$ ([1] uses the notation $e \nearrow e'$) is asymmetric, so that rather than e and e' being unable to coexist in a configuration, e' cannot be added to a configuration that contains e. The converse relation $e' \lhd e$ can be seen as precedence or weak causation, where if both events are in a configuration then e' was added first, as illustrated by Example 2.4. An AES-morphism is defined in the same way as a PES morphism, but replacing symmetric conflict with asymmetric. This gives the category **AES**.

Example 2.4. $\mathscr{E}_2 = (E, <, \lhd)$ *where* $E = \{a,b,c\}$ *and* $a < b$ *and* $b \lhd c$ *has configurations* $\emptyset$, $\{a\}$, $\{c\}$, $\{a,b\}$, $\{a,c\}$, *and* $\{a,b,c\}$, *and therefore* $D_a(\mathscr{E}_2)$ *is the domain seen in Figure 2b.*

General event structures, or simply *event structures*, work somewhat differently from PESs or AESs. Instead of causation and conflict, they have an enabling relation and a consistency relation.

Definition 2.5 (Event structure [19])**.** *An event structure (ES) is a triple* $\mathscr{E} = (E, \mathsf{Con}, \vdash)$, *where* E *is a set of events,* $\mathsf{Con} \subseteq_{\text{fin}} 2^E$ *is the consistency relation, such that if* $X \in \mathsf{Con}$ *and* $Y \subseteq X$ *then* $Y \in \mathsf{Con}$, *and* $\vdash \subseteq \mathsf{Con} \times E$ *is the enabling relation, such that if* $X \vdash e$ *and* $X \subseteq Y \in \mathsf{Con}$ *then* $Y \vdash e$.

Configurations are finitely consistent sets of events, where each event is deducible via the enabling relation. Once again we define an ES-morphism, giving us the category **ES** [19]. The idea behind them is much the same as for PES- and AES-morphisms. Enabling sets are treated in much the same way as causes, and consistent sets in the opposite way from conflict.

Stable event structures [19] form a full subcategory **SES** of **ES**. The idea is that in any given configuration, each event will have a unique enabling set.

Example 2.6. $\mathscr{E}_3 = (E, \mathsf{Con}, \vdash)$ *where* $E = \{a,b,c\}$, $\mathsf{Con} = \{\emptyset, \{a\}, \{b\}, \{a,c\}, \{b,c\}\}$, *and* $\emptyset \vdash a$, $\emptyset \vdash b$, $\{a\} \vdash c$, *and* $\{b\} \vdash c$ *can be represented by the domain* $D_s(\mathscr{E}_3)$ *seen in Figure 2c.*

3 Reversible Prime and Asymmetric Event Structures

Our goal is to define the categories and functors in the lower part of Figure 3.

We start by adding reversibility to PESs. When discussing reversible events we will use $\underline{e}$ to denote reversing e and e^* to denote that e may be performed or reversed. *Reversible prime event structures* [13] (Definition 3.1) consist of a set of events, E, some of which may be reversible, causality and conflict similar to a PES, reverse causality, which works similarly to causality, in that $e \prec \underline{e'}$ means e' can only be reversed in configurations containing e, and prevention, which resembles the asymmetric conflict of AESs, in that $e \rhd \underline{e'}$ means that e' can only be reversed in configurations not containing e.

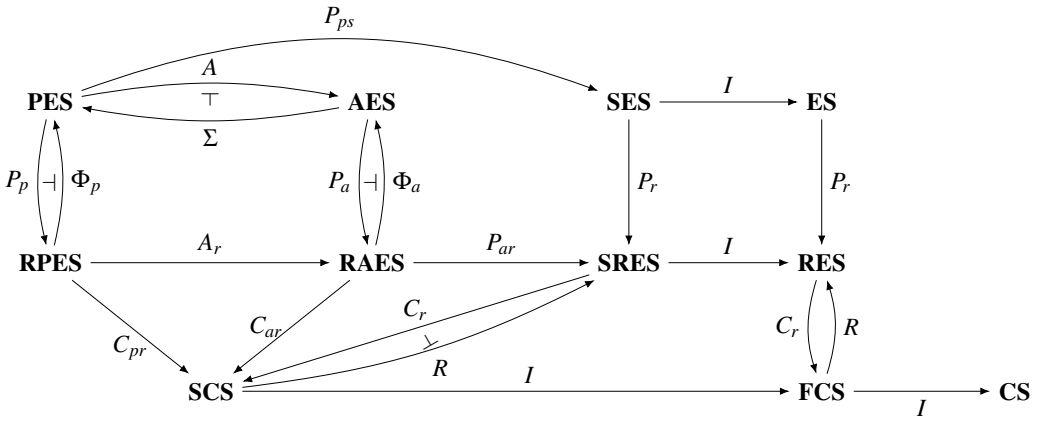


Figure 3: Categories of event structures and functors between them: We extend Figure 1 by categorically defining RPESs, RAESs, CSs, P_p, Φ_p, P_a, Φ_a, C_p, C_{pr}, C_a, C_{ar}, and A_r [13] and RESs and P_r [15]. The categories **SRES**, **SCS**, and **FCS**, and functors P_{pr}, P_{ar}, C_r, C, and R are new, as well as the noted adjunctions.

Definition 3.1 (RPES [13])**.** *A reversible prime event structure (RPES) is a sextuple* $\mathscr{E} = (E, F, <, \sharp, \prec, \rhd)$ *where* E *is the set of events,* $F \subseteq E$ *is the set of reversible events, and*

- $<$ *is an irreflexive partial order such that for every* $e \in E$, $\{e' \in E \mid e' < e\}$ *is finite and conflict-free*
- $\sharp$ *is irreflexive and symmetric such that if* $e < e'$ *then not* $e \sharp e'$
- $\rhd \subseteq E \times \underline{F}$ *is the prevention relation*
- $\prec \subseteq E \times \underline{F}$ *is the reverse causality relation where for each* $e \in F$, $e \prec \underline{e}$ *and* $\{e' \mid e' \prec \underline{e}\}$ *is finite and conflict-free and if* $e \prec \underline{e'}$ *then not* $e \rhd \underline{e'}$
- $\sharp$ *is hereditary with respect to sustained causation* $\ll$ *and* $\ll$ *is transitive, where* $e \ll e'$ *means that* $e < e'$ *and if* $e \in F$ *then* $e' \rhd \underline{e}$

As previously, in order to define the category **RPES**, we need a notion of morphism. An RPES-morphism can be seen as a combination of a PES-morphism for the forwards part and an AES-morphism for the reverse part, and reversible events can only synchronise with other reversible events. The category **RPES** has coproducts (Definition 3.2). A coproduct can be described as a choice between two event structures to behave as, as illustrated by Example 3.3.

Definition 3.2 (RPES coproduct)**.** *Given RPESs* $\mathscr{E}_0 = (E_0, F_0, <_0, \sharp_0, \prec_0, \rhd_0)$ *and* $\mathscr{E}_1 = (E_1, F_0, <_1, \sharp_1, \prec_1, \rhd_1)$*, their coproduct* $\mathscr{E}_0 + \mathscr{E}_1$ *is* $(E, F, <, \sharp, \prec, \rhd)$ *where:*

- $E = \{(0,e) \mid e \in E_0\} \cup \{(1,e) \mid e \in E_1\}$ *and* $F = \{(0,e) \mid e \in F_0\} \cup \{(1,e) \mid e \in F_1\}$
- *injection* i_j *exist such that for* $e \in E_j$, $i_j(e) = (j,e)$ *for* $j \in \{0,1\}$
- $(j,e) < (j',e')$ *iff* $j = j'$ *and* $e <_j e'$
- $(j,e) \sharp (j',e')$ *iff* $j \neq j'$ *or* $e \sharp_j e'$
- $(j,e) \prec \underline{(j',e')}$ *iff* $j = j'$ *and* $e \prec_j \underline{e'}$
- $(j,e) \rhd \underline{(j',e')}$ *iff* $e' \in F_{j'}$ *and* $j \neq j'$*, or* $e \rhd_j \underline{e'}$

Example 3.3 (RPES coproduct). *Given RPESs $\mathscr{E}_0 = (E_0, F_0, <_0, \sharp_0, \prec_0, \rhd_0)$ and $\mathscr{E}_1 = (E_1, F_1, <_1, \sharp_1, \prec_1, \rhd_1)$ where $E_0 = \{a,b\}$, $F_0 = \{a,b\}$, $a <_0 b$, $a \prec_0 \underline{b}$ and $E_1 = \{c,d\}$, $F_1 = \{c\}$, and $d \rhd_1 \underline{c}$, the coproduct $\mathscr{E}_0 + \mathscr{E}_1$ is $(E, F, <, \sharp, \prec, \rhd)$, where $E = \{(0,a),(0,b),(1,c),(1,d)\}$, $F = \{(0,a),(0,b),(1,c)\}$, $(0,a) < (0,b)$, $(0,a) \prec (0,b)$, $(0,a) \sharp (1,c)$, $(0,a) \sharp (1,d)$, $(0,b) \sharp (0,c)$, $(0,b) \sharp (0,d)$, $(0,a) \rhd (1,c)$, $(0,b) \rhd (1,c)$, $(1,c) \rhd (0,a)$, $(1,d) \rhd (0,a)$, $(1,c) \rhd (0,b)$, $(1,d) \rhd (0,b)$, and $(1,d) \rhd \underline{(1,c)}$.*

As we did with PESs, we will now add reversibility to AESs. *Reversible asymmetric event structures* (RAES) [13] (Definition 3.4) consist of events, some of which may be reversible, as well as causation and precedence, similar to an AES, except that $\prec$ is no longer a partial order, and instead just well-founded. In addition, both work on the reversible events, similarly to the RPES.

Definition 3.4 (RAES [13]). *A reversible asymmetric event structure (RAES) is a quadruple $\mathscr{E} = (E, F, \prec, \lhd)$ where E is the set of events, $F \subseteq E$ is the set of reversible events, and*

- *$\lhd \subseteq (E \cup \underline{F}) \times E$ is the irreflexive precedence relation*
- *$\prec \subseteq E \times (E \cup \underline{F})$ is the causation relation, which is irreflexive and well-founded, such that for all $\alpha \in E \cup \underline{F}$, $\{e \in E \mid e \prec \alpha\}$ is finite and has no $\lhd$-cycles, and for all $e \in F$, $e \prec \underline{e}$*
- *for all $e \in E$ and $\alpha \in E \cup \underline{F}$ if $e \prec \alpha$ then not $e \rhd \alpha$*
- *$e \prec\!\!\lhd e'$ implies $e \lhd e'$, where $e \prec\!\!\lhd e'$ means that $e \prec e'$ and if $e \in F$ then $e' \rhd \underline{e}$*
- *$\prec\!\!\lhd$ is transitive and if $e \sharp e'$ and $e \prec\!\!\lhd e''$ then $e'' \sharp e'$*

Once again we create a category **RAES** by defining RAES-morphisms. This definition is nearly identical to that of an AES-morphism, with the added condition that, like in the RPES morphism, reversible events can only synchronise with other reversible events. The category **RAES** has coproducts, defined very similarly to the RPES coproduct, though without symmetric conflict and combining both causation relations into one.

4 Reversible General Event Structures

The last kind of event structure we add reversibility to is the general event structure. The *reversible (general) event structure* differs from the general event structure, not only by allowing the reversal of events, but also by including a preventing set in the enabling relation, so that $X \circledcirc Y \vdash e$ means e is enabled in configurations that include all the events of X but none of the events of Y. An example of an RES can be seen in Figure 4b. In all examples we will use $X \circledcirc Y \vdash e^*$ as shorthand for $X' \circledcirc Y \vdash e^*$ whenever $X \subseteq X' \in \mathsf{Con}$

Definition 4.1 (RES [15]). *A reversible event structure (RES) is a triple $\mathscr{E} = (E, \mathsf{Con}, \vdash)$ where E is the set of events, $\mathsf{Con} \subseteq_{\text{fin}} 2^E$ is the consistency relation, which is left-closed, $\vdash \subseteq \mathsf{Con} \times 2^E \times (E \cup \underline{E})$ is the enabling relation, and (1) if $X \circledcirc Y \vdash e^*$ then $(X \cup \{e\}) \cap Y = \emptyset$, (2) if $X \circledcirc Y \vdash \underline{e}$ then $e \in X$, and (3) if $X \circledcirc Y \vdash e^*$, $X \subseteq X' \in \mathsf{Con}$, and $X' \cap Y = \emptyset$ then $X' \circledcirc Y \vdash e^*$.*

To define the category **RES**, we need to define a RES-morphism (Definition 4.2). With the exception of the requirements regarding preventing sets, it is identical to the definition of an ES-morphism. We treat the preventing set similarly to (asymmetric) conflict in PES, AES, RPES, and RAES-morphisms.

Definition 4.2 (RES morphism). *Let $\mathscr{E}_0 = (E_0, \mathsf{Con}_0, \vdash_0)$ and $\mathscr{E}_1 = (E_1, \mathsf{Con}_1, \vdash_1)$ be RESs. A morphism $f : \mathscr{E}_0 \to \mathscr{E}_1$ is a partial function $f : E_0 \to E_1$ such that*

- *for all $e \in E_0$, if $f(e) \neq \bot$ and $X \circledcirc Y \vdash_0 e^*$ then there exists a $Y_1 \subseteq E_1$ such that for all $e_0 \in E_0$, if $f(e_0) \in Y_1$ then $e_0 \in Y$ and $f(X) \circledcirc Y_1 \vdash_1 f(e)^*$*

- *for any* $X_0 \in \mathsf{Con}_0$, $f(X_0) \in \mathsf{Con}_1$
- *for all* $e, e' \in E_0$, *if* $f(e) = f(e') \neq \bot$ *and* $e \neq e'$ *then no* $X \in \mathsf{Con}_0$ *exists such that* $e, e' \in X$

As with **RPES** and **RAES**, **RES** has coproducts (Definition 4.3).

Definition 4.3 (RES coproduct)**.** *Given RESs* $\mathscr{E}_0 = (E_0, \mathsf{Con}_0, \vdash_0)$ *and* $\mathscr{E}_1 = (E_0, \mathsf{Con}_0, \vdash_0)$*, their coproduct* $\mathscr{E}_0 + \mathscr{E}_1$ *is* $(E, \mathsf{Con}, \vdash)$ *where:*

- $E = \{(0,e) \mid e \in E_0\} \cup \{(1,e) \mid e \in E_1\}$
- *injections* i_j *exist such that for* $e \in E_j$ $i_j(e) = (j,e)$ *for* $j \in \{0,1\}$
- $X \in \mathsf{Con}$ *iff* $\exists X_0 \in \mathsf{Con}_0 . i_0(X_0) = X$ *or* $\exists X_1 \in \mathsf{Con}_1 . i_1(X_1) = X$
- $X \odot Y \vdash (j,e)^*$ *iff* $\exists X_j, Y_j \in E_j$ *such that* $X_j \odot Y_j \vdash e^*$, $i_j(X_j) = X$, $Y = i_j(Y_j) \cup (E \setminus i_j(E_j))$

We also define the product of RESs (Definition 4.4). A product can be described as a parallel composition of two RESs. The reason we did not define the products of RPESs or RAESs is, that while the ES product defined in [19] easily translates to RESs, definitions of PES products, such as the one based on mapping the PESs into a domain and back seen in [17], are far more complex and difficult to translate directly to a reversible setting. Since we do not have mappings from CSs to RPESs or RAESs, this is not a possible solution. Example 4.5 shows the product of two RESs.

Definition 4.4 (RES product)**.** *Given RESs* $\mathscr{E}_0 = (E_0, \mathsf{Con}_0, \vdash_0)$ *and* $\mathscr{E}_1 = (E_1, \mathsf{Con}_1, \vdash_1)$*, their partially synchronous product* $\mathscr{E}_0 \times \mathscr{E}_1$ *is* $(E, \mathsf{Con}, \vdash)$ *where:*

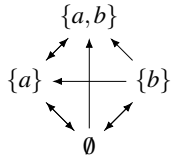
- $E = E_0 \times_* E_1 = \{(e,*) \mid e \in E_0\} \cup \{(*,e) \mid e \in E_1\} \cup \{(e,e') \mid e \in E_0 \text{ and } e' \in E_1\}$
- *there exist projections* π_0, π_1 *such that for* $(e_0, e_1) \in E$, $\pi_i((e_0,e_1)) = e_i$
- $X \in \mathsf{Con}$ *if* $\pi_0(X) \in \mathsf{Con}_0$, $\pi_1(X) \in \mathsf{Con}_1$, *and for all* $e, e' \in X$, *if* $\pi_0(e) = \pi_0(e')$ *or* $\pi_1(e) = \pi_1(e')$ *then* $e = e'$
- $X \odot Y \vdash e^*$ *if*
  - *if* $\pi_0(e) \neq *$ *then* $\pi_0(X) \odot \pi_0(Y) \vdash \pi_0(e)^*$
  - *if* $\pi_1(e) \neq *$ *then* $\pi_1(X) \odot \pi_1(Y) \vdash \pi_1(e)^*$
  - *if* $e^* = \underline{e}$ *then* $e \in X$

Example 4.5 (RES product)**.** *Given RESs* $\mathscr{E}_0 = (E_0, \mathsf{Con}_0, \vdash_0)$ *and* $\mathscr{E}_1 = (E_1, \mathsf{Con}_1, \vdash_1)$*, where* $E_0 = \{a,b\}$, $\mathsf{Con}_0 = 2^{E_0}$, $\emptyset \odot \emptyset \vdash_0 a$, $\{a\} \odot \emptyset \vdash_0 b$, $\{a,b\} \odot \emptyset \vdash_0 \underline{b}$, *and* $\{a\} \odot \emptyset \vdash_0 \underline{a}$ *and* $E_1 = \{c\}$, $\mathsf{Con}_1 = \{\emptyset, \{c\}\}$, $\emptyset \odot \emptyset \vdash_1 c$, *and* $\{c\} \odot \vdash_1 \underline{c}$, *the product* $\mathscr{E}_0 \times \mathscr{E}_1$ *is* $(E, \mathsf{Con}, \vdash)$ *where* $E = \{(a,*),(b,*),(a,c),(b,c),(*,c)\}$, $\mathsf{Con} = \{\emptyset, \{(a,*)\}, \{(b,*)\}, \{(a,c)\}, \{(b,c)\}, \{(*,c)\}, \{(a,*),(b,*)\}, \{(a,*),(b,c)\}, \{(a,*),(*,c)\}, \{(a,c),(b,*)\}, \{(b,*),(*,c)\}, \{(a,*),(b,*),(*,c)\}\}$, $\emptyset \odot \emptyset \vdash (a,*)$, $\{(a,*) \odot \emptyset \vdash (b,*)$, $\{(a,c)\} \odot \emptyset \vdash (b,*)$, $\emptyset \odot \emptyset \vdash (a,c)$, $\{(a,*) \odot \emptyset \vdash (b,c)$, $\{(a,c)\} \odot \emptyset \vdash (b,c)$, $\emptyset \odot \emptyset \vdash (*,c)$, $\{(a,*)\} \odot \emptyset \vdash \underline{(a,*)}$, $\{(b,*),(a,*)\} \odot \emptyset \vdash \underline{(b,*)}$, $\{(b,*),(a,c)\} \odot \emptyset \vdash \underline{(b,*)}$, $\{(a,c)\} \odot \emptyset \vdash \underline{(a,c)}$, $\{(b,c),(a,*)\} \odot \emptyset \vdash \underline{(b,c)}$, *and* $\{(*,c)\} \odot \emptyset \vdash \underline{(*,c)}$.

We also create functors from **RPES** and **RAES** to **RES**. While not all AESs have ESs which map to the same domain, RAESs map into RESs using the preventing set to model asymmetric conflict as described in Definition 4.6.

Definition 4.6 (From **RAES** to **RES**)**.** *The mapping* $P_{ar} : \boldsymbol{RAES} \to \boldsymbol{RES}$ *is defined as:*

- $P_{ar}(\mathscr{E}) = (E, \mathsf{Con}, \vdash)$ *where*
  $\mathsf{Con} = \{X \subseteq E \mid \lhd \text{ is well-founded on } X\}$
  $X \odot Y \vdash e$ *if* $\{e' \mid e' \prec e\} \subseteq X \in \mathsf{Con}$, $Y = \{e' \mid e' \rhd e\}$, $X \cap Y = \emptyset$, *and* $e \in E$
  $X \odot Y \vdash \underline{e}$ *if* $\{e' \mid e' \prec \underline{e}\} \subseteq X \in \mathsf{Con}$, $Y = \{e' \mid e' \rhd \underline{e}\}$, $X \cap Y = \emptyset$, *and* $e \in F$
- $P_{ar}(f) = f$



(a) CS $\mathscr{C}$

$\mathscr{E} = (E, \mathsf{Con}, \vdash)$ where
$E = \{a, b\}$
$\mathsf{Con} = \{\emptyset, \{a\}, \{b\}, \{a,b\}\}$
$\emptyset \odot \emptyset \vdash a$, $\emptyset \odot \emptyset \vdash b$,
$\{b\} \odot \emptyset \vdash \underline{b}$, $\{a\} \odot \{b\} \vdash \underline{a}$

(b) RES $\mathscr{E}$

Figure 4: A CS and the corresponding RES such that $R(\mathscr{C}) = \mathscr{E}$ and $C_r(\mathscr{E}) = \mathscr{C}$.

5 Configuration Systems

Configuration systems perform a similar role in the reversible setting to domains in the forward-only setting, though they have a more operational character. A configuration system [13] (Definition 5.1) consists of a set of events, E, some of which, F, are reversible, a set C of configurations on these, and an optionally labelled transition relation $\rightarrow$ such that if $X \xrightarrow{A \cup \underline{B}} Y$ then the events of A can happen and the events of B can be undone in any order starting from configuration X, resulting in Y. We also leave out Y when describing such a transition, since it is implied that $Y = (X \setminus B) \cup A$. A CS is shown in Figure 4a.

Definition 5.1 (Configuration system [13])**.** *A configuration system (CS) is a quadruple $\mathscr{C} = (E, F, \mathsf{C}, \rightarrow)$ where E is a set of events, $F \subseteq E$ is a set of reversible events, $\mathsf{C} \subseteq 2^E$ is the set of configurations, and $\rightarrow \subseteq \mathsf{C} \times 2^{E \cup \underline{F}} \times \mathsf{C}$ is an optionally labelled transition relation such that if $X \xrightarrow{A \cup \underline{B}} Y$ then:*

- *$A \cap X = \emptyset$, $B \subseteq X \cap F$, and $Y = (X \setminus B) \cup A$*
- *for all $A' \subseteq A$ and $B' \subseteq B$, we have $X \xrightarrow{A' \cup \underline{B'}} Z \xrightarrow{(A \setminus A') \cup \underline{(B \setminus B')}} Y$, meaning $Z = (X \setminus B') \cup A' \in \mathsf{C}$*

We define a notion of morphism (Definition 5.2), creating the category **CS**.

Definition 5.2 (CS-morphism)**.** *Let $\mathscr{C}_0 = (E_0, F_0, \mathsf{C}_0, \rightarrow_0)$ and $\mathscr{C}_1 = (E_1, F_1, \mathsf{C}_1, \rightarrow_1)$ be configuration systems. A configuration system morphism is a partial function $f : E_0 \rightarrow E_1$ such that*

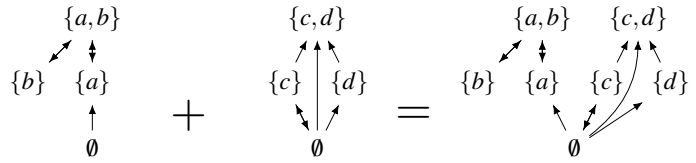
- *for any $X, Y \in \mathsf{C}_0$, $A \subseteq E_0$, and $B \subseteq F_0$, if $X \xrightarrow{A \cup \underline{B}}_0 Y$ then $f(X) \xrightarrow{f(A) \cup f(\underline{B})}_1 f(Y)$*
- *for any $X \in \mathsf{C}_0$, $f(X) \in \mathsf{C}_1$*
- *for all $e_0, e_0' \in E_0$, if $f(e_0) = f(e_0') \neq \perp$ and $e_0 \neq e_0'$ then there exists no $X \in \mathsf{C}_0$ such that $e_0, e_0' \in X$*

We also define the coproduct of two CSs (Definition 5.3). This is illustrated with CSs modelling the RPESs and RESs from Examples 3.3 and 4.5 in Example 5.4.

Definition 5.3 (CS coproduct)**.** *Given CSs $\mathscr{C}_0 = (E_0, F_0, \mathsf{C}_0, \rightarrow_0)$ and $\mathscr{C}_1 = (E_1, F_1, \mathsf{C}_1, \rightarrow_1)$, their coproduct $\mathscr{C}_0 + \mathscr{C}_1 = (E, F, \mathsf{C}, \rightarrow)$ where:*

- *$E = \{(0,e) \mid e \in E_0\} \cup \{(1,e) \mid e \in E_1\}$ and $F = \{(0,e) \mid e \in F_0\} \cup \{(1,e) \mid e \in F_1\}$*
- *injections i_j exist such that for $e \in E_j$ $i_j(e) = (j,e)$ for $j \in \{0,1\}$*
- *$X \in \mathsf{C}$ iff $\exists X_0 \in \mathsf{C}_0 . i_0(X_0) = X$ or $\exists X_1 \in \mathsf{C}_1 . i_1(X_1) = X$*
- *$X \xrightarrow{A \cup \underline{B}} Y$ iff there exists $j \in \{0,1\}$ such that there exist $X_j, Y_j, A_j, B_j \subseteq E_j$ such that $i_j(X_j) = X$, $i_j(Y_j) = Y$, $i_j(A_j) = A$, $i_j(B_j) = B$, and $X_j \xrightarrow{A_j \cup \underline{B_j}}_j Y_j$.*

Example 5.4 (Coproduct)**.**

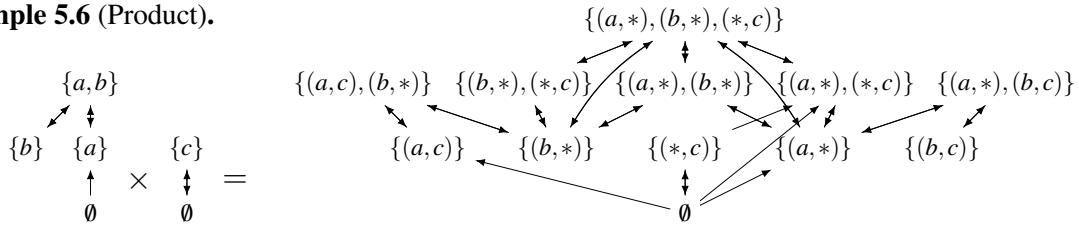


We also define the product of CSs (Definition 5.5). This is illustrated in Example 5.6, where the CSs represent the RESs of Example 4.5.

Definition 5.5 (CS product)**.** *Given CSs $\mathscr{C}_0 = (E_0, F_0, C_0, \to_0)$ and $\mathscr{C}_1 = (E_1, F_1, C_1, \to_1)$, their partially synchronous product $\mathscr{C}_0 \times \mathscr{C}_1 = (E, F, C, \to)$ where:*

- $E = E_0 \times_* E_1 = \{(e,*) \mid e \in E_0\} \cup \{(*,e) \mid e \in E_1\} \cup \{(e,e') \mid e \in E_0 \text{ and } e' \in E_1\}$
- $F = F_0 \times_* F_1 = \{(e,*) \mid e \in F_0\} \cup \{(*,e) \mid e \in F_1\} \cup \{(e,e') \mid e \in F_0 \text{ and } e' \in F_1\}$
- *there exist projections π_0, π_1 such that for $(e_0,e_1) \in E$, $\pi_i((e_0,e_1)) = e_i$*
- *$X \in C$ if $\pi_0(X) \in C_0$, $\pi_1(X) \in C_1$, and for all $e,e' \in X$, if $\pi_0(e) = \pi_0(e')$ or $\pi_1(e) = \pi_1(e')$ then $e = e'$*
- *$X \xrightarrow{A \cup \underline{B}} Y$ if $B \subseteq X$ and*
  - *if $\pi_0(A \cup B) \neq \emptyset$ then $\pi_0(X) \xrightarrow{\pi_0(A) \cup \underline{\pi_0(B)}}_0 \pi_0(Y)$*
  - *if $\pi_1(A \cup B) \neq \emptyset$ then $\pi_1(X) \xrightarrow{\pi_1(A) \cup \underline{\pi_1(B)}}_1 \pi_1(Y)$*

Example 5.6 (Product)**.**



We define a functor C_r from **RES** to **CS** (Definition 5.7).

Definition 5.7 (From **RES** to **CS**)**.** *The mapping C_r : **RES** $\to$ **CS** is defined as*

- *$C_r((E, Con, \vdash)) = (E, F, C, \to)$, where (1) $e \in F$ if there exists X, Y such that $X \ominus Y \vdash \underline{e}$, (2) $C \in C$ if for all $X \subseteq_{\text{fin}} C$, $X \in Con$, and (3) for $X, Y \in C$, $X \xrightarrow{A \cup \underline{B}} Y$ if*
  - *$Y = (X \setminus B) \cup A$, $A \cap X = \emptyset$, $B \subseteq X$, and $X \cup A \in C$*
  - *for all e in A, $X' \ominus Z \vdash e$ for some X', Z such that $X' \subseteq_{\text{fin}} X \setminus B$ and $Z \cap (X \cup A) = \emptyset$*
  - *for all $e \in B$, $X' \ominus Z \vdash \underline{e}$ for some X', Z such that $X' \subseteq_{\text{fin}} X \setminus (B \setminus \{e\})$ and $Z \cap (X \cup A) = \emptyset$*
- *$C_r(f) = f$*

Applying this functor to a RES results in a *finitely enabled CS* (FCS), that is to say a CS such that there does not exist a transition from an infinite configuration $X \xrightarrow{A \cup \underline{B}}$, such that there does not exist a finite configuration $X' \subseteq_{\text{fin}} X$ such that $X' \xrightarrow{A \cup \underline{B}}$ and whenever $X' \subseteq X'' \subseteq X$ there exists a transition $X'' \xrightarrow{A \cup \underline{B}}$. We call the category of these CSs and the CS-morphisms between them **FCS**, and describe a functor, R, from this category to **RES** in Definition 5.8. An example of C_r and R can be seen in Figure 4.

Definition 5.8 (From **FCS** to **RES**)**.** *The mapping R : **FCS** $\to$ **RES** is defined as:*

- *$R((E, F, C, \to)) = (E, Con, \vdash)$ where $X \in Con$ if $X \subseteq_{\text{fin}} C \in C$ and:*

– *If* $X \xrightarrow{\{e^*\}}$ *and*
  * $X' \subseteq X$, $X' \xrightarrow{\{e^*\}}$, *and whenever* $X' \subseteq X'' \subseteq X$ *there exists a transition* $X'' \xrightarrow{\{e^*\}}$
  * *no* $X'' \subset X'$ *exists such that* $X'' \xrightarrow{\{e^*\}}$, *and whenever* $X'' \subseteq X''' \subseteq X$ *there exists a transition* $X''' \xrightarrow{\{e^*\}}$
  * *no* $X'' \supset X$ *exists such that* $X'' \xrightarrow{\{e^*\}}$, *and whenever* $X' \subseteq X''' \subseteq X''$ *there exists a transition* $X''' \xrightarrow{\{e^*\}}$

  then
  * *if* $e^* = e$, *then for all* $X'' \in \mathit{Con}$ *such that* $X' \subseteq X'' \subseteq X \cup \{e\}$, $X'' \otimes E \setminus X \cup \{e\} \vdash e$
  * *if* $e^* = \underline{e}$, *then for all* $X'' \in \mathit{Con}$ *such that* $X' \subseteq X'' \subseteq X$, $X'' \otimes E \setminus (X \setminus \{e\} \vdash \underline{e}$

- $R(f) = f$

As Theorem 6.8 states, C_r and R are in many cases inverses of each other.

6 Stable Reversible Event Structures and Configuration Systems

Similarly to the stable event structures, we define the *stable reversible event structures* (Definition 6.1), and create the category **SRES** consisting of SRESs and the RES-morphisms between them. SRESs and SESs are defined identically, with the exception that in an SRES the preventing sets are included as well, and treated in much the same way as the enabling sets. Like in a SES, an event in a configuration of a SRES will have one possible cause as long as the configuration has been reached by only going forwards.

Definition 6.1 (Stable RES)**.** *A stable reversible event structure (SRES) is an RES* $\mathscr{E} = (E, \mathit{Con}, \vdash)$ *such that for all* $e^* \in E$ *if* $X \otimes Y \vdash e^*$, $X' \otimes Y' \vdash e^*$, *and* $X \cup X' + e^* \in \mathit{Con}$ *then* $X \cap X' \otimes Y \cap Y' \vdash e^*$.

Similarly, we can define a *stable configuration system* (Definition 6.2). This has the property that if $\mathscr{E}$ is a SRES then $C_r(\mathscr{E})$ is a SCS, and if $\mathscr{C}$ is a SCS then $R(\mathscr{C})$ is a SRES.

Definition 6.2 (Stable CS)**.** *A stable CS (SCS) is an FCS* $\mathscr{C} = (E, F, \mathsf{C}, \to)$ *such that*

1. C *is downwards closed*
2. *For all* $e \in F$, *there exists a transition* $X \xrightarrow{\underline{e}}$
3. *For* $X_1, X_2, X_3 \in \mathsf{C}$:
   (a) *if* $X_1 \subseteq X_2 \subseteq X_3$, $X_1 \xrightarrow{A \cup \underline{B}}$, *and* $X_3 \xrightarrow{A \cup \underline{B}}$, *then* $X_2 \xrightarrow{A \cup \underline{B}}$
   (b) *if* $((X_1 \cup X_2) \setminus B) \cup A \in \mathsf{C}$, $X_1 \xrightarrow{A \cup \underline{B}}$, *and* $X_2 \xrightarrow{A \cup \underline{B}}$, *then* $X_1 \cup X_2 \xrightarrow{A \cup \underline{B}}$ *and* $X_1 \cap X_2 \xrightarrow{A \cup \underline{B}}$
   (c) *if* $X_0, X_1, X_2, X_3 \in \mathsf{C}$, $A_0, A_1, B_0, B_1 \subseteq E$ *and there exist transitions* $X_0 \xrightarrow{A_0 \cup \underline{B_0}} X_1$, $X_0 \xrightarrow{A_1 \cup \underline{B_1}} X_2$, $X_1 \xrightarrow{A_1 \cup \underline{B_1}} X_3$, *and* $X_2 \xrightarrow{A_0 \cup \underline{B_0}} X_3$, *then* $X_0 \xrightarrow{A_0 \cup A_1 \cup \underline{B_0 \cup B_1}} X_3$

Figure 4a shows a stable CS. One way to make it not stable would be to remove the transition from $\emptyset$ to $\{a, b\}$, since that would violate Item 3c.

As [13] did for RPESs and RAESs, we define a subcategory of *cause-respecting* RESs in Definition 6.4. This is based on the idea that if e' enables e, then e' cannot be reversed from a configuration which does not have another possible enabling set for e. Unlike causal reversibility [5] however, a configuration fulfilling these conditions does not guarantee that reversing is possible.

Definition 6.3 (Minimal enabling configurations for RES $m_{RES}(e)$)**.** *Given an RES* $\mathscr{E} = (E, \mathit{Con}, \vdash)$ *the set of minimal enabling configurations of an event* $e \in E$ *is defined as:*

$m_{RES}(e) = \{X \in \mathit{Con} \mid \exists Y. X \otimes Y \vdash e \text{ and } \forall X', Y'. X' \otimes Y' \vdash e \Rightarrow X' \not\subset X\}$

Definition 6.4 (CRES). *A CRES $\mathscr{E} = (E, \mathsf{Con}, \vdash)$ is an RES such that for all $e, e' \in E$, $e' \in X \in m_{RES}(e)$ iff whenever $X' \odot Y' \vdash \underline{e'}$, we have $e \in Y'$ or there exists an $X'' \subseteq X' \setminus \{e'\}$ such that $X'' \in m_{RES}(e)$.*

Moreover we define a *cause-respecting* CS in much that same way as a CRES (Definition 6.6). This has the property that if $\mathscr{E}$ is a CRES then $C_r(\mathscr{E})$ is a CCS, and if $\mathscr{C}$ is a finitely enabled CCS then $R(\mathscr{C})$ is a CRES. In addition, the functors C_r and R are inverses of each other (Theorem 6.8).

We can then prove Theorem 6.9, which is analogous to a property of cause-respecting RPESs and RAESs proved in [13]. The CS in Figure 4a is cause-respecting, but removing the transition from $\emptyset$ to $\{a\}$ would change that.

Definition 6.5 (Minimal enabling configurations for CS $m_{CS}(e)$). *Given a CS $\mathscr{C} = (E, F, \mathsf{C}, \to)$ the set of minimal enabling configurations of an event $e \in E$ is defined as*

$$m_{CS}(e) = \{X \in \mathsf{C} \mid X \xrightarrow{\{e\}} \text{ and } \forall X'.X' \xrightarrow{\{e\}} \Rightarrow X' \not\subset X\}$$

Definition 6.6 (CCS). *A cause-respecting CS $\mathscr{C} = (E, F, \mathsf{C}, \to)$ is a CS such that if $e' \in X \in m_{CS}(e)$, then whenever $X' \xrightarrow{\{\underline{e'}\}} Y'$ and $e \in X'$, there exists an $X'' \subseteq Y'$ such that $X'' \in m_{CS}(e)$.*

Proposition 6.7. *If $\mathscr{E}$ is a CSRES then $C_r(\mathscr{E})$ is a CSCS, and if $\mathscr{C}$ is a CSCS then $R(\mathscr{C})$ is a CSRES.*

Theorem 6.8. *Given a SCS $\mathscr{C} = (E, F, \mathsf{C}, \to)$, $C_r(R(\mathscr{C})) = \mathscr{C}$ if C is downwards closed, and for all $e \in F$ there exists a transition $X \xrightarrow{\underline{e}}$. If $\mathscr{E} = (E, \mathsf{Con}, \vdash)$ is a SRES with no "unnecessary" enablings $X \odot Y' \vdash e^*$ such that $X \odot Y \vdash e^*$ for $Y \subset Y'$ then $R(C_r(\mathscr{E})) = \mathscr{E}$.*

Theorem 6.9. *If $\mathscr{C} = (E, F, \mathsf{C}, \to)$ is a CSCS then every reachable configuration is forwards reachable.*

7 Conclusion

We have defined categories for configuration systems (CS), reversible prime event structures (RPES), reversible asymmetric event structures (RAES), and reversible general event structures (RES), and functors between them, showing all the event structures can be modelled as CSs and conversely finitely enabled CSs can be modelled as RESs in a way that preserves morphisms, with the two directions being inverses in the stable setting (Theorem 6.8). We also defined coproducts for each of these categories, though products only for RESs and CSs.

With a view to the semantics of causal reversible process calculi, we have also defined stable and cause-respecting subcategories of RESs, in which every reachable configuration is forwards reachable (Theorem 6.9).

Future Work: Defining a product of RPESs and RAESs will likely be trickier than for RESs, since definitions of products of prime event structures are far more complex than those of general event structures [17], and we note that the product of asymmetric event structures is as yet undefined. We plan to formulate a notion of 'causal' RES which strengthens the 'cause-respecting' safety condition with a liveness condition.

Acknowledgements: We thank the referees for their helpful comments. This work was partially supported by EPSRC DTP award; EPSRC projects EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1 and EP/N028201/1; EU FP7 612985 (UPSCALE); and EU COST Action IC1405.

References

[1] Paolo Baldan, Andrea Corradini & Ugo Montanari (2001): *Contextual Petri Nets, Asymmetric Event Structures, and Processes*. *Information and Computation* 171(1), pp. 1 – 49, doi:10.1006/inco.2001.3060.

[2] William J Bowman, Roshan P James & Amr Sabry (2011): *Dagger traced symmetric monoidal categories and reversible programming*. In: *Workshop on Reversible Computation, RC 2011*. Available at `https://williamjbowman.com/resources/cat-rev.pdf`.

[3] Ioana Cristescu, Jean Krivine & Daniele Varacca (2013): *A Compositional Semantics for the Reversible pi-Calculus*. In: *IEEE Symposium on Logic in Computer Science*, LICS '13, IEEE Computer Society, Washington, DC, USA, pp. 388–397, doi:10.1109/LICS.2013.45.

[4] Ioana Cristescu, Jean Krivine & Daniele Varacca (2016): *Rigid Families for the Reversible π-Calculus*. In: *Reversible Computation - 8th International Conference, RC 2016, Bologna, Italy, July 7-8, 2016, Proceedings*, *Lecture Notes in Computer Science* 9720, Springer, pp. 3–19, doi:10.1007/978-3-319-40578-0_1.

[5] Vincent Danos & Jean Krivine (2004): *Reversible Communicating Systems*. In Philippa Gardner & Nobuko Yoshida, editors: *CONCUR*, *LNCS* 3170, Springer, Berlin, Heidelberg, pp. 292–307, doi:10.1007/978-3-540-28644-8_19.

[6] Vincent Danos & Jean Krivine (2007): *Formal Molecular Biology Done in CCS-R*. *Electronic Notes in Theoretical Computer Science* 180(3), pp. 31 – 49, doi:10.1016/j.entcs.2004.01.040.

[7] Vincent Danos, Jean Krivine & Paweł Sobociński (2007): *General Reversibility*. In: *EXPRESS*, 175(3), pp. 75 – 86, doi:10.1016/j.entcs.2006.07.036.

[8] Ivan Lanese, Claudio Antares Mezzina & Jean-Bernard Stefani (2010): *Reversing Higher-Order Pi*. In Paul Gastin & François Laroussinie, editors: *CONCUR*, *LNCS* 6269, Springer, Berlin, Heidelberg, pp. 478–493, doi:10.1007/978-3-642-15375-4_33.

[9] Ivan Lanese, Claudio Antares Mezzina & Francesco Tiezzi (2014): *Causal-Consistent Reversibility*. *Bulletin of the EATCS* 114, p. 17. Available at `https://hal.inria.fr/hal-01089350`.

[10] Mogens Nielsen, Gordon Plotkin & Glynn Winskel (1979): *Petri nets, event structures and domains*. In Gilles Kahn, editor: *Semantics of Concurrent Computation*, *LNCS* 70, Springer, Berlin, Heidelberg, pp. 266–284, doi:10.1007/BFb0022474.

[11] Iain Phillips & Irek Ulidowski (2006): *Reversing Algebraic Process Calculi*. In Luca Aceto & Anna Ingólfsdóttir, editors: *FOSSACS*, *LNCS* 3921, Springer, Berlin, Heidelberg, pp. 246–260, doi:10.1007/11690634_17.

[12] Iain Phillips & Irek Ulidowski (2007): *Reversibility and Models for Concurrency*. *Electr. Notes Theor. Comput. Sci.* 192(1), pp. 93–108, doi:10.1016/j.entcs.2007.08.018.

[13] Iain Phillips & Irek Ulidowski (2015): *Reversibility and asymmetric conflict in event structures*. *Journal of Logical and Algebraic Methods in Programming* 84(6), pp. 781 – 805, doi:10.1016/j.jlamp.2015.07.004.

[14] Iain Phillips, Irek Ulidowski & Shoji Yuen (2013): *A Reversible Process Calculus and the Modelling of the ERK Signalling Pathway*. In Robert Glück & Tetsuo Yokoyama, editors: *RC*, *LNCS* 7581, Springer, Berlin, Heidelberg, pp. 218–232, doi:10.1007/978-3-642-36315-3_18.

[15] Iain Phillips, Irek Ulidowski & Shoji Yuen (2013): *Modelling of Bonding with Processes and Events*. In Gerhard W. Dueck & D. Michael Miller, editors: *RC*, *LNCS* 7948, Springer, Berlin, Heidelberg, pp. 141–154, doi:10.1007/978-3-642-38986-3_12.

[16] Irek Ulidowski, Iain Phillips & Shoji Yuen (2014): *Concurrency and Reversibility*. In Shigeru Yamashita & Shin-ichi Minato, editors: *RC*, *LNCS* 8507, Springer, Cham, pp. 1–14, doi:10.1007/978-3-319-08494-7_1.

[17] Frits W Vaandrager (1989): *A simple definition for parallel composition of prime event structures*. CS-R8903, Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands. Available at `http://www.sws.cs.ru.nl/publications/papers/fvaan/CS-R8903.pdf`.

[18] Glynn Winskel (1982): *Event structure semantics for CCS and related languages*. In Mogens Nielsen & Erik Meineche Schmidt, editors: *ICALP*, *LNCS* 140, Springer, Berlin, Heidelberg, pp. 561–576, doi:10.1007/BFb0012800.

[19] Glynn Winskel (1987): *Event structures*. In W. Brauer, W. Reisig & G. Rozenberg, editors: *Petri Nets: Applications and Relationships to Other Models of Concurrency. ACPN*, *LNCS* 255, Springer, Berlin, Heidelberg, pp. 325–392, doi:10.1007/3-540-17906-2_31.

Best-by-Simulations: A Framework for Comparing Efficiency of Reconfigurable Multicore Architectures on Workloads with Deadlines

Sanjiva Prasad
Indian Institute of Technology Delhi
New Delhi, India
sanjiva@cse.iitd.ac.in

Energy consumption is a major concern in multicore systems. Perhaps the simplest strategy for reducing energy costs is to use only as many cores as necessary while still being able to deliver a desired quality of service. Motivated by earlier work on a dynamic (heterogeneous) core allocation scheme for H.264 video decoding that reduces energy costs while delivering desired frame rates, we formulate operationally the general problem of executing a sequence of actions on a reconfigurable machine while meeting a corresponding sequence of absolute deadlines, with the objective of reducing cost. Using a transition system framework that associates costs (e.g., time, energy) with executing an action on a particular resource configuration, we use the notion of amortised cost to formulate in terms of simulation relations appropriate notions for comparing deadline-conformant executions. We believe these notions can provide the basis for an operational theory of optimal cost executions and performance guarantees for approximate solutions, in particular relating the notion of simulation from transition systems to that of competitive analysis used for, e.g., online algorithms.

1 Introduction

Video decoding [21], an almost ubiquitous application on machines ranging from mobile phones to server machines, is amenable to execution on *embedded multicore* platforms — multi-threaded implementations of the H.264 codec [12] run on processors such as Intel Silvermont (homogeneous multicore) [13] and ARM Cortex A15 (heterogeneous multicore, based on the delightfully named big.LITTLE architectural model). High video quality means better resolution and higher frame rates, which in turn requires more computation and thus more energy. The required frame rate determines a *budgeted per-frame decode time*, and thus a *series of deadlines* for decoding each of a series of frames. The standard implementations utilise as many cores as available on the multicore platform in order to meet performance requirements.

To reduce energy consumption, Pal *et al.* proposed and implemented dynamic core allocation schemes in which cores are switched on or off using clock gating (or in heterogeneous multicores, smaller cores used instead of larger ones) according to the per-frame decoding requirements [20]. The basic idea is that since frames are often decoded well within the budgeted decode time, if deadlines can still be met by using fewer/smaller cores for decoding a frame, then the same performance can be achieved with lower energy consumption. By measuring slack and overshoot over the budgeted decode time and amortising these across frames, their schemes are able to save energy without missing any performance deadlines. Simulations on Sniper [9] for timing and McPAT [19] for energy measurements show energy savings of 6% to 61% while strictly adhering to the required performance of 75 fps on homogeneous multicore architectures, and 2% to 46% while meeting a performance of 25 fps on heterogeneous multicore architectures.

V.T. Vasconcelos and P. Haller (Eds.): Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES'17)
EPTCS 246, 2017, pp. 61–71, doi:10.4204/EPTCS.246.10

There, however, is no corresponding theoretical framework for (1) justifying the correctness of such schemes, or (2) comparing the performance of difference multicore (re)configurations on a given workload. While there are algorithmic optimisation approaches for structured problems in which the trade-offs between achieving an objective in a timely manner and the cost incurred for doing so are expressed, there are few formulations in *operational semantic* terms.

In this paper, we generalise the video decoding problem to the following abstract setting: "Suppose we are given a workload consisting of a sequence of actions, each of which has to be performed by a given deadline. Suppose there are different computational machine configurations (let *Conf* denote the set of these configurations) on which these actions may be executed, with possibly different costs per action-configuration combination.

1. Can the sequence of actions be performed on some machine configuration while meeting each deadline?
2. Is a given reconfiguration scheme (strategy/heuristic) correct (i.e., meets all deadlines)?
3. How can we compare the cost of execution according to a reconfiguration heuristic/strategy versus that on the baseline configuration?
4. Is it possible to express performance guarantees of a reconfiguration scheme with respect to an optimal strategy?

This generalisation allows us to examine the execution of arbitrary programs, expressed as a sequence of atomic tasks or workloads (not just video decoding) on a variety of architectures (not only multicores), particularly those that support reconfiguration, where we seek to reduce the cost of execution (not merely energy), subject to some performance deadlines.

The trade-offs involved are non-trivial, since different actions require differing processing times, with there being no simple method for anticipating the number and kinds of future actions (the problem is posed as an "online" one). For example, it is not entirely obvious whether while trying to save energy by using a slower computational configuration to perform an action, we will have enough time for processing subsequent actions without missing deadlines. On the other hand, being too conservative and operating only on the fastest configurations may mean forgoing opportunities for saving energy. Note that the problem is not of task scheduling but rather of resource allocation to meet a performance constraint (and then of finding close-to-optimal-cost executions; also see §1.1).

In this work, we present an operational semantics framework for specifying the execution of a workload in terms of *cumulative weighted transition systems*, which lets us record execution times (and then energy consumption). We then use the notion of simulation to express the execution of fixed workloads on different computational resource configurations as well as the specification of a deadline-meeting execution (§2). An important feature of our framework is that it is not confined to dealing with finite-state systems and finite workloads, and so applies to both finite and infinite runs of a system. In §2.1, we compare the capabilities of different resource configurations in executing a specified workload, with Propositions 2–5 providing some useful properties. The framework is extended to deal with *reconfiguration* (§2.2), following which we show the correctness of the scheme proposed by Pal *et al.* in [20] (Theorem 1). The weighted transition systems are extended to account for energy consumption in §3, using which we are able to formally state that the scheme of Pal *et al.* performs better than the baseline configuration (Theorem 2). The formulation allows us to examine an instance where there is a trade-off between efficiency in energy consumption versus satisfying timeliness constraints. We continue in §4 with a discussion on how one may formulate comparisons of performance with *optimal executions*, and propose a notion of simulation with performance within a constant factor c. We envisage this is the first

step towards relating operational formulations of correctness with the *competitive analysis* of approximation algorithmic schemes in the case of possibly infinite executions. §5 mentions a possible application in security that illustrates how the framework can address problems that go beyond meeting time deadlines. We then briefly discuss how the framework can be modified to deal with online scheduling of concurrently enabled threads during program execution on a reconfigurable machine. We conclude with a short statement on our future goals of developing further connections between operational notions such as simulation and approaches used in the analysis of relative and absolute performance guarantees of (online) algorithms.

1.1 Related Work

Timed automata are the preferred operational framework for specifying time-related properties of systems. In particular, the cost-optimal reachability problem has been studied both in a single-cost [3] and in multi-cost settings [18]. Bouyer *et al.* have studied issues relating to minimising energy in infinite runs within the framework of weighted (priced) timed automata [8]. Specifically, they have examined the construction of infinite schedules for *finite* weighted automata and one-clock weighted timed automata, subject to boundary constraints on the accumulated weight. However, we are unaware of an automata-based formulation of our general deadline-constrained execution problem, especially with respect to minimising cost (energy consumption), where the times/costs are cumulative and unbounded, i.e., where the state spaces and value domains (and possibly the alphabet) are *not finite*.

The seminal work in the use of process algebra for performance analysis is by Hermanns *et al.* [11]. Götz et al [10] have used stochastic process algebra in studying correctness and performance analysis of multiprocessors and distributed system design. Klin and Sassone [15, 16] have explored using monoidal structures for stochastic SOS, an elegant approach that unifies various different operational semantic models into a single algebraic frame. This approach has been taken further by Bernardo *et al.* [7] in finding a unifying structure for dealing with probabilistic, stochastic and time-dependent nondeterminism. The theory of weighted automata has been studied by Almagor, Kupferman and others [2]. Their weighted automata approach allows optimisation problems to be formulated as runs for finite words yielding values in a tropical semiring.

The dynamic reconfiguration scheme we study may be transformed to an instance of *dynamic speed scaling* in task scheduling [1], where tasks have strict deadlines and a scheduler has to construct feasible schedules while minimising energy consumption. Instead of using multiple cores, dynamic speed scaling allows the speed of the processor to be changed, assuming a model where power consumption increases exponentially with the speed of the processor ($P(s) = s^{\alpha}$). The polynomial-time YDS algorithm [22] finds optimal schedules in the offline case when all tasks and their requirements are known *a priori* ($\mathcal{O}(n^3)$ for a naive implementation, which can be improved to $\mathcal{O}(n^2 log n)$). The main idea is to find maximum density intervals, and schedule tasks occurring within them according to an earliest deadline first (EDF) policy. Tasks may be left unexecuted, and may be pre-empted. On the one hand, YDS deals with the more general problem of task scheduling, but on the other hand assumes a given relationship between power and speed, unlike our formulation, which leaves this relationship un(der)specified. Results about the competitive analysis of online versions of the algorithm (Average Rate and Optimal Available) have been given[1], assuming the exponential power-speed relationship. These bounds have been shown to be essentially tight [6]. Bansal *et al.* have also used the concept of slack and urgency in a variant

[1] An online algorithm ALG is called c-competitive if for every input task sequence, the objective function value of ALG is within c times the value of an optimal solution for that input.

of the problem, where deadlines may be missed but throughput maximisation is the objective function, presenting an online algorithm that is 4-competitive [5].

2 Getting the Job Done: An Operational Model

Preliminaries. We define a weighted transition system, workloads, deadlines and executing a workload respecting deadlines.

Definition 1. *A weighted transition system $\mathscr{T} = (Q, \mathscr{A}, \mathscr{W}, \rightarrow, Q_0, O)$ consists of a set of states Q; an input alphabet $\mathscr{A}$; an output domain $\mathscr{W}$; a cost-weighted transition relation $\_ \xrightarrow[\square]{\square} \_ : Q \times \mathscr{A} \times Q \times \mathscr{W}$; a set of initial states $Q_0 \subseteq Q$; and an observation function $O : Q \to \mathscr{W}$.*

A weighted transition system is a minor modification of an input-output Moore-style transition system. The major difference is that instead of an output set/alphabet we have a (monoidal) weight domain, and the transition relation, written $q \xrightarrow[w]{a} q'$, which maps a transition from q on a to q' to a weight $w \in \mathscr{W}$. This may be thought of the combination of a transition relation $\Delta \subseteq Q \times \mathscr{A} \times Q$ and a cost function $c : \Delta \to \mathscr{W}$. Further, we assume additional structure on the weight domain — (1) it is a partially ordered set $\langle \mathscr{W}, \leq_{\mathscr{W}} \rangle$ (2) it is also a *monoid* $\langle \mathscr{W}, \oplus, \mathrm{o} \rangle$, where o is the identity element for $\oplus$. The operation $\oplus$ is monotone and expansive w.r.t. $\leq_{\mathscr{W}}$, i.e., for all $x, y, z \in \mathscr{W}$, $x \leq_{\mathscr{W}} y$ implies $x \oplus z \leq_{\mathscr{W}} y \oplus z$, and $x \leq_{\mathscr{W}} x \oplus y$ and $y \leq_{\mathscr{W}} x \oplus y$. For a finite sequence $a_1 \ldots a_n$, we define $q_0 \xrightarrow[w]{a_1 \ldots a_n} q_n = w = \bigoplus_{i=1}^{n} w_i$ where $q_{i-1} \xrightarrow[w_i]{a_i} q_i$ $(i \in \{1, \ldots, n\})$. When $n = 0$, the weight $w = \mathrm{o}$, and otherwise $\bigoplus_{i=1}^{n} w_i = (\ldots(\mathrm{o} \oplus w_1) \ldots \oplus w_n)$ — the notation is unambiguous even if $\oplus$ is not commutative. A weighted transition system is *cumulative* if whenever $q \xrightarrow[w]{a} q'$ then $O(q') = O(q) \oplus w$ (and consequently, $O(q) \leq_{\mathscr{W}} O(q')$). It is sometimes useful to extend $\mathscr{W}$ to contain a maximum and annihilating element ω for $\oplus$, i.e., $x \oplus \omega = \omega = \omega \oplus x$ and $x \leq_{\mathscr{W}} \omega$ for all x. We write $q \xrightarrow[\_]{a} q'$ if $q \xrightarrow[w]{a} q'$ for some weight $w <_{\mathscr{W}} \omega$, and so can write $q \xrightarrow[\omega]{a} q'$ whenever $q \xrightarrow[\_]{a} q'$. For the motivating example, we will consider $\mathscr{W} = (\mathbb{R}^{\infty}, +, 0)$ (with $\omega = \infty$), which allows us to model time and deadlines.

We recast the notion of simulation for weighted transition systems. Note that our formulation uses the observation function O to compare weights.

Definition 2. *Suppose $\mathscr{T}_1 = (Q_1, \mathscr{A}, \mathscr{W}, \rightarrow, Q_{1o}, O_1)$ and $\mathscr{T}_2 = (Q_2, \mathscr{A}, \mathscr{W}, \rightarrow, Q_{2o}, O_2)$ are weighted transition systems on the same input alphabet $\mathscr{A}$ and weight domain $\mathscr{W}$. A simulation relation between $\mathscr{T}_1$ and $\mathscr{T}_2$ is a binary relation $R \subseteq Q_1 \times Q_2$ such that $(p, q) \in R$ implies (i) $O_2(q) \leq_{\mathscr{W}} O_1(p)$; and (ii) whenever $p \xrightarrow[\_]{a} p'$, there exists q' such that $q \xrightarrow[\_]{a} q'$ and $(p', q') \in R$.*

We say q simulates p if (p, q) is in *some* simulation. Transition system $\mathscr{T}_2$ simulates $\mathscr{T}_1$ if for *all* $p \in Q_{1o}$ there is a $q \in Q_{2o}$ such that q simulates p. That is, from q one can do everything that the other can from p, and with a lower weight.

Proposition 1. *Simulation relations include identity and are closed under composition and unions: (i) The identity relation $\{(p, p) \mid p \in Q\}$ is a (weighted) simulation; (ii) If R_1 and R_2 are weighted simulations, then so is $R_1 \circ R_2$. (iii) If R_i $(i \in I)$ are simulation relations, then so is $\bigcup_{i \in I} R_i$.*

The largest simulation relation is thus a quasi-order (reflexive and transitive).

Workloads with Deadlines. A workload is a (finite or infinite) sequence $\mathbf{a} = a_1 a_2 \ldots$, such that each $a_i \in \mathscr{A}$. Suppose with each a_i, we have a corresponding *budgeted time* $b_i \in \mathbb{R}$. Assume that the

actual time taken to perform each task a_i on a machine configuration $r \in Conf$ is given by $\tau(r, a_i) = t_i$. For simplicity, we assume τ is a function, though in practice the same computational task a_i may take differing amounts of time under different circumstances (e.g., ambient temperature, memory resources consumed by other tasks, etc.).

A naïve formulation of being able to satisfy this workload on configuration r is that $\forall i, 0 \leq i : t_i \leq b_i$, i.e., the actual time taken for each frame is less than the budgeted time. For frame-decoding, the budgeted time is the inverse of the desired frame rate. However, this is overly conservative, since it does not allow for the fact that one can begin processing the next frame early, thus amortising across frames using the slack earned by decoding a frame well within its budgeted time to offset overshoot incurred when taking longer than the budgeted time to decode another frame. Therefore, we consider a cumulative formulation, choosing to model a workload **a** together with a corresponding sequence of *absolute deadlines* $\mathbf{d} = d_1 d_2 \ldots$, where $d_i = \Sigma_{j \leq i}\, b_j$.

We can *specify* a workload $\mathbf{a} = a_1 \ldots$ with corresponding deadlines $\mathbf{d} = d_1 \ldots$ as a *deterministic* transition system **Spec** as $0 \xrightarrow[b_1]{a_1} d_1 \ldots d_{i-1} \xrightarrow[b_i]{a_i} d_i \ldots$, with $Q \subset \mathbb{R}$ and $O(d_i) = d_i$, where the b_i's are the budgeted times for each action.

The transition system $\mathscr{T}_r^{\mathbf{a}}$ for executing workload **a** on a machine configuration r can be modelled in terms of (r paired with) the cumulative time taken so far, i.e., $Q \subset Conf \times \mathbb{R}$, $O(\langle r, t\rangle) = t$, and $\langle r, t\rangle \xrightarrow[w]{a} \langle r, t'\rangle$ if $t' = \tau(r, a) + t$. The initial state is $\langle r, 0\rangle$. Note that for a given workload this also is a *deterministic* transition system, i.e., a *path*.

Definition 3. *We say that execution on a machine configuration r "by-simulates" a specified workload* **Spec** *(***a** *with corresponding deadlines* **d***) if there is a simulation relation between* **Spec** *and $\mathscr{T}_r^{\mathbf{a}}$ for this workload.*

That is, the execution sequence on machine r meets each deadline. The machine configuration r is then said to be *capable* of executing the specified workload with the expected quality of service; otherwise this configuration is incapable of doing so.

2.1 Good Enough: Comparing Configurations Based on Capability

Consider a workload specification **Spec** (action sequence **a** with corresponding deadlines **d**) and two computational resource configurations r and r'. We say that r is *at least as capable as* r' in performing **Spec**, written $r' \preceq_{\mathbf{Spec}} r$, if $\mathscr{T}_{r'}^{\mathbf{a}}$ can by-simulate **Spec** implies that so can $\mathscr{T}_r^{\mathbf{a}}$. We say that r and r' are *equi-capable* in performing **Spec**, written $r \sim_{\mathbf{Spec}} r'$ if $\mathscr{T}_{r'}^{\mathbf{a}}$ by-simulates **Spec** if and only if $\mathscr{T}_r^{\mathbf{a}}$ does. In other words, both resource configurations are capable of meeting the sequence of deadlines.

Proposition 2. *For every workload* **Spec**, *the relation $\preceq_{\mathbf{Spec}}$ is a preorder, and $\sim_{\mathbf{Spec}}$ an equivalence.*

Without any additional conditions, we cannot say much about the relationship between the capabilities of different computational resources on *different* workloads. Note that it is possible for $r \sim_{\mathbf{Spec}} r'$ for some workload **Spec** but $r \not\sim_{\mathbf{Spec}'} r'$ for some other workload **Spec**$'$. We say that r is *elementarily at least as capable as* r' if for each possible action $a : \tau(r, a) \leq \tau(r', a)$.

Proposition 3. *If r is elementarily at least as capable as r', then for any workload* **Spec**, *$r' \preceq_{\mathbf{Spec}} r$.*

This notion captures the intuition that the capability of a resource configuration is an inherent property (e.g., its speed) rather than peculiarly dependent on the action to be executed. The following proposition relate capability with sub-sequences of actions.

Capability and equi-capability are prefix-closed (Proposition 4) and the notions also suffix-compose (Proposition 5).

Proposition 4. *Let* **Spec** *be a workload. If* $r' \preceq_{\mathbf{Spec}} r$ *(respectively* $r \sim_{\mathbf{Spec}} r'$*) then for each prefix* $\mathbf{Spec}'$ *of* **Spec**, $r' \preceq_{\mathbf{Spec}'} r$ *(respectively* $r \sim_{\mathbf{Spec}'} r'$*).*

Proposition 5. *Let* **Spec** *be a finite workload of actions* $a_1,\ldots,a_m$ *with deadlines* $d_1,\ldots,d_m$ *and* $\mathbf{Spec}'$ *be another (finite or infinite) workload of actions* $a'_1,\ldots a'_j,\ldots$*, with deadlines* $d'_1,\ldots,d'_j\ldots$*. Consider the sequenced workload* $\mathbf{Spec}'' = a_1,\ldots,a_m,a'_1,\ldots,a'_j,\ldots$*, with deadlines* $d_1,\ldots,d_m,d'_1+d_m,\ldots,d'_j+d_m,\ldots$*. Then, if* $r' \preceq_{\mathbf{Spec}} r$ *and* $r' \preceq_{\mathbf{Spec}'} r$ *(respectively* $r \sim_{\mathbf{Spec}} r'$ *and* $r \sim_{\mathbf{Spec}'} r'$*), then* $r' \preceq_{\mathbf{Spec}''} r$ *(respectively* $r \sim_{\mathbf{Spec}''} r'$*).*

In particular, if $r' \preceq_{\mathbf{Spec}} r$ (respectively $r \sim_{\mathbf{Spec}} r'$), then for any workload $\mathbf{Spec}'$ of which **Spec** is a prefix, $r' \preceq_{\mathbf{Spec}'} r$ (respectively $r \sim_{\mathbf{Spec}'} r'$).

Note however that if $\mathbf{Spec}'' = a_1,\ldots,a_m,a'_1,\ldots,a'_n$, with deadlines $d_1,\ldots,d_m,d'_1+d_m,\ldots,d'_n+d_m$, and $r' \preceq_{\mathbf{Spec}''} r$ (respectively $r \sim_{\mathbf{Spec}''} r'$), while by Proposition 4, for $\mathbf{Spec} = a_1,\ldots,a_m$, with deadlines $d_1,\ldots,d_m$ we necessarily have $r' \preceq_{\mathbf{Spec}} r$ (respectively $r \sim_{\mathbf{Spec}} r'$), it may *not* be the case that for $\mathbf{Spec}' = a'_1,\ldots,a'_n$, with deadlines $d'_1,\ldots,d'_n$, that we will have $r' \preceq_{\mathbf{Spec}'} r$ (respectively $r \sim_{\mathbf{Spec}'} r'$), because the resource configurations are capable of performing the latter part of the workload before the specified deadlines *only* because of "credit" earned by completing the prefix **Spec** *sufficiently early*.

Resource Lattice. In the video decoding applications, we assume that we have the *elementarity* property, based on the assumptions made by Pal *et al*: (0) Decoding a frame can be cleanly decomposed into decoding of independent slices/macroblocks, assigned to different cores. (1) The decoding time for a frame is monotone non-increasing in the number of cores; (2) In heterogeneous architectures, decoding time for a frame does not increase when moving from a small core to a big core. Therefore, we can assume a lattice structure with ordering $\preceq$ applicable to *any* workload **Spec**, with a *maximal* resource configuration r_{max} being the one where all cores of all kinds are given work, and a minimal configuration r_{min} which is one in which all cores are switched off (of course, not much happens on that minimal configuration).

In the sequel, we will only consider workloads **Spec** which *can* be successfully executed (meeting all deadlines) on the maximal resource configuration r_{max}. This will be considered the baseline configuration.

2.2 Reconfiguration

Till now we have considered only deterministic transition systems (paths) that arise for a given workload on a given configuration, and have compared different configurations on their ability to handle a given workload. We now consider *reconfigurable machines*. Let $\delta_{r,r'}$ denote the cost of changing configuration from r to r', with $\delta_{r,r}$ being 0. For simplicity, we assume any change of configuration to have a constant cost δ. We can now define *reconfigurable* execution to be the *non-deterministic* transition system $\mathscr{N}$, obtained by modifying the earlier weighted transitions as follows: $\langle r,t\rangle \xrightarrow[w]{a} \langle r',t'\rangle$ if $t' = t+\delta_{r,r'}+\tau(r',a)$, denoting the cost of changing configuration to r' and then executing a. $w = \delta_{r,r'}+\tau(r',a)$. The start state is $\langle r_{max},0\rangle$. The branching structure captures the various possibilities in choosing to reconfigure the machine at any stage in the execution.

A reconfiguration scheme (algorithm/heuristic) defines a sub-transition system (a pruning) $\mathscr{T}$ of $\mathscr{N}$. In general, this may be a non-deterministic transition system, embodying the possibility of reconfiguration according to the scheme, which is why we use simulation relations to consider and compare every execution path with the specification. $\mathscr{T}$ by-simulates a workload **Spec** if *every* path of $\mathscr{T}$ (by-)simulates

Spec. That is, every possible reconfiguration path in $\mathscr{T}$ meets all deadlines when executing the specified actions.

The scheme proposed by Pal et al., [20], permits reconfiguration from r to a weaker configuration r' only when sufficient slack has been earned to permit a slower execution of the next action plus time for reconfiguration (before and possibly after), i.e., $\langle r, t_{i-1}\rangle \xrightarrow[w_i]{a_i} \langle r', t_i\rangle$ if $d_i - t_{i-1} \geq 2 * \delta + \tau(r', a_i)$, where $t_i = t_{i-1} + \delta + \tau(r', a_i)$ (i.e., $w_i = \delta_{r,r'} + \tau(r', a_i)$). That is, the sum of the slack earned so far and the budgeted time for a_i should exceed the time for reconfiguring and executing on a slower configuration, with a further allowance for a possible reconfiguration to a faster configuration to avoid missing future deadlines. Otherwise, a faster configuration (r_{max}, to be safe) is chosen. It thus defines a non-deterministic transition system $\mathscr{P}$ which is a subtransition system of the transition system $\mathscr{N}$ mentioned above. Theorem 1 states the correctness of this scheme (and so of any deterministic algorithm based on it).

Theorem 1. *If r_{max} can execute each action a_i of a workload within its corresponding budgeted time b_i, then the scheme of Pal et al. defines a transition system $\mathscr{P}$ that by-simulates* **Spec**.

Note that we have been able to state a general proof of the correctness of the scheme in the abstract, without positing any model relating configurations to speeds, and without any bounds on the times for any task in $\mathscr{A}$. Note also that the scheme does not consider idling between actions, since that would be counter-productive to meeting deadlines.

3 Better: Comparing Resources Based on Energy Efficiency

The motivation for dynamic reconfiguration is to save energy, since weaker configurations consume less energy, providing an opportunity to trade off time versus energy cost. We focus on amortising total energy consumption, subject to the constraint of meeting all deadlines (other objectives can also be formulated). Accordingly, we modify the transition system to have weights that also consider cumulative energy costs. We assume that energy costs for an action are given by a function $\gamma(r,a)$, again making the simplifying assumption that energy costs are determined only by the configuration r and the action a. Let the energy cost of reconfiguration from r to r' be denoted $\theta_{r,r'}$ which for simplicity we assume to be 0 when $r = r'$ and a constant θ otherwise.

The reconfigurable energy-aware transition system $\mathscr{E}$ for executing workloads can be modelled with $Q = Conf \times \mathbb{R} \times \mathbb{R}$; $O(\langle r,t,e\rangle) = e$; and $\langle r,t,e\rangle \xrightarrow[w]{a} \langle r',t',e'\rangle$ if $t' = t + \delta_{r,r'} + \tau(r',a)$ and $e' = e + \theta_{r,r'} + \gamma(r',a)$. The initial state is $\langle r,0,0\rangle$. In the general setting, the weight domain can be seen as a composite monoid.

Consider a workload **Spec** and two paths of π, π' of $\mathscr{E}$ that *both* by-simulate **Spec**. We say that π is *more efficient* than π' if π simulates π'. That is, π does whatever actions π' can (within the deadlines), but at lower cumulative energy cost at each step.

The notion can be extended to transition systems $\mathscr{P}$ and $\mathscr{P}'$ that both by-simulate **Spec**. $\mathscr{P}$ is *more efficient* than $\mathscr{P}'$ in executing **Spec** if for every execution path π' of $\mathscr{P}'$, there exists a path π of $\mathscr{P}$ such that π is *more efficient* than π'. This is a simulation relation between the transition systems.

Note that a simulation relation allows $\mathscr{P}$ to contain paths that are *not* more efficient than any path in $\mathscr{P}'$. We therefore modify the notion of simulation to yield that of a *betterment*:

Definition 4. *Suppose $\mathscr{T}_1 = (Q_1, \mathscr{A}, \mathscr{W}, \rightarrow, Q_{1o}, O_1)$ and $\mathscr{T}_2 = (Q_2, \mathscr{A}, \mathscr{W}, \rightarrow, Q_{2o}, O_2)$ are cumulative weighted transition systems on the same input alphabet $\mathscr{A}$ and weight domain $\mathscr{W}$. A betterment relation between $\mathscr{T}_1$ and $\mathscr{T}_2$ is a binary relation $R \subseteq Q_1 \times Q_2$ such that $(p,q) \in R$ implies (i) $O_2(q) \leq_{\mathscr{W}} O_1(p)$;*

and (ii) whenever $p \xrightarrow[-]{a} p'$ *then there exists at least one* q' *such that* $q \xrightarrow[-]{a} q'$*, and for* every q' *such that* $q \xrightarrow[-]{a} q'$*,* $(p',q') \in R$*.*

We say q *betters* p if (p,q) is in *some* betterment relation. Transition system $\mathscr{T}_2$ betters $\mathscr{T}_1$ if for all $p \in Q_{1o}$, and *every* $q \in Q_{2o}$, (p,q) is in a betterment relation. That is, every path in $\mathscr{T}_2$ is at least as efficient as any path in $\mathscr{T}_1$. In other words, $\mathscr{T}_2$ is in "every way better" than $\mathscr{T}_1$. Note that if $\mathscr{T}_2$ is deterministic, a betterment reduces to a simulation.

The identity relation on transition systems may not be a betterment. However, betterments are closed under composition and union.

Proposition 6. . *(i) If* R_1, R_2 *are betterments, then so is* $R_1 \circ R_2$*. (ii) If* $R_i \ \ (i \in I)$ *are betterment relations between two given transition systems, then so is* $\bigcup_{i \in I} R_i$*.*

The scheme in [20] additionally examines the energy savings when opportunistically deciding to reconfigure, i.e., $\langle r, t_{i-1}, e_{i-1} \rangle \xrightarrow[e]{a_i} \langle r', t_i, e_i \rangle$ if (i) $d_i - t_{i-1} \geq 2 * \delta + \tau(r', a_i)$; (ii) $\gamma(r, a_i) \geq \gamma(r', a_i) + 2\theta$ where $t_i = t_{i-1} + \delta + \tau(r', a_i)$, and $e_i = e_{i-1} + e$, where $e = \theta_{r,r'} + \gamma(r', a_i)$.

Theorem 2. *If baseline configuration* r_{max} *can execute each action* a_i *of a workload* **Spec** *within its corresponding budgeted time* b_i*, then any execution under the Pal et al. energy-saving scheme [20] is a better (more efficient) by-simulation than execution on the baseline configuration* r_{max}*.*

4 What's Best?

The scheme in [20] is *not optimal* for arbitrary workloads. For *finite* workloads it is possible to determine optimal executions using offline techniques such as the YDS algorithm [22], or *dynamic programming* techniques for related problems. However, it may not be pragmatic to use such offline algorithmic techniques because of the size of the workload and the available memory and computational resources. Hence the problem is posed in a manner resembling an *online algorithm* with an *estimate* of the maximum time and energy required for executing the next action. However, one would like to ask how far from the optimal (either in absolute or relative terms) the approximation given by any given scheme is. We propose that simulation relations on cumulative weighted transition systems can provide a framework for reasoning about relative performance guarantees of approximations. We extend the weight domain to being a semiring, $\langle \mathscr{W}, \oplus, 0, \odot, 1 \rangle$, where 0 is the identity element for $\oplus$, and 1 is the identity element for $\odot$.

Definition 5. *Suppose* $\mathscr{T}_1 = (Q_1, \mathscr{A}, \mathscr{W}, \rightarrow, Q_{1o}, O_1)$ *and* $\mathscr{T}_2 = (Q_2, \mathscr{A}, \mathscr{W}, \rightarrow, Q_{2o}, O_2)$ *are weighted transition systems on the same input alphabet* $\mathscr{A}$ *and weight domain* $\mathscr{W}$*. Let* c *be any constant in* $\mathscr{W}$*. A constant-factor* c*-simulation relation between* $\mathscr{T}_1$ *and* $\mathscr{T}_2$ *is a binary relation* $R_c \subseteq Q_1 \times Q_2$ *such that* $(p,q) \in R_c$ *implies (i)* $O_2(q) \leq_{\mathscr{W}} c \odot O_1(p)$*; and (ii) whenever* $p \xrightarrow[-]{a} p'$*, there exists* q' *such that* $q \xrightarrow[-]{a} q'$ *and* $(p',q') \in R_c$*.*

Constant factor simulations include the identity relation and are closed under relational composition (which corresponds to $\odot$ on the indexing constants). Moreover they are monotone increasing with respect to the indexing constant. For any c, c-simulations are closed under union.

Proposition 7. *(i) The identity relation* $\{(p,p) \mid p \in Q\}$ *is a* 1*-factor simulation; (ii) If* $(p,q) \in R_c$ *and* $(q,s) \in R_{c'}$*, then* $(p,s) \in R_{c \odot c'}$ *(iii) If* $c \leq c'$ *then if* q *can simulate* p *up to constant factor* c*, then so can it up to constant factor* c'*. (iv) If* $R_i \ \ (i \in I)$ *are all* c*-simulations, then* $\bigcup_{i \in I} R_i$ *is also a* c*-simulation.*

An algorithm A_2 has a competitive ratio of c with respect to another A_1 if there is a c-simulation between the (deterministic) transition systems defined by them on any given input sequence of actions. That competitive ratio c between two algorithms is *tight* if there is no c'-simulation between them for any $c' < c$. Note that if α is the ratio of the speeds between the fastest and slowest configurations, then the scheme of Pal *et al.* will be α-competitive. This is however a weak bound.

5 Conclusions

Inspired by practical problems encountered in multicore architectures, we have presented an operational formulation of a general problem that involves finding feasible executions of a series of actions each to be completed within hard budgetary constraints (deadlines), and then comparing the cost of the feasible executions. There are several trade-offs that can be explored once the problem is amenable to an operational framework. While finite instances of such problems may be optimally solved "offline", using techniques such as dynamic programming or automata-based programming techniques, we pose the general problem in an online form, allowing for infinite executions, and unbounded state and data spaces. Such a formulation allows us to extend well-studied notions in concurrency theory such as simulation relations to the class of weighted transition systems, and thence to a general notion of algorithmic correctness and efficiency. The quantitative and timing aspects of the problem have motivated the use of interesting algebraic structures such as cumulative monoids. Typically semirings (e.g., a *min*-+ algebra, also called a tropical semiring) are employed for formulating and comparing the behaviour of systems, especially in optimisation problems.

Other applications. To illustrate that our formulation is not merely about meeting deadlines and that it is not confined to video decoding, let us consider another application involving multicore machines, this time related to security. Consider the problem of executing a series of actions each to be completed within a prescribed energy budget. Such problems are increasingly important in energy-oriented compiler optimisations. It is by now well established that an attacker can gain side-channel information about a computation by observing the power consumption characteristics of a machine performing a computation [17]. Such attacks exploit information leakage from mobile devices (smartphones, wireless payment devices etc.) that are widely used today. Therefore we have the additional objective of minimising information leakage through this "side channel". A common approach to thwarting the attacker's capability involves generating noise to obfuscate the power-consumption profile of the actions (instruction/job/task). The noise generator can be run on another core in parallel with the main computation, but this is at the cost of extra power consumption. Amortising energy consumption across the actions, we can minimise the leakage of power-profile based information from a subsequence of actions (using any energy credit earned when performing earlier tasks well within their budgeted energy).

Concurrent actions. Our formulation involved resource allocation rather than task scheduling, since the problem was presented as a *sequence of (atomic) actions* to be executed — only one task is enabled at a time. However, our problem finds an obvious generalisation that involves scheduling as well, when we are presented with a *sequence of sets of actions* where each set of actions must be concurrently executed. At each step the set of concurrently-enabled actions are to be executed within their given deadlines. If the deadlines can be met by an interleaving of the atomic actions, then one can allocate a minimal set of required cores, thus minimising energy consumption while meeting all deadlines. A scheduler tries to find such an interleaving for the set of concurrently enabled actions. In case two or more concurrent

actions must be mutually exclusively executed, they are suitably interleaved in a feasible schedule (if one exists). Similar conditions apply if one task has to be executed in preference to another. Otherwise, if the set of concurrent actions cannot be interleaved, the scheduler tries to allocate disjoint sets of cores for the parallel execution of the actions, in a manner that minimises energy consumption while still meeting the deadlines of each task. In these cases, we may additionally need to consider the costs of allocating cores and assigning tasks, as well as idling costs when tasks wait at a synchronisation point. Note that the scheduler needs to work online, in that the particulars of sets of actions that will materialise in the future are not known to it.

Future work. To our knowledge, the connections between algorithmic efficiency and performance guarantees on the one hand, and operational formulations such as simulations and bisimulations on the other have not been adequately explored. We recently became aware of a particular subclass of problems for which this connection has been well formulated, namely the connection proposed by Aminof *et al.* between weighted *finite-state automata* and online algorithms [4]. Their main insight is to relate the "unbounded look ahead" of optimal offline algorithms with nondeterminism, and the "no look ahead" of online algorithms with determinism. Our proposed relationship can be seen as an extension from finite state automata to general transition systems, replacing language equality with relations such as simulations (and bisimulations and prebisimulations).

We are currently looking at formulating and analysing online algorithms that may have better competitive ratios for the general energy minimisation problem, using, e.g., branch and bound techniques, etc., with the intention of proving tighter bounds. We are considering the cases where there is a limit on how far ahead one can execute actions (because of, say, a bounded buffer for decoded frames) and when the online algorithm can look ahead at the characteristics of the next k frames when deciding what configuration to choose. In the future, we would also like to examine the connections between *absolute* performance guarantees and the framework of amortised bisimulations [14].

References

[1] Susanne Albers (2011): *Algorithms for Dynamic Speed Scaling*. In: *28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011), Leibniz International Proceedings in Informatics (LIPIcs)* 9, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 1–11, doi:10.4230/LIPIcs.STACS.2011.1.

[2] Shaull Almagor, Udi Boker & Orna Kupferman (2011): *What's Decidable about Weighted Automata?*, pp. 482–491. Springer, Berlin, Heidelberg, doi:10.1007/978-3-642-24372-1_37.

[3] Rajeev Alur, Salvatore La Torre & George J. Pappas (2001): *Optimal Paths in Weighted Timed Automata*. In: *Hybrid Systems: Computation and Control, 4th International Workshop, HSCC 2001, Rome, Italy, March 28-30, 2001, Proceedings*, pp. 49–62, doi:10.1007/3-540-45351-2_8.

[4] Benjamin Aminof, Orna Kupferman & Robby Lampert (2010): *Reasoning About Online Algorithms with Weighted Automata*. *ACM Trans. Algorithms* 6(2), pp. 28:1–28:36, doi:10.1145/1721837.1721844.

[5] Nikhil Bansal, Ho-Leung Chan, Tak-Wah Lam & Lap-Kei Lee (2008): *Scheduling for Speed Bounded Processors*, pp. 409–420. Springer, Berlin, Heidelberg, doi:10.1007/978-3-540-70575-8_34.

[6] Nikhil Bansal, Tracy Kimbrel & Kirk Pruhs (2007): *Speed scaling to manage energy and temperature*. *J. ACM* 54(1), pp. 3:1–3:39, doi:10.1145/1206035.1206038.

[7] Marco Bernardo, Rocco De Nicola & Michele Loreti (2013): *A Uniform Framework for Modeling Nondeterministic, Probabilistic, Stochastic, or Mixed Processes and Their Behavioral Equivalences*. *Inf. Comput.* 225, pp. 29–82, doi:10.1016/j.ic.2013.02.004.

[8] Bouyer, Patricia and Fahrenberg, Uli and Larsen, Kim G. and Markey, Nicolas and Srba, Jiří (2008): *Infinite Runs in Weighted Timed Automata with Energy Constraints*, pp. 33–47. Springer, Berlin, Heidelberg, doi:10.1007/978-3-540-85778-5_4.

[9] Trevor E. Carlson, Wim Heirman & Lieven Eeckhout (2011): *Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations*. In: *International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, pp. 52:1–52:12, doi:10.1145/2063384.2063454.

[10] Norbert Götz, Ulrich Herzog & Michael Rettelbach (1993): *Multiprocessor and Distributed System Design: The Integration of Functional Specification and Performance Analysis Using Stochastic Process Algebras*. In: *Performance Evaluation of Computer and Communication Systems, Joint Tutorial Papers of Performance '93 and Sigmetrics '93*, Springer-Verlag, London, UK, UK, pp. 121–146, doi:10.1007/BFb0013851.

[11] Holger Hermanns, Ulrich Herzog & Joost-Pieter Katoen (2002): *Process algebra for performance evaluation*. *Theoretical Computer Science* 274(1-2), pp. 43–87, doi:10.1016/S0304-3975(00)00305-4.

[12] ISO/IEC 14496-10: *Advanced Video Coding for Generic Audiovisual Services*. In: *http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=11466*.

[13] David Kanter (2013): *Silvermont: Intel's Low Power Architecture*. In: *http://www.realworldtech.com/silvermont*.

[14] Astrid Kiehn & S. Arun-Kumar (2005): *Amortised Bisimulations*. In: *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, pp. 320–334, doi:10.1007/11562436_24.

[15] Bartek Klin (2009): *Structural Operational Semantics for Weighted Transition Systems*, pp. 121–139. Springer, Berlin, Heidelberg, doi:10.1007/978-3-642-04164-8_7.

[16] Bartek Klin & Vladimiro Sassone (2013): *Structural operational semantics for stochastic and weighted transition systems*. *Inf. Comput.* 227, pp. 58–83, doi:10.1016/j.ic.2013.04.001.

[17] Paul C. Kocher, Joshua Jaffe & Benjamin Jun (1999): *Differential Power Analysis*. In: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, Springer-Verlag, pp. 388–397, doi:10.1007/3-540-48405-1_25.

[18] Kim Guldstrand Larsen & Jacob Illum Rasmussen (2008): *Optimal reachability for multi-priced timed automata*. *Theoretical Computer Science* 390(2-3), pp. 197–213, doi:10.1016/j.tcs.2007.09.021.

[19] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen & Norman P. Jouppi (2009): *McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures*. In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, pp. 469–480, doi:10.1145/1669112.1669172.

[20] Rajesh Kumar Pal, Ierum Shanaya, Kolin Paul & Sanjiva Prasad (2016): *Dynamic core allocation for energy efficient video decoding in homogeneous and heterogeneous multicore architectures*. *Future Generation Comp. Syst.* 56, pp. 247–261, doi:10.1016/j.future.2015.09.018.

[21] Michael Roitzsch (2007): *Slice-Balancing H.264 Video Encoding for Improved Scalability of Multicore Decoding*. In: *Proceedings of the 7th ACM/IEEE International Conference on Embedded Software*, pp. 269–278, doi:10.1145/1289927.1289969.

[22] F. Yao, A. Demers & S. Shenker (1995): *A Scheduling Model for Reduced CPU Energy*. In: *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, IEEE Computer Society, Washington, DC, USA, pp. 374–, doi:10.1109/SFCS.1995.492493.