AGERE! at SPLASH 2015

5th International Workshop on Programming based on Actors, Agents, and Decentralized Control

Workshop held at ACM SPLASH 2015
26 October 2015
Pittsburgh, PA, USA

Companion Proceedings

AGERE! is an ACM SIGPLAN workshop

Introduction

ago, agis, egi, actum, **agere**
latin verb meaning to act, to lead, to do,
common root for actors and agents

The fundamental turn of software into concurrency and distribution is not only a matter of performance, but also of design and abstraction. It calls for programming paradigms that, compared to current mainstream ones, would allow us to think about, design, develop, execute, debug, and profile – more naturally – systems exhibiting different degrees of concurrency, asynchrony, and physical distribution. To this purpose, the AGERE! workshop is aimed at focusing on programming systems, languages and applications based on actors, agents and other programming paradigms promoting a decentralized-control mindset in solving problems and developing software. The workshop is designed to cover both the theory and the practice of design and programming, bringing together researchers working on the models, languages and technologies, and practitioners developing real-world systems and applications.

This volume contains the work-in-progress and short papers accepted and presented at the 5th edition of the AGERE! workshop, not included in the formal proceedings available on the ACM Digital Library.

Acknowledgment

The organizing committee would like to thank all program committee members, authors and participants. Thank you to ACM and SPLASH organizers for their support. We look forward to a productive workshop.

Committee

Program Committee

Gul Agha, University of Illinois at Urbana-Champaign
Sylvan Clebsch, Imperial College London
Rem Collier, University College Dublin
Travis Desell, University of North Dakota
Amal El Fallah Seghrouchni, LIP6 Univ. P and M. Curie, Paris, France
Ludovic Henrio, INRIA
Jomi Hübner, Federal University of Santa Catarina
Shams Imam, Rice University
Stefan Marr, INRIA, France
Francisco Sant'Anna, PUC-Rio
Tom Van Cutsem, Alcatel-Lucent Bell Labs
Wei-Jen Wang, National Central University
Takuo Watanabe, Tokyo Institute of Technology
Nabuko Yoshida, Imperial College London
Damien Zuffereu, MIT

Organizing Committee & PC Chairs

Eliza Gonzalez Boix, Vrije Universiteit Brussel, Belgium
Philipp Haller, KTH Royal Institute of Technology, Sweden
Alessandro Ricci, University of Bologna, Italy
Carlos Varela, Rensselaer Polytechnic Institute, NY, USA

Steering Committee

Gul Agha, University of Illinois at Urbana-Champaign, USA
Rafael H. Bordini, FACIN–PUCRS, Brazil
Assaf Marron, Weizmann Institute of Science, Israel
Alessandro Ricci, University of Bologna, Italy

List of the papers included in this volume

Actario: A Framework for Reasoning About Actor Systems.
Shohei Yasutake and Takuo Watanabe – Tokyo Institute of Technology, Japan

Programming Abstractions for Augmented Worlds.
Angelo Croatti and Alessandro Ricci – University of Bologna, Italy

A Model-based Approach to Secure Multi-party Distributed Systems.
Najah Ben Said, Saddek Bensalem, Marius Bozga – Univ. Grenoble Alpes, VERIMAG, CNRS, France
Takoua Abdellatif – University of Carthage, Tunis

Bulk-Synchronous Communication Mechanisms in Diderot
John Reppy and Lamont Samuels – University of Chicago, US

A Performance and Scalability Analysis of Actor Message Passing and Migration in SALSA Lite
Travis Desell – University of North Dakota, US
Carlos A. Varela – Rensselaer Polytechnic Institute, US

Optimizing Communicating Event-Loop Languages with Truffle
Stefan Marr and Hanspeter Mössenböck – Johannes Kepler University Linz, Austria

Connect.js – A cross mobile platform actor library for multi-networked mobile applications
Elisa Gonzalez Boix Christophe Scholliers Nicolas Larrea Wolfgang De Meuter – Vrije Universiteit Brussel, Belgium

Towards Verified Privacy Policy Compliance of an Actor-based Electronic Medical Record Systems
Tom MacGahan, Claiborne Johnson, Armando L. Rodriguez, Mark Apple, Jianwei Niu, Jeffery von Ronne – The University of Texas at San Antonio, US

Actario: A Framework for Reasoning About Actor Systems

Shohei Yasutake
Tokyo Institute of Technology
yasutake@psg.cs.titech.ac.jp

Takuo Watanabe
Tokyo Institute of Technology
takuo@acm.org

Abstract

The two main characteristics of the Actor model are asynchronous message passing and dynamic system topology. The latter relies on the on-the-fly creation of actor names that often complicates the formal treatment of systems described in the Actor model. In this paper, we introduce Actario, a formalization of the Actor model in Coq. Actario incorporates a name creation mechanism that is formally proved to generate a consistent set of actor names. The mechanism helps proper handling of names in modeling and reasoning about actor-based systems. Actario also provides a code extraction mechanism that generates Erlang programs.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

General Terms Actors, Formal Models

Keywords Actor Model, Formalization, Actario, Coq, Erlang

1. Introduction

The Actor model[3] is a kind of concurrent computation model, in which a system is expressed as a collection of autonomous computing entities called actors that communicate each other only with asynchronous messages. On receiving a message, an actor may (1) send messages to other actors (or itself) whose names are known to the sender, (2) create new actors and (3) change its behavior for the next message.

Starting from the 1970s, the Actor model and its variations such as concurrent objects[15] have a long research history. They are today regarded as popular high-level abstractions for concurrent and parallel programming used in some industrial strength language and libraries such as Erlang[7], Scala[10] and Akka[4]. Because of this situation, establishing a mechanized formal verification method for actor-based systems is a pressing issue.

Several methods and systems for formally verifying actor-based systems have been presented recently. Rebeca[11] is a modeling language that allows model-checking. For deductive verification using proof assistants, formalizations using Athena[9] and Coq[8] have been presented.

A *name*[1] in the Actor model is a unique conceptual location associated with each actor. The concept of *name uniqueness* denotes that each name uniquely refers an actor and each actor should be referred by a single name. In the implementations of actor systems including Erlang, Scala, and Akka, naming of actors is implicit; we don't need to manually assign a fresh name to a newly created actor. The name uniquness may be broken if the naming is explicit in complex systems. Implicit naming, however, might complicate the formal treatment of actor-based systems. Thus, some formalization adopts explicit naming.

In this paper, we propose Actario[1], a Coq framework for implementing and verifying actor-based systems. The framework (1) supports Erlang-like notation for describing an actor system, (2) allows verifying desired properties of the system using the proof mechanism in Coq, and (3) generates executable Erlang code from the system description.

To be close to realistic actor languages and libraries, we designed Actario to support implicit naming. This is the main difference between our formalization and formalizations using Athena[9] or Coq[8]. The naming mechanism behind the scene is formally proved to satisfy the name uniqueness. We also proved other properties including the persistence of actors and messages. The proof scripts of these properties are available in the GitHub repository of Actario[1].

The layout of the rest of this paper is as follows. The next section describes the overview of Actario. In Section 3, we give the operational semantics of the Actor model formalized in Actario. Section 4 outlines the proof of the uniqueness property on dynamically generated names. In Section 5, we discuss fairness properties formalized in Actario. The code extraction mechanism is described in Section 6. Finally,

AGERE!@SPLASH, Oct., 2015, Pittsburgh, PA, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

[1] The term *mail address* is used in some other literature.

Section 7 overviews related work and Section 8 concludes the paper.

2. Overview of Actario

2.1 Programming in Actario

Actario is a Coq framework for defining and verifying actor-based systems. A typical workflow using Actario is as follows.

1. Describe an actor system using types and notations defined in the framework.
2. Specify and verify desired properties of the system.
3. Extract the Erlang version of the system using the code extraction mechanism of Coq.

Note that Actario does not provide a dedicated language for describing actor systems. The framework offers a set of Coq vocabularies (types and notations described in Section 2.2) for that purpose.

Example: Recursive Factorial System We use a simple example to illustrate a system description in Actario. Figure 1 shows the definition of an actor system that implements the continuation-passing style factorial function adapted from [3]. In this definition, the function `factorial_system` sets up a system that initially consists of a single factorial actor whose behavior is defined as `factorial_behv`. The actor can receive a tuple of a natural number and the name of a *customer* actor (`cust`) that is intended to receive the result. If the first component of tuple is more than zero, *i.e.*, it matches the successor pattern `S n`, the actor creates a new continuation actor (`cont`) and recursively sends itself a pair of `n` and `cont`. The behavior of continuation actors is specified as `factorial_cont_behv`.

```
1  Definition factorial_cont_behv (val : nat)
2                                 (cust : name) :=
3    receive (fun msg ⇒
4      match msg with
5       | nat_msg arg ⇒
6         cust ! nat_msg (val * arg);
7         become empty_behv
8       | _ ⇒ become empty_behv
9      end).
10
11 CoFixpoint factorial_behv :=
12   receive (fun msg ⇒
13     match msg with
14      | tuple_msg (nat_msg 0) (name_msg cust) ⇒
15        cust ! nat_msg 1;
16        become factorial_behv
17      | tuple_msg (nat_msg (S n))
18                  (name_msg cust) ⇒
19        cont ← new
20                (factorial_cont_behv (S n) cust);
21        me ← self;
22        me ! tuple_msg (nat_msg n)
23                       (name_msg cont);
24        become factorial_behv
25      | _ ⇒ become factorial_behv
26     end).
27
28 Definition factorial_system (n : nat)
29                               (cust : name) :=
30   init "factorial" (
31     x ← new factorial_behv;
32     x ! tuple_msg (nat_msg n) (name_msg cust);
33     become empty_behv
34   ).
```

Figure 1. Recursive Factorial System in Actario

2.2 Types and Notations

2.2.1 Types

Figure 2 shows the inductively defined type of messages delivered among actors, each of whose constructors corresponds to a kind of messages. In the current version of Actario, a message may be empty, an actor name, a value of basic types (Boolean, natural number or string), or a tuple of two messages.

Figure 3 defines two mutually coinductive types: `actions` and `behavior`. They specify sequences of actions performed by actors and behaviors of actors respectively. Each constructor of `actions` corresponds to a single action embedded in an action sequence as follows.

`new` b f creates a new actor with initial behavior b and applies f to the name of the created actor. Then continues to the action sequence that f returns.

`send` n m α sends message m to the actor with name n and then continues to action sequence α.

`self` f retrieves the name of the actor that executes this action and applies f to it. Then continues to the action sequence that f returns.

`become` b sets b as the next behavior of the actor that executes this action. This action should end an action sequence.

In the Actor model, an actor persists indefinitely. Thus, as shown in Figure 1, that a behavior may have `become` actions that specify itself or other behaviors eventually recurring to the original one. The reason for using `CoFixpoint` and defining `actions` and `behavior` coinductively is to model such behaviors.

2.2.2 Notations

In addition to the types defined above, Actario provides a collection of notations (syntactic sugaring) described in Figure 4. Using the notations, we can write actor behaviors intuitively without being complicated by CPS. Figure 5 com-

```
Inductive message : Set :=
 | empty_msg : message
 | name_msg : name → message
 | str_msg : string → message
 | nat_msg : nat → message
 | bool_msg : bool → message
 | tuple_msg : message → message →
    message.
```

Figure 2. Message Type

```
CoInductive actions : Type :=
 | new : behavior → (name → actions) →
    actions
 | send : name → message → actions →
    actions
 | self : (name → actions) → actions
 | become : behavior → actions
with behavior : Type :=
 | receive : (message → actions) →
    behavior.
```

Figure 3. Types for Actions and Behaviors

```
Notation "n '←' 'new' behv ; cont" :=
  (new behv (fun n ⇒ cont))
  (at level 0, cont at level 10).
Notation "n '!' m ';' a" :=
  (send n m a) (at level 0, a at level 10).
Notation "me '←' 'self' ';' cont" :=
  (self (fun me ⇒ cont))
  (at level 0, cont at level 10).
```

Figure 4. Notations for Actions

```
new b (fun x ⇒
self (fun s ⇒
send x (name_msg s)
(become b')))
```

(a) without notations

```
x ← new b;
s ← self;
x ! (name_msg s);
become b'
```

(b) with notations

Figure 5. Example Use of Notations

pares the descriptions of a simple action sequence without/with the notations.

```
Inductive name : Set :=
 | toplevel : string → name
 | generated : nat → name → name.
```

Figure 6. name

```
Record actor := {
  actor_name : name;
  remaining_actions : actions;
  next_num : gen_number
}.
```

Figure 7. actor

3. Semantics

In this section, we explain the formalization of the operational semantics of the Actor model in Actario. First, for the explanation of formalization of operational semantics, we describe `name` type, `actor` type, `in_flight_message` type, and `config` type. And then, we explain how to formalize the operational semantics in Actario.

3.1 Actor Name

In Actario, actor name is defined as the disjoint sum of the case of an actor with no parent and the case of an actor generated by another actor (Figure 6). We call the actors having no parent *top level actor*. Top level actor represents initial actors in the system. And we call the actors generated by another actor *generated actor*. The name of a generated actor consists of the name of parent actor and the number that the parent actor generated so far. We call the number *generation number*. To keep name uniqueness, we introduce generation number. For more detail about name uniqueness, see Section 4.

3.2 Actor

We explain how `actor` is defined in Actario. Actor consists of its name, sequence of remaining actions, and next generation number to use in generating next child (Figure 7). If remaining actions are only become, the actor is ready for receiving a message.

3.3 Messages in Flight

Next, we define `in_flight_message` type which represents messages in flight in the configuration. `in_flight_message` consists of the name of the destination, the name of the sender, and the content of the message (Figure 8).

3.4 Configuration

configuration represents a snapshot of the actor system. configuration is used to formulate operational semantics of the

```
Record in_flight_message := {
  to : name;
  from : name;
  content : message
}.
```

Figure 8. in flight message

```
Record config := {
  in_flight_messages :
    list in_flight_message;
  actors : list actor
}.
```

Figure 9. config

```
Inductive label :=
 | Receive (to : name) (from : name)
     (content : message)
 | Send (from : name) (to : name)
     (content : message)
 | New (child : name)
 | Self (me : name).
```

Figure 10. label

Actor model. In Actario, a configuration consists of a list of actors and a list of messages in flight.

3.5 Transition Label

Actario formulates operational semantics of the Actor model as labeled transition system, so we define label (Figure 10). The explanations of each label are as follows.

`Receive (to : name) (from : name) (content : message)`
This represents that the actor named `to` receives the message `content` sent from the actor named `from`.

`Send (from : name) (to : name) (content : message)`
This represents that the actor named `from` sends the message `content` to the actor named `to`.

`New (child : name)`
This represents that the actor named `child` is generated.

`Self (me : name)`
This represents that the actor named `me` gets the name itself.

$$
\begin{array}{lcll}
c & \in & \textit{Configuration} & = & \textit{Set(InFlight)} \times \textit{Set(Actor)} \\
a & \in & \textit{Actor} & = & \textit{Name} \times \textit{Actions} \times \mathbb{N} \\
n & \in & \textit{Name} & ::= & \mathsf{toplevel}(s) \\
 & & & | & \mathsf{generated}(g, n) \\
m & \in & \textit{Message} & = & \textit{Name} + \textit{PrimVal} + \\
 & & & & \textit{Message} \times \cdots \times \textit{Message} \\
i & \in & \textit{InFlight} & = & \textit{Name} \times \textit{Name} \times \textit{Message} \\
b & \in & \textit{Behavior} & = & \textit{Message} \to \textit{Actions} \\
\alpha & \in & \textit{Actions} & ::= & \mathsf{send}(n, m, \alpha) \\
 & & & | & \mathsf{new}(b, \kappa) \\
 & & & | & \mathsf{self}(\kappa) \\
 & & & | & \mathsf{become}(b) \\
l & \in & \textit{Label} & ::= & \mathsf{Receive}(n, n, m) \\
 & & & | & \mathsf{Send}(n, n, m) \\
 & & & | & \mathsf{New}(n) \\
 & & & | & \mathsf{Self}(n) \\
\kappa & \in & \textit{Name} \to \textit{Actions} & & \\
g & \in & \mathbb{N} & &
\end{array}
$$

Figure 11. Configuration

3.6 Semantics

We formulate operational semantics of the Actor model as labeled transition system. For the later explanation, we define the symbols as shown in Figure 11.

The labeled transition system used in Actario is defined like Figure 12. The explanations for each of transitions are the followings.

RECEIVE
RECEIVE is the transition for Receive label. The actor which is ready to receive a message, in other words, the actor whose remaining actions are only become, receives a message and generate new remaining actions decided by the behavior and the content of the message.

SEND
SEND is the transition for Send label. The actor which want to send a message sends a message, and then the message is added into messages in flight.

NEW
NEW is the transition for New label. An actor generates its child actor by the given behavior. And then, do the followings:

- The child actor is added into the configuration. The next generation number of child actor is 0.
- The next generation number of the parent actor increases by 1.
- The child actor is ready to receive a message.

SELF
SELF is the transition for Self label. An actor gets the self name and applies it to the continuation.

The definition in Actario is in Appendix A.

$$(I \uplus \{(n_{\text{to}}, n_{\text{from}}, m)\}, A \cup \{(n_{\text{to}}, \mathsf{become}(b), g)\}) \overset{\mathsf{Receive}(n_{\text{to}}, n_{\text{from}}, m)}{\rightsquigarrow} (I, A \cup \{(n_{\text{to}}, b(m), g)\}) \quad \text{(RECEIVE)}$$

$$(I, A \cup \{(n_{\text{from}}, \mathsf{send}(n_{\text{to}}, m, \alpha), g)\}) \overset{\mathsf{Send}(n_{\text{from}}, n_{\text{to}}, m)}{\rightsquigarrow} (I \uplus \{(n_{\text{to}}, n_{\text{from}}, m)\}, A \cup \{(n_{\text{from}}, \alpha, g)\}) \quad \text{(SEND)}$$

$$(I, A \cup \{(n, \mathsf{new}(b, \kappa), g)\}) \overset{\mathsf{New}(n')}{\rightsquigarrow} (I, A \cup \{(n, \kappa(n'), g+1), (n', \mathsf{become}(b), 0)\}) \quad \text{where } n' := \mathsf{generated}(g, n) \quad \text{(NEW)}$$

$$(I, A \cup \{(n, \mathsf{self}(\kappa), g)\}) \overset{\mathsf{Self}(n)}{\rightsquigarrow} (I, A \cup \{n, \kappa(n), g\}) \quad \text{(SELF)}$$

Figure 12. labeled transition semantics

```
Theorem actor_persistence :
  ∀ c c' l n,
    n \in map actor_name (actors c) →
    c ~(l)⇝ c' →
    n \in map actor_name (actors c').
```

Figure 13. Actor Persistence

```
Theorem message_persistence :
  ∀ c c' l m (n : nat),
    n == count_mem m (in_flight_messages c) →
    c ~(l)⇝ c' →
    if l == Receive (to m) (from m)
                    (content m) then
      count_mem m (in_flight_messages c')
                                      == n.-1
    else
      if l == Send (from m) (to m)
                   (content m) then
        count_mem m (in_flight_messages c')
                                      == n.+1
    else
      count_mem m (in_flight_messages c') == n.
```

Figure 14. Message Persistence

3.7 Actor Persistence and Message Persistence

In this semantics, actor persistence property (the property that actors do not disappear) and message persistence property (the property that messages in flight do not disappear except of receiving) are formally proved. Each of the definitions is shown in Figure 13 and Figure 14. The number of lines of the proofs is less than 100 lines in Actario [2].

[2] https://github.com/amutake/actario/blob/d9e5084c87e7e0bc630ffa0f96b0b3b49d65fa9a/src/persistence.v

4. Name Uniqueness

In programming languages or libraries providing the Actor model such as Erlang or Akka, the system automatically generates actors with fresh names without specifying the name explicitly by the programmer. In Actario, the proposition that all actor names in the configuration are not duplicate by any transitions is proved.

To prove, we define an invariant about actor names preserved between any transitions. It is named *trans invariant*. The trans invariant consists of the following three predicates for configuration.

$$\texttt{trans\_invariant}(c) := \texttt{chain}(c) \wedge \texttt{gen\_fresh}(c) \wedge \texttt{no\_dup}(c)$$

The brief explanations of `chain`, `gen_fresh`, and `no_dup` are followings:

`chain`
: For each actor in the configuration, if the actor is generated by another actor, then the parent actor is also in the configuration.

`gen_fresh`
: For each actor in the configuration, actor name genereted by the actor in the next is fresh.

`no_dup`
: For all actor name in the configuration are unique.

4.1 Functions

Before starting the explanation and the proof, we define some functions used in this section.

`actors` : ***Configuration*** $\to$ ***Set(Actor)***
: `actors` returns the set of actors in the given configuration.

`parent` : ***Actor*** $\to$ ***Actor***
: `parent` returns the parent actor of the given actor. If the given actor is toplevel actor, the function returns nothing.

`gen_number` : ***Actor*** $\to \mathbb{N}$
: `gen_number` returns generated number of the name of

the given actor. If the given actor is toplevel actor, the function returns nothing.

`next_number` : ***Actor*** $\rightarrow \mathbb{N}$

`next_number` returns next generation number of the given actor.

`name` : ***Actor*** $\rightarrow$ ***Name***

`name` returns the name of the given actor.

`names` : ***Set(Actor)*** $\rightarrow$ ***Set(Name)***

`names` returns names of the given set of actors.

4.2 Chain Property

We define a predicate of configuration, called `chain`. `chain` is the predicate that, for each actor in the given configuration, if it is generated by another actor, the parent actor is also in the configuration. `chain` is defined as the following.

$$\texttt{chain}(c) := \\ \forall a \in \texttt{actors}(c), \forall p, p = \texttt{parent}(a) \Rightarrow p \in \texttt{actors}(c)$$

Then, we can prove *chain preservation property* that chain is preserved between any transitions. The proof is by case analysis on the label. `chain` is decided by only actor names, and the transition which have a possibility to change the names in the configuration is only NEW transition. Therefore, we consider only the case of NEW transition.

LEMMA 1. *chain preservation*

$$\forall c, c' \in \textit{Configuration}, \forall l \in \textit{Label}, \\ \texttt{chain}(c) \land c \overset{l}{\rightsquigarrow} c' \Rightarrow \texttt{chain}(c')$$

4.3 Gen Fresh Property

We define `gen_fresh` predicate that, for each actor in the configuration, the name of its child is always fresh. The definition of `gen_fresh` is complicated a little. We translate the proposition that next generated name is fresh to the following.

$$\texttt{gen\_fresh}(c) := \\ \forall a \in \texttt{actors}(c), \forall p \in \texttt{actors}(c), p = \texttt{parent}(a) \Rightarrow \\ \texttt{gen\_number}(a) < \texttt{next\_number}(p)$$

It is guaranteed that the actor name generated in the next is fresh if satisfying `gen_fresh` predicate by the relation of next generation numbers and actor names. However, the actor name generated after the next is not always fresh name. For example, if there are two actors (A and B) that have the same name and the same next generation number and actor A generates a child actor and actor B generates a child actor, although `gen_fresh` holds, these child actors have the same name. Furthermore, if the parent of the actor A does not exist in the configuration and the parent of the parent exists in the configuration, and the parent of the parent actor generates an actor and it also generates an actor, then the name is possible to have the same as A's one.

Thus, to prove *gen fresh preservation* proposition that `gen_fresh` is preserved between transitions, it is necessary to use `chain` and `no_dup` as hypotheses.

LEMMA 2. *gen fresh preservation*

$$\forall c, c' \in \textit{Configuration}, \forall l \in \textit{Label}, \\ \texttt{chain}(c) \land \texttt{gen\_fresh}(c) \land \texttt{no\_dup}(c) \land c \overset{l}{\rightsquigarrow} c' \Rightarrow \\ \texttt{gen\_fresh}(c')$$

4.4 No Dup Property

We define `no_dup` predicate that all actor names in the given configuration are unique. This is the property we have to prove. `no_dup` is defined as the following.

$$\texttt{no\_dup}(c) := \\ \forall a \in \texttt{actors}(c), \texttt{name}(a) \notin \texttt{names}(\texttt{actors}(c) \setminus \{a\})$$

We proved *no dup preservation* property defined as the following. It represents that if the actor names in the configuration is not duplicate and the next generated actor name is fresh, then `no_dup` holds in the next configuration.

LEMMA 3. *no dup preservation*

$$\forall c, c' \in \textit{Configuration}, \forall l \in \textit{Label}, \\ \texttt{gen\_fresh}(c) \land \texttt{no\_dup}(c) \land c \overset{l}{\rightsquigarrow} c' \Rightarrow \texttt{no\_dup}(c')$$

4.5 Proof of Name Uniqueness Property

Then, we start to prove name uniqueness. First, we prove trans invariant preservation that trans invariant is preserved between transitions. This is obvious by chain preservation, gen fresh preservation and no dup preservation.

LEMMA 4. *trans invariant preservation*

$$\forall c, c' \in \textit{Configuration}, \forall l \in \textit{Label}, \\ \texttt{trans\_invariant}(c) \land c \overset{l}{\rightsquigarrow} c' \Rightarrow \\ \texttt{trans\_invariant}(c')$$

Next, we prove that if trans invariant holds in initial configuration, trans invariant holds after arbitrary transitions.

LEMMA 5. *trans invariant preservation star*

$$\forall c, c' \in \textit{Configuration}, \forall l \in \textit{Label}, \\ \texttt{trans\_invariant}(c) \land c \overset{l}{\rightsquigarrow} \star\, c' \Rightarrow \\ \texttt{trans\_invariant}(c')$$

$c \overset{l}{\rightsquigarrow} \star\, c'$ represents reflexive transitive closure of transition. The proof is by induction of reflexive transitive closure of transition and trans invariant preservation.

Finally, we can prove name uniqueness.

THEOREM 1. *name uniqueness*

$$\forall c, c' \in \textit{Configuration}, \forall l \in \textit{Label}, \\ \texttt{trans\_invariant}(c) \land c \overset{l}{\rightsquigarrow} \star\, c' \Rightarrow \texttt{no\_dup}(c')$$

This is obvious by trans invariant preservation star because `no_dup` is in `trans_invariant`.

5. Fairness

fairness is a property that reception of a message does not delay infinitely. There are two variants of fairness property, weak fairness and strong fairness. Weak fairness is that if an actor is infinitely always ready to receive the message, the message is eventually received. Strong fairness is that if an actor is infinitely often ready to receive the message, the message is eventually received. The Actor model satisfies strong fairness. We have not proved any properties using strong fairness yet, but for a case study, we explain how to define strong fairness in Actario.

5.1 Transition Path

Generally, fairness is represented in using operators of temporal logic. We have to encode temporal logic because Coq does not support temporal logic. We use transition path, which represents transition sequence of configuration, to define fairness as a predicate of transition path. This method is used in Applπ [2].

We define transition path as a function of $\mathbb{N}$ to `option config`. In this definition, $\mathbb{N}$ represents the number of transitions from initial configuration and the reason why the return value is wrapped with `option` is that it may be no more transitions.

```
Definition path := nat → option config.
```

And we define the predicate that the given path is correct transition path.

```
Definition is_transition_path
    (p : path) : Prop :=
  ∀ n,
    (∀ c, p n = Some c →
      (∃ c' l, p (S n) = Some c' /\
        c ~(l)⇝ c') \/
      p (S n) = None) /\
    (p n = None → p (S n) = None).
```

5.2 Enabled

We define the predicate that the transition from the given configuration with the given label is possible, called `enabled`. In Actario, `enabled` is defined as there exists a configuration after transitioning from the configuration with the label, as follows.

```
Definition enabled (c : config)
    (l : label) : Prop :=
  ∃ c', c ~(l)⇝ c'.
```

5.3 Infinitely Often Enabled

We define the predicate that the transition is infinitely often enabled in the transition path. It is named `infinitely often enabled`.

```
Definition infinitely_often_enabled
    (l : label) (p : path) : Prop :=
  ∀ n c, p n = Some c →
    enabled c l →
    ∃ m c', m > n /\
      p m = Some c' /\
      enabled c' l.
```

5.4 Eventually Processed

We define `eventually processed` that is the predicate of label and transition path. It represents that the transition with the label is processed eventually in the path. It is defined as follows.

```
Definition eventually_processed
    (l : label) (p : path) : Prop :=
  ∃ n c c',
    p n = Some c /\
    p (S n) = Some c' /\
    c ~(l)⇝ c'.
```

5.5 Definition of Fairness

Then we can define `fairness` predicate for transition path. For the given transition path and for each label, if `infinitely often enabled` holds, then `eventually processed` holds. `is postfix of` predicate is used for representing 'infinite'. If `is postfix of` is not used, the transition may not be processed after the transition is processed although the transition is processed in whole the path. To prevent it, if `inifinitely often enabled` holds then `eventually processed` holds for arbitrary postfix path by using `is postfix path`.

```
Definition is_postfix_of
    (p' p : path) : Prop :=
  ∃ n, (∀ m, p' m = p (m + n)).

Definition fairness : Prop :=
  ∀ p p', is_postfix_of p' p →
    (∀ l,
      infinitely_often_enabled l p' →
      eventually_processed l p').
```

6. Extraction

Extraction is a Coq feature which enables to convert Coq programs to the programs of other languages. Normal Coq can extract programs to OCaml, Haskell, and Scheme. If

```
(* Inductive nat :=  *)
(* | O : nat         *)
(* | S : nat → nat. *)

O (* ⇒ {o} *)
S (S (S O)) (* ⇒ {s, {s, {s, o}}} *)
```

Figure 15. example of extraction of algebraic data types

```
CoFixpoint behvA :=
  receive (fun msg ⇒
    match msg with
      | name_msg sender ⇒
        me ← self;
        sender ! name_msg me;
        become behvA
      | _ ⇒
        child ← new behvB;
        child ! msg;
        become behvA
    end)
```

Figure 16. Extraction example: Actario code

we want to extract to other languages or use custom extraction algorithm, we have to implement it as plugins or patches. Actario has custom extraction mechanism for the programs using Actario. It can extract to Erlang. In Actario, `ActorExtraction` command is defined for extracting actor systems. It is used like traditional `Extraction` command.

6.1 Data Types

Values of algebraic data types are extracted to a tuple with the label. Value constructor is extracted to a label, and arguments are extracted to the second and the following elements of the tuple. Figure 15 is an example of extraction of the natural number type.

However, actions of actors, for example, send, new, self, become and `behavior` are implemented as value constructor of `actions` and `behavior` type We handle these constructors as special to generate the corresponding syntax of Erlang.

For example, Actario code shown in Figure 16 is extracted to Erlang code shown in Figure 17.

6.2 Name

In Actario, a programmer does not make actor names from constructors, so that all of actor names are in variables. Therefore, all of actor names in extracted code are variables. These variables are bound by values of name type in Actario, but in Erlang, these variables are bound by process ids.

```
behvA() →
  receive Msg → case Msg of
    {name_msg, Sender} →
      Me = self(),
      Sender ! {name_msg, Me},
      behvA()
    _ →
      Child = spawn(fun() → behvB() end),
      Child ! Msg
      behvA()
  end.
```

Figure 17. Extraction example: Erlang code

6.3 Execution

The program extracted by Actario is impossible to execute by itself. So Actario's programmers have to write executor to execute the extracted Actor system in Erlang. For example, we consider factorial system described in Section 2.

```
Definition factorial_system (n : nat)
    (parent : name) : config :=
  init "factorial" (
    x ← new factorial_behv;
    x ! tuple_msg (nat_msg n)
          (name_msg parent);
    become empty_behv
  ).
```

`factorial_system` is extracted to the following Erlang code.

```
factorial_system(N, Parent) →
    X = spawn(fun() →
        factorial_behv()
    end),
    X ! {tuple_msg, {nat_msg, N},
          {name_msg, Parent}},
    empty_behv().
```

To execute this, we have to write executor like Figure 18. `nat2int` and `int2nat` are auxiliary functions for converting Coq's natural number and Erlang's integer.

6.4 Future Work for Erlang Extraction

Currently, it is not proved that the extraction mechanism does not change the meanings of Actario programs and Erlang programs and properties such as name uniqueness. In order to show these properties, we have to formalize Erlang in Coq and extraction mechanism, write extraction mechanism in Coq, and prove the preservation of a certain property.

```
-module(fact_main).
-export([fact/1]).

fact(N) →
  _ = spawn(factorial, factorial_system,
        [int2nat(N), self()]),
  receive
    {nat_msg, Result} →
      io:fwrite("fact(~w) = ~w~n",
        [N, nat2int(Result)]);
    _ →
      io:fwrite("error~n")
  end.
```

Figure 18. Example: user code to execute factorial system

Furthermore, we like to provide bridge library between user code and extracted code for convenience, for example, `nat2int` and `int2nat` in Figure 18.

7. Related Work

Applπ is a Coq library for modeling and verifying concurrent programs [2]. Actario is very inspired by Applπ, for example, the definition of fairness, continuation passing style in `actions` and framework design. The main difference of Applπ and Actario is that Applπ adopts π-calculus for its concurrent computation basic, but Actario adopts the Actor model for its concurrent computation basic.

Musser and Varela[9] formalized the Actor model for the Athena theorem prover[5]. Within their formalization, maintaining the uniqueness of actor names is formally proved. However, one must manually specify a fresh name for each new actor. In contrast, the automatic actor naming mechanism in Actario eases the specification of complex systems. In addition, Actraio provides an extraction mechanism of runnable Erlang code.

Verdi is a framework for constructing and verifying fault-tolerant distributed systems [14]. A system assumed no network failure is converted to the system which tolerates dropping packets, duplication of packets, and machine failure. One of the purposes of Actario is also to build and verify fault-tolerant distributed systems. We will introduce *supervisor* mechanism to achieve building fault-tolerant systems generally used in Erlang and Akka.

Tony Garnock-Jones, Sam Tobin-Hochstadt, and Matthias Felleisen give a formalization of the Actor model using Coq [8]. In this paper, the operational semantics is formalized so that transition is decidable. Due to this, it is difficult to apply the formalization to realistic concurrent systems.

8. Concluding Remarks

In this paper, we present Actario, a Coq framework for describing and verifying actor-based systems. Actario is designed to support implicit naming of actors. This simplifies the description of actor systems. We have formally proved that the underlying execution model provided in the framework satisfies important properties including name uniqueness, actor persistence and message persistence. The fact implies that a system described using Actario is guaranteed to have these actor properties.

Actario is currently under development and still does not provide convenient libraries of predicates, lemmas, tactics and so forth. Thus, verifying a user-defined actor system may involve a large amount of work. Providing such libraries should be included in the future work.

In addition, we like to extend Actario to support extended Actor models. For example, extensions that support high-level synchronization mechanisms such as [6], coordination models[12], and reflective models such as [13].

A. Labeled Transition Semantics in Actario

The full labeled transition semantics described in Section 3 is shown in Figure 19. The each of inductive constructors corresponds to each transitions of Figure 12.

References

[1] Actario. Actario: A framework for verifying actor based systems. `https://github.com/amutake/actario`.

[2] R. Affeldt and N. Kobayashi. A Coq library for verification of concurrent programs. In *Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004)*, volume 199 of *Electronic Notes in Theoretical Computer Science*, pages 17–32, 2008. . URL `http://www.sciencedirect.com/science/article/pii/S1571066108000765`.

[3] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986. URL `http://mitpress.mit.edu/books/actors`.

[4] Akka. Akka. `http://akka.io/`.

[5] K. Arkoudas. Athena. `http://proofcentral.org/athena`.

[6] J. De Koster, T. Van Cutsem, and T. D'Hondt. Domains: Safe sharing among actors. In *Proceedings of the 2nd edition on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions (AGERE!@SPLASH 2012)*, pages 11–22. ACM, ACM, 2012. .

[7] Erlang. Erlang programming language. `http://www.erlang.org/`.

[8] T. Garnock-Jones, S. Tobin-Hochstadt, and M. Felleisen. The network as a language construct. In *Programming Languages and Systems (ESOP 2014)*, volume 8410 of *Lecture Notes in Computer Science*, pages 473–492. Springer-Verlag, 2014. . URL `http://www.ccs.neu.edu/home/tonyg/esop2014/`.

[9] D. R. Musser and C. A. Varela. Structured reasoning about actor systems. In *Workshop on Programming based on Actors,*

```
Reserved Notation "c1 '~(' t ')⇝' c2" (at level 60).
Inductive trans : label → config → config → Prop :=
(* receive transition *)
| trans_receive :
    ∀ to from content f gen sendings_l sendings_r actors_l actors_r,
      (sendings_l ++ Build_sending to from content :: sendings_r)
                ⋈ (actors_l ++ Build_actor to (become (receive f)) gen :: actors_r)
        ~(Receive to from content)⇝
        (sendings_l ++ sendings_r) ⋈ (actors_l ++ Build_actor to (f content) gen :: actors_r)
(* send transition *)
| trans_send :
    ∀ from to content cont gen sendings_l sendings_r actors_l actors_r,
      (sendings_l ++ sendings_r)
                ⋈ (actors_l ++ Build_actor from (send to content cont) gen :: actors_r)
         ~(Send from to content)⇝
         (sendings_l ++ Build_sending to from content :: sendings_r)
           ⋈ (actors_l ++ Build_actor from cont gen :: actors_r)
(* new transition *)
| trans_new :
    ∀ parent behv cont gen sendings actors_l actors_r,
      sendings ⋈ (actors_l ++ Build_actor parent (new behv cont) gen :: actors_r)
        ~(New (generated gen parent))⇝
        sendings ⋈
          (Build_actor (generated gen parent) (become behv) 0 ::
          actors_l ++
          Build_actor parent (cont (generated gen parent)) (S gen) ::
          actors_r)
(* self transition *)
| trans_self :
    ∀ me cont gen sendings actors_l actors_r,
      sendings ⋈ (actors_l ++ Build_actor me (self cont) gen :: actors_r)
        ~(Self me)⇝
        sendings ⋈ (actors_l ++ Build_actor me (cont me) gen :: actors_r)
where "c1 '~(' t ')⇝' c2" := (trans t c1 c2).
```

Figure 19. Labeled Transition Semantics in Actario

Agents, and Decentralized Control (AGERE!@SPLASH 2013), pages 37–48. ACM, oct 2013. .

[10] Scala. The Scala programming language. `http://scala-lang.org/`.

[11] M. Sirjani and M. M. Jaghoori. Ten years of analyzing actors: Rebeca experience. In G. Agha, O. Danvy, and J. Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 20–56. Springer-Verlag, 2011. .

[12] C. Talcott, M. Sirjani, and S. Ren. Comparing three coordination models: Reo, ARC, and PBRD. *Science of Computer Programming*, 76(1):3–22, 2011. .

[13] T. Watanabe. Towards a compositional reflective architecture for actor-based systems. In *Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!@SPLASH 2013)*, pages 19–24. ACM, oct 2013. .

[14] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 357–368, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. . URL `http://doi.acm.org/10.1145/2737924.2737958`.

[15] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA '86 Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 258–268, 1986. .

Programming Abstractions for Augmented Worlds

Angelo Croatti
DISI, University of Bologna
Via Sacchi, 3 – Cesena, Italy
a.croatti@unibo.it

Alessandro Ricci
DISI, University of Bologna
Via Sacchi, 3 – Cesena, Italy
a.ricci@unibo.it

ABSTRACT

The impressive development of technologies is reducing the gulf between the physical and the digital matter, reality and virtuality. In the short future, the design and development of *augmented worlds* – as software systems extending the physical space and environment with computational functionalities and an augmented reality-based appearance – could become an important aspect of programming, calling for novel programming abstractions and techniques. In this paper we introduce this vision, discussing *mirror worlds* as augmented worlds programmed in terms of agent-oriented abstractions.

1. INTRODUCTION

In the short future, the design and development of *augmented worlds* could become an important aspect of programming, not only related to specific application domains—such as the ones traditionally targeted by augmented/mixed/hyper reality [3, 2, 28, 6].

The notion of *augmentation* that we consider in this paper concerns different aspects, spanning from *augmented/mixed reality* as a primary one, to *human augmentation* – enabled by mobile/wearable technologies – and *environment augmentation* – based on pervasive computing and Internet of Things (IoT).

Mixed reality refers to the merging of real and virtual worlds to produce new environments and visualizations where physical and digital objects co-exist and interact in real time [5]. As defined by P. Milgram and F. Kishino, it is "anywhere between the extrema of the virtuality continuum" [13], that extends from the completely real through to the completely virtual environment with augmented reality (AR) and augmented virtuality ranging between. In recent years, there has been an impressive development of hardware technologies related to mobile and wearable supporting different degrees of AR—the main example is given by *smart glasses*. Conceptually, this kind of devices allows to extend people cognitive capabilities, improving the way in which they perform their tasks [26]. This can be interpreted as a form of *human augmentation*, in a way that recalls the Augmented System conceptual framework introduced by the computer science pioneer Doug Engelbart more than 50 years ago [8]. As remarked in [31], it is not only a matter of an hardware augmentation: the software plays a key role here such that we can talk about *software-based* augmentation.

These technologies can have a huge impact from an application point view, allowing for rethinking the way in which people work, interact and collaborate—escaping from the limits that current mobile devices such as smartphones and tablets enforce. A main one is about requiring users to use their hands and watch 2D screens. Smart-glasses allow to free users' hands and perceive application output while perceiving the physical reality where they are immersed.

The vision of pervasive/ubiquitous computing [29, 22, 11, 18], which is more and more entering in the mainstream with the IoT [10, 33], provides a further and related form of augmentation of the physical world. In this case, the augmentation is given by open ecosystems of connected and invisible computing devices, embedded in physical objects and spread in the physical environment, equipped with different kinds of sensors and actuators. Data generated by these devices is typically collected and managed in the cloud, and accessed by mobile/cloud applications. This embedded computing layer *augments* the functionalities of the physical things that people use everyday and, again, are going to have a strong impact on how people work, interact and collaborate [16]. This view about environment augmentation strongly recalls the idea of *computer-augmented environments* by P. Wellner et al. [30], in which the digital (cyber) world is merged with the physical one.

In literature, these forms of augmentation are discussed in separated contexts, mainly focusing on solving issues related to the enabling hardware/software technologies and to develop ad hoc systems for specific application domains. In this paper we are interested to consider these augmentations from a *programming perspective*. In particular we see a convergence that can be captured by the idea of *computation-as-augmentation* of the physical reality, and – more specifically – of computer programs designed to be an extension of the physical environment where people live and work in. To capture this viewpoint, in this work we propose the notion of *augmented worlds* as a conceptual framework to start explor-

ing the main concepts, principles and problems underlying these kinds of programs, their design and development. An augmented world is a computer program augmenting the functionalities of some physical environment by means of full-fledge computational objects *located in the space* that users can perceive and interact with by means of proper mobile/wearable devices. These *augmented entities* – being them objects, actors or agents – can model also extensions of existing physical things, so that their state can be affected by their physical counterpart, and, viceversa, this one can be affected by events/actions occurring at the digital level.

In the remainder of this paper, we first provide an overview of main concepts and features related to augmented worlds (Section 3), abstracting from any specific programming paradigm that can be adopted to implement them. Then, we focus on programming, first providing a general overview of the programming issues that concern the adoption of specific paradigms (Section 4) and then discussing *mirror worlds* (Section 5), which provide a concrete agent-oriented programming model to develop augmented worlds.

Mirror worlds have been introduced in literature in the context of smart environments modelled in terms of agents and multi-agent systems [20]. Inspired by them, the idea of augmented worlds represent a generalization aimed at capturing the main principles concerning these new kinds of systems, as well as discussing the main aspects involved by their design and programming.

Besides introducing the idea of augmented worlds, the aim of this paper is also to provide an overview of some main research challenges and open issues (Section 6), eventually defining a first research agenda for future work.

Before describing more extensively the main concepts that characterize augmented worlds, in next section we review the main research work and technologies that are related to the vision proposed here.

2. BACKGROUND

The level of sophistication achieved by the techniques developed in the context of Augmented/Mixed Reality and Mobile Augmented Reality (MAR) research [3, 5, 24, 23] – supported by the availability of more and more powerful hardware (sensors, processors, displays, wearables) – makes it possible today to focus on the design of augmented worlds assuming that the basic enabling functionalities – such as indoor/outdoor *tracking*, to determine the location and orientation of things in the real world – are available, provided by a bottom layer of the software stack. This does not concern only the *location*—which is however an essential aspect in our case, like in location-based applications. It may include also other elements that more generally define the user *context*—in fact, the techniques developed in context-aware computing [7] are another enabling brick of the vision.

The maturity of these technologies is witnessed by the products and solutions that are entering into the mainstream. Among the others, a main recent one is Microsoft HoloLens [1]—in which we find many points of the augmented world vision. By exploiting an holographic helmet, Microsoft HoloLens generates a multi-dimensional image visible to the user wearing the helmet so that he or she perceives holographic objects in the physical world. Holographic objects are similar to GUI objects whose canvas is the real world—they can be pinned, or anchored, to physical locations chosen by the user, moved according to their own rules, or remain in a specific location within user's field of view regardless of where the user is or in which direction she/he is looking. Beside the many similarities, the kind of augmentation provided in augmented worlds is conceptually different. In particular, holographic objects – as far as authors' understanding from the information currently available about HoloLens [1] – are meaningful only if there is a user who (generates and) perceives them. Conversely, an augmented world has an *objective existence* which does not depend on the users that are located inside: it is first of all a computational augmentation of the environment, which can be then perceived by users with proper devices.

Augmented worlds are typically multi-user systems—a main goal is to ease the development of applications supporting *collaborative* hands-free human activities. From this point view, the augmented worlds vision shares many similarities with online multi-user distributed collaborative environments such as Croquet [25]. The main difference is that, instead of being purely virtual, augmented worlds are deployed in the physical world.

The kind of augmentation typically provided by AR/MAR system is information-oriented, i.e., the objective is to overlay on the physical reality a set of 2D/3D objects that provide some information content about that physical reality, that can be perceived by proper AR browsers [12]. An important exception is given by MAR *games* [27], which have many points in common with the augmented world view. In fact, they typically create multi-user immersive environments blended with the physical reality composed not only by informational objects but computational elements with a behavior—e.g., game bots and characters. In this paper we argue that the usefulness of this kind of extended augmentation goes beyond games, making the investigation of general purpose programming frameworks and abstractions to support them a relevant research topic.

Finally, the vision proposed in this paper is strongly related to any research work that investigates computing systems in which the *physical space* become a first-order concept of the computation layer, beyond the perspectives already developed in distributed and mobile computing. To authors' knowledge, in literature this view has been developed so far by *spatial computing* [32]—whose perspective, however, is quite different from the one discussed in this paper. In fact, spatial computing systems are typically based on very large set of mobile computing devices locally-connected and location-aware that support the execution of distributed computations in which the spatial configuration of the elements has a primary role.

3. AUGMENTED WORLDS – CONCEPTS

In this section we identify and discuss the main features that characterize augmented worlds. These features are mostly independent from the specific programming paradigms that can be adopted to implement them – for this reason we will try to abstract as much as possible from them.

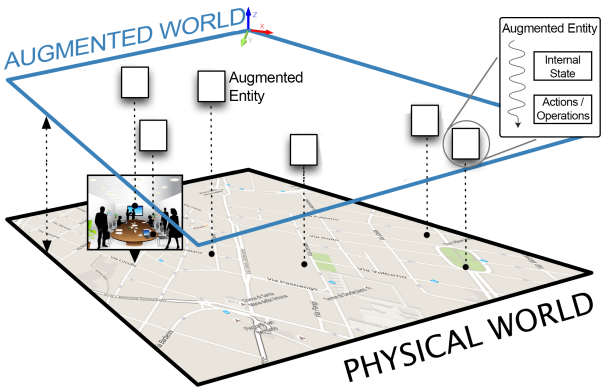


Figure 1: A simple representation of an Augmented World, focusing on spatial coupling aspect of augmented entities.

3.1 Spatial Coupling

The main distinguishing feature that characterizes augmented worlds is to be composed by computational objects that are situated in some specific location of the physical reality (see Figure 1). In the following we will refer to these computational objects as *augmented entities*, abstracting from the specific programming paradigm that be used to implement them – they could be objects in OOP, actors in actor programs, etc.

An augmented entity is instantiated at runtime by specifying also its location, that could be either directly a geographical position or indirectly the reference to a physical object living in the physical reality. The location can be specified with respect to a local system of reference defined by the augmented world. An important point here is that augmented entities are meant to model not just data items – like in the case of POIs (Point-of-Interests) as found in MAR applications – but any full-fledged computational entity, eventually encapsulating a state and a behaviour. This is the basic fundamental feature which is useful in particular to build applications where the augmentation is not just an information layer about physical things, but more complex services or functionalities involving processing and interaction.

It is worth clarifying that an augmented entity is bound to some physical location not (necessarily) because it is running on some hardware device physically situated at *that* location—like in the case of e.g., spatial computing [32]. The code, data and runtime state of augmented entities are meant to be managed by one or multiple computing server nodes hosting augmented worlds execution, possibly in the cloud.

Besides the location, an augmented entity can have a *geometry* modeling the extension of the object in the physical space, like in classic AR application. This geometry can be described e.g., as a set of 3D points defined with respect to a local system of reference.

Both the location and the geometry are part of the computational state of the entity, and can change at runtime then by virtue of the computation and interaction occurring inside the augmented world.

3.2 Discovery and Observability

The spatiality of augmented entities can be exploited to enrich the ways in which these computational objects are discovered, observed and more generally how they interact with each other—including interaction with users.

About discovery, inside an augmented world (as a system in execution), the reference to an augmented entity could be retrieved by *lookup* functions – provided by the augmented world runtime – specifying the region of the world to be queried, among the possible filters. This is essentially a pull-based approach to discovery.

Besides, a push-based/event-driven modality is useful as well, that is: retrieving the reference to an augmented entity as soon as it becomes *observable*, given its position with respect to the position of the *observing* entity. In order to support this modality, the augmented entity abstraction can be further characterized by two observation-related properties: an *observability neighborhood* and an *observation neighborhood*, defining a spatial-based constraint on the set of entities that – respectively – can observe/perceive the augmented entity and can be observed/perceived by the augmented entity. The simplest kind of spatial neighborhood can be defined in terms of distances between the position where the entities are located. In this case, these properties can be simply referred as an *observability radius* and *observation radius* (see Figure 2, left).

As a simple example about the usefulness of these properties, an augmented entity representing a message can be left in a specific point in a city and can be observed (read) only by entities that are in message's proximity (e.g., the ones representing human users immersed in the augmented world). As another example, a counter entity left in a specific geographical location with the purpose to count how much entities pass near the counter, may be defined to perceive all entities in a very short observation radius, but the observability radius can be set to zero, so other entities does not necessarily perceive the counter.

Also the properties related to observability can change at runtime, depending on the functionality of the augmented entity, its computing behaviour and interaction within the world.

3.3 User Modeling and Interaction

User interaction is a main aspect of augmented worlds, whose primary objective is to model applications where human users – and robots, eventually – are engaged in some kind activity in some physical environment.

An augmented world is typically a multi-user application, where multiple users perceive, share and interact within the same augmented entities. An effective way to model a user in an augmented world is to associate her/him with an augmented entity – as a kind of *augmented body* – with features that are useful for controlling user input/output and a computational state reifying the dynamic information about the user (state) that can be relevant at the application level. By

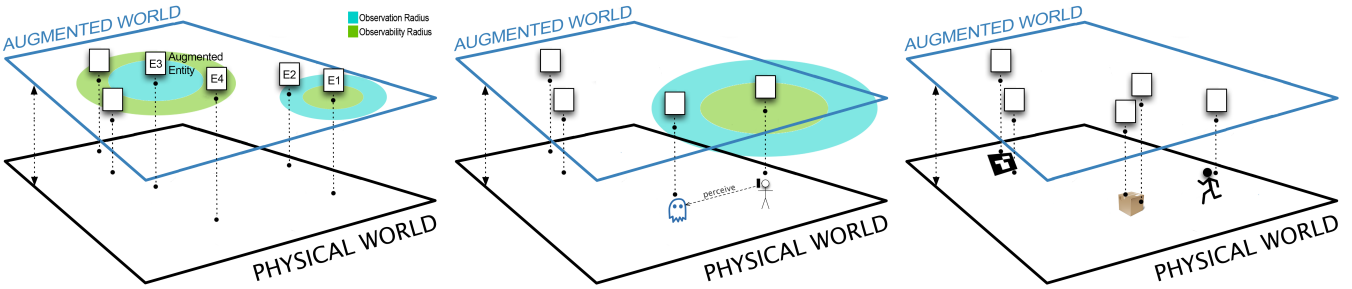


Figure 2: (left) In this time instant, the entity E1 can perceive E2 but cannot be observed by other entities (no entities within its observability radius). Viceversa, E3 can perceive E4 and can be observed by all others entities within its observability radius. (center) An human wearing glasses can perceive ghost entity through its associated assistant entity. (right) Physical Embedding and Coupling issue representation.

exploiting the observation-related features, it is possible to determine which entities of the world the user is perceiving and their observable state (see Figure 2, center). This state can include also the geometry of the entity, which can be eventually exploited for constructing the view perceived by the user.

Besides, a key aspect of this modeling is that it allows for programming augmented entities that perceive the presence of the users, by exploiting the same mechanisms. This feature can be useful in particular to simplify the design of smart environments that react to events related to user state and behavior, reified in the corresponding augmented body.

3.4 Physical Embedding

The notion of augmentation in augmented worlds eventually means also the extension of the functionalities of existing physical objects (see Figure 2, right). Such an extension can be useful at two different (but related) levels.

A first one is for human users interacting with the physical objects, extending the object affordances by means of virtual interfaces to control or inspect the state of the physical object.

A second one is for enriching the physical object functionalities, exploiting the computing capabilities given by the augmented layer (either embedded or not in the object itself). For instance, an alarm clock on the bedside table – exploiting the associated augmented entity – can provide to a user the functionality to seamless schedule wake-up times, considering morning appointments on calendar or – in the short future – considering the monitored sleep status (by another augmented device) and adjusting wake-up timing consequently.

This kind of augmentation requires in general some kind of *coupling* between the physical objects and the corresponding augmented entities providing their extension, such that changes to the physical state of the objects are tracked and reified in the augmented level. In other words, the augmented level always needs to be informed about changes into the physical world. When this is not possible, due to e.g. a lack of connectivity, the augmented world's infrastructure has to update a kind of meta-information for each information/data related to physical world (like a "degree of freshness") to infer if and when a specific information/data could be no longer aligned to the real state.

4. TOWARDS A PROGRAMMING MODEL

After sketching the main characteristics of augmented worlds, we consider now the issue of how to implement them, what kind of programming abstractions can be adopted.

OOP provides an effective approach to model the basic structural aspects of augmented worlds. Augmented entities can be directly mapped into simple objects – encapsulating augmented entity state and behaviour – possibly exploiting interfaces and classes to define the type and hierarchies of augmented entities. An augmented object would have basic operations to manage aspects such as the position inside the augmented world. An augmented world in this case can be modelled as the container of the augmented objects, providing services for their management. The observer pattern could be used in this case to model the observability features discussed in previous section.

This OOP modeling however is not effective to capture concurrency and asynchronous interactions, which are main fundamental aspects of this kind of systems. Augmented worlds are inherently concurrent systems. From a control point of view, augmented entities are independent computing entities, whose computations can be triggered asynchronously by virtue of different kind of events—user(s) actions and other entities requests. At the logical level, they are distributed, given their spatial coupling.

These aspects can effectively captured by concurrency model integrating objects with concurrency, such actors and actor-like entities such as active/concurrent objects. This choice allows for strengthening the level of encapsulation, devising augmented worlds as collections of *autonomous* entities encapsulating a state, a behaviour and the control of the behaviour. Modelling augmented entities as actors means using direct asynchronous message passing to model every kind of interaction occurring inside an augmented world. In this case, the observation-related features can be implemented as a framework on top, by means of, e.g., pre-defined actors providing functionalities related to lookup/discovery and spatial-driven observation. Alternatively, these features

can be embedded in the actor framework/language, extending the basic model.

The observation-related features can be directly captured instead by adopting an *agent-oriented* modelling. In that case, an augmented world can be modeled in terms of autonomous agents situated into a virtual environment making the bridge between the physical and augmented layer. The augmented entities would be the basic bricks composing such environment, which can be observed and manipulated by the agents by means of the environment interface (i.e. the set of actions/percepts).

In next section we go deeper in this modeling, by discussing a first example of agent-oriented augmented world programming based on *mirror worlds*.

5. THE CASE OF MIRROR WORLDS

Mirror words (MW) have been introduced in [20] as an agent-oriented approach to conceive the design of future smart environments, integrating in the same unifying model perspectives and visions from Distributed Systems [9], Ambient Intelligence (AmI) and Augmented Reality. In the MW vision, smart spaces are modelled as digital cities shaped in terms of the physical world to which they are coupled, inhabited by open societies and organisations of *software agents* playing the role of the inhabitants of those cities. *Mirroring* is given by the fact that physical things, which can be perceived and acted upon by humans in the physical world, have a digital counterpart (or augmentation, extension) in the mirror, so that they can be observed and acted upon by agents. Viceversa, an entity in the MW that can be perceived and acted upon by software agents, may have a physical appearance (or extension) in the physical world—e.g., augmenting it, in terms of AR, so that it can be observed and acted upon by humans.

Mirror worlds are based on the A&A (Agents and Artifacts) [14] meta-model, introduced in agent-oriented software engineering to exploit also the agent environment as a first-order abstraction aside to agents to model and design multi-agent systems. In particular, A&A introduces *artifacts* as first-class abstractions to model and design the application environments where agents are logically situated. An artifact can be used to model and design any kind of (non-autonomous) resources and tools used and possibly shared by agents to do their job [21]. Artifacts have an *observable state* that agents can perceive and a set of *operations*, that agents can request as *actions* to affect the world. The observable state is represented by a set of *observable properties*, whose value can change dynamically depending on the actions executed on the artifact. Like objects in OOP, artifacts can have also a hidden state, not accessible to agents. The concept of observable state is provided to support event-driven forms of interaction between agents and artifacts, so that an agent observing an artifact is asynchronously notified with an event each time the state of the artifact is updated. Artifacts are collected in *workspaces*, which represent logical containers defining the topology of the multi-agent system, which may be distributed over the network.

The interactions among agents and artifacts are fully asynchronous. Actions on artifacts executed by agents – which encapsulate their own logical thread of control, like actors – are executed by separate threads of control with respect to the agent one. Operations are executed *transactionally*, so the execution of multiple actions concurrently on the same artifact is safe [21].

5.1 Mirror Worlds as Augmented Worlds

Given the augment worlds conceptual framework introduced in this paper, a mirror world can be described as an agent-oriented augmented world, implementing some of the principles that have been discussed in Section 3.

Artifacts and workspaces are used in mirror worlds to model the bridge between the augmented world layer and the physical reality layer. In particular, a MW is modelled in term of a set of *mirror workspaces*, which extend the notion of workspace defined in A&A with a *map* specifying which part of the physical world is coupled by the MW. It could be a part a city, a building, a room. The map defines a local system of reference to locate *mirror artifacts* inside. Mirror artifacts are simply artifacts anchored to some specific location inside the physical world, as defined by the map. Such location could be either a geo-location, or some trackable physical marker/object. Thus, mirror artifacts realize the spatial coupling defined in Section 3.

About observability, in a MW an agent can perceive and observe a mirror artifact in two basic ways. One is exactly the same as for normal artifacts, that is explicitly *focusing* on the artifact, given its identifier [21]—focusing is like a subscribe action in publish/subscribe systems. The second one instead is peculiar to mirror workspace and is the core feature of agents living in mirror workspaces, that is: perceiving an artifact depending on its position, without the need of explicit subscription (focusing). To that purpose, an agent joining a mirror workspace can create a *body* artifact, which is a built-in mirror artifact useful to situate the agent in a specific location of the workspace. We call *mirror agent* an agent with a body in a mirror workspace. Observability is ruled by two parameters – as defined in Section 3 – the observability radius and the observation radius. The observability radius is defined for each mirror artifact and defines the maximum distance within which an agent (body) must be located in order to perceive the artifact. The observation radius instead is defined for each agent body and defines the maximum distance within which a mirror artifact must be located in order to be perceived by an agent. Thus, a mirror artifact X located at the position X_{pos}, with an observability radius X_r is observable by a mirror agent with a body B, located in B_{pos}, with an observation radius B_R, iff $d <= X_r$ and $d <= B_R$, being d the distance between X_{pos} and B_{pos}. Both parameters can be changed at runtime.

The user interface in MW is currently realized by user assistant mirror agents with a body coupled to the physical location of the human user, by means of a smart device—glass, phone, whatever. Such agents perceive mirror artifacts in the nearby of the user location, their observable state, and can select and represent them on the smart-glass worn by the user. The representation can range from simple messages and cues to full-fledged augmented reality rendering, eventually superimposing virtual 3D objects on the image

of the physical reality.

About the physical embedding discussed in Section 3, mirror artifacts can be either completely virtual, or coupled to some object of the physical reality. In the first case, the geo-position inside the mirror (and the physical environment) is specified when instantiating the artifact, and it can be updated then by operations provided by the artifact. In the second case, at the infrastructure level, the state and position of the mirror artifact is kept *synchronized* to the state and position of the physical object by the MW runtime, by means of suitable sensors/embedded devices.

5.2 Programming Mirror Worlds

A first implementation of Mirror Worlds has been developed on top of the JaCaMo agent framework [19], which directly supports the A&A meta-model. Agents are programmed using the Jason agent programming language [4], which is a practical extension and implementation of AgentSpeak(L) [17]; Artifact-based environments are programmed using the CArtAgO framework [21], which provides a Java API for that purpose.

The MW API are currently a layer on top of JaCaMo, including:

- `MirrorArtifact` artifact template, extending the `Artifact` CArtAgO base class and representing the basic template to be further extended and specialized to implement specific kinds of mirror artifacts. The usage interface of this artifact includes:
  - a `pos` observable property, containing the current location in the mirror workspace of the artifact;
  - `observabilityRadius` observable property, storing the current observability radius of the artifact;
  - specific operations (`setPos`, `setObservability Radius`) for updating the position and the observability radius.

  The usage interface of agent bodies includes also an `observationRadius` observable property, storing the current observation radius of the agent, and the related operation `setObservationRadius` for updating such radius.
- a new set of actions to be used by agents for creating mirror workspaces and mirror artifacts inside, including agent bodies.
- some *utility* artifacts are available providing functionalities useful for agents working in the mirror. An example is given by the `GeoTool`, which provides functionalities for converting the coordinates and computing distances.

In the remainder of the section we give an overview of MW programming in JaCaMo by considering a simple example of mirror world, a kind of *hello, world*. An overview of the main elements of the example is shown in Figure 3. The mirror world is composed by a single mirror workspace (called `mirror-example`) mapped onto a city zone in the

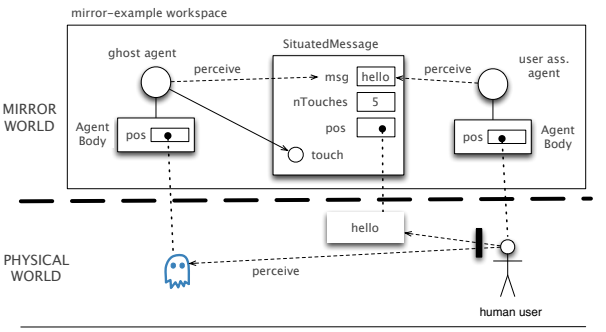


Figure 3: Main elements included in the example discussed in Section 5: a mobile user walking along the streets and the corresponding user assistant agent, with a body located at the position detected by the GPS. A situated message, modelled as a `SituatedMessage` mirror artifact. A ghost, as a mirror agent autonomously walking through the streets.

center of a city (Cesena, in this case, in Italy). The mirror workspace is dynamically populated of mirror artifacts representing messages situated in some specific point of the city (template `SituatedMessage`). Besides keeping track of a message, these artifacts embed a counter and a `touch` operation to increment it. Mobile human users walk around the streets along with their user assistant agents, running on their mobile device (e.g, the smartphone or directly the smart glasses). As soon as user assistant agents perceive a situated message, they display it on the smart glasses worn by the users. In the MW there are also some `ghost` mirror agents that are moving around autonomously along some streets of the city, perceiving and interacting with the situated messages as well. They react to situated messages perceived while walking, eventually executing a `touch` action on the mirror artifact encountered. Besides, if/when a ghost agent reaches a human user, it *hugs* her/him, which is physically perceived by a trembling of the smartphone. This occurs by executing a `tremble` action on the artifact modeling the user mobile device.

The example includes also a *control room* (see Figure 4), which is a remote desktop application which allows to track the real-time state of the running mirror world, showing the position of mirror agents (red circles) – actually the body of mirror agents – and the position of mirror artifacts, i.e. the situated messages in the example.

This simple example contains most of the main ingredients of an augmented/mirror world. The situated messages represent stateful augmented entities, with a simple behaviour; Such augmented entities are shared, perceived and manipulated by both the human users (indirectly through the user assistant agents) and the other autonomous agents living in the mirror—i.e., the ghosts. Through the mirror, such agents can have an effect also on the physical world (trembling of the smartphones). An aspect which is quite overlooked in the example is the visual representation of augmented objects, which is currently fairly simple (just messages)—more sophisticated AR-based views are planned

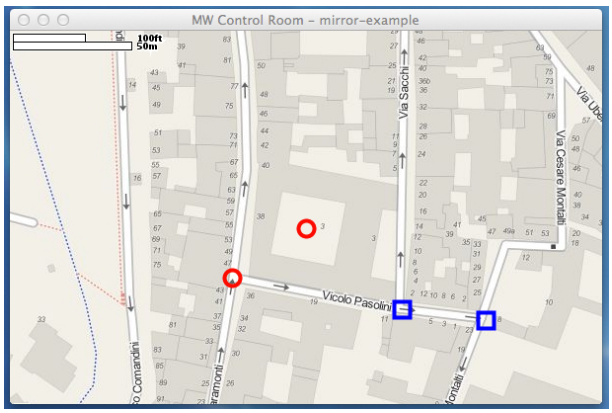


Figure 4: The map visualised by the control room, showing the position of mirror agents (red circles) – that is, the body of mirror agents – and the position of mirror artifacts, i.e. situated messages in the example.

in the future.

In order to have a concrete taste of MW programming, in the following we show some details about how the mirror agents and artifacts are programmed—the full code is available in [15], along with the experimental JaCaMo distribution supporting mirror worlds.

5.2.1 *Defining Mirror Artifacts*

The situated messages of the example are implemented by the the `SituatedMessage` mirror artifact—Figure 5 shows the implementation of the Java class representing the artifact template. Artifacts in CArtAgO can be defined as classes extending the `Artifact` base class. Methods annotated with `@OPERATION` define the operations available to agents. Observable properties are managed by means of predefined primitives (e.g. `defineObsProperty`, `getObsProperty`, ...)—implemented as protected methods of the base class. Operation execution is atomic, so – similarly to monitors – only one operation at a time can be running inside an artifact. Changes to the observable properties are made observable to agents only when an operation has completed. Further details about the artifact model are described in [21].

A mirror artifact can be defined by extending the base `MirrorArtifact` class—which is an extension itself of the CArtAgO `Artifact` class. `SituatedMessage` has two observable properties (besides the ones inherited by `MirrorArtifact`), `msg` and `nTouches`, storing the content of the message and the counter keeping track of the number of times that the message has been *touched*. The `touch` operation allows to increment the counter.

5.2.2 *Implementing Agents in the Mirror*

Agents in the mirror are normal Jason agents, with more actions available—given by the new artifacts introduced by the MW framework. In particular, specific actions are available for creating mirror workspaces and instantiating mirror artifacts. As an example, Figure 6 shows the code of

```
1  public class SituatedMessage extends MirrorArtifact {
2
3      public void init(String msg){
4          super.init(msg);
5          defineObsProperty("msg",msg);
6          defineObsProperty("nTouches",0);
7      }
8
9      @OPERATION void touch(){
10         updateObsProperty("nTouches",
11               getObsProperty("nTouches").intValue()+1);
12     }
13 }
```

Figure 5: Source Code of the mirror artifact representing a situated message.

```
1  /* initial beliefs */
2
3  /* the center of the mirror -- latitude/longitude */
4  poi("isi_cortile", 44.13983, 12.24289).
5
6  /* the point of interests, where to put the messages */
7  poi("pasolini_montalti",44.13948, 12.24384).
8  poi("sacchi_pasolini",44.13952, 12.24340).
9
10 /* initial goal*/
11 !setupMW.
12
13 /* the plans */
14
15 +!setupMW
16   <- ?poi("isi_cortile",Lat,Long);
17      createMirrorWorkspace("mirror-example",Lat,Long);
18      joinWorkspace("mirror-example");
19      /* create an aux artifact to help coordinate conversion */
20      makeArtifact("geotool","GeoTool",[Lat,Long]);
21      /* create the situated messages */
22      !create_messages;
23      println("MW ready.").
24
25 /* to create the situated message mirror artifacts */
26 +!create_messages
27   <- ?poi("pasolini_montalti",Lat,Lon);
28      toCityPoint(Lat,Lon,Loc);
29      createMirrorArtifactAtPos("a1","SituatedMessage",
30          ["hello #1"],Loc,2.5);
31      ?poi("sacchi_pasolini",Lat2,Lon2);
32      toCityPoint(Lat2,Lon2,Loc2);
33      createMirrorArtifactAtPos("a2","SituatedMessage",
34          ["hello #2"],Loc2,2.5).
```

Figure 6: Code of the `majordomo` agent.

a *majordomo* agent, whose task is to setup the initial environment of the MW of our example. The agent creates a mirror workspace, called `mirror-example` (line 17), and a couple of `SituatedMessage` mirror artifacts (plan at lines 26-34), located at two POIs tagged as `pasolini_montalti` and `sacchi_pasolini` (which are two street intersections on the map). The action `createMirrorArtifactAtPos` makes it possible to instantiate a new mirror artifact, specifying its logical name, the template (the Java class name), its location `pos` and the observability radius (in meter). The utility artifact (`GeoTool` template) used by the agent provides functionalities to manage geographical coordinates, in particular to convert global latitude/longitude coordinates into the local one of the mirror workspace (`toCityPoint` operation).

The first example of mirror agent is given by *user assistant agents*, whose code is shown in Figure 7. The agent first cre-

ates (in its default/local workspace) a `SmartGlassDevice` artifact (line 8), to be used as output device to display messages, by means of the `displayMsg` operation. Then, the agent joins the mirror workspace and creates its body, with observation radius of 10 meters—to this purpose the `createAgentBody` action is used, specifying the observing and observability radii (line 14). The body is bound to a `GPSDeviceDriver` device driver artifact (line 18), previously created (line 16). The device driver implements the coupling between the position detected by the GPS sensor, available on the smartphone of the user. When the human user approaches a point in the physical world where a situated message is located, the user assistant agent perceives the message and reacts by simply displaying it on the glasses (lines 23-25). When (if) the human user moves away from the mirror artifact, the belief about the message is removed and the use assistant agent reacts by displaying a further message (lines 27-29).

Figure 8 shows the code of the ghost mirror agents, autonomously walking through some streets of the mirror world. They have a `walk_around` goal (line 8), and the plan for that goal (line 12) consists in repeatedly doing the same path, whose nodes (a list of point-of-interests) is stored in the `path` belief (line 5). They move by changing the position of their body, by executing a `moveTowards` action available in each mirror artifact—specifying the target point (to define the direction) and the distance to be covered (in meters). The plan for reaching an individual destination of the path (lines 23-29) simply computes the distance from the target (exploiting the `computeDistanceFrom`, provided by the `GeoTool` artifact) and then, if the distance is greater than one meter, it moves the body of 0.5 meter and then goes on reaching, by requesting recursively the subgoal `!reach_dest`; otherwise it completes the plan (the destination has been reached). Ghosts too react to messages perceived while walking (plan at lines 38-41), eventually executing a `touch` action on each message encountered and printing to the console the current number of touches observed on the message. Instead, when a ghost perceives a human (lines 43-46) – by perceiving the body of the user assistant agent – it reacts by making a trembling on the smartphone owned by the human user. `body` is an observable property provided by each agent body artifact, containing the identifier of the user assistant agent which created the body. Trembling happens by executing a `tremble` action on the artifact which the user assistant agent created to enable the physical interaction with the corresponding human user. By convention, in the example, these artifacts are created with the name `user-dev-X`, where `X` is name of the user assistant agent. This convention allows the ghost agent to retrieve the identifier of the artifact dynamically given its logic name, by doing a lookup (line 45).

5.3 Remarks

The example, in spite of its simplicity and of the details about Jason/CArtAgO programming, should provide a first idea about the level of abstraction provided by agent-oriented programming for designing and programming augmented worlds. The main strength is that it allows to model the augmented world in a way which is similar to the real world—even more similar in our opinion than the modeling provided by paradigms such as actors or concurrent objects.

```
1  /* User assistant agent */
2
3  /* goal of the agent */
4  !monitor_and_display_messages.
5
6  +!monitor_and_display_messages
7    <- /* setup the smart glass device */
8       makeArtifact("viewer","SmartGlassDevice",[],Viewer);
9       /* keep track of the device id with a viewer belief */
10      +viewer(Viewer);
11      /* join the mirror workspace */
12      joinWorkspace("mirror-example");
13      /* create the agent body */
14      createAgentBody(1000,10,Body);
15      /* create the artifact used as MW coupling device */
16      makeArtifact("driver","GPSDeviceDriver",Dev);
17      /* bind the body to the device */
18      bindTo(Body)[artifact_id(Dev)];
19      println("ready.").
20
21 /* plans reacting to situated messages perceived in the mirror worlds */
22
23 +msg(M) : viewer(Dev)
24   <- .concat("new message perceived: ",M,Msg);
25      displayMsg(100,50,Msg)[artifact_id(Dev)].
26
27 -msg(M) : viewer(Dev)
28   <- .concat("message ",M," no more perceived. ",Msg);
29      displayMsg(100,50,Msg)[artifact_id(Dev)].
```

Figure 7: Code of the user-assistant agents.

```
1  /* ghost agent initial beliefs */
2
3  start_pos("pasolini_chiaramonti").
4  /* path of the walk - 2 steps*/
5  path(["sacchi_pasolini","pasolini_montalti"]).
6
7  /* initial goal */
8  !walk_around.
9
10 /* plans */
11
12 +!walk_around <- !setup; !moving.
13
14 +!moving <- ?path(P); !make_a_trip(P); !moving.
15
16 +!make_a_trip([POI|Rest])
17   <- ?poi(POI,Lat,Lon);
18      !reach_dest(Lat,Lon);
19      !make_a_trip(Rest).
20 +!make_a_trip([])
21   <- ?start_pos(Start); ?poi(Start,Lat,Lon); !reach_dest(Lat,Lon).
22
23 +!reach_dest(Lat,Lon) : myBody(B)
24   <- toCityPoint(Lat,Lon,Target);
25      computeDistanceFrom(Target,Dist)[artifact_id(B)];
26      if (Dist > 1){
27        moveTowards(Target,0.5)[artifact_id(B)];
28        .wait(50);
29        !reach_dest(Lat,Lon)}.
30
31 +!setup
32   <- joinWorkspace("mirror-example",Mirror);
33      lookupArtifact("geotool",Tool); focus(Tool);
34      ?start_pos(Point); ?poi(Point,Lat,Lon); toCityPoint(Lat,Lon,P);
35      createAgentBodyAtPos(P,1000,10,Body);
36      +myBody(Body); .my_name(Me); +me(Me).
37
38 +msg(M) [artifact_id(Id)]
39   <- touch [artifact_id(Id)];
40      ?nTouches(C)[artifact_id(Id)];
41      println("new message perceived: ",M," - touch count: ",C).
42
43 +body(Who) : me(Me) & Who \== Me
44   <- .concat("user-dev-",Who,Dev);
45      lookupArtifact(Dev,DevId);
46      tremble [artifact_id(DevId)].
```

Figure 8: Code of ghost agents.

In particular, both in the real world and the augmented worlds, a main role is played by indirect interactions (vs. direct message passing) based on the (asynchronous) observation of events occurring in the environment. This is directly captured by the agent/environment abstractions.

6. CHALLENGES AND FUTURE WORK

The development of augmented worlds relies on the availability of enabling technologies that deal with issues and challenges of different kinds. An example is given by tracking and registration, which are a main concern of the AR and MAR layer [3, 5].

Besides the challenges in the enabling layers, there are further issues that specifically concern the augmented worlds model.

A main general one is related to the level of real-time coupling and synchronization between the computational augmented layer and the physical layer. This coupling/synchronization is critical from users' perspective, since it impacts on what users perceive of an augmented world, and then how they reason about it and act consequentially. Being a multi-user system, two users must perceive the same observable state of the shared augmented entities. If a part of the augmented world is temporarily disconnected – because of, e.g., some network transient failure – users must be able to realize this.

These aspects are challenging in particular because – like in distributed systems in general – it is not feasible in an augmented world to assume a single clock defining a centralized notion of time. Conversely, it is fair to assume that each augmented entity has its own local clock and the events generated inside an augmented world can be only *partially* ordered. In spite of the distribution, causal consistency must be guaranteed, in particular related to chains of events that span from the physical to the digital layer and viceversa. That is, if an augmented entity produces a *sequence* of two events concerning the change of its observable state, the same sequence must be observed by different human users immersed in the augmented world.

Similar challenges are found in online multi-user distributed collaborative systems. As a main example, Croquet is based on TeaTime [25], a scalable real-time multi-user architecture which manages the communication among objects, and their synchronization. The spatial coupling and physical embedding properties of augmented worlds introduce further elements and complexities, that are not fully captured by strategies adopted in purely virtual systems.

Finally, the aim of this paper was to introduce the vision about augmented worlds, along with a first conceptual framework discussing some main features of their programming abstractions. Clearly, a more rigorous and comprehensive approach is needed to tackle the development and engineering of non-naive augmented worlds, and, more generally, to achieve a deeper understanding of the *computation-as-augmentation* view. This understanding includes, e.g., investigating if and how spatial coupling impacts on system modularity, compositionality, extensibility. To this purpose, the definition of formal models capturing the main aspects and properties of this kind of programs appears an important future work. Another important investigation concerns the design of proper tools supporting the development/debugging/profiling of augmented worlds. These tools must provide specific features to deal with the characteristics described in Section 3. To this purpose, the design of proper real-time *simulators* – allowing to run an augmented world like, e.g., a first-person perspective video-game – appears an interesting solution to explore.

7. CONCLUSION

The fruitful integration of enabling technologies concerning augmented reality, mobile/wearable computing and pervasive computing makes it possible to envision a new generation of software systems in which computation and programming can be exploited to shape various forms of augmentation of the physical reality.

Augmented worlds – introduced in this paper – are programs that are meant to extend the physical world by means of full-fledge computational entities logically situated in some physical location, possibly enriching the functionalities of existing physical objects. Mirror words [20, 19] provide a concrete agent-oriented programming model for building augmented worlds—based on the A&A conceptual model and the JaCaMo platform.

The main contribution of this work is to lay down the first basic bricks about the augmented worlds vision, and to trigger further research and practical investigations, including the engineering of real-world and robust applications based on these ideas.

8. REFERENCES

[1] Microsoft HoloLens, Official web site. https://www.microsoft.com/microsoft-hololens.

[2] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre. Recent advances in augmented reality. *Computer Graphics and Applications, IEEE*, 21(6):34–47, 2001.

[3] R. T. Azuma et al. A survey of augmented reality. *Presence*, 6(4):355–385, 1997.

[4] R. H. Bordini, J. F. Hübner, and M. Wooldrige. *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. John Wiley & Sons, 2007.

[5] E. Costanza, A. Kunz, and M. Fjeld. Human machine interaction. chapter Mixed Reality: A Survey, pages 47–68. Springer-Verlag, Berlin, Heidelberg, 2009.

[6] K. Curran, D. McFadden, and R. Devlin. The role of augmented reality within ambient intelligence. *Int. Journal of Ambient Computing and Intelligence*, 3(2):16–33, 2011.

[7] A. K. Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7, 2001.

[8] D. C. Engelbart and W. K. English. A research center for augmenting human intellect. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 395–410, New York, NY, USA, 1968. ACM.

[9] D. H. Gelernter. *Mirror Worlds: or the Day Software Puts the Universe in a Shoebox...How It Will Happen*

and What It Will Mean. Oxford, 1992.

[10] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Gener. Comput. Syst.*, 29(7):1645–1660, 2013.

[11] S. Kurkovsky. Pervasive computing: Past, present and future. *5th IEEE International Conference on Information and Communications Technology (ICICT)*, 2007.

[12] T. Langlotz, T. Nguyen, D. Schmalstieg, and R. Grasset. Next-generation augmented reality browsers: Rich, seamless, and adaptive. *Proceedings of the IEEE*, 102(2):155–169, Feb 2014.

[13] P. Milgram and F. Kishino. A taxonomy of mixed reality visual displays. *IEICE Trans. Information Systems*, E77-D(12):1321–1329, Dec. 1994.

[14] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, 2008.

[15] Pervasive Software Lab – DISI, University of Bologna. JacaMo-MW – mirror worlds in JaCaMo. https://bitbucket.org/pslabteam/mirrorworlds, 2015.

[16] R. Poovendran. Cyber-physical systems: close encounters between two parallel worlds. *Proceedings of the IEEE*, 98(8), 2010.

[17] A. S. Rao. Agentspeak (l): Bdi agents speak out in a logical computable language. In *Agents Breaking Away*, pages 42–55. Springer, 1996.

[18] S. Reeves. Envisioning ubiquitous computing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 1573–1582. ACM, 2012.

[19] A. Ricci, A. Croatti, P. Brunetti, and M. Viroli. Programming Mirror-Worlds: an Agent-Oriented Programming Perspective. In *Engineering Multi-Agent Systems Third International Workshop, EMAS 2015, Revised Selected Papers*, LNCS. Springer, 2015. To Appear.

[20] A. Ricci, M. Piunti, L. Tummolini, and C. Castelfranchi. The mirror world: Preparing for mixed-reality living. *IEEE Pervasive Computing*, 14(2):60–63, 2015.

[21] A. Ricci, M. Piunti, and M. Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, Sept. 2011.

[22] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8:10–17, 2001.

[23] D. Schmalstieg, T. Langlotz, and M. Billinghurst. Augmented reality 2.0. In G. Brunnett, S. Coquillart, and G. Welch, editors, *Virtual Realities*, pages 13–37. Springer Vienna, 2011.

[24] D. Schmalstieg and G. Reitmayr. The world as a user interface: Augmented reality for ubiquitous computing. In G. Gartner, W. Cartwright, and M. Peterson, editors, *Location Based Services and TeleCartography*, Lecture Notes in Geoinformation and Cartography, pages 369–391. Springer Berlin Heidelberg, 2007.

[25] D. A. Smith, A. Kay, A. Raab, and D. P. Reed. Croquet-a collaboration system architecture. In *Creating, Connecting and Collaborating Through Computing, 2003. C5 2003. Proceedings. First Conference on*, pages 2–9. IEEE, 2003.

[26] T. Starner. Project Glass: An Extension of the Self. *Pervasive Computing, IEEE*, 12(2):14–16, April 2013.

[27] B. H. Thomas. A survey of visual, mixed, and augmented reality gaming. *Comput. Entertain.*, 10(3):3:1–3:33, Dec. 2012.

[28] J. Tiffin and N. Terashima. *HyperReality: Paradigm for the Third Millenium*. Routledge, 2001.

[29] M. Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, July 1999.

[30] P. Wellner, W. Mackay, and R. Gold. Computer-augmented environments: back to the real world. *Communications of the ACM*, 36(7), 1993.

[31] C. Xia and P. Maes. The design of artifacts for augmenting intellect. In *Proceedings of the 4th Augmented Human International Conference*, pages 154–161. ACM, 2013.

[32] F. Zambonelli and M. Mamei. Spatial computing: An emerging paradigm for autonomic computing and communication. In M. Smirnov, editor, *Autonomic Communication*, volume 3457 of *Lecture Notes in Computer Science*, pages 44–57. Springer Berlin Heidelberg, 2005.

[33] D. Zhang, L. T. Yang, and H. Huang. Searching in internet of things: Vision and challenges. *in Proc. IEEE 9th Int. Symp. Parallel Distrib. Process. Appl. (ISPA)*, 2011.

A Model-based Approach to Secure Multi-party Distributed Systems

Najah Ben Said, Saddek Bensalem, Marius Bozga
Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
CNRS, VERIMAG, F-38000 Grenoble, France
FirstName.LastName@imag.fr

Takoua Abdellatif
University of Carthage, Tunis
Tunisia Polytechnic School, Tunisia
takoua_abdellatif@yahoo.fr

ABSTRACT

Securing multi-party distributed systems is still a challenge. In such distributed systems with completely distributed interactions between parties with mutual distrust, it is hard to control the illicit flowing of private information to unintended parties. Unlike some existing solutions dealing with verification of low-level cryptographic protocol in multi-party interactions, we propose a novel approach based on model transformations to build secure-by-construction multi-party distributed systems. The user has to describe his system in a component-based model and annotate it to define the system security policy. Then, the system is checked and when valid, a secure code, consistent with the desired security policy, is automatically generated. To validate the approach, we present a framework that implements our method and we use it to secure an online social network application.

Keywords

component-based systems, distributed systems, model transformation, information flow security

1. INTRODUCTION

Multi-party distributed systems involve different parties and interactions with generally mutual distrust and unsecure underlying communication channels. In such systems, it is still challenging to protect people privacy and make sure that classified information are only disclosed to intended parties. The typical example is the online social network (OSN) where members within groups exchange information and events and do not necessarily control how their data are disseminated within OSN users and storage. Furthermore, it is difficult to control when some public information implicitly reveal some secret one (for instance, a travel ticket price can disclose the travel destination). Indeed, many studies on OSN such as Facebook show that the security configuration settings fall short to ensure intended policies. Another example of multi-party application is the Health-Service-App, used to calculate patients profiles and make statistics in collaboration with a set of distributed hospital applications. It is obvious that some parties are not allowed to access all health private details of patients. Considering an other example of data-mining, privacy preservation is rather treated by a cryptograph-based research known as multi-party secure computation [12]. In this field, researchers adopt formal frameworks for verifying low-level adopted cryptographic protocols.

In this work, we are interested in verifying multi-party security systems at an abstract level model before even defining protocols used in the system. The advantage of such approach is that the system designer can describe separately his system component behavior and interactions at a high-level and then configures security in an intuitive way. Then, the designer automatically checks the intended security policy without worrying about the burden of the application protocol details. Hence, we guarantee that security specifications, that can be modified within the system life cycle, are treated independently and in parallel with functional specifications. For security checking, we adopt the non-interference property [11] to enforce privacy and make sure that information are not leaking in an explicit or implicit way.

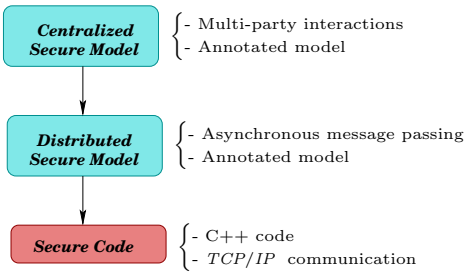


Figure 1: Model Transformation

In this paper, we present a practical automated method to build secure-by-construction distributed systems using a model-based approach. Starting from a multi-party centralized model, the system is described as a set of components and interactions. Annotations at the level of variables and interfaces of components and interactions allow configuring the system security policies. The centralized model is transformed to a decentralized send/receive (S/R) model that is, scheduling and communication protocol components are inserted to implement interactions and set-up distributed communication protocols. Security annotations are propagated to the new distributed component-based architecture following a set of rules to preserve the non-interference property. In the last step, the S/R model is used to produce the distributed code. The model transformations are illustrated in Figure 1. To the best of our knowledge, this is the first approach to securely decentralize high-level component-based systems with multiparty interactions. We proved that, whenever the input model is secure, that is, satisfies conditions for event and data non-interference of [5] , the decen-

tralized model is also secure. The first transformation has been designed such that to preserve, by construction, the non-interference property. For the code generation, the implementation is directly derived from the S/R model using secure communication primitives. Our contribution can be summarized in the following points:

- We define an automated method based on model transformations to build a secure-by-construction multi- party distributed system; the user has only to design his system in a component-based model with multiparty interactions.
- We provide formal definitions of non-interference for component-based models and we correctness proofs of different transformation steps by showing the preservation of non-interference property. We define two kinds of non-interference: event and data non-interference for a more rigorous and fine-grained verification in distributed systems.
- We present a framework that implements our method and use it to secure an OSN application called *Whens-App* for event organization. This application is general enough to cover many multi-party applications with different security configurations.

The paper is structured as follows. Section 2 introduces the *Whens-App* application and motivates the need for our work and approach. Section 3 presents the main concepts of the component-based framework adopted in this work as well as non-interference definitions and sufficient conditions. In section 4, we describe the automated distribution approach to derive secure executable code. Section 5 presents the implementation Tool-set and evaluation section 6 discusses the related work. The section 7 concludes and presents some perspectives for future work. All proofs of technical results are given in a technical report[1].

2. CASE STUDY

Throughout the paper, we consider *Whens-App*, an OSN application for organizing events, such as business meeting where participants can exchange data when these meetings take place. Figure 2 shows an overview of a fragment of the system which consists of a finite number of *Event-Creators*, each communicating with a set of *Event-Receivers*. Communication channels are represented by lines in the figure.

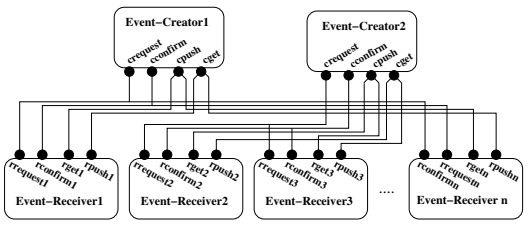


Figure 2: High-level description of the Whens-App system.

As social network application, *Whens-App*, entails a large variety of security requirements, in this paper however, we focus on some relevant requirements related to information flow security: Assuming that components are trustful and the network is unsecure, *(1)* the interception and observation of exchanged data messages does not reveal any information about event participants and *(2)* confidentiality of classified data as well as events is always preserved and kept secret inter- as well as intra-components. Both requirements are ensured by using security annotations model for tracking events and data in the system and checking that the formal model satisfies the security constraints given in Section 3.3. We additionally enforce privacy of participants at implementation by hiding the identity of components participating in a secret interaction.

[1]http://www-verimag.imag.fr/Technical-Reports,264.html?lang=en&-number=TR-2014-6

3. SECURE COMPONENT-BASED MODEL

The centralized secure model [5] presents a component oriented model well adapted in describing complex systems like heterogeneous and distributed ones. It has been carefully made to achieve high expressivity for component composition, while maintaining separation of concerns between components behavior and their coordination. Thanks to its modularity, it offers a flexible way to develop and manage complex systems. Particularly, for information flow security, the explicit system architecture it provides allows tracking easily intra and inter-components information flow. Furthermore, the centralized model allows the development of scalable validation and verification methods and tools, exploiting compositionality principles. We recall hereafter the main concepts behind component model with a particular focus on security annotations and the different notions of non-interference and their verification.

3.1 Component-Based Model

The system functional model is expressed as a set of atomic components, that is, finite state automata or 1-safe Petri nets, extended with data. Communications inter-components are achieved using interactions that express synchronization constraints and do the transfer of data between the interacting components. In the following, we recall the key concepts of the used component-based model which are further relevant for dealing with information flow security. In particular, we give a formal definition of atomic components and their composition through multiparty interactions.

DEFINITION 1 (ATOMIC COMPONENT). *An atomic component B is a tuple (L, X, P, T) where L is a set of states, X is a set of variables, P is a set of ports and $T \subseteq L \times P \times L$ is a set of port labelled transitions. For every port $p \in P$, we denote by X_p the subset of variables exported and available for interaction through p. For every transition $\tau \in T$, we denote by g_τ its guard, that is, a Boolean expression defined on X and by f_τ its update function, that is, a parallel assignment $\{x := e^x_\tau\}_{x \in X}$ to variables of X.*

Let $\mathcal{D}$ be the data domain of variables. Given a set of variables Y, we call valuation on Y any function $\mathbf{y} : Y \to \mathcal{D}$ mapping variables to data. We denote by $\mathbf{Y}$ the set of all valuations defined on Y. The semantics of an atomic component $B = (L, X, P, T)$ is defined as the labelled transition system $\text{LTS}(B) = (Q_B, \Sigma_B, \underset{B}{\longrightarrow})$ where the set of states $Q_B = L \times \mathbf{X}$, the set of labels is $\Sigma_B = P \times \mathbf{X}$ and the set of

labelled transitions $\xrightarrow[B]{}$ is defined by the rule:

$$\textsc{Atom}\ \frac{\tau = \ell \xrightarrow{p} \ell' \in T \quad \mathbf{x}''_p \in \mathbf{X}_p \quad g_\tau(\mathbf{x}) \quad \mathbf{x}' = f_\tau(\mathbf{x}[X_p \leftarrow \mathbf{x}''_p])}{(\ell, \mathbf{x}) \xrightarrow[B]{p(\mathbf{x}''_p)} (\ell', \mathbf{x}')}$$

That is, $(\ell', \mathbf{x}')$ is a successor of $(\ell, \mathbf{x})$ labelled by $p(\mathbf{x}''_p)$ iff (1) $\tau = \ell \xrightarrow{p} \ell'$ is a transition of T, (2) the guard g_τ holds on the current valuation $\mathbf{x}$, (3) $\mathbf{x}''_p$ is a valuation of exported variables X_p and (4) $\mathbf{x}' = f_\tau(\mathbf{x}[X_p \leftarrow \mathbf{x}''_p])$ meaning that, the new valuation $\mathbf{x}'$ is obtained by applying f_τ on $\mathbf{x}$ previously modified according to $\mathbf{x}''_p$. Whenever a p-labelled successor exist in a state, we say that p is *enabled* in that state.

Example 1. *Figure 3 shows three atomic components, Event-Creator,* Event-Receiver1 *and Event-Receiver2. The Event-Creator component contains three control states l_1, l_2 and l_3 and a set of ports $\{crequest, cconfirm, cget, cpush, cancel\}$. Initially at the state l_1, the Event-Creator sends a request to both Event-Receivers by executing the transition labelled with crequest port. If the event participants send back a confirm and transition labelled by cconfirm is executed, both atomic components can exchange data variables notif and info through the Event-Creator, otherwise, the event creation is canceled. The dashed squares represent security annotations that will be presented in the coming sections.*

The system composition is obtained by binding atomic components $\{B_i = (L_i, X_i, P_i, T_i)\}_{i=1,n}$ trough specific composition operators. We consider that atomic components have pairwise disjoint sets of states, ports, and variables i.e., for any two $i \neq j$ from $\{1..n\}$, we have $L_i \cap L_j = \emptyset$, $P_i \cap P_j = \emptyset$, and $X_i \cap X_j = \emptyset$. We denote $P = \bigcup_{i=1}^{n} P_i$ the set of all the ports, $L = \bigcup_{i=1}^{n} L_i$ the set of all states, and $X = \bigcup_{i=1}^{n} X_i$ the set of all variables.

An *interaction* a between atomic components is a triple (P_a, G_a, F_a), where $P_a \subseteq P$ is a set of ports, G_a is a guard, and F_a is an update function. By definition, P_a uses at most one port of every component, that is, $|P_i \cap P_a| \leq 1$ for all $i \in \{1..n\}$. Therefore, we simply denote $P_a = \{p_i\}_{i \in I}$, where $I \subseteq \{1..n\}$ contains the indices of the components involved in a and for all $i \in I, p_i \in P_i$. G_a and F_a are both defined on the variables exported by the ports in P_a (i.e., $\bigcup_{p \in P_a} X_p$).

Definition 2 (composite component). *A composite component $C = \gamma(B_1, \ldots, B_n)$ is obtained by applying a set of interactions γ to a set of atomic components $B_1, \ldots B_n$.*

Let $B = \gamma(B_1, \ldots, B_n)$ be a composite component. Let $B_i = (L_i, X_i, P_i, T_i)$ and $\textsc{lts}(B_i) = (Q_i, \Sigma_i, \xrightarrow[B_i]{})$ their semantics, for all $i = 1, n$. The semantics of C is the labelled transition system $\textsc{lts}(C) = (Q_C, \Sigma_C, \xrightarrow[C]{})$ where the set of states $Q_C = \otimes_{i=1}^{n} Q_i$, the set of labels $\Sigma_C = \gamma$ and the set of labelled transitions $\xrightarrow[C]{}$ is defined by the rule:

$$\textsc{Comp}\ \frac{\begin{array}{c} a = (\{p_i\}_{i \in I}, G_a, F_a) \in \gamma \\ G_a(\{\mathbf{x}_{p_i}\}_{i \in I}) \quad \{\mathbf{x}''_{p_i}\}_{i \in I} = F_a(\{\mathbf{x}_{p_i}\}_{i \in I}) \\ \forall i \in I.\ (\ell_i, \mathbf{x}_i) \xrightarrow[B_i]{p_i(\mathbf{x}''_{p_i})} (\ell'_i, \mathbf{x}'_i) \qquad \forall i \notin I.\ (\ell_i, \mathbf{x}_i) = (\ell'_i, \mathbf{x}'_i) \end{array}}{((\ell_1, \mathbf{x}_1), \ldots, (\ell_n, \mathbf{x}_n)) \xrightarrow[C]{a} ((\ell'_1, \mathbf{x}'_1), \ldots, (\ell'_n, \mathbf{x}'_n))}$$

For each $i \in I$, $\mathbf{x}_{p_i}$ above denotes the valuation $\mathbf{x}_i$ restricted to variables of X_{p_i}. The rule expresses that a composite component $C = \gamma(B_1, \ldots, B_n)$ can execute an interaction $a \in \gamma$ *enabled* in state $((\ell_1, \mathbf{x}_1), \ldots, (\ell_n, \mathbf{x}_n))$, iff (1) for each $p_i \in P_a$, the corresponding atomic component B_i can execute a transition labelled by p_i, and (2) the guard G_a of the interaction holds on the current valuation of variables exported on ports participating in a. Execution of interaction a triggers first the update function F_a which modifies variables exported by ports $p_i \in P_a$. The new values obtained, encoded in the valuation $\mathbf{x}''_{p_i}$, are then used by the components' transitions. The states of components that do not participate in the interaction remain unchanged.

Any finite sequences of interactions $w = a_1 ... a_k \in \gamma^*$ executable by the composite component starting at some given initial state q_0 is named a trace. The set of all traces w from state q_0 is denoted by $\textsc{traces}(C, q_0)$.

Example 2. *Figure 3 presents a simplified composite component from the Whens-App application previously presented in Section 2. The composition represents an event creation between two Event-receiver components. Here, interactions are represented using connectors (lines) between the interacting ports. All interactions between components Event-Creator and Event-Receiver are strong synchronized binary interactions. The interactions {get,push} implements a data transfer between both Event-receivers , that is, an assignments at exportation between variables "info" and "notif".*

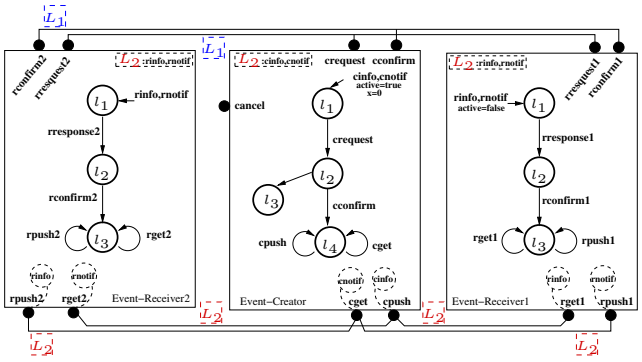


Figure 3: Composite component

3.2 Information Flow Security

We consider information flow policies [8, 4, 11] with focus on the non-interference property. In order to track information we adopt the classification technique and we define a classification policy where we annotate the information by assigning security levels to different parts of the centralized component model (data variables, ports and interactions). The policy describes how information can flow from one classification with respect to the other.

The system parameters are annotated using the *Decentralized Label Model* (DLM) introduced in [13]. In DLM, labels are defined as pair of confidentiality and integrity policies denoted $\{c; d\}$, where c is the confidentiality policy and d is the integrity policy. In the rest of the paper, in order to simplify the notations and since integrity is treated dually, we concentrate only on confidentiality. The main entity used to express policies is the principal. A principal is a atomic

component that has the power to observe and change certain aspects of the system. Principals are ordered using the *can-acts-for* relation ($\preceq$), which is a delegation mechanism that enables a principal to pass some of his rights to another principal.

A confidentiality label L contains an owner set, denoted $O(L)$, that are principals whose data was observed in order to construct the data value; they are the original sources of the information. Label L also contains for each owner $o \in O(L)$ a set of readers, denoted $R(L,o)$, representing principals to whom the owner o is willing to release the information value. The association of an owner o and a set of readers $R(o)$ defines a policy. Confidentiality label is expressed using set of policies. For example, considering the confidentiality label L_2 assigned to exported variables $cinfo$,$rinfo$, $cnotif$ and $rnotif$ in figure 3, where L_2 :{*Event-Creator*: *Event-Receiver1*, *Event-Receiver2*} where *Event-Creator* can act for *Event-Receiver1* and *Event-Receiver2*. We denote by $(\_)$ the less restrictive authority, for instance label $L_1 : \{\_ : \_\}$ assigned to the *request* interaction that is considered a public event.

A security domain is a lattice of the form $\langle S, \subseteq, \cup, \cap \rangle$ where:

- S is a finite set of security labels.
- $\subseteq$ is a partial order "can flow to" on S that indicates that information can flow from one security level to an equal or a more restrictive one. For two labels L_1 and L_2, we consider that $L_1 \subseteq L_2$ if and only if $O(L_1) \preceq O(L_2)$ and $\forall o \in O(L_1), R(L_1, o) \preceq R(L_2, o)$.
- $\cup$ is a "join" operator for any two labels in S and that represents the upper bound (LUB) of them. The join of two labels L_1 and L_2 denoted $L_1 \cup L_2$ contains an owner set $O(L_1 \cup L_2) = O(L_1) \cup O(L_2)$ and $\forall o \in O(L_1 \cup L_2)$ there is reader set $R(L_1 \cup L_2, o) = R(L_1, o) \cap R(L_2, o)$
- $\cap$ is a "meet" operator for any two levels in S that represents the lower bound (GLB)of them. The meet of two labels L_1 and L_2 denoted $L_1 \cap L_2$ do contains an owner set $O(L_1 \cup L_2) = O(L_1) \cup O(L_2)$ and $\forall o \in O(L_1 \cup L_2)$ there is reader set$R(L_1 \cup L_2, o) = R(L_1, o) \cup R(L_2, o)$

The intuition behind the definition of $\subseteq$ relation is that *(1)* the information can only flow from one owner o_1 to either the same or a more powerful owner o_2 where o_2 can acts for o_1 and *(2)* the readers allowed by $R(L_2, o)$ must be a subset of the readers allowed by $R(L_1, o)$ where we consider that the readers allowed by a policy include not only the principals explicitly mentioned by the policy but also any principal able to act for the explicitly mentioned reader is also able to read the data.

Let $C = \gamma(B_1, \ldots B_n)$ be a composite component, fixed. Let X (resp. P) be the set of all variables (resp. ports) defined in all atomic components $(B_i)_{i=1,n}$. Let $\langle S, \subseteq, \cup, \cap \rangle$ be a security domain, fixed.

DEFINITION 3 (SECURITY ASSIGNMENT σ). *A security assignment for component C is a mapping $\sigma : X \cup P \cup \gamma \to S$ that associates security levels to variables, ports and interactions such that, moreover, the security levels of ports matches the security levels of interactions, that is, for all $a \in \gamma$ and for all $p \in P$ it holds $\sigma(p) = \sigma(a)$.*

In atomic components, the security levels considered for ports and variables allow to track intra-component information flows and control the intermediate computation steps. Moreover, inter-components communication, that is, interactions with data exchange, are tracked by the security levels assigned to interactions. For example, ports, variables and interactions of previously presented example in Figure 3 are tagged with L_1, L_2 security levels (graphically represented with dashed squares).

We will now formally introduce the notions of non-interference for our component model. We start by providing few additional notations and definitions. Let σ be a security assignment for C, fixed. For a security level $s \in S$, we define $\gamma{\downarrow_s^\sigma}$ the restriction of γ to interactions with security level at most s that is formally, $\gamma{\downarrow_s^\sigma} = \{a \in \gamma \mid \sigma(a) \subseteq s\}$.

For a security level $s \in S$, we define $w|_s^\sigma$ the projection of a trace $w \in \gamma^*$ to interactions with security level lower or equal to s. Formally, the projection is recursively defined on traces as $\epsilon|_s^\sigma = \epsilon$, $(aw)|_s^\sigma = a(w|_s^\sigma)$ if $\sigma(a) \subseteq s$ and $(aw)|_s^\sigma = w|_s^\sigma$ if $\sigma(a) \not\subseteq s$. The projection operator $|_s^\sigma$ is naturally lifted to sets of traces W by taking $W|_s^\sigma = \{w|_s^\sigma \mid w \in W\}$.

For a security level $s \in S$, we define the equivalence $\approx_s^\sigma$ on states of C. Two states q_1, q_2 are equivalent, denoted by $q_1 \approx_s^\sigma q_2$ iff (1) they coincide on variables having security levels at most s and (2) they coincide on control states having outgoing transitions labeled with ports with security level at most s. We are now ready to define the two types of non-interference respectively *event non-interference (ENI)* and *data non-interference (DNI)*. We consider that deducing event-related information represent a risk that should be handled while controlling the system's information flow in addition to data flows.

DEFINITION 4 (EVENT/DATA NON-INTERFERENCE). *The security assignment σ ensures event (ENI) and data non-interference (DNI) of $\gamma(B_1, \ldots, B_n)$ at security level s iff,*

$$(ENI)\ \ \forall q_0 \in Q_C^0 : \textsc{traces}(\gamma(B_1, \ldots, B_n), q_0)|_s^\sigma = \textsc{traces}((\gamma{\downarrow_s^\sigma})(B_1, \ldots, B_n), q_0)$$

$$(DNI)\ \ \forall q_1, q_2 \in Q_C^0 :\ q_1 \approx_s^\sigma q_2 \Rightarrow \forall w_1 \in \textsc{traces}(C, q_1), w_2 \in \textsc{traces}(C, q_2) :\ w_1|_s^\sigma = w_2|_s^\sigma \Rightarrow \forall q_1', q_2' \in Q_C :\ q_1 \xrightarrow[C]{w_1} q_1' \wedge q_2 \xrightarrow[C]{w_2} q_2' \Rightarrow q_1' \approx_s^\sigma q_2'$$

Moreover, σ is said secure *for a component $\gamma(B_1, \ldots, B_n)$ iff it ensures both event and data non-interference, at all security levels $s \in S$.*

Here non-interference is expressed as indistinguishability between several states and traces of the system. For instance, an attacker that can observe the system's variables and occurences of interactions at security level L_1 must not be able to distinguish neither changes on variables or occurrence of interactions having higher security level L_2.

3.3 Checking Non-interference

We established sufficient syntactic conditions that aim to simplify the verification of non-interference and reduce it to local constrains check on both transitions (inter-component verification) and interactions (intra-component verification). We recall these conditions in order to be used later in section 4 for rechecking security correctness of the decentralized component model. Indeed, these conditions offer an easy way to automate verification and there preservation proofs the system non-interference.

DEFINITION 5 (SECURITY CONDITIONS). *Let $C = \gamma(B_1, \ldots, B_n)$ be a composite component and let σ be a security assignment. We say that C satisfies the security conditions for security assignment σ iff:*

(i) the security assignment of ports, in every atomic component B_i is locally consistent, that is:

- *for every pair of causal transitions:*

$$\forall \tau_1, \tau_2 \in T_i : \ \tau_1 = \ell_1 \xrightarrow{p_1} \ell_2, \ \tau_2 = \ell_2 \xrightarrow{p_2} \ell_3 \Rightarrow \ell_1 \neq \ell_2 \Rightarrow \ \sigma(p_1) \subseteq \sigma(p_2)$$

- *for every pair of conflicting transitions:*

$$\forall \tau_1, \tau_2 \in T_i : \ \tau_1 = \ell_1 \xrightarrow{p_1} \ell_2, \ \tau_2 = \ell_1 \xrightarrow{p_2} \ell_3 \Rightarrow \sigma(p_1) = \sigma(p_2)$$

(ii) all assignments $x := e$ occurring in transitions within atomic components and interactions are sequential consistent, in the classical sense:

$$\forall y \in use(e) : \ \sigma(y) \subseteq \sigma(x)$$

(iii) variables are consistently used and assigned in transitions and interactions, that is,

$$\forall \tau \in \cup_{i=1}^{n} T_i \forall x, y \in X : x \in def(f_\tau), y \in use(g_\tau) \Rightarrow \sigma(y) \subseteq \sigma(p_\tau) \subseteq \sigma(x)$$
$$\forall a \in \gamma \ \ \forall x, y \in X \ : x \in def(F_a), y \in use(G_a) \Rightarrow \sigma(y) \subseteq \sigma(a) \subseteq \sigma(x)$$

(iv) all atomic components B_i are port deterministic:

$$\forall \tau_1, \tau_2 \in T_i : \ \tau_1 = \ell_1 \xrightarrow{p} \ell_2, \tau_2 = \ell_1 \xrightarrow{p} \ell_3 \Rightarrow (g_{\tau_1} \wedge g_{\tau_2}) \text{ is unsatisfiable}$$

The first family of conditions (i) is similar to Accorsi's conditions [1] for excluding causal and conflicting places for Petri net transitions having different security levels. Similar conditions have been considered in [9, 10] and lead to more specific definitions of non-interferences and bisimulations on annotated Petri nets. The second condition (*ii*) represents the classical condition needed to avoid information leakage in sequential assignments. The third condition (*iii*) tackles covert channels issues. Indeed, (*iii*) enforces the security levels of the data flows which have to be consistent with security levels of the ports or interactions (e.g., no low level data has to be updated on a high level port or interaction). Such that, observations of public data would not reveal any secret information. Finally, condition (*iv*) enforces deterministic behavior on atomic components.

The following result, proven in [5], states that the above security conditions are sufficient to ensure both event and data non-interference.

THEOREM 1. *Whenever the security conditions hold, the security assignment σ is secure for the composite component C.*

As an illustration, consider the composite component in Figure 3. It can be relatively easily checked that the security conditions hold. Indeed since *Event-Creator* acts for *Event-Receiver1* and *Event-Receiver2*, $L1 \subseteq L_2$ and the security level L_2 of variables at the assignment $cinfo := rinfo$ is consistent with the security level L_1 of the guard's variable *active*. Besides, the security level of ports involved in all interactions is also consistent (equal). Henceforth, the composite component is secure.

4. SECURE DECENTRALIZED MODEL

In this section we first recall the key steps for decentralizing the functional model from Section 3.1 following a transformation method from [6]. This method relies on a systematic transformation of centralized atomic components and replacement of multiparty interaction by protocols expressed using send/receive (S/R) interactions. S/R interactions are binary point-to-point and directed interactions from one sender component (port), to one receiver component (port) implementing message passing. The transformation guarantees that the receive port is always enabled when the corresponding send port becomes enabled, and therefore S/R interactions can be safely implemented using asynchronous message passing primitives (e.g., TCP/IP network communication, MPI communication, etc...). To this end, each atomic component B_i is transformed into a decentralized B_i^{SR} component.

DEFINITION 6 (COMPOSITE S/R COMPONENT). *$C^{SR} = \gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR})$ is a S/R composite component if we can partition the set of ports of B^{SR} into three sets P_s, P_r, P_u that are respectively the set of send-ports, receive-ports and unary interaction ports.*

- Each interaction $a = (P_a, G_a, F_a) \in \gamma^{SR}$ is either (1) a S/R interaction with $P_a = (s, r_1, r_2, ..., r_k)$, $s \in P_s$, $r_1, ..., r_k \in P_r$ and $G_a = true$ and F_a copies the variables exported by the port s to the variables exported by the port $r_1, r_2 ..., r_k$ or (2) a unary interaction $P_a = \{p\}$ with $p \in P_u$, $G_a = true$, F_a is the identity function.
- If s is a port in P_s, then there exists one and only one S/R interaction $a = (P_a, G_a, F_a) \in \gamma^{SR}$ with $P_a = (s, r_1, r_2, ..., r_k)$ and all ports $r_1, r_2, ..., r_k$ are receive ports. We say that $r_1, r_2, ..., r_k$ are the receive ports associated to F.
- If $a = (P_a, G_a, F_a)$ with $P_a = (s, r_1, r_2, ..., r_k)$ is a S/R interaction in γ^{SR} and s is enabled in some global state of C^{SR} then all its associated receive-ports $r_1, r_2, ..., r_k$ are also enabled at that state.

From a functional point of view, the main challenge when transforming functional models with multiparty interactions towards distributed models with send/receive interactions is to enhance parallelism for execution of concurrently enabled interactions and computations within components. That is, in a distributed setting, each atomic component executes independently and thus has to communicate with other components in order to ensure correct execution with respect to the original semantics. The existing method for distributed implementation relies on introducing an interaction protocol layer to handle interactions between decentralized atomic components layer. The first layer (S/R atomic component) includes transformed atomic components. Each atomic component will publish its offer, that is the list of its enabled ports, and then wait for a notification indicating which interaction has been chosen for execution. The second layer (*IP*) deals with distributed execution of interactions by implementing specific interaction protocols. The interaction protocol evaluates the guard of each interaction and executes the associated update function. The interface between this layer and the component layer provides ports for receiving offers and notifying the ports selected for execution.

In the rest of this section, we extend the above decentralization method such that to encompass and preserve information flow security. Here, we impose additional modifications in S/R component behavior to encompass multi-level security annotations, as well as restrictions on the partitioning of the *IP* components. That is, interactions must be statically partitioned according to their security levels to enforce isolation at event level. Despite this partitioning, we show that different security level data still can be managed within a single security interaction manager. Besides, we provide label propagation rules to automatically enforce the non-interference at decentralized model which enforces security at implementation level.

Let $C = \gamma(B_1, \cdots B_n)$ be a composite component and σ be a security assignment for C with domain S, fixed. Moreover, assume that σ satisfies the security conditions defined in subsection 3.3 for C. Furthermore, to simplify presentation, consider that atomic components are deterministic, that is, for every state ℓ and port p there exists at most one transition outgoing ℓ and labelled by p.

4.1 Atomic Components Layer

The transformation at atomic component level consists on breaking the atomicity of there transitions. Precisely, each transition is split into two consecutive steps: (1) an offer that publishes the current state of the component, and (2) a notification that triggers the update function. The intuition behind this transformation is that the offer transition correspond to sending information about component's intention to interact to some scheduler and the notification transition corresponds to receiving the answer from the scheduler, once some interaction has been completed. Update functions can be then executed concurrently and independently by components upon notification reception.

In contrast to [6], to protect the information flow in the distributed context, some changes are needed. A distinct offer port o_s and a different participation number n_s are defined for every security level used within the corresponding atomic component in centralized model. Thus, we ensure that offers and their corresponding notifications have the same security level. Equally, information about execution of interactions having different security levels is not revealed through the observation of n_s variable.

Definition 7 (Transformed atomic component). *Let $B = (L, X, P, T)$ be an atomic component within C. The corresponding transformed S/R component is $B^{SR} = (L^{SR}, X^{SR}, P^{SR}, T^{SR})$:*

- $L^{SR} = L \cup L^{\perp}$, *where* $L^{\perp} = \{\perp_\ell \mid \ell \in L\}$
- $X^{SR} = X \cup \{x_p\}_{p \in P} \cup \{n_s | s \in S\}$ *where each x_p is a Boolean variable indicating whether port p is enabled, and n_s is an integer called a participation number (for security level s).*
- $P^{SR} = P \cup \{o_s \mid s \in S\}$. *The offer ports o_s export the variables $X^{SR}_{o_s} = \{n_s\} \bigcup \{\{x_p\} \cup X_p \mid \sigma(p) = s\}$ that is the participation number n_s, the new Boolean variables x_p and the variables X_p associated to ports p having security level s. For all other ports $p \in P$, we define $X^{SR}_p = X_p$.*
- *For each state $\ell \in L^{SR}$, let S_ℓ be the set of security levels assigned to ports labelling all outgoing transitions of ℓ. For each security level $s \in S_\ell$, we include the following transition $\tau_{o_s} = (\perp_\ell \xrightarrow{o_s} \ell) \in T^{SR}$, where the guard g_{o_s} is true and f_{o_s} is the identity function.*
- *For each transition $\tau = \ell \xrightarrow{p} \ell' \in T$ we include a notification transition $\tau_p = (\ell \xrightarrow{p} \perp_{\ell'})$ where the guard g_p is true. The function f_p applies the original update function f_τ on X, increments n_s and updates the Boolean variables x_r, for all $r \in P$. That is, $x_r := g_\tau$ if $\exists \tau = \ell' \xrightarrow{r} \ell'' \in T$, and false otherwise.*

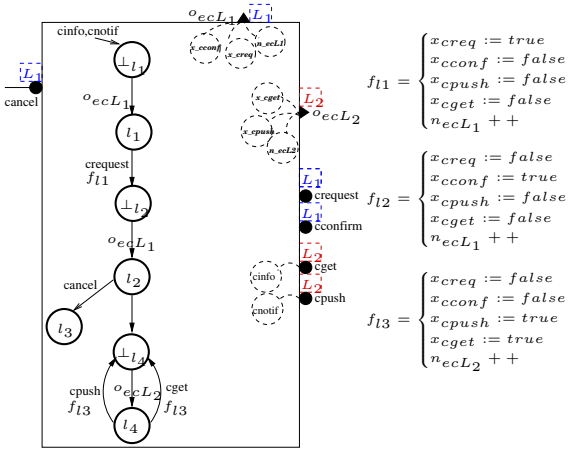


Figure 4: Transformation of the Event-Creator component (Figure 3).

Example 3. *Figure 4 represents the transformed S/R version of the Event-Creator component, presented in Figure 3. The component is initially in control state $\perp_{l_1}$. It sends an offer through the corresponding offer port o_{ecL_1} containing the current enabled port x_{creq} and the participation number n_{ecL_1}, then reaches state l_1. In that state, it waits for a notification from crequest port triggers the execution of the update function which consists on incrementing the value of n_{ecL_1} and re-evaluating x_{creq} and x_{cconf}. At state $\perp_{l_2}$, the Event-Creator sends an offer through port o_{ecL_1} to create an event between participants. The event is created only if all participants are ready to join, otherwise, the event is cancelled.*

Definition 8 (security assignment σ^{SR} for B^{SR}). *The security assignment σ^{SR} is defined as an extension of the original security assignment σ. For variables X^{SR} and ports P^{SR} of a transformed atomic components B^{SR}, define*

$$\sigma^{SR}(x) = \begin{cases} \sigma(p) & \text{if } x = x_p \text{ for some } p \in P \\ s & \text{if } x = n_s \text{ for some } s \in S \\ \sigma(x) & \text{otherwise, for any other } x \in X^{SR} \end{cases}$$

$$\sigma^{SR}(p) = \begin{cases} s & \text{if } p = o_s \text{ for some } s \in S \\ \sigma(p) & \text{otherwise, for any other } p \in P^{SR} \end{cases}$$

As an illustration, reconsider the example depicted in Figure 4. Following the above definition, ports *crequest*, *cconfirm* and o_{ecL_1} are tagged as (L_1) and respectively ports *cpush*, *cget* and o_{ecL_2} are tagged with (L_2) where $L_1 \subseteq L_2$.

Lemma 2. *If the security assignment σ satisfies the security conditions for the atomic component B then the security assignment σ^{SR} satisfies the security conditions for the transformed S/R component B^{SR}.*

4.2 Interaction Protocol (IP) Layer

This layer consists of a set of components, each in charge of execution of a subset of interactions in the initial centralized model. Each component represent a scheduler that receives messages from S/R components then calculates the enabled interaction and selects them for execution. The use of the *IP* components allow parallel execution at components level as well as interactions level in a distributed environment. In this section we show how to construct a secure *IP* schedulers without introducing unexpected behavior nor disallowing interleavings, which represents a compromise between liveness and security property in distributed systems. Indeed, a parallel execution of two distinct security level interaction would not need to suspend one of them to maintain security, thus there will be no internal timing leak.

Conflicts between interactions executed by the same *IP* component are resolved by that component locally. Two interactions a_1 and a_2 are in conflict iff either, they share a common port p (i.e $p \in a_1 \cap a_2$), or there exist two conflicting transitions at a local state ℓ of a component B_i that are labelled with ports p_1 and p_2, where $p_1 \in a_1$ and $p_2 \in a_2$. *IP* components behaviors used in this layer are similar to the ones introduced in [6]. However, to enforce event noninterference we enforce the partitioning of interactions according to there security levels. Let us remark that, for a centralized component model satisfying security conditions, the above partitioning can be proven *conflict-free*, that is, no conflicts exist between interactions having different security levels. Henceforth, all conflicts can be resolved locally by *IP* components. Such a solution is practical and preserve initial system behavior without introducing additional waiting time or priorities at execution to some interactions of certain level over others in schedulers (IP components) to preserve security.

Nonetheless, the security of this centralized solution is not granted for the data part. Remind that in the centralized model, interactions at some security level s are allowed to transfer and/or perform arbitrary computations on data variables with levels other than s. Therefore, in contrast to interactions, annotations of variables requires a bit of care to maintain the security conditions. A concrete example is presented in Figure 6 and discussed later.

Let $C = \gamma(B_1, \cdots, B_n)$ be a composite component and $\gamma_s = \{a_i = (P_i, F_i, G_i) | a_i \in \gamma, \sigma(a_i) = s\}$ the set of interactions of level s. Let *participants*(γ_s) (resp. *ports*(γ_s)) be the set of atomic components (resp. ports) participating (resp. occurring) in interactions from γ_s.

Definition 9 (IP component at level s). *The component $IP_s = (L^{IP}, X^{IP}, P^{IP}, T^{IP})$ handling γ_s is defined as:*

- *Set of places $L^{IP} = \{w_i, r_i \mid B_i \in participants(\gamma_s)\} \cup \{s_p \mid p \in ports(\gamma_s)\}$.*
- *Set of variables $X^{IP} = \{n_{is} \mid B_i \in participants(\gamma_s)\} \cup \{\{x_p\} \cup X_p \mid p \in ports(\gamma_s)\}$*
- *Set of ports $P^{IP} = \{o_{si} \mid B_i \in participants(\gamma_s)\} \cup \{p \mid p \in ports(\gamma_s)\}$ where offer ports o_{is} are associated to variables n_{is}, x_p, and X_p from component B_i and ports p are associated to variables X_p.*
- *Set of transitions $T^{IP} \subseteq 2^{L^{IP}} \times P^{IP} \times 2^{L^{IP}}$. A transition τ is a triple $({}^\bullet\tau, p, \tau^\bullet)$, where ${}^\bullet\tau$ is the set of input places of τ and $\tau^\bullet$ is the set of output places of τ. We introduce three types of transitions:*
  - *receiving offers (w_i, o_{si}, r_i) for all components $B_i \in participants(\gamma_s)$.*
  - *executing interaction $(\{r_i\}_{i\in I}, a, \{s_{pi}\}_{i\in I})$ for each interaction $a \in \gamma_s$ such that $a = \{p_i\}_{i\in I}$, where I is the set of components involved in a. To this transition we associate the guard $[G_a \wedge \bigwedge_{p\in a} x_p]$ and we apply the original update function F_a on $\cup_{p\in a} X_p$.*
  - *sending notification (s_p, p, w_i) for all ports p and component $B_i \in participants(\gamma_s)$.*

Example 4. *Figure 5 illustrates the IP_{L_1} component constructed for interactions $\gamma_{L_1} = \{request, confirm\}$ for the example shown in Figure 3. For all B_i^{SR} components involved in interactions γ_{L_1}, we introduce a waiting (w_i) and receiving (r_i) places (i.e, (w_{er1} ,w_{ec}, w_{er2}) and (r_{er1}, r_{ec}, r_{er2}). For all ports p involved in γ_{L_1} we introduce a sending place s_{p_i} (i.e, (s_{rreq1}, s_{creq}, s_{rreq2}, s_{rconf1}, s_{rconf2} and s_{cconf}). The IP_{L_1} component moves from w_i to rcv_i whenever it receives an offer from the corresponding component B_i^{SR}. After choosing and executing interactions, the IP_{L_1} component moves to sending (s_p) places to send notification through ports p to the corresponding component.*

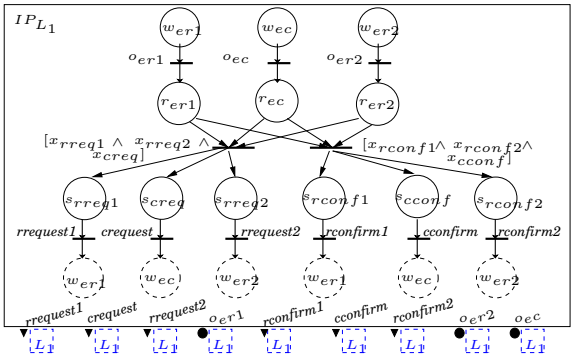


Figure 5: IP_{L_1} Event-secure interactions scheduler component

Definition 10 (security assignment σ^{SR} for IP_s). *The security assignment σ^{SR} is built from the original security assignment σ. For variables X^{IP} and ports P^{IP} of the IP_s component that handles γ_s, we define*

$$\sigma^{SR}(x) = \begin{cases} \sigma(x) & \text{if } x \in X_p \text{ and } s \subseteq \sigma(x) \\ s & \text{otherwise} \end{cases}$$

$$\sigma^{SR}(p) = s, \text{ forall } p \in P^{IP}$$

Informally, the security assignment σ^{SR} maintains the same security level for all variables having their level greater than s in the original model and *upgrades* the others to s. That is, all variables within the IP_s component will have security levels at least s. This change is mandatory to ensure consistent copy of data in offers (resp. notifications) from (resp. to) components to the IP.

Example 5. *Figure 6 (a) presents a data transfer between Event-Creator and an Event-Receiver1 on get interaction where the variable $rinfo$ from component Event-Receiver1*

is assigned to the variable cinfo in component Event-Creator. Here we assume that variable rinfo is tagged with L_1 annotation and variables cnotif is tagged with L_2 where $L_1 \subseteq L_2$. In the decentralized model presented in Figure 6 (b), the IP_{L_2} component executes the interaction get. To this end, variable cinfo is imported into a same security level variables cinfo′, while the variable rinfo is imported into a higher security level variable rinfo′, through the corresponding offer port, such that we preserve the security level consistency between interaction (ports) and the transferred variables . Once the interaction takes place, cinfo′ is copied back to cinfo on the notification transition. No copy is performed back to the rinfo, hence, we manage variables of different levels into the same IP scheduler while preserving there initially defined levels.

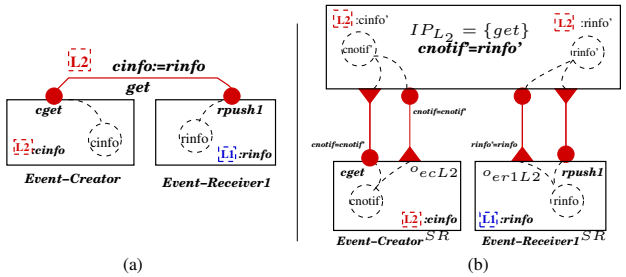


Figure 6: Secure data exchange between atomic and IP components.

LEMMA 3. *IP components satisfies the security conditions with security assignment σ^{SR}.*

Figure 7 represents the distributed model of the system shown in Figure 3 with two distinct security level interaction management. Indeed, low-level security interactions *request* and *confirm* are managed with a single IP_{L_1}, where high level interaction *push* and *get* are managed with the IP_{L_2}.

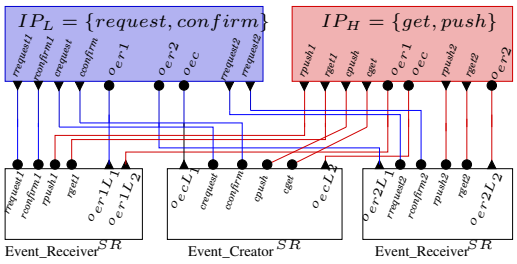


Figure 7: Centralized interaction management

4.3 Cross-layer Interactions

Hereafter, we define the interactions in the S/R composition model Following Definition 6, we introduce S/R interactions by specifying the sender and the associated receivers. Given a composite component $C = \gamma(B_1, \cdots, B_n)$, and partitions $\gamma_{1s}, \cdots, \gamma_{ms}$ of $\gamma_s \subseteq \gamma$, for every security levels $s \in S$, the transformation produces a S/R composite component $C^{SR} = \gamma^{SR}(B_1^{SR}, \cdots, B_n^{SR}$, $(IP_{1s}, \cdots, IP_{ms})_{s \in S})$. We define the S/R interactions of γ^{SR} as follows:

- For every atomic component B_i^{SR} participating in interactions of security level s, for every IP components $IP_{1s}, \cdots, IP_{ms}$ handling γ_s, include in γ^{SR} the *offer* interaction $off_s = (B_i^{SR}.o_s, IP_{1s}.o_{is}, \cdots, IP_{ms}.o_{is})$.
- For every port p in component B_i^{SR} and for every IP_{js} component handling an interaction involving p, we include in γ^{SR} the *response* interaction res_p=$(IP_{js}.p, B_i^{SR}.p)$

DEFINITION 11 (SECURITY ASSIGNMENT σ^{SR} FOR γ^{SR}). *The security assignment σ^{SR} is build from the security assignment σ. For interactions γ^{SR} between all atomic components of the transformed model, we define $\sigma^{SR}(a) = s$ for any interaction a involving an IP_{js} component handling interactions with security level s.*

LEMMA 4. *All the cross-layer interactions of C^{SR} are secure with σ^{SR}.*

The following theorem states the correctness of our transformation, that is, the constructed S/R model satisfies the security conditions by construction.

THEOREM 5 (SECURITY-BY-CONSTRUCTION). *If the component C satisfies security conditions for the security assignment σ then the transformed component C^{SR} satisfies security conditions for the security assignment σ^{SR}.*

PROOF. From lemma 2,3 and 4 we ensure the preservation of all security conditions at all S/R model layer and transformation steps. □

5. IMPLEMENTATION

In this section, we illustrate the complete design flow for generating secure distributed code represented in Figure 8. The white strong lined boxes represent modules that we implemented while the shaded strong lined ones represent modules that already exists and we modified to encompass security. Based on *BIP* framework [14], we implement these modules in Java language and we generate a C++ code. In this architecture, the flow consists on configuring security at two levels, first at the abstract model and second depending on target platform. Hereafter, we discuss the different steps and design choices.

5.1 Abstract Model Configuration

Additionally to the system functional model (.bip), the system designer provide a configuration file (Annotations.xml) that contains the DLM annotations from Section 3.2 where we define the acts_for relations and labels to different ports and data in each component. Figure 9 presents fragments of the configuration file for the *Whens-App* abstract model. We extend the system model parser to extract labels from Annotations.xml file. Then, we associate annotations to their corresponding ports and data types stored in the secure component model. Next, the *secureBIP checker* tool browses all atomic components and interactions in the model to extract events dependencies at each local state (incoming and outgoing port labelled transitions) and data dependencies at different transition's and interaction's actions and checks their

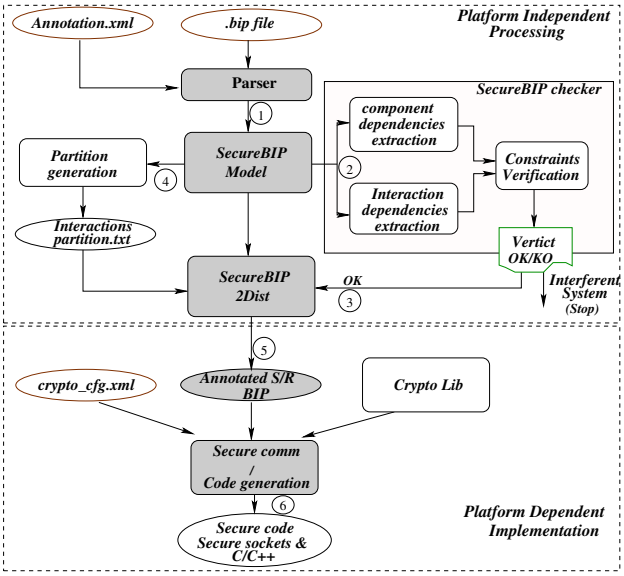


Figure 8: Tool-set Architecture Overview.

label consistency. In the case where tool verdict is positive, the tool generates automatically an interaction partition file that describes the set of interactions that each *IP*component would manage. This file is used as input by *secureBIP 2Dist* to generate an annotated S/R model. The *secureBIP 2Dist* generator is modified to encompass modifications in decentralized model as well as rules for annotations propagation.

```
- <config>
  - <acts_for>
      <authority>Event_Creator:Event_receiver1,Event_Receiver2</authority>
    </acts_for>
  - <var_config>
      <variable name="cinfo" component="Event_Creator"
        label="Event_Creator:Event_receiver1,Event_Receiver2" />
      <variable name="rinfo1" component="Event_receiver1"
        label="Event_Creator:Event_receiver1,Event_Receiver2" />
      <variable name="rinfo2" component="Event_receiver2"
        label="Event_Creator:Event_receiver1,Event_Receiver2" />
      ...
    </var_config>
  - <port_config>
      <port name="crequest" component="Event_Creator" label="_:_" />
      <port name="cconfirm" component="Event_Creator" label="_:_" />
      <port name="cpush" component="Event_Creator"
        label="Event_Creator:Event_receiver1,Event_Receiver2" />
      <port name="cget" component="Event_Creator"
        label="Event_Creator:Event_receiver1,Event_Receiver2" />
      ...
    </port_config>
</config>
```

Figure 9: Configuration file for the abstract model.

5.2 Platform-Dependent Configuration

Here the system designer provides configuration file that contains the cryptographic mechanisms to be used to ensure confidentiality for data and ports to secure interactions between atomic S/R and IP components. To preserve confidentiality we use encryption. We assume that the generated code is running on trusted hosts where it is safe to generate and store encryption keys. The *Crypto Lib* library contains the different encryption protocols and functions that, following the configuration file, the code generator selects messages to secure at communications using secure TCP/IP sockets.

The configuration states the encryption mechanisms for each defined security level, that is, for variables and ports that need to be secured following the secure abstract annotations. A data security is enforced using authentication, encryption and signature mechanisms to encrypt and sign the data at socket buffer before sending it. However we consider that encryption does only provide a degree of privacy with variables. Hence, we enforce the security of ports, if it is configured so, by hiding the message identity at sending to enforce privacy of message sender and receiver.This is done by encapsulating the sent message such that no information can be deduced by observing message transfer between components. In this message source and receiver are encrypted under a shared key between sender and receiver component. The *message_index* (common encrypted pass-world shared between sender and receiver) will be used by the receiver to retrieve the sent message. According to domain application, there exist some privacy extensions allowing the identities of the communicating parties to be hidden from third parties.

Following this defined configuration, we automatically generate stand-alone C++ processes for every S/R components (atomic and *IP*) communicating with secure TCP/IP sockets channels that can be deployed and run on a distributed network. Each C++ process can be run on a host that ensures at least the upper bound security level of annotated data and ports in it. Obviously, it is easier to find a set of hosts that are trusted to run a process of specific security level at most than it is to find a host that can run the whole multi-level system.

5.3 Evaluation

Here we introduce configuration according to the propagated annotation in the distributed model using the configuration file where we specify authentication and encryption mechanisms. The executions is performed on an Intel Code 2Duo 2GHz with 4GB RAM memory running Linux Ubuntu. For generation of the certificates and encryption we use OpenSSL library which contains tested C libraries and here we use X.509 certificates for signature and an asymmetric encryption algorithm (RSA) with 2048bit key size.

Components	Compilation (s)	Execution(s)	
		$\gamma(B^{SR})$	Sec-$\gamma(B^{SR})$
3	3.02	4.6	7.1
11	3.84	8.6	10.3
25	4.15	12.5	15.7
101	4.91	22.1	25.2

Table 1: Whens-App configuration and execution time (in s)

As an Evaluation of our approach performance, Table 1 presents some experiments over compilation time that include security check and model transformation and the execution time of the generated code for different component number of *Event-Receivers* in the *Whens-App* system with the use of security mechanisms (*Sec-*$\gamma(B^{SR})$) and without ($\gamma(B^{SR})$). The number of decentralized multi-party interactions for a system with 100 components is 50. the number of binary interactions executed in the decentralized model has reached 480 interaction for a system of 100 components managed with two IP components. Here we can see, that there is no significant overhead at compilation time with the increase of system component. The use of cryptographic mechanisms induces an overhead of 20%, however this per-

formance can be improved if we choose to use,for instance, symmetric encryption instead of the asymmetric one currently implemented. Despite that our prototype implementation is based on the use of secure TCP/IP sockets, the interaction protocol implementation can be flexible and encompass other communication types depending on application domain, such as RMI.

6. RELATED-WORK

Previous works are mainly related to model-based security and deals wit both event and data IFC and the multi-party security.

Model-based security: Several works on model-based security aim at simplifying security configuration and coding [7, 3]. In [3], authors, propose modeling security policy in UML and target automating security code generation for business applications like JEE and .net applications. Other works [7] use model-based approach to simplify secure code deployment on heterogeneous platforms. Compared to these, our work is not restricted to point-to-point access control and deals with information flow security. Recent works on information flow security in web services rely on Petri-nets for modeling composed services [2]. Petri-net graphs are generated from *BPEL* orchestration processes and are, next, modified by the developer to represent shared resources and to annotate interactions. Developer's modification is necessary here since Petri-nets capture event-based interactions only. Our model allows representing both data and events. Furthermore, in [2], proposed tools allowing only non-interference checking and no security enforcement is proposed.

Multi-party security: This domain deals with "data-mining privacy preservation", [12]. The goal is that a set of parties with private inputs wishes to jointly compute some function of their inputs. The parties learn the correct output and nothing else. A typical application is the one that need to publicize tables that sum up some statistics. This task is extremely dangerous because census questionnaires contain a lot of sensitive information, and it is crucial that it not be possible to identify a single user in the publicized tables. Compared to these works that generally deal with low level cryptography protocols verification, we provide a more abstract model that helps system designers to check their system security more rapidly. Indeed, it is very important to check security configuration early in the life cycle of system development. Before detailing the communication protocols and fine-grained system coding, the designer checks security configurations starting from a high level description of component's behavior and interactions.

7. CONCLUSION

In this paper, we present a practical approach to automatically secure information flow in distributed systems. Starting from an abstract component-based model with multi-party interactions, we verify security policy preservation as defined by the user, that is, verifying non-interference property at both event and data levels. Then, we generate a distributed model where multiparty interactions are replaced with asynchronous message passing. The intermediate distributed model is formally proved "secure-by-construction". Here we provide a framework with a set of tools allowing to build distributed systems and automatically generating secure code that implements the desired security policy defined at the centralized level. This work is now being extended to verify this approach on a parametric model with dynamic labelling annotations. We are also trying to apply our decentralization method to a relaxed version of non-interference (e.g, intransitive or with declassification mechanisms), since for some systems, the transitive definition of non-interference is relatively strict for practical use.

8. REFERENCES

[1] Accorsi, R., Lehmann, A.: Automatic information flow analysis of business process models. In: Proceedings of the 10th international conference on Business Process Management, BPM'12 (2012)

[2] Accorsi, R., Wonnemann, C.: Static information flow analysis of workflow models. In: Conference on Business Process and Service Computing, volume 147 of Lecture Notes in Informatics (2010)

[3] Basin, D., Doser, J., Lodderstedt, T.: Model driven security: from uml models to access control infrastructures. ACM Transactions on Software Engineering and Methodology p. 2006

[4] Bell, E.D., La Padula, J.L.: Secure computer system: Unified exposition and multics interpretation (1976)

[5] Ben Said, N., Abdellatif, T., Bensalem, S., Bozga, M.: Model-driven information flow security for component-based systems. In: Proceedings of Etaps workshop 2014, From Programs to Systems,The Systems Perspective in Computing (2014)

[6] Borzoo, B., Marius, B., Mohamad, J., Jean, Q., Joseph, S.: A framework for automated distributed implementation of component-based models. Distributed Computing (2012)

[7] Chollet, S., Lalanda, P.: Security Specification at Process Level. 2008 IEEE International Conference on Services Computing (2008)

[8] Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM (1977)

[9] Focardi, R., Rossi, S., Sabelfeld, A.: Bridging language-based and process calculi security. In: In Proc. of Foundations of Software Science and Computation Structures (FOSSACS'05), volume 3441 of LNCS, pp. 299–315. Springer-Verlag (2005)

[10] Frau, S., Gorrieri, R., Ferigato, C.: Formal aspects in security and trust (2009)

[11] Goguen, J., Meseguer, J.: Security policies and security models. In: Proceedings of the 1982 ieee symposium on security and privacy, pp. 11–20. IEEE Computer Society

[12] Lindell, Y., Pinkas, B.: Secure multiparty computation for privacy-preserving data mining. IACR Cryptology ePrint Archive (2008)

[13] Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. (2000)

[14] Zdancewic, S., Zheng, L., Nystrom, N., Myers, A.C.: Secure program partitioning. ACM Trans. Comput. Syst. (2002)

Bulk-Synchronous Communication Mechanisms in Diderot

John Reppy
University of Chicago
jhr@cs.uchicago.edu

Lamont Samuels
University of Chicago
lamonts@cs.uchicago.edu

Abstract

Diderot is a parallel domain-specific language designed to provide biomedical researchers with a high-level mathematical programming model where they can use familiar tensor calculus notations directly in code without dealing with underlying low-level implementation details. These operations are executed as parallel independent computations, called *strands*, in a bulk synchronous parallel (BSP) fashion. The original BSP model of Diderot limited strand creation to initialization time and did not provide any mechanisms for communication between strands. For algorithms, such as particle systems, where strands are used to explore the image space, it is useful to be able to create new strands dynamically and share data between strands.

In this paper, we present an updated BSP model with three new features: a spatial mechanism that retrieves nearby strands based on their geometric position in space, a global mechanism for global computations (*i.e.*, parallel reductions) over sets of strands and a mechanism for dynamically allocating new strands. We also illustrate through examples how to express these features in the Diderot language. More, generally, by providing a communication system with these new mechanisms, we can effectively increase the class of applications that Diderot can support.

Keywords Actor Model, Domain Specific Languages, Image Analysis, Scientific Visualization, Parallelism,

1. Introduction

Biomedical researchers use imaging technologies, such as *computed tomography* (CT) and *magnetic resonance* (MRI) to study the structure and function of a wide variety of biological and physical objects. The increasing sophistication of these new technologies provide researchers with the ability to quickly and efficiently analyze and visualize their complex data. But researchers using these technologies may not have the programming background to create efficient parallel programs to handle their data. We have created a language called Diderot that provides tools and a system to simplify image data processing.

Diderot is a parallel domain specific language (DSL) that allows biomedical researchers to efficiently and effectively implement image analysis and visualization algorithms. Diderot supports a high-level mathematical programming model that is based on continuous tensor fields. We use *tensors* to refer to scalars, vectors, and matrices, which contain the types of values produced by the medical imaging technologies stated above and values produced by taking spatial derivatives of images. Algorithms written in Diderot can be directly expressed in terms of tensors, tensor fields, and tensor operations, using the same mathematical notation that would be used in vector and tensor calculus. Diderot is targeted towards image analysis and visualization algorithms that use real image data, where the data is better processed as parallel computations.

We model these independent computations as autonomous lightweight threads called *strands*. Currently, Diderot only supports applications that allow strands to act independently of each other, such as direct volume rendering [7] or fiber tractography [9]. Many of these applications only require common tensor operations or types (*i.e.*, reconstruction and derivatives for direct volume rendering or tensor fields for fiber tractography), which Diderot already provides. However, Diderot is missing features needed for other algorithms of interest, such as particle systems,where strands are used to explore image space and may need to create new strands dynamically or share data between other strands.

In this paper, we present new communication mechanisms for our parallelism model that include support for inter-strand communication, global computations over sets of strands, and dynamic allocation of strands. Inter-strand communication is a spatial mechanism that retrieves the state information of nearby strands based on their geometric position in space. Global communication is a mechanism based on sharing information on a larger scale within the program using parallel reductions. Finally, new strands can be created

during an execution step and will begin running in the next iteration.

The paper is organized as follows. In the next section, we discuss our parallelism model and applications that benefit from this new communication system. We then present the communication system's design details in Section 3 and describe important aspects of our implementation in Section 4. A discussion of related work is presented in Section 5. We briefly summarize our system and describe future plans for it in Section 6.

2. Background

Diderot is based around two fundamental design aspects: a high-level mathematical programming model, and an efficient execution model. The mathematical model is based on linear algebra and properties of tensor calculus. More background details about our mathematical model and its design are covered in an earlier paper [5]. This paper focuses more on the execution model of Diderot. This deterministic model executes these independent strands in a bulk-synchronous parallel (BSP) fashion [20] [21]. This section provides a brief overview of our execution model and discusses potential new applications that benefit from these new features.

2.1 Program Structure

Before discussing our execution model, it would be beneficial to provide a simple example that describes the structure of a Diderot program. A program is organized into three sections: global definitions, which include program inputs; strand definitions, which define the computational core of the algorithm; and initialization, which defines the initial set of strands. We present a program that uses Heron's method for computing square roots (shown in Figure 1) to illustrate this structure.

Lines 1–3 of Figure 1 define the global variables of our program. Line 3 is marked as an **input** variable, which means it can be set outside the program (input variables may also have a default value, as in the case of `eps`). Lines 1–2 load the dynamic sequence of integers from the file `"numbers.nrrd"`, binds the sequence to the variable `args` and retrieves the number of elements in `args`.

Similar to a kernel function in CUDA [18] or OpenCL [14], a strand defintion in Diderot encapsulates the computational core of the application. Each strand has parameter(s) (*e.g.*, `arg` on Line 5), a *state* (Line 7) and an **update** method (Lines 8–12). The strand state variables are initialized when the strand is created. State variables can be declared as one of our five concrete types: booleans, integers, strings, tensors, and fixed-size sequences of values; some variables may be annotated as **output** variables (Line 7), which define the part of the strand state that is reported in the program's output. Heron's method begins with choosing an arbitrary initial value (the closer to the actual root of *arg*, the better). In this case, we assign the initial value of *root* to be our real number *arg*. Unlike globals, strand state variables are mutable. In addition, strand methods may define local variables (the scoping rules are essentially the same as C's).

```
1  int{} args = load("numbers.nrrd");
2  int nArgs = length(args);
3  input real eps = 0.00001;
4
5  strand SqRoot (real arg)
6  {
7    output real root = arg;
8    update {
9      root = (root + arg/root) / 2.0;
10     if (|root^2 - arg| / arg < eps)
11       stabilize;
12     }
13 }
14
15 initially { SqRoot(args{i}) |
16                        i in 0 .. nArgs-1 };
```

Figure 1: A complete Diderot program that uses Heron's method to compute the square root of integers loaded from a file.

The **update** method of the `SqRoot` strand performs the approximation step of Heron's method (Line 9). The idea is that if *root* is an overestimation to the square root of *arg* then $\frac{arg}{root}$ will be an underestimate; therefore, the average of these two numbers provides a better approximation of the square root. In Line 10, we check to see if we achieved our desired accuracy as defined by `eps`, in which case we *stabilize* the strand (Line 11), which means the strand ceases to be updated.

The last part of a Diderot program is the initialization section, which is where the programmer specifies the initial set of strands in the computation. Diderot uses a comprehension syntax, similar to those of Haskell or Python, to define the initial set of strands. When the initial set of strands is specified as a collection, it implies that the program's output will be a one-dimension array of values; one for each stable strand. In this program, each square root strand will produce the approximate square root of an integer.

2.2 Execution Model

Our BSP execution model is shown in Figure 2. In this model, all active strands execute in parallel execution steps called *super-steps*. There are two main phases to a super-step: a strand update phase and a global computation phase.

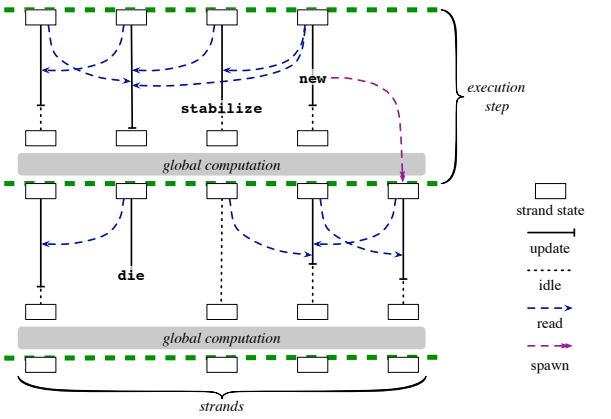


Figure 2: Illustrates two iterations of our current bulk synchronous model.

The strand update phase executes a strand's update method, which changes its state. During this phase, strands may read the state of other strands but cannot modify it. Strands see the states as they were at the *beginning* of a super-step. This property means we must maintain two copies of the strand state. One for strand reading purposes and one for updating a strand state during the execution of an update method. Also, strands can create new strands that will begin executing in the next super-step. The idle periods represent the time from when the strand finishes executing its update method to the end of the strand update phase. Stable strands remain idle for the entirety of its update phase. Dead strands are similar to stable strands where they remain idle during their update phase but also do not produce any output. Before the next super-step, an optional global computation phase is executed. Inside this phase, global variables can be updated with new values. In particular, global variables can be updated using common reduction operations. These updated variables can be used in the next super-step, but note they are immutable during the strand update phase. Finally, the program executes until all of the strands are either stable or dead.

2.3 Supporting Applications

Particle systems is a class of applications that greatly benefit from this updated BSP model. One example is an algorithm that distributes particles on implicit surfaces. Meyer uses a class of energy functions to help distribute particles on implicit surfaces within a locally adaptive framework [16]. The idea of the algorithm is to minimize the potential energy associated with particle interactions, which will distribute the particles on the implicit surface. Each particle creates a potential field, which is a function of the distances between the particle and its neighbors that lie within the potential field. The energy at each particle is defined to be the sum of the potentials of its interacting neighbors. The global energy of the system is then the sum of all the individual particle energies. The derivative of the global energy function produces a repulsive force that defines the necessary velocity direction. By moving each particle in the direction of the energy gradient, a global minimum is found when the particles are evenly spaced across the surface. The various steps within this algorithm require communication in two distinct ways: interactions between neighboring particles (*i.e.*, computing the energy at a particle is the sum of its neighbors' potentials) and interactions between all particles (*i.e.*, computing the global energy of the entire system). These distinctions motivate us to design a communication system that provides mechanisms for both local and global interactions. Strands need a way for only interacting with a subset of strands or with all the strands in the system.

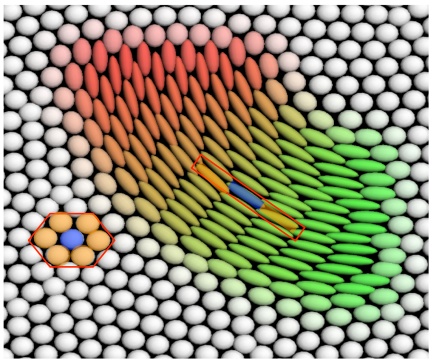


Figure 3: Glyph packing on synthetic data [15]. The red grids are specialized queries for retrieving neighboring glyphs. The orange glyphs represent the neighbors of the blue glyph performing the query.

Another design goal for inter-strand communication is to provide different ways of collecting nearby neighbors. For example, Kindlmann and Westin [15] uses a particle system to locate tensors at discrete points according to tensor field properties and visualize these points using tensor glyphs. Particles are distributed throughout the field by a dense packing method. The packing is calculated using the particle system where particles interactions are determined via a potential energy function derived from the tensor field. Since the potential energy of a particle is affected by its surrounding particles, neighboring particles can be chosen based on the current distribution of the particles. Figure 3 illustrates an example of using glyph packing on a synthetic dataset. We use this figure to show two possible ways in which neighbors are collected: hexagonal encapsulation and rectangular encapsulation. Both mechanisms only retrieve the particles defined within those geometric encapsulations and use queries that represent the current distribution in their area. In this example, using specialized queries can lead to a better representation on the underlying continuous features of the field. We need to provide various means of strand interac-

tion, where researchers have the option of choosing the best query that suits their algorithm or their underlying data.

3. Communication design

A major design goal of Diderot is to provide a programming notation that makes it easy for programmers to implement new algorithms. With the addition of our communication system, we want to continue that philosophy by designing these new features to ease the burden of developing programs. This section provides an overview of the design of the new communication system and examples of its syntax.

3.1 Spatial Communication

The idea behind spatial communication is shown in Figure 4. A Strand Q needs information about its neighboring strands. One way to retrieve Strand Q's neighbors is by encapsulating it within a spherical shape (the green circle) given a radius r. Any strand contained within this circle is returned to Strand Q as a collection of strands (*i.e.*, Strands A, B, and C). In Diderot, this process is done using predefined *query* functions. The queries are based on the strand's position in world space. Currently we only support this spherical or circle (in the 2D case) type of query, but plan to support various other type of queries, such as encapsulating the strand within a box. Once the collection is returned by the query, Strand Q can then retrieve state information from the collection of neighbors.

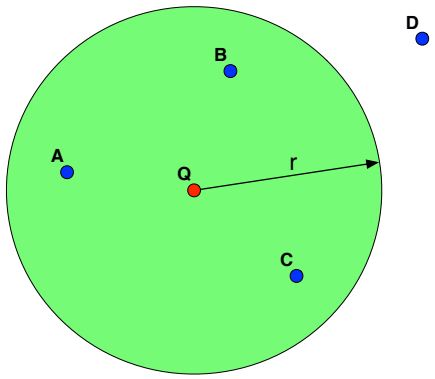


Figure 4: An example of showing a spherical query (*i.e.*, a circle in 2D). Strand Q (red) produces an encapsulating circle (green) with a predefined radius. Any strands within the encapsulating circle is returned to Strand Q. In this case, the query would return Strands A, B, and C.

Query functions produce a sequence of strand states. Using the strand state, a strand can then gain access to its neighbor's state variables. As mentioned earlier, queries are based on a strand's position in world space; therefore, the strand state needs to contain a state variable called *pos*. The position variable needs to be defined as a **`real`** `pos`, **`vec2`** `pos`, or **`vec3`** `pos`. The position of a strand is a dynamic value that can be updated during a super-step. Thus, query functions could return varying strand sequences in future super-steps if strands are moving through out the world.

Processing the queried sequence of strands is performed using a new Diderot mechanism called the **`foreach`** statement. The foreach statement is very similar to for statements used in many modern languages, such as Python and Java. An iteration variable will be assigned to each strand in the collection returned by the query function. During each iteration of the loop, a strand can use this iterator variable to gain access to its neighbor's state. Inside foreach block, strands can access the neighbor's state by using the selection operator (*i.e.*, the . symbol) followed by the name of the strand state variable and then the name of the field (similar to accessing a field in a struct variable in the C language).

```
1  real{} posns = load("positions.nrrd");
2  real{} energy = load("energies.nrrd");
3
4  strand Particle (real e, real x,real y) {
5    vec2 pos = [x,y];
6    real energy = e;
7    output real avgEnergy = 0.0;
8    update {
9      int neighborCount = 0;
10     real total = 0.0;
11     foreach(Particle p in sphere(10.0)){
12       neighborCount += 1;
13       total += p.energy;
14     }
15     avgEnergy = total/count;
16     stabilize;
17   }
18 }
```

Figure 5: A snippet of code demonstrating the spatial communication mechanism.

An example of using spatial communication is shown in Figure 5. This code snippet calculates the average energy of all the neighbors for each particle. The code loads positions and energies from a file (Lines 1-2) and assigns (Lines 5-6) each particle (*i.e.*, a strand) a position and energy. It declares accumulator variables (Line 9-10) to hold the total energy and count of the neighbors. The foreach statement (Line 11) declares variable *p* with its typing being the strand name. The program uses a spherical query with a given radius (*e.g.*, the value of 10 in this example) to retrieve the collection of neighbors. Each neighbor will then be assigned to *p* for an iteration of the loop. The accumulator variable, neighborCount, is incremented for each neighbor within the collection (Line 12). We use the selection operator (Line 13) to retrieve the energy state variable and added it to the total energy. Finally, we use the accumulator variables (Line 15) to calculate the final output, *avgEnergy*.

3.2 Global Communication

As mentioned earlier, strands may want to interact with a subset of strands, where they may not be spatially close to one another. This mechanism can be seen as allowing strand state information to flow through a group of strands to produce a result that gives insightful information about the entire state of a group. Once the result is computed, a strand can then use it as a way updating their state during the update phase or updating a global property (*i.e.*, a global variable) of the system. We call this flow of information global communication. This feature is performed by using common reduction operations, such as **`product`** or **`sum`**.

Figure 6 shows the syntax of a global reduction in Diderot. *r* represents the name of the reduction. Table 1 provides the names of the reductions and a detailed description of their semantics. The expression e has strand scope. Only strand state and global variables can be used in e along with other basic expression operations (*e.g.*, arithmetic or relational operators). The variable *x* is assigned to each strand state within the set *t*. This variable can then be used within the expression e to gain access to state variables. The set *t* can contain either all active strands, all stable strands, or both active and stable strands for an iteration. *N* represents the name of the strand definition in the program. Syntactically, if a **`strand`** P was defined as the strand definition name then a program can retrieve the sets as follows: *P.active* (all active Ps), *P.stable* (all stable Ps), or *P.all* (all active and stable Ps).

$$
\begin{array}{rcll}
t & ::= & N.\texttt{all} & \text{strand sets} \\
 & | & N.\texttt{active} & \\
 & | & N.\texttt{stable} & \\
e & ::= & \ldots & \text{previously defined expressions} \\
 & | & r\{e \mid x \ in \ t\} & \text{reduction expression}
\end{array}
$$

Figure 6: Syntax of a global reduction

The reduction operations reside in a new definition block called **`global`**. Global reductions can only be assigned to local and global variables in this block. Reductions are not permitted to be used inside the strand definition. Before the addition of the global block, global variables were immutable but now they can be modified within this block.

A trivial program that finds the maximum energy of all active and stable strands is shown in Figure 7. Similar to the spatial code, this code loads energies from a file (Line 1) and assigns each strand an energy value (Line 6). The program also defines a global variable *maxEnergy* (Line 3) that holds the maximum energy of all the strands. Inside the global block (Lines 16-18), the **`max`** reduction is used to find the maximum of the energy state variable. The reduction scans through each *P.energy* value from all active and stable strands and assigns *maxEnergy* the highest value. Inside the update method (Lines 7-9), the strand with the maximum energy prints it out and all strands stabilize. An important aspect to note about this particular code is the need for the *iter* global variable. Remember, the first evaluation of the global computation phase is only after the first execution of the strand update phase. Thus, *maxEnergy* will not have the actual maximum energy until the second iteration. This caveat is why the strands stabilize and the maximum energy is printed when *iter* reaches two.

Table 1: The meanings of reduction operations

Reduction	Semantics	Identity
`all`	For each strand, *S*, in the set t compute the value e and return TRUE if all the e values in t evaluate to TRUE, otherwise FALSE.	true
`max`	For each strand, *S*, in the set t compute the value e and return the maximum e value from the set s.	$-\infty$
`min`	For each strand, *S*, in the set t compute the value e and return the minimum e value from the set t.	$+\infty$
`exists`	For each strand, *S*, in the set t compute the value e and return TRUE if any one of the e values in t evaluate to TRUE, otherwise FALSE.	false
`product`	For each strand, *S*, in the set t compute the value e and return the product of all the e values in t.	1
`sum`	For each strand, *S*, in the set t compute the value e and return the sum of all the e values in t.	0
`mean`	For each strand, *S*, in the set t compute the value e and return the mean of all the e values in t.	0
`variance`	For each strand, *S*, in the set t compute the value e and return the variance of the e values in t.	0

3.3 Dynamic Strand Allocation

Diderot does allow the number of strands to vary dynamically, via its die statement, but there is no way to increase the number of active strands in a program. New strands can be allocated during the strand update phase by using the **`new`** statement within the **`update`** method. Using the new statement:

```
new P (arguments);
```

where *P* is the name of the strand definition, a new strand is created and initialized from the given *arguments*. Figure 8 shows a snippet of code that uses the new statement. If a

```
int{} energies = load("energies.nrrd");
int nEnergies= length(energies);
real  maxEnergy = -∞;
int iter = 1;
strand Particle (real initEnergy) {
  real energy = initEnergy;
  update {
    if(iter == 2 && maxEnergy == energy)
      print(maxEnergy);
    if(iter == 2)
      stabilize;
  }
}

global {
  iter += 1;
  maxEnergy = max{P.energy |
                  P in Particle.all};
}
initially [ Particle(energies(vi)) |
            vi in 0..(nEnergies-1)];
```

Figure 7: Retrieves the maximum energy of all active and stable strands in the program.

strand is wandering alone in the world then it may want to create more strands in its neighborhood. In this case, new strands are created by using the strand's position plus some desired distance from the strand.

```
real d =  5.0;
// desired distance of a new particle.
...
strand Particle (real x, real y) {
  vec2 pos = [x,y];
  update {
    int count = 0;
    foreach(Particle p in sphere(10.0)){
      count = count + 1;
       ...
    }
    if(count < 1) {
      new Particle(pos[0] + d, pos[1] + d);
    }
    ...
  }
}
```

Figure 8: A snippet of code showing how new strands are created when there are no surrounding neighbors.

4. Implementation

The addition of strand communication to the Diderot compiler produced minimal changes for its front end and intermediate representations. But we added a significant amount of code to our runtime system. In particular, implementing the spatial scheme for spatial communication and determining an efficient way of executing global reductions was required. In this section, we give a brief overview of the Diderot compiler and runtime system and discuss the implementations details of the communication system.

4.1 Compiler Overview

The Diderot compiler is a multi-pass compiler that handles parsing, type checking, multiple optimization passes and code generation [5]. A large portion of the compiler deals with translating from the high-level mathematical surface language to efficient target code. This process occurs over a series of three intermediate representations (IRs), ranging from a high-level IR that supports the abstractions of fields and derivatives to a low-level language that decomposes these abstractions to vectorized C code. Also at each IR level, we perform a series of traditional optimizations, such as unused-variable elimination and redundant-computation elimination using value numbering [4]. The code the compiler produces depends on the target specified. We have separate backends for various targets: sequential C code with vector extensions [10], parallel C code, OpenCL [14]. Because these targets are all block-structured languages, the code generation phase converts the lowest IR into a block-structured AST. The target-specific backends translate this representation into the appropriate representation and augment the code with type definitions and runtime support. The generated code is then passed to the host system's compiler.

4.2 Runtime Targets

The runtime system implements the Diderot execution model on the specified target and provides additional supporting functions. For a given program, the runtime system combines target-specific library code and specialized code produced by the compiler. We current support three versions of the runtime system:

4.2.1 Sequential C

The runtime implements this target as a loop nest, with the outer loop iterating once per super-step and the inner loop iterating once per strand. This execution is done on a single-processor and vectorized operations.

4.2.2 Parallel C

The parallel version of the runtime is implemented using the Pthreads API. The system creates a collection of worker threads (the default is one per hardware core/processor) and manages a work-list of strands. To keep synchronization overhead low, the strands in the work-list are organized into blocks of strands (currently 4096 strands per block). During a super-step, each worker grabs and updates strands until the work-list is empty. Barrier synchronization is used to coordinate the threads at the end of a super step. Note,

however, that the compiler will omit barrier synchronization when it is not required (*e.g.*, when the program does not use strand communication), which results in better performance.

4.2.3 GPUs

Lastly, the GPU runtime is implemented using the OpenCL API. In OpenCL, work items (*i.e.*, threads) are separated into workgroups and execution is done by *warps* (*i.e.*, 32 or 64 threads running a single instruction). Similar to the parallel C runtime, strands are organized into blocks. Each strand block contains a warp's worth of strands for execution. Instead of the runtime using the GPU scheduler, the system implements the idea of *persistent thread* to manage workers [12]. We use multiple workers per compute unit to hide memory latency. Currently, the communication system is implemented only for the sequential and parallel C targets. We plan to add communication support for GPUs in the future.

4.3 Spatial Execution

When choosing a spatial scheme, it is important to consider how it affects a program's performance. For instance, if a Strand Q queried for its neighbors using a spherical query then a naive implementation would sequentially perform pairwise tests with the rest of the strand population to determine if a strand lies within the sphere's radius. For n strands, this requires: $O(n^2)$ pairwise tests [8]. A Diderot program can contain thousands of active strands at any given super step. This scheme can become too expensive even with a moderate number of strands due to the quadratic complexity. We use a tree-based spatial partitioning scheme, specifically, k-d trees [1][11]. A k-d tree allows one to divide a space along one dimension at a time. We use the traditional approach of splitting along x, y, and z in a cyclic fashion. With this spatial scheme, nearest neighbor searches can be done more efficiently and quickly because we are searching smaller regions versus the entire system. This process thereby reduces the number of comparisons needed.

As stated earlier, the position state variable of a strand is used for constructing the spatial tree. We use a parallel version of the medians of medians algorithm [3] to select the splitting value. As the tree is built, we cycle through the axes and use the median value as the splitting value for a particular axis. A strand's position can potentially change during the execution of the update method, or new strands can be created during an update. Thus, the tree is rebuilt before the beginning of the update method to take into account these changes. To improve performance, this reconstruction process is done in parallel.

4.4 Global Reductions

As stated previously, the global reductions reside in the new global block. This block of code allows for the modification of global variables and is executed at the end of the superstep. The parallel C target uses one of its threads to execute the block in a sequential fashion with the exception of reductions. The process of executing reductions is described below:

4.4.1 Reduction Phases

The reductions inside the global block are executed in parallel phases. Each reduction is assigned and grouped into execution phases with other reductions. After parsing and type checking, a typed AST is produced and converted into a simplified representation, where temporaries are introduced for intermediate values and operators are applied only to variables. It is during this stage of the compiler reductions are assigned their phase group. The phase in which a reduction is assigned is dependent on whether an another reduction is used to calculate the final value for a global variable. Figure 9 shows an example on how reductions are grouped. Initially, each global variable assigned to a reduction will automatically begin in phase 0 and added to a hash-map that contains all global reduction variables. If the right hand side (rhs) expression contains other global reduction variables the phase assigned to the variable will be: $1 + \theta$, where θ is the highest phase among the rhs global reduction variables. Spitting reductions into phases is important for increasing performance. Calculating a reduction in parallel occurs a large amount of overhead for running and synchronizing threads. Thus, reductions that can be executed in parallel with other reductions reduces this overhead.

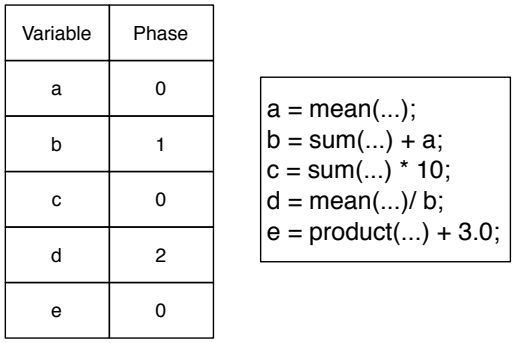


Figure 9: Reduction variables are assigned into phase groups for execution. A phase group for a variable is incremented depending on whether other reduction variables reside in the same expression. This process greatly reduces the amount of overhead of performing a reduction versus individual execution.

4.4.2 Reduction Lifting

Also during phase identification, we perform an operation called reduction lifting. This process is shown in Figure 9. Each reduction expression is replaced with a temporary variable and lifted into a new block called the *reduction block*. Lifting reductions simplifies the grouping of reductions into their correct phase groups during code generation.

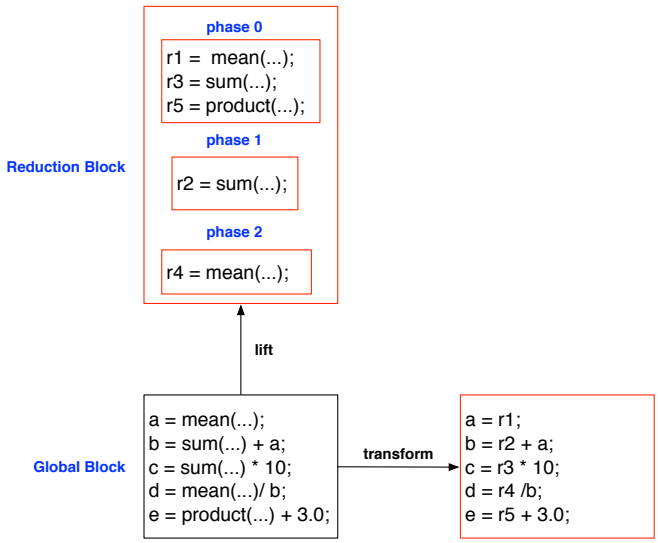


Figure 10: Reduction lifting makes it easier to group reductions when generating target code. Each reduction is replaced with a new variable in the expression and is lifted into a special code block for reductions.

4.4.3 Phase Execution

The code generation phase breaks the reductions into their assigned phases. This phase also determines at what time to execute a particular phase. The assignments inside the global block are scanned for reduction variable usage. If a reduction variable is used on the rhs then we look up its phase group and insert a phase execution call before the assignment as shown in Figure 9. The phase execution call is only needed before the first occurrence of any reduction variable in that particular phase.

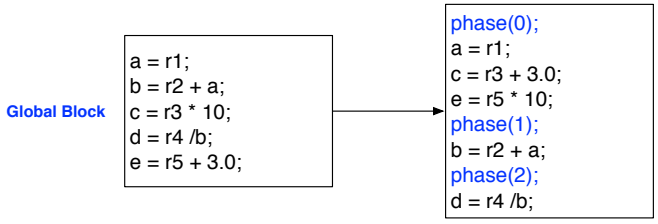


Figure 11: Phase execution calls are inserted before the execution of the assignment that uses the reduction.

4.5 Allocating Strands

Diderot maintains a contiguous chuck of memory that represents the strand states for a program. During world initialization (i.e., before the first update method call), We allocate this chunk of memory with enough space to allocate the initial set of strands along with an additional amount of strands that will remain uninitialized. Remember, worker threads process blocks of strands during a super-step. If a worker thread needs to allocate a new strand then it can request an uninitialized strand from that block memory, which it then can initialize once received.

5. Related Work

The work presented in this paper is novel to the area of visualization and image analysis languages. Currently, we are unaware of any other languages that provide the spatial query mechanism that is directly built into the language itself. Although, the concepts of spatial and global communication are studied in various other research fields.

The ideas behind spatial communication in Diderot was influenced by previous works that use agent-based models [13]. These models use simulations based on local interactions of agents in an environment. These agents can represent variety of different objects such as: plants, animals, or autonomous characters in games. With regards to spatial communication, we explored models that are spatially explicit in their nature (*i.e.*, agents are associated with a location in geometric space).

Craig Reynolds's boids simulation is an example of an spatially explicit environment [19]. This algorithm simulates the flocking behavior of various species. Flocking comes from the practice of birds flying or foraging together in a group. A flock is similar to groups of other animals, such as the swarming of insects. Reynolds developed a program called Boids that simulates local agents (*i.e.*, boids) that move according to three simple rules: separation, alignment, and cohesion. The boids simulation influenced our spatial communication design in regards to how it retrieved its neighbors. When boids are searching for neighboring boids, they are essentially performing a query similar to our queries in Diderot. In particular, they are performing a circle query that encapsulates the querying boid and any nearby boids bounded within a circle. However, our query performs much faster because we use tree-based scheme to retrieve neighbors, while Reynold's query runs in $O(n^2)$ because it requires each boid to be pairwise tested with all other boids.

Agent interactions have also been modeled using interprocess communication systems [17] and computer networks [22]. In these models, processes or nodes can send and receive messages (*i.e.*, data or complex data structures that represent tasks) to other processes. Once agents are mapped to processes, they then can use the messaging protocol defined by the system to query about nearby agents or exchange state information with each other. However, this process requires an application to explicitly adapt or layer its spatial communication model to work within these systems. In Diderot, there is only an implicit notion that strands are defined within a geometric space and one uses query functions to retrieve state information of neighboring strands, which differs from the layering requirement needed for these other communication systems.

The execution of global reductions in Diderot is similar to the MapReduce model developed by Dean and Ghemawat [6]. MapReduce is a programming model used to process parallelize problems that use large data sets, where data mapped into smaller sub-problems and is collected together to reduce to a final output. Many data parallel languages have supported parallel map-reduce, such as the NESL language [2], which has also influenced our design of global reductions. The global block allows reductions to be mapped to global variables. After each super-step, the global phase (*i.e.*, our "reduce" step) executes the global block, which performs the actual computation for each reduction. Programmers do not need to worry about lifting the reductions into their own phase because this process is handled by the Diderot compiler.

6. Discussion and Future Work

The original design and implementation of Diderot was limited in that it only supported computations involving completely autonomous strands [5]. This limitation excluded algorithms of interest, such as particle systems. These algorithms require additional communication mechanism to be supported by the language. With the addition of spatial communication and global reductions we have given programmers the ability to implement and explore more applications with the Diderot language. Although, there are few areas in we plan to improve and work on in the future to provide a better communication system.

Currently we only support a limited number of query functions. We plan to provide additional query functions such as ellipsoidal and hexagonal, to bring more diversity to the options researchers can use within their algorithms. We also are exploring the ability to support abstraction spatial relationships. For example, defining a query to retrieve the 26-neighbors in a 3D grid, or support mesh based methods, where a strand corresponds to a triangle in a finite-element mesh. Query functions are the basis behind spatial communication in Diderot, so allowing for various query options gives a larger range in the types of algorithms we can support.

The new BSP communication mechanism poses a difficult implementation challenge for our GPU target. GPUs are not as flexible in terms of allocating memory dynamically, which can only be done on the host side device. This restriction means that we have to produce a scheme for managing memory efficiently. This scheme needs to determine the appropriate times to dynamically allocate more memory, which can incur a large overhead cost if done naively. With having various components of a Diderot program being allocated on the GPU (*i.e.*, strand state information, spatial tree information, GPU scheduler information, and potential image data), we can potentially run out of memory on the device. If this happens then we may need to come up with a scheme that offloads certain components to the CPU and load only the data that is need for a given iteration. These complications need to be considered when implementing strand communication on the GPU.

Acknowledgments

Portions of this research were supported by the National Science Foundation under award CCF-1446412. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

This research was also previously presented at the Compilers for Parallel Computing Workshop in January 2015 at Imperial College, London, UK. The workshop did not include a published proceedings but rather it was a workshop for international researchers coming together to present their research ideas.

References

[1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975. ISSN 0001-0782. . URL http://doi.acm.org/10.1145/361002.361007.

[2] G. E. Blelloch. NESL: A nested data-parallel language. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[3] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, Aug. 1973. ISSN 0022-0000. . URL http://dx.doi.org/10.1016/S0022-0000(73)80033-9.

[4] P. Briggs, K. D. Cooper, , and L. T. Simpson. Value numbering. *Software – Practice and Experience*, 27(6):701–724, June 1997.

[5] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: A parallel DSL for image analysis and visualization. In *Proceedings of the 2012 SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*, pages 111–120, New York, NY, June 2012. ACM.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51 (1):107–113, Jan. 2008.

[7] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pages 65–74, New York, NY, USA, 1988. ACM. ISBN 0-89791-275-6. . URL http://doi.acm.org/10.1145/54852.378484.

[8] C. Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. ISBN 1558607323.

[9] A. Filler. The history, development and impact of computed imaging in neurological diagnosis and neurosurgery:

CT, MRI, and DTI. *Nature Precedings*, 2009. URL http://dx.doi.org/10.1038/npre.2009.3267.5.

[10] *Using vector instructions through built-in functions*. Free Software Foundation. URL http://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html.

[11] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, Sept. 1977. ISSN 0098-3500. . URL http://doi.acm.org/10.1145/355744.355745.

[12] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *Inovative Parallel Computing (InPar '12)*, May 2012.

[13] N. R. Jennings. Agent-based computing: Promise and perils. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, IJCAI '99, pages 1429–1436, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-613-0. URL http://dl.acm.org/citation.cfm?id=646307.687432.

[14] *The OpenCL Specification (Version 1.1)*. Khronos OpenCL Working Group, 2010. Available from http://www.khronos.org/opencl.

[15] G. Kindlmann and C.-F. Westin. Diffusion tensor visualization with glyph packing. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1329–1336, Sept. 2006. ISSN 1077-2626. . URL http://dx.doi.org/10.1109/TVCG.2006.134.

[16] M. D. Meyer. Robust particle systems for curvature dependent sampling of implicit surfaces. In *In SMI 05: Proceedings of the International Conference on Shape Modeling and Applications 2005 (SMI 05*, pages 124–133. IEEE Computer Society, 2005.

[17] MPS. The message passing interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpi/. Accessed: 20/03/2013.

[18] *NVIDIA CUDA C Programming Guide (Version 4.0)*. NVIDIA, May 2011. Available from http://developer.nvidia.com/category/zone/cuda-zone.

[19] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, Aug. 1987. ISSN 0097-8930. . URL http://doi.acm.org/10.1145/37402.37406.

[20] D. Skillicorn, J. M. Hill, and W. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[21] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[22] D. C. Walden. A system for interprocess communication in a resource sharing computer network. *Commun. ACM*, 15 (4):221–230, Apr. 1972. ISSN 0001-0782. . URL http://doi.acm.org/10.1145/361284.361288.

A Performance and Scalability Analysis of Actor Message Passing and Migration in SALSA Lite

Travis Desell
University of North Dakota
3950 Campus Road Stop 9015
Grand Forks, ND 58203, USA
tdesell@cs.und.edu

Carlos A. Varela
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180, USA
cvarela@cs.rpi.edu

ABSTRACT

This paper presents a newly developed implementation of remote message passing, remote actor creation and actor migration in SALSA Lite. The new runtime and protocols are implemented using SALSA Lite's lightweight actors and asynchronous message passing, and provide significant performance improvements over SALSA version 1.1.5. Actors in SALSA Lite can now be *local*, the default lightweight actor implementation; *remote*, actors which can be referenced remotely and send remote messages, but cannot migrate; or *mobile*, actors that can be remotely referenced, send remote messages and migrate to different locations. Remote message passing in SALSA Lite is twice as fast, actor migration is over 17 times as fast, and remote actor creation is two orders of magnitude faster. Two new benchmarks for remote message passing and migration show this implementation has strong scalability in terms of concurrent actor message passing and migration. The costs of using remote and mobile actors are also investigated. For local message passing, remote actors resulted in no overhead, and mobile actors resulted in 30% overhead. Local creation of remote and mobile actors was more expensive with 54% overhead for remote actors and 438% for mobile actors. In distributed scenarios, creating mobile actors remotely was only 6% slower than creating remote actors remotely, and passing messages between mobile actors on different theaters was only 5.55% slower than passing messages between remote actors. These results highlight the benefits of our approach in implementing the distributed runtime over a core set of efficient lightweight actors, as well as provide insights into the costs of implementing remote message passing and actor mobility.

Keywords

Actor model, Lightweight Actors, Distributed Actor Benchmarks, Distributed Computing, Actor Migration

1. INTRODUCTION

AGERE! 2015 Pittsburgh, Pennsylvania, USA

As programming environments continue to increase in parallelism in terms of numbers of processors and cores, the need for efficient and effective concurrent and distributed programming languages becomes ever more important. In many ways, the common method of using object, threads and communication over synchronous sockets is not well suited to these environments, as evidenced by the large body of work on detecting, preventing and avoiding deadlocks and other race conditions [23, 28, 18, 49, 27, 31, 16, 1, 26, 17]. Many of these issues arise due to the fact that objects do not encapsulate their state, so member fields must be protected with mutexes or other blocking synchronization constructs to prevent concurrent access. This blocking behavior, coupled with the blocking behavior of using sockets synchronously makes it quite easy for deadlocks and other race conditions to occur.

As threads move from object to object, without any programmatic way of knowing where they came from, in many ways they present a harmful situation similar to the much maligned `GOTO` statement [15, 42]. As Dijkstra eloquently stated, *"The unbridled use of the* **go to** *statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress."* In many ways, the unbridled use of threads presents a similar situation where it becomes terribly hard to find a meaningful set of coordinates in which to describe a *threads* progress.

The Actor model, formalized over 40 years ago [22, 21] and later extended to open distributed systems [2], provides a strong alternative model without these pitfalls. Actors are independent, concurrent entities that communicate by exchanging messages. Each actor encapsulates a state with a logical thread of control which manipulates it. Communication between actors is purely asynchronous. The actor model assumes guaranteed message delivery and fair scheduling of computation. Actors only process information in reaction to messages. While processing a message, an actor can carry out any of three basic operations: altering its state, creating new actors, or sending messages to other actors (see Figure 1). Actors are therefore inherently independent, concurrent and autonomous which enables efficiency in parallel execution [32] and facilitates mobility [3, 44]. In the actor model, a thread of control only operates on a single actor, and activity passes through the system via asynchronous messages, which have sources and targets that enable an easier understanding of program flow. This model, if strictly adhered to, actually makes it challenging to program a system which deadlocks and completely prevents concurrent

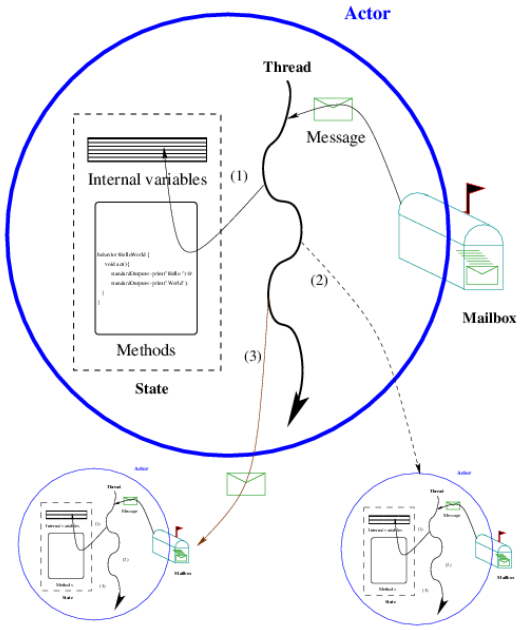


Figure 1: Actors are reactive entities. In response to a message, an actor can (1) change its internal state, (2) create new actors, and/or (3) send messages to other actors (image from [43]).

memory access issues.

Many of the early implementations of actor languages involved heavyweight actors, such as Erlang [7], Scala [19] and SALSA [44, 4], where each actor had an actual thread of control. This type of actor has a fair bit of overhead over objects, and also has low limits on how many actors can exist imposed by the operating system and hardware on how many threads are allowed. Because of this, languages like SALSA, Scala, and Kilim [38] allow the use of both objects and actors, which can unfortunately lead to potential violations of the actor model [14] and the use of a mixture of concurrency models [40].

In part due to these issues, and in part due to a desire to seek higher levels of performance in actor languages, systems using lightweight actors have been developed and are of high interest. In Java based actor libraries and languages, Kilim utilizes lightweight threads [38] and Scala also allows actors to run using a thread pool [19] or as lightweight actors using the Akka framework [5, 9]. Charm++ [29] provides a lightweight actor inspired library based on C++ for use in cluster computing environments, and more recently libcppa [12], has evolved into the C++ Actor Framework (CAF) which utilizes lightweight actors and also enables GPGPU computing [10, 11].

SALSA Lite follows in this path by rebuilding SALSA from the ground up using lightweight actors with a strong emphasis on performance. Previous work has evaluated the performance of SALSA Lite in non-distributed concurrent settings [14]. This work presents how these lightweight actors have been used to develop an efficient distributed computing runtime that enables both remote and mobile actors. Few actor languages and libraries support transparent distribution of actors, and even fewer support mobility. To the authors' knowledge, only the ActorFoundry [8] (based on Kilim), ActorNet [34, 33] (an actor platform for wireless sensor networks), JavAct [6], Actor Architecture [25], SALSA and now SALSA Lite provide transparent actor mobility [30].

2. APPROACH

The design philosophy of SALSA Lite is in essence to practice what we preach in regards to the benefits of the Actor model, *i.e.*, the language should be built using the Actor model as opposed to objects and threads. Further, to borrow from Unix, the common case should be executed fast. As such, SALSA Lite has been rebuilt from the ground up to provide a core of lightweight actors which can send messages and be created extremely quickly. Using this simple efficient core, language semantics and runtime services are then built using these actors as opposed to objects and threads. Other results have shown the efficiency of SALSA Lite actors in concurrent non-distributed settings [14, 11], providing justification for using them as a foundation for the language.

This lightweight actor implementation has been built in a novel way based on hashing to eliminate any synchronization bottlenecks [14]. For example, if actors or messages need a unique identifier, a common way to generate that is to take the host, port and start time of the theater and append a counter. However, that counter needs to be accessed atomically which makes it a singular point of synchronization. This can lead to applications which appear concurrent, but actually are operating sequentially based on those synchronization bottlenecks. SALSA Lite avoids this by using the hashcode of the actor requesting a service and selecting one of N copies of a service by the hashcode modulo the number of services. Collisions are not an issue as mutliple actors are expected to use the limited number of N services. The number of the various services can be specified at runtime, allowing SALSA Lite applications to easily scale to the number of cores available.

Instead of developing a runtime and services for remote and mobile actors using objects and threads, as done in the previous implementation of SALSA, the lightweight core of actors has been used to develop a distributed computing environment which utilizes the Actor model to eliminate deadlocks and achieve high levels of concurrency and performance. Lastly, as SALSA Lite is being developed with performance in mind, we have decided to allow programmers to specify if an actor will be local, remote or mobile, as adding this functionality does come at a cost. This allows programmers to use the type of actor with the best performance for the task at hand.

3. IMPLEMENTATION

SALSA Lite's runtime is based on the concept of *stages* (see Figure 2), which essentially act as a unified mailbox and thread of control for groups of actors assigned to them. This allows local actors to be implemented as simple Java objects with only a reference to the stage they are "performing" on. When a message is sent to an actor, the message is placed in its stage's mailbox using this reference and the stage's `putMessageInMailbox` method. While to the authors' knowledge, SALSA Lite is the only actor langauge to utilize this type of runtime, however others such as E's

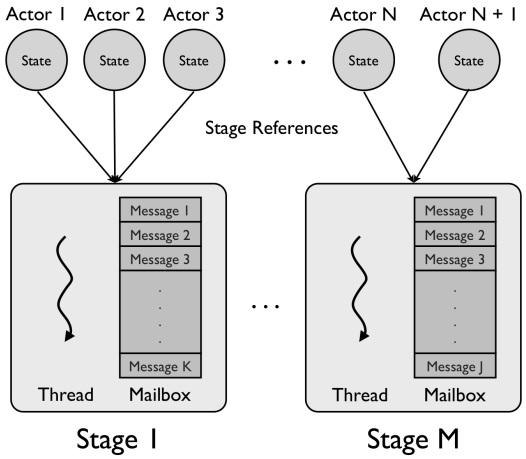


Figure 2: The SALSA Lite runtime environment. Heavyweight actors called *stages* are used to process messages on multiple lightweight actors, simulating their concurrent execution. A stage will repeatedly get the first message from its mailbox and process that message on the message's target actor. Every actor is assigned to a stage. A Message sent to an actor is placed at the end of its assigned stage's mailbox (image from [14]).

```
1: //Create a remote actor at the local theater
2: MyRemoteActor a = new MyRemoteActor()
3:     called ("a");

4: //Create a remote actor at a remote theater
5: MyRemoteActor b = new MyRemoteActor()
6:     called ("b") at (host, port);

7: //Create a name server
8: NameServer ns = new NameServer()
9:     called ("my_nameserver");

10: //Create a mobile actor at the local theater
11: MyMobileActor c = new MyMobileActor()
12:     called ("c") using (ns);

13: //Create a mobile actor at a remote theater
14: MyMobileActor c = new MyMobileActor()
15:     called ("c") using (ns)
16:     at (host, port);
```

Figure 3: SALSA Lite has simplified syntax for creating remote and mobile actors. Unique names are specified with the `called` keyword, the host and port of the theater the actor is created on are specified with the `at` keyword, and the name server actor a mobile actor is registered at is specified with the `using` keyword.

vats [36, 35] and SEDA [48, 37] use a similar approach for high efficiency. With this design in mind, remote and mobile actors were implemented in a way to not impact or degrade the performance of these local actors.

3.1 Theaters

Distributed computing in SALSA is done using the concept of *theaters*. Each theater serves as a separate process in which multiple actors perform on a set of stages. The number of stages in a theater can be dynamically specified at runtime, and actors can either be automatically assigned to stages, have new stages generated for them, or be assigned to particular stages programatically (see [14] for further details). Theaters listen on a particular port for incoming connections to other theaters which can be distributed over local area networks or the Internet.

Each theater has a `TheaterActor` repeatedly listening for incoming socket connections. When one occurs, it spawns a `IncomingTheaterConnection` actor who handles receiving messages and migrating actors from that theater. Messages and migrating actors are sent via a theater's `TransportService` which has a set of static methods which put messages in the appropriate `OutgoingTheaterConnection` mailboxes and create new `OutgoingTheaterConnection` actors when necessary. The `TheaterActor`, `IncomingTheaterConnection` and `OutgoingTheaterConnection` actors are all heavyweight, each running on their own stage as to not block the execution of other actors when they are blocked listening for connections or waiting to receive data over a socket.

The `IncomingTheaterConnection` actors put messages in the appropriate stage's mailbox as they are received. Making sure references to actors are correctly kept as messages are serialized and de-serialized is described in Section 3.2.1.

The `OutgoingTheaterConnection` actors simply send messages and actors across a socket to the `IncomingTheaterConnection` actor they are paired with. As described in Section 3.3.1, implementing these services as actors also allows for the easy implementation of a protocol to update remote references to mobile actors as they migrate around a set of theaters.

3.2 Remote Actors

As remote actors do not migrate, it is always the case that a reference to a remote actor refers to the actual actor when it is present at the same theater, or that it is a reference to a remote actor on another theater. In the first case, implementation of remote actors is identical to that of a local actor, with the exception that the remote actor needs a unique name so that it can be referred to and looked up by other actors. Figure 3 presents the syntax for creating the various types of actors in SALSA Lite and Figure 4 presents the syntax for generating references to actors using their name and location. The remote actor also needs to be added to a `RemoteActorRegistry` which is a `HashMap` of names to the actual remote actor lightweight actor object, so incoming messages can be directed towards it accordingly. This adds some overhead to the creation of a remote actor, while sending messages to it locally can be performed the same as with local actors as the other actors simply have a reference to the actual remote actor.

In the second case, where it is a reference to a remote actor on another theater, when a message is invoked on that reference (implemented as a local actor), it instead uses SALSA Lite's transport service to put it in the mailbox of the appropriate `OutgoingTheaterConnection` actor, which sends it over a socket to the appropriate theater. When the mes-

```
1: //Reference a remote actor at the local theater
2: MyRemoteActor a = reference
3:     MyRemoteActor called ("a");

4: //Reference a remote actor at a remote theater
5: MyRemoteActor b = reference
6:     MyRemoteActor called ("b")
7:     at (host, port);

8: //Get a reference to a remote name server
9: NameServer ns = reference NameServer
10:    called ("my_nameserver")
11:    at (host, port);

12: //Reference a mobile actor registered at
13: //that name server
14: token MyMobileActor c = reference
15:    MyMobileActor called ("c")
16:    using (ns);
```

Figure 4: SALSA Lite has also simplified syntax for referencing remote and mobile actors. Instead of the `new` keyword, the `reference` keyword is used. The actor's names are specified with the `called` keyword, the host and port of the theater the actor is created on are specified with the `at` keyword, and the name server actor a mobile actor is registered at is specified with the `using` keyword.

sage is received by the target theater's `IncomingTheaterConnection`, the actual remote actor is looked up in the `RemoteActorRegistry` and the message is sent to it.

3.2.1 Actor Reference Propagation

Some challenges arise in that a message sent to a remote actor on another theater can contain references to other local or remote actors. If these messages were blindly serialized while being sent to the other theater, this would result in unintended copies of these actors. To overcome this, SALSA Lite uses Java's `readResolve()` and `writeReplace()` methods instead of default object serialization. When a local or mobile actor is serialized, its `writeReplace()` method is called, which creates a serialized reference only containing the hashcode, host and port in the case of a local actor, or the unique name, host and port in the case of a remote actor. For local actors, a reference to the local actor is also placed in a `LocalActorRegistry` so it can be looked up if messages are sent to it from another theater. This also can drastically reduce the size of the messages being sent as only the minimum amount of data required to lookup the actor or generate a reference is sent. It should be noted that the implementation of local actors has remained completely unchanged, apart from now providing `readResolve()` and `writeReplace()` methods for serialization when references to them propagate to remote theaters.

When the serialized reference is received by a theater, the `readResolve()` method is called on the serialized reference. This performs a lookup in either the `LocalActorRegistry` or `RemoteActorRegistry`. If the actor is present, the `readResolve()` method returns the actual reference to that actor, otherwise it returns a remote reference object which sends messages to the `OutgoingTheaterConnection` actor instead of actually processing them. This prevents copies of actors from occurring, and also ensures that there is only one remote reference to an actor at any one theater (which will aid in implementing distributed garbage collection). These registries have been implemented using the hashing strategy described in Section 2, so multiple copies can be made which are selected by the actor's hashcode, preventing the registries from acting as a singular bottleneck.

3.3 Mobile Actors

Unlike local and remote actors, there are significant challenges in implementing mobile actors as a single object (either as a remote reference or the actual actor), as migration would then involve having to update the references to it held by all other actors every time it migrates. Keeping track of these reverse references can lead to significant memory and performance overhead. Similar to how actors are implemented in SALSA, in SALSA Lite, mobile actors are divided into reference and state objects. When a mobile actor is created, its state is placed in a `MobileActorStateRegistry`, which is the only object with a reference to the actor's state. The reference acts in the place of a lightweight actor on a stage. When a message is invoked on the reference, it performs a lookup in the `MobileActorStateRegistry` which invokes the message on the state if the actor is present. When the actor migrates, the state object is put in a message to the `OutgoingTheaterConnection` it is being sent over, and the state object is removed from the `MobileActorRegistry` and is replaced with a reference to the `OutgoingTheaterConnection` actor that sends messages to the theater the actor migrated to. If the lookup returns the connection, the message is placed in the `OutgoingTheaterConnection`'s mailbox to be sent to that theater. In this way, the only time the mobile actor's state is serialized is when it migrates.

This also allows references to mobile actors to propagate in a manner similar to local and remote actors. This propagation is handled the same way by using `readResolve()`, `writeReplace()`, a serialized reference and `MobileActorReferenceRegistries`.

Note that every time a message is invoked on a mobile actor, a lookup in a `MobileActorStateRegistry` needs to be performed, which adds overhead to message passing.

3.3.1 Finding Mobile Actors

In addition to mobile actors requiring a unique name, host and port in their reference and state, mobile actors also need to be registered at a *name server*. The name server is used as a lookup service for getting a reference to a mobile actor (see Figure 4). When the mobile actor is created, it sends a `PUT` message asynchronously to the name server it will be registered at. When the actor migrates, it sends an asynchronous `UPDATE` message to the name server, which updates its location on the name server. When another actor wants to get a reference to a mobile actor, it can contact the name server with a `GET` message which will return a reference to that actor. This is done transparently when the `reference` keyword is used.

In contrast with SALSA, where name servers are run as standalone daemons, in SALSA Lite, name servers are implemented using remote actors and are first class entities within the runtime (see Figure 3 and 4). This makes the use of name servers much easier, as they can be easily created within SALSA Lite programs, and also allows them to use SALSA Lite's remote message sending infrastructure.

Another major difference is that in SALSA Lite, name servers operate asynchronously. In SALSA, whenever an actor migrates, it synchronously performs an `UPDATE` on the name server and only migrates after it completes, as name servers are used synchronously within the protocol for looking up actors if a message arrives at a theater and the actor had migrated away in the meantime. In SALSA Lite, name servers only asynchronously provide a reference to the actor and the run time updates itself as to where the actor is located.

This is done with the following protocol: if a message received by an `IncomingTheaterConnection` actor has mobile actor as its target, and that mobile actor is not present at the theater, it performs a lookup as to where the actor had migrated using the `MobileActorStateRegistry`. It sends the message on to the theater the actor had migrated to, but also sends an `updateActorLocation` message to the theater actor at the source of the message. It keeps a list of actors it has sent `updateActorLocation` messages to and has not yet heard an acknowledgement back from yet, to prevent spamming the source theater with multiple `updateActorLocation` messages. In this way, as an actor migrates and messages are sent to it, the theaters update their `MobileActorStateRegistry` with references to where the mobile actors have moved to.

All these messages are sent asynchronously using SALSA Lite's remote messaging, to prevent deadlocks and improve performance. Also, this means that the name server responds with a reference to an incorrect theater, as an actor had completed migration before the `UPDATE` message was processed, that reference will be updated to the actor's current location using this protocol without requiring any further calls to the name server.

4. RESULTS

For reproducibility, source code for SALSA Lite is freely available on GitHub.[1] The benchmarks used can be found in the `benchmarks` directory of the repository. This section presents a performance analysis of the newly developed remote and mobile actors and compares them to local actors in SALSA Lite as well as SALSA version 1.1.5.

4.1 Runtime Environment

All results were gathered using a small Beowulf HPC cluster with 4 dual quad-core compute nodes (for a total of 32 processing cores). Each compute node has 64GBs of 1600MHz RAM, two mirrored RAID 146GB 15K RPM SAS drives, two quad-core E5-2643 Intel processors which operate at 3.3Ghz, and run the Red Hat Enterprise Linux (RHEL) 6.2 operating system. All 32 nodes within the cluster are linked by a private 56 gigabit (Gb) InfiniBand (IB) FDR 1-to-1 network. Java version 1.6.0_26 was used, with the Java(TM) SE Runtime Environment (build 1.6.0_26-b03) and the Java HotSpot(TM) 64-Bit Server VM (build 20.1-b02, mixed mode). Runs were performed 10 times each, with freshly created theaters and name servers, so startup times are included, and the various figures display the mean runtime as well as the standard deviation of the different runs. Runs done with SALSA version 1.1.5 were done with garbage collection turned off by using the `-Dnogc` system property for a more accurate comparison as distributed garbage collection in SALSA Lite remains an area of future work. This is in part due to challenges in correctly and efficiently implementing distributed garbage collection that can also handle pathological cases such as distributed circular references.

[1]https://github.com/travisdesell/salsa_lite

4.2 Local Message Passing Performance

Figure 5 shows the performance of local, remote and mobile actors in SALSA Lite, with all actors running on a single stage.[2] The `ThreadRing` benchmark was identical for all three, except that actors either were local, or extended the `RemoteActor` or `MobileActor` behaviors. One stage was used to avoid introducing effects from thread scheduling which could significantly impact performance. 31 actors were created (as typical for the benchmark) and 500,000 to 1,000,000 messages were passed around the ring.

While this figure shows a fair amount of overhead for using remote and mobile actors, this is mostly due to increased startup times. Remote actors require a theater to be created, which opens a socket and listens for incoming communication from other theaters, mobile actors require this in addition to a name server which they are registered with. A linear regression was performed for each of these runs, with r-values (correlation coefficients) greater than 0.999 for all three actor types. Figure 5 shows the slope and intercept for these regressions, showing that remote actors have a ∼2.5x increase in startup costs and that mobile actors have a ∼3.26x increase over purely local actors. Apart from the increased startup costs, the performance of actual message sending is quite good, with local and remote actors having practically identical message passing performance, and mobile actors having ∼30% overhead.

4.3 Local Actor Creation Performance

Figure 6 shows the performance of creating local, remote and mobile actors using a simple actor creation micro benchmark. As in the previous benchmark, only one stage was used to avoid introducing effects from thread scheduling. The benchmark created between 100,000 and 600,000 actors, with each actor responding to the master actor with an acknowledgement message that it had been created. After an acknowledgement had been received for each actor, the benchmark would terminate. Here, in addition to the startup costs seen in the `ThreadRing` benchmark, there is significantly more overhead for creating remote and mobile actors.

Linear regression on these runs produced an r-value above 0.999 for the local actors, and r-values above 0.977 for remote and mobile actors. Creation of remote actors resulted in ∼54% overhead, while mobile actors had ∼438% overhead. This was expected however, as in SALSA Lite, local actors are little more than objects with a reference to the stage they are running on, while remote actors also need to

[2]Previous results have compared SALSA Lite to SALSA for the `ThreadRing` benchmark, along with a set of other actor based programming languages [14] (Kilim, Scala, and Erlang). SALSA Lite has been found to be three times faster than Kilim and an order of magnitude faster than Erlang, Scala and SALSA on this benchmark. This performance has also been recently reproduced by Charousset *et* al. [11]. As this paper focuses on the performance of remote and mobile actors in SALSA Lite, we refer the reader to those works for further performance comparisons between different actor languages.

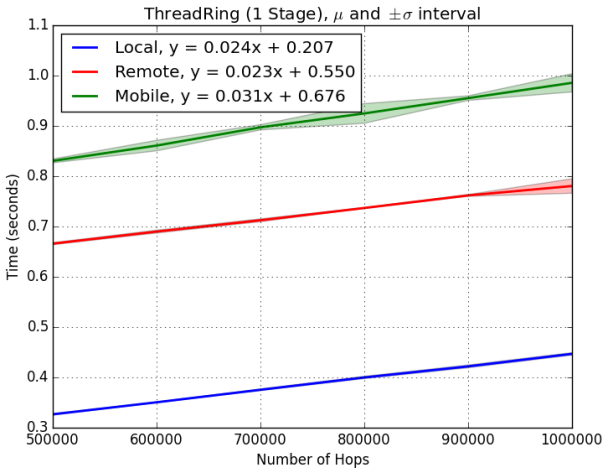


Figure 5: This figure shows the performance of the `ThreadRing` benchmark for local, remote and mobile actors in SALSA Lite. This serves as a measure of the basic amount of overhead in message passing and theater startup costs for using remote and mobile actors. In each benchmark, 31 actors were created and 500,000 to 1,000,000 messages were sent around the ring. The mean and standard deviation of 10 runs for each data point are shown.

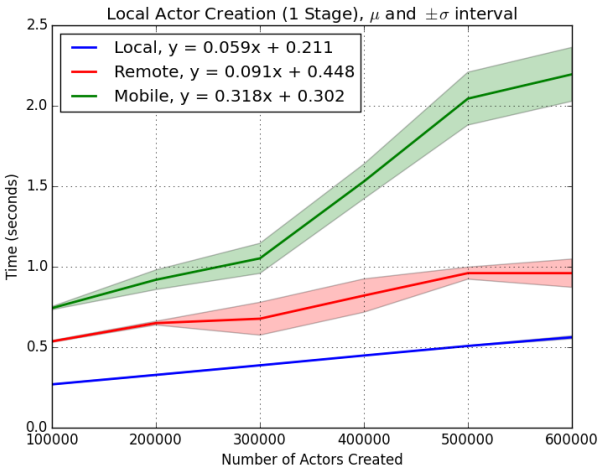


Figure 6: This figure shows the performance of a actor creation micro-benchmark in SALSA Lite. This serves as another measure of theater startup costs, as well as the overhead of creating remote and mobile actors. 100,000 to 600,000 actors were created, each sending an acknowledgement to a master actor. The mean and standard deviation of 10 runs for each data point are shown.

have a name, host and port. In addition to that, mobile actors also require a reference to a name server, and also send a `PUT` message to register at the name server when they are created. Further, both also need to be stored in their respective registries. As the actors created in this micro-benchmark only have a reference to the master actor that created them, these costs can represent significant overhead.

4.4 Remote Actor Creation Performance

Figure 7 shows the performance of creating both remote and mobile actors at a remote theater. Additionally, the cost of creating an actor locally and then migrating it to the remote theater is presented. Only one stage was used, however the theater actors are created on their own stages, so some of the performance differences could be attributed to thread context switching.

This micro-benchmark is identical to the previous local version except for the actors being created on the remote theater. For the version with migration, the acknowledgement message to the master was only sent after the migration had completed. For these tests, 1000 to 5000 actors were created. Linear regression produced r-values greater than 0.994 for the local actors, 0.979 for the remote actors, and 0.983 for the mobile actors. Interestingly, creating the mobile actors locally and migrating them performed better than creating them remotely, however this makes some sense in that creating the mobile actors remotely returns a token (similar to a future) which must be received by a `TokenDirector` actor created by the runtime, so it entails the creation of an extra actor and two extra messages (one from the remote theater to the `TokenDirector` at local theater and another from the `TokenDirector` to any actor which wants to use the reference to the remotely created actor). Even so, remote creation of mobile actors was only ~6% slower than remote creation of remote actors.

It is also worth mentioning that while local creation and migration performed faster than remote creation of mobile actors, in general this is probably not the case. For example, if an actor generates any large amount of data in its constructor, creating it locally and migrating it could be significantly more expensive due to extra bandwidth required. As such, it is good to be able to have the option to use either method.

Results were also gathered using SALSA version 1.1.5 however these were not added to the figures as for above 100 actors deadlocks as well as issues with reaching a limit on the number of threads available occurred. However, for 100 actors, remote creation took an average of 1.84 seconds with a standard deviation of 0.033 over 10 runs, and local creation and migration took on average 1.24 seconds with a standard deviation of 0.0092 seconds. This is an order of magnitude slower than the time taken for SALSA Lite to create 1,000 actors; so the new implementation shows significant performance gains over SALSA version 1.1.5.

4.5 Remote Message Passing Performance

Figure 8 presents results for a new benchmark called `TheaterRings` which examines the performance of remote message sending. This benchmark operates similarly to the `ThreadRing` benchmark, except in this case a single actor is created at each theater. Additionally, multiple rings of actors are created which send messages concurrently. All messages hop around the ring of actors in the same direction. For this benchmark, 1 to 320 rings were generated, with each ring sending 1000 messages. Each ring had one actor created on one of the four nodes in the Beowulf cluster. Each theater had one stage for these actors, however

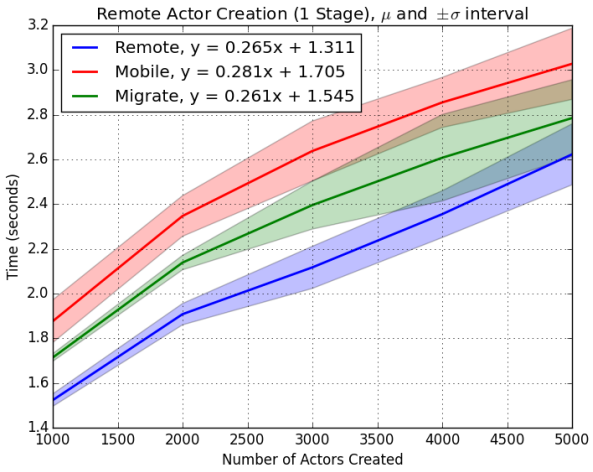


Figure 7: This figure shows the costs of creating remote and mobile actors at a remote theater, along with creating mobile actors locally and migrating them to the remote theater. After each actor was created remotely, or completed its migration, it sent an acknowledgement message back to the master actor. The mean and standard deviation of 10 runs for each data point are shown.

other stages were created for actors in the theater runtime. Figure 8a shows results for 1 to 10 rings, and Figure 8b shows results calculated with 10, 20, 40, 80, 160, and 320 rings. Results were gathered using remote and mobile actors in SALSA Lite, as well as actors in SALSA version 1.1.5.

From 1 to 10 rings some rather interesting behavior occurred. First, once SALSA reached 7 rings, the runtime started varying dramatically and deadlocks started to occur, as is shown by the large increase in the standard deviation of the runtime. Additionally, for 1 and 2 rings, mobile actors exhibited some very poor performance, running almost 5 times slower than remote actors and SALSA. Also, for all three, runtime generally *decreased* as more rings were added. Performance was the fastest for SALSA and remote actors at 6 rings, with an average runtime of 2.001 and 1.518 seconds, respectively, and mobile actors were the fastest at 20 rings, with an average runtime of 1.971 seconds.

For SALSA, after 10 rings, deadlocks occurred even more frequently, however the runtime stabilized for the runs which completed (the times shown are the average runtime of runs which completed). Presumably, from 7 to 9 rings, the issue causing the deadlock could resolve itself resulting in the larger span of run times, however with 10 or more rings the issue was not resolvable. After 10 rings, the runtime increased quite linearly, with the linear regression having r-values of greater than 0.999, 0.998 and 0.999 for remote, mobile and SALSA actors, respectively. Using the values from the linear regression from 10 to 320 stages, mobile actors were ∼5.55% slower than remote actors (similar to results from the remote actor creation micro-benchmark). Further, both remote and mobile actors were around twice as fast as SALSA (not counting SALSA deadlocks).

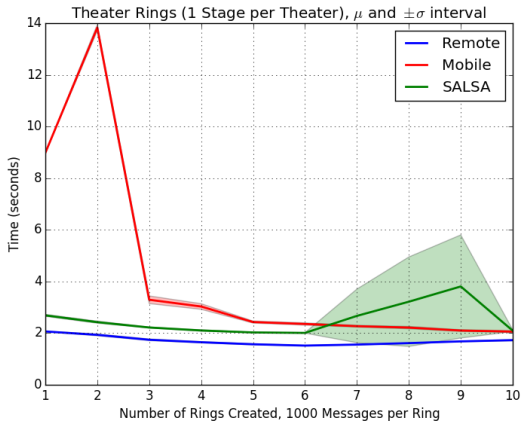
Given these results, it seems that the `TheaterRings` benchmark presents a pathological case for mobile actors when there are 1 or 2 rings. These results are somewhat similar to the case of the `ThreadRing` benchmark where each actor is given their own stage (*i*.e., their own thread), where SALSA Lite performs similarly to SALSA, at about an order of magnitude slower than having the `ThreadRing` actors entirely on one stage. Because of this, the poor performance seems to be due to the cost of context switching between the stage of the `IncomingTheaterConnection` and the stage of the `TheaterRingWorker`. When there is only one message being passed around, the thread of each stage wakes up from a notification when the message is placed in its mailbox, and after processing the message the thread goes back to sleep waiting for the next message as its mailbox is empty. This behavior would explain how increasing the number of rings improved performance, as this would further reduce context switching time. After 10 or so rings, the latency and bandwidth between theaters became the bottleneck, resulting in the linear performance from there as more rings were added.

4.6 Actor Migration Performance

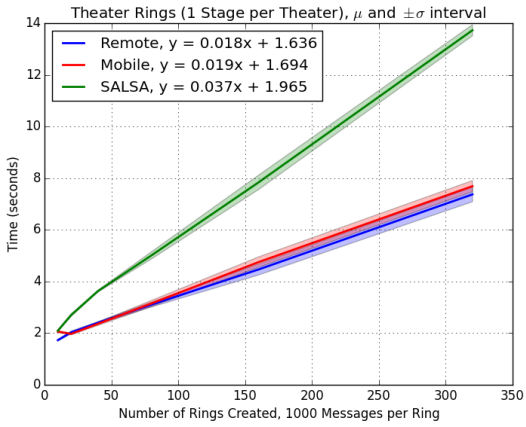
Figure 9 presents results for another new benchmark called `MigrationRings`, which examines the performance of migrating actors in a distributed system. The benchmark creates N actors, and each are given the same list of theaters in the system. Each actor starts at the same origin theater, and then migrates around the theaters in a ring until M migrations have been performed. For this benchmark, 1 to 320 actors were created with results being measured for 10, 20, 40, 80, 160 and 320 actors. Each actor performed 1000 migrations. These actors migrated around four theaters, one on each node in the Beowulf cluster. Each theater was created with one stage for these actors, however additional stages were created for the theater runtime actors. After each actor completed the given number of migrations, an acknowledgement is sent to the master actor, which terminates when it receives an acknowledgement for each actor. Results were gathered using both SALSA version 1.1.5 and mobile actors in SALSA Lite.

Similar to the results for the `TheaterRings` benchmark, mobile actors show weak performance with a single actor, however with more than one ring performance is very good. Additionally, these results also show a similar decrease in runtime when more concurrency is added, presumably due to less context switching. It should also be noted that unlike the remote creation and `TheaterRings` benchmarks, no deadlocks were detected in SALSA version 1.1.5 for these runs. Using the linear regression from results with 10 to 320 actors, with r-values greater than 0.999 for mobile actors and 0.984 for SALSA have mobile actors migrating 17.88 times faster than SALSA, which is a dramatic improvement in performance.

There are many factors in regards to this large performance improvement. First, SALSA actors are heavyweight, each with their own thread, so migration involves starting up and destroying threads as the actor migrates between theaters. Additionally, while the name server in SALSA Lite is used asynchronously as a boot strapping method, in SALSA, migration involves synchronously updating the actor's entry in the name server before performing migration. In many ways, this makes the name server a synchronous bottleneck to performance. While this can be somewhat alleviated by having multiple name servers, it is still a significant performance hit.

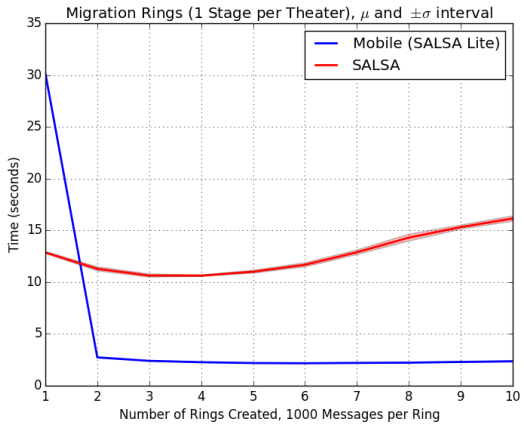


(a) 1 to 10 Rings

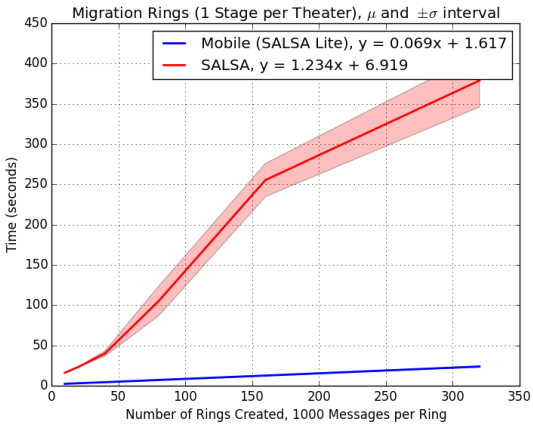


(b) 10 to 320 Rings

Figure 8: This figure shows results from the `TheaterRings` benchmark. 1 to 320 rings of actors were created, one actor per theater, and each ring sent 1000 messages around similar to the `ThreadRing` benchmark. Each ring operated concurrently. Results were gathered using SALSA version 1.1.5 as well as remote and mobile actors in SALSA Lite. The mean and standard deviation of 10 runs for each data point are shown.



(a) 1 to 10 Rings



(b) 10 to 320 Rings

Figure 9: This figure shows results from the `MigrationRings` benchmark. 1 to 320 actors were created, and each migrated 1000 times around four theaters. Results were gathered using SALSA version 1.1.5 and with mobile actors in SALSA Lite. The mean and standard deviation of 10 runs for each data point are shown.

5. CONCLUSIONS AND FUTURE WORK

This paper presents a new distributed runtime to enable remote and mobile actors in SALSA lite. This runtime is built using SALSA Lite's lightweight actors as a foundation to enable high performance and scalability. Performance improvements over SALSA version 1.1.5 are significant: remote message passing in SALSA Lite is 1.94 times faster for mobile actors, and 2.05 times faster for remote actors; migration of mobile actors in SALSA Lite is 17.88 times faster than SALSA; and remote creation of mobile and remote actors in SALSA Lite is two orders of magnitude faster. Additionally, with two new benchmarks, SALSA Lite is shown to have strong scalability in terms of concurrent actor execution. Further, in some of the more complicated benchmarks with lots of actor concurrency, SALSA version 1.1.5 suffered from deadlocks, while SALSA Lite did not, adding more justification for building the runtime using lightweight actors.

The performance overhead of using remote and mobile actors was also compared to SALSA Lite's local actor implementation. For message passing within a theater, remote actors resulted in 0% overhead, and mobile actors resulted in 30% overhead. Local creation of remote and mobile actors was more expensive with 54% overhead for remote actors and 438% for mobile actors. In distributed scenarios, creating mobile actors remotely was only 6% slower than creating remote actors remotely, and passing messages between mobile actors on different theaters was only 5.55% slower than passing messages between remote actors. In general, this overhead is found to be fairly low given the requirements of

remote and mobile actors.

This investigation opens up many avenues for future work and analysis. In particular, some pathological cases for message passing were found in SALSA Lite when the cost of thread context switching becomes quite high. SALSA Lite uses Java's `LinkedList` class along Java's `synchronized` keyword to make access to it thread safe.[3] These pathological cases could be potentially eliminated by using lock-free data structures as used by CAF [10], or thread pools as in Scala [20] and Akka [9, 5]. Another potential area for improved performance would be the use of an asynchronous IO framework such as Netty [41] instead of Java's synchronous sockets, which could allow the `IncomingTheaterConnection` and `OutgoingTheaterConnection` actors to be lightweight instead of heavyweight.

Additionally, for a more in depth investigation of SALSA Lite's performance and comparison to other Actor programming languages, we intend to fully implement the Savina benchmark suite [24] which can potentially uncover other areas where performance can be improved and compare the performance remote messaging and actor migration to other modern implementations. Further, it would be beneficial to extend this suite with more benchmarks such as the `MigrationRings` benchmark discussed in this paper which can more fully test and evaluate the performance of actor migration. The syntax for remote actor referencing described in Section 3 can potentially be simplified even further using syntax described in [45]. Lastly, as touched on in Section 3.2.1, using Java's `readResolve()` and `writeReplace()` methods for serialization lays groundwork for investigating efficient implementations of distributed actor garbage collection [47, 13, 46, 39].

6. REFERENCES

[1] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and run-time monitoring. In *Hardware and Software, Verification and Testing*, pages 191–207. Springer, 2006.

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

[3] G. Agha and N. Jamali. Concurrent programming for distributed artificial intelligence. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to DAI.*, chapter 12. MIT Press, 1999.

[4] G. Agha and C. Varela. Worldwide computing middleware. In M. Singh, editor, *Practical Handbook on Internet Computing*, pages 38.1–21. CRC Press, 2004.

[5] J. Allen. *Effective Akka.* " O'Reilly Media, Inc.", 2013.

[6] S. R. J.-P. Arcangeli, F. Migeon, and S. Rougemaille. Javact: a java middleware for mobile adaptive agents. *Lab. IRIT, University of Toulouse, February 5th*, 2008.

[7] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[8] M. Astley. The actor foundry: A java-based actor programming environment. *University of Illinois at Urbana-Champaign: Open Systems Laboratory*, 1998.

[9] J. Bonér, V. Klang, R. Kuhn, et al. Akka library. *http://akka.io*.

[3]Java's concurrent list implementations were experimented with but resulted in poorer performance.

[10] D. Charousset, R. Hiesgen, and T. C. Schmidt. Caf-the c++ actor framework for scalable and resource-efficient applications. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, pages 15–28. ACM, 2014.

[11] D. Charousset, R. Hiesgen, and T. C. Schmidt. Revisiting actor programming in c++. *arXiv preprint arXiv:1505.07368*, 2015.

[12] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch. Native actors: a scalable software platform for distributed, heterogeneous environments. In *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control*, pages 87–96. ACM, 2013.

[13] S. Clebsch and S. Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. *ACM SIGPLAN Notices*, 48(10):553–570, 2013.

[14] T. Desell and C. A. Varela. Salsa lite: A hash-based actor runtime for efficient local concurrency. *Springer Lecture Notes in Computer Science: Concurrent Objects and Beyond*, 8665, 2013.

[15] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[16] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

[17] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *ACM SIGPLAN Notices*, 39(1):256–267, 2004.

[18] A. Gosain and G. Sharma. A survey of dynamic program analysis techniques and tools. In *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, pages 113–122. Springer, 2015.

[19] P. Haller and M. Odersky. Actors that unify threads and events. In *Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION)*, pages 171–190, 2007.

[20] P. Haller and M. Odersky. Actors that unify threads and events. In *Coordination Models and Languages*, pages 171–190. Springer, 2007.

[21] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8-3:323–364, June 1977.

[22] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[23] S. Hong and M. Kim. A survey of race bug detection techniques for multithreaded programmes. *Software Testing, Verification and Reliability*, 25(3):191–217, 2015.

[24] S. Imam and V. Sarkar. Savina-an actor benchmark suite. In *4th International Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE*, 2014.

[25] M.-W. Jang. The actor architecture manual. *Department of Computer Science, University of*

Illinois at Urbana-Champaign, 2004.

[26] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM, 2010.

[27] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. *ACM Sigplan Notices*, 44(6):110–120, 2009.

[28] A. Jyoti and V. Arora. Debugging and visualization techniques for multithreaded programs: A survey. In *Recent Advances and Innovations in Engineering (ICRAIE), 2014*, pages 1–6. IEEE, 2014.

[29] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[30] R. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM, 2009.

[31] N. Kaveh and W. Emmerich. Deadlock detection in distribution object systems. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 44–51. ACM, 2001.

[32] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Proceedings of Supercomputing'95*, pages 39–48, 1995.

[33] Y. Kwon, K. Mechitov, and G. Agha. Design and implementation of a mobile actor platform for wireless sensor networks. In *Concurrent Objects and Beyond*, pages 276–316. Springer, 2014.

[34] Y. Kwon, S. Sundresh, K. Mechitov, and G. Agha. Actornet: An actor platform for wireless sensor networks. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1297–1300. ACM, 2006.

[35] M. Miller, E. Tribble, and J. Shapiro. Concurrency among strangers. *Trustworthy Global Computing*, pages 195–229, 2005.

[36] M. S. Miller and J. S. Shapiro. *Robust composition: Towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006.

[37] T. Salmito, A. L. de Moura, and N. Rodriguez. A flexible approach to staged events. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 661–670. IEEE, 2013.

[38] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In J. Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 104–128. Springer, 2008.

[39] C.-H. Tang, T.-Y. Song, M.-F. Tsai, and W.-J. Wang. Collecting mobile-agent garbage using actor-based weighted reference counting. *Advanced Science Letters*, 9(1):157–161, 2012.

[40] S. Tasharofi, P. Dinges, and R. E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *ECOOP 2013–Object-Oriented Programming*, pages 302–326. Springer, 2013.

[41] The Netty Project. Netty. [Accessed Online 2015] http://netty.io.

[42] R. Van Renesse. Goal-oriented programming, or composition using events, or threads considered harmful. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 82–87. ACM, 1998.

[43] C. Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, U. of Illinois at Urbana-Champaign, 2001. http://osl.cs.uiuc.edu/Theses/varela-phd.pdf.

[44] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.

[45] C. A. Varela. *Programming Distributed Computing Systems: A Foundational Approach*. The MIT Press, 2013.

[46] W.-J. Wang. Conservative snapshot-based actor garbage collection for distributed mobile actor systems. *Telecommunication Systems*, 52(2):647–660, 2013.

[47] W.-J. Wang and C. A. Varela. Distributed garbage collection for mobile actor systems: The pseudo root approach. In *Advances in Grid and Pervasive Computing*, pages 360–372. Springer, 2006.

[48] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, 2001.

[49] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *ECOOP 2005-Object-Oriented Programming*, pages 602–629. Springer, 2005.

Optimizing Communicating Event-Loop Languages with Truffle

[Work In Progress Paper]

Stefan Marr*
Johannes Kepler University Linz, Austria
stefan.marr@jku.at

Hanspeter Mössenböck
Johannes Kepler University Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Communicating Event-Loop Languages similar to E and AmbientTalk are recently gaining more traction as a subset of actor languages. With the rise of JavaScript, E's notion of *vats* and non-blocking communication based on promises entered the mainstream. For implementations, the combination of dynamic typing, asynchronous message sending, and promise resolution pose new optimization challenges.

This paper discusses these challenges and presents initial experiments for a Newspeak implementation based on the Truffle framework. Our implementation is on average 1.65x slower than Java on a set of 14 benchmarks. Initial optimizations improve the performance of asynchronous messages and reduce the cost of encapsulation on microbenchmarks by about 2x. Parallel actor benchmarks further show that the system scales based on the workload characteristics. Thus, we conclude that Truffle is a promising platform also for communicating event-loop languages.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Concurrent programming structures; D.3.4 [**Processors**]: Optimization

Keywords

Actors, Event-Loops, Concurrency, Optimization, Truffle

1. INTRODUCTION

Communicating event-loop languages such as E [14] and AmbientTalk [16] are appealing for application development because they are free from low-level data races and deadlocks, and their concurrency model is comparably simple. Communicating event loops (CEL) are isolated from each other so that any form of *communication* needs to be done explicitly via message passing instead of implicitly via shared state. This prevents low-level data races and thereby raises the abstraction level of programs. In E, these CELs are called *vats* and contain heaps of objects. In this paper, we simply call them *actors*. As a consequence of the deadlock-free design, synchronization is modeled with asynchronous messages and promises. Consequently, data and synchronization dependencies become explicit to the programmers, which can avoid hidden race conditions and deadlocks.

Recently, the CEL model got adopted by languages used predominantly in shared memory settings. JavaScript [5] add a variation of this model with Web Workers[1] and its support for promises. With this extension, many languages targeting JavaScript VMs as execution platforms also adopt the model, e. g., Dart,[2] ClojureScript,[3] Scala.js,[4] and TypeScript.[5] This trend brings the CEL model from a distributed setting into the realm of shared memory multicore systems ranging from mobile devices to server applications. Hence, an efficient utilization of multicore processors becomes more relevant than distributed messaging performance. For language implementers, this brings new optimization challenges since JavaScript—as language and platform—is dynamically typed. To our knowledge, existing research on the performance of communicating event-loop languages either restricted itself to simple interpreters as was the case for E and AmbientTalk, or used static type systems, e. g., JCoBox [15] used one to optimize message sends. Similarly, languages or frameworks that follow more closely the Hewitt [6] and Agha [1] style of actors focused on type systems to improve performance, e. g., SALSA [4] or Pony.[6]

This work is an initial exploration of the optimization challenges for dynamically-typed communicating event-loop languages for shared-memory multicore systems. We present SOM_{NS}, an implementation of Newspeak[7] [2] based on the Truffle framework, which executes on top of a JVM. Beside a brief sketch of Newspeak's concurrency model, we discuss the optimization challenges for asynchronous execution, ensuring isolation between actors, and promise resolution. We evaluate initial strategies to optimize asynchronous message sends and argument handling to ensure isolation.

*Stefan Marr's research is funded by Oracle Labs.

Submitted as Work-In-Progress Paper to the AGERE'15 Workshop co-located with SPLASH'15, Pittsburgh, PA, USA.

[1] *Web Workers*, W3C, May 2012 `www.w3.org/TR/workers/`
[2] *Dart*, Google `dartlang.org`
[3] *ClojureScript*, `github.com/clojure/clojurescript/`
[4] *Scala.js*, Sébastien Doeraene `scala-js.org`
[5] *TypeScript*, Microsoft `typescriptlang.org`
[6] *Pony*, `ponylang.org`
[7] *Newspeak Programming Language Specification*, Gilad Bracha, ver. 0.095 `http://bracha.org/newspeak-spec.pdf`

2. NEWSPEAK: A DYNAMIC CONCURRENT EVENT-LOOP LANGUAGE

For our exploration we chose Newspeak, a dynamically typed class-based language with actors based on E's communicating event-loop model. For concurrency research, it has the major advantage that it does not have a notion of global or static state. Instead, state has to be passed explicitly following the object capability model [3]. The resulting language is simple and self-consistent, which avoids many special cases while retaining the convenience of classes. Nonetheless, it is similar to widely used object-oriented languages, and thus, represents a wide range of languages used on the Web. Furthermore, Newspeak also has JavaScript and Dart backends to run in browsers and on servers. Our implementation, SOM_{NS}[8] is designed for research on shared-memory concurrency with good performance. For instance, in addition to the CEL model, SOM_{NS} implements Newspeak's `Value` objects, which are deeply immutable and thus can only refer to deeply immutable objects themselves. Hence, it is safe to share such *values* between actors and to allow them to access value objects synchronously.

A Self-Optimizing Truffle Interpreter. To achieve performance within small factors of highly optimizing VMs, we built on Truffle [12, 18]. This means, SOM_{NS} is an abstract-syntax-tree (AST) interpreter running on top of a Java Virtual Machine (JVM) with the Graal just-in-time compiler [17]. Truffle and Graal together enables meta-compilation based on self-optimizing ASTs that rewrite themselves at run time to optimize for the characteristics of the executed program and its data. Once a SOM_{NS} method has been executed often enough, Graal generates native code for it by taking the AST and applying partial evaluation, aggressive inlining, as well as classic compiler optimizations. The end result is the compilation of a SOM_{NS} method to native code. With this meta-compilation approach, the compiler is independent from the implemented language, and has been used for example for JavaScript, Python, R, and Ruby. The SOM_{NS} interpreter uses self-optimization to determine types of operations, to optimize the object layout based on types, for polymorphic inline caches [9] to cache method lookups, and other performance relevant issues of dynamic languages. Details are discussed in previous work on SOM [12, 13].

3. OPTIMIZATION CHALLENGES

The first major hurdle for performance is SOM_{NS}' dynamically typed nature, which it shares with JavaScript, Python, Smalltalk, and others. However, the ideas of speculative optimizations and adaptive compilation [8] have been successfully applied to eliminate the cost of dynamic typing and late binding. At times, the results even outperform statically compiled code, because speculative optimizations can make optimistic assumptions leading to faster native code.

For this work however, the main focus is on optimization challenges for event loop languages. Thus, we investigate the performance challenges revolving around asynchronous message sends, ensuring isolation between actors, and the efficient handling of promise resolution.

[8] SOM_{NS} is a derivate of SOM (Simple Object Machine), a family of interpreter implementations: `som-st.github.io`

Asynchronous Execution. Newspeak has four ways to initiate an asynchronous execution. We distinguish between sending of asynchronous messages to (i) far references, (ii) near references, and (iii) promises. Furthermore, we also consider the execution of the (iv) success or failure handlers, i. e., callbacks, registered on promises.

For the sending of asynchronous messages, one challenge is to determine the method to be executed on the receiver side efficiently. In a distributed setting, we assume that the lookup is done every time a received message is processed. Even if one would try to use something like a polymorphic inline cache [9] in an event loop, we assume the event loop to result in megamorphic behavior, because all messages and receiver types are funneled through a single point. Especially for dynamic languages with complex lookup semantics, the repeated lookups represent a significant cost compared to the cost of method invocation in the sequential case. Another issue for reaching peak performance is that information about the calling context is typically lost. For optimizing dynamic languages, this is however highly relevant to enable optimistic type specializations. Imagine a simple method adding numbers, depending on the caller a method might be used exclusively for adding integers, or in another setting exclusively for adding doubles. When an optimizer is able to take such calling-context information into account, it might be able to produce two separate compilations of the method for the different callers, which then can be specialized to either integers or doubles avoiding unnecessary run time checks and value conversions. For handlers registered on promises, lookup is no issue because the handle directly corresponds to the code to be executed. The calling context however might also be an issue if the same handler is used in multiple distinct situations.

Ensuring Isolation Between Actors. The second optimization challenge is to minimize the overhead of guaranteeing isolation between actors. Many pragmatic systems forgo isolation because of performance concerns. Examples include many actor libraries for the JVM [11] including Akka and Jetlang, as well as JCSP[9] and Go,[10] which implement *communicating sequential processes* [7]. SOM_{NS} provides this guarantee because we see it as an essential properties that make CELs useful from an engineering perspective.

To guarantee isolation, SOM_{NS} needs to ensure that the different types of objects are handled correctly when being passed back and forth between actors. Specifically, mutable objects need to be wrapped in far references so that other actors have no synchronous access. Far references on the other hand need to be checked whether they reference an object that is local to the receiving actor to unwrap them and guarantee that objects owned by an actor are always directly accessible. When promises are passed between actors, SOM_{NS} needs to create a new promise chained to the original one. This is necessary to ensure that asynchronous sends to promises that resolve to value objects are executed on the lexically correct actor. Since value objects do not

[9] *Communicating Sequential Processes for Java (JCSP)*, Peter Welch and Neil Brown, access date: 2015-07-05 `www.cs.kent.ac.uk/projects/ofa/jcsp/`

[10] *The Go Programming Language*, `golang.org`

have owners, we bind promises to actors and resolve the new promise with the original one when passing them between actors. Similar to asynchronous sends, handlers registered on promises need to be scheduled on the correct actor, i. e., the one that registered them. Thus, promises need to be handled differently from other objects passed between actors. For value objects, it needs to be determined efficiently whether they are deeply immutable, so that they can be passed safely as direct references.

For message sending, distinguishing between all these different cases has a negative impact on performance. Thus, finding ways to minimize the number of checks that need to be performed would reduce the cost of guaranteeing isolation. One conceptual benefit of Newspeak, and with it SOM_{NS}, is that the message send semantics do not require copying of objects graphs, which is required, for instance, for message sending between JavaScript Web Workers.

Efficient Promise Resolution. The optimization of promise resolution and scheduling of their messages and handlers is hard because promises can form tree-shaped dependencies, and in this dependency tree, promises from different actors can be involved. Ideally, resolving a promise would only require to schedule a single action on the event loop of the actor owning the promise. In this case, guaranteeing isolation and scheduling the corresponding action on the event loop could be done in code that is straightforwardly compiled to efficient native code. However, when a promise p_A is resolved with a promise p_B, p_A's resolution handlers are not scheduled immediately, instead, they will only be triggered once p_B has been resolved. Similarly, sending an asynchronous message to a promise means that the message is sent to the value to which the promise is eventually resolved. In the general case, this means, the resolution process has to traverse a tree structure of dependent promises and schedule all the handlers and messages registered on these promises on their corresponding event loops. Since the dependent promises can originate from different actors, we also need to check at each point whether the resolved value is properly wrapped to guarantee isolation. Another pitfall with promise resolution is that a naive implementation could easily cause a stack overflow in the implementation, which would cause a crash when resolving long dependency chains of promises.

4. FIRST OPTIMIZATIONS

The previous section outlined some of the challenges for optimizing SOM_{NS}. This section introduces initial optimizations that address them.

Send-site Lookup Caching. To avoid the repeated lookup overhead for asynchronous messages, we rely on SOM_{NS} executing on a shared memory system. This allows us to introduce polymorphic inline caches (PIC) for the send-site of asynchronous messages. Specifically, we use Truffle's notion of a `RootNode`, which essentially correspond to functions. At each send site of an asynchronous message, a root node is constructed that contains a normal SOM_{NS} synchronous message send operation, which already utilizes a PIC. This root node is eventually used when processing the asynchronous message in the event loop. This approach has two major benefits. On the one hand, we achieve specialization based on the send site. Ideally, the send is monomorphic, as most methods call are, so that is requires only a simple identity check of the receiver's class before executing the cached method. Since we reuse the normal synchronous send operation at the receiver site, Truffle also does method splitting and thus, enable the use of profile information based on the specific send site. On the other hand, creating the root node allows us to put these performance critical operations within the scope of Truffle's meta-compilation. This means, when the event loop takes a message for execution, it will eventually call directly into compiled code from the event loop and does not perform any generic operations that cannot be optimized. By constructing the root node that is send-site specific, but executes and performs the caching only in the target event loop, this optimization is applicably to all types of sends in SOM_{NS}. Thus, asynchronous sends to far references, to direct references, and to promises are optimized in the same way.

Compared to normal synchronous method invocation, asynchronous messages have the cost of the message queuing and cannot be inlined into the caller, since this would violate the semantics. Beside that however, we enable the other classic compiler optimizations, which can eliminate the cost of SOM_{NS}' dynamicity.

Guaranteeing Isolation. As discussed before, guaranteeing isolation requires to check at run time whether objects need to be wrapped in far references or have to be near references, make sure that promises are treated correctly to ensure their desired behavior, or to check whether an object is a deeply immutable value object.

To efficiently check deep immutability of values, we rely on Newspeak's semantics. All objects of classes that include the `Value` mixin are supposed to refer only to deeply immutable objects themselves. Since object constructors can however execute arbitrary code, we chose to check at the end of a constructor whether all fields of a value object contain only value objects themselves. For these checks, we assume that objects are usually going to be initialized with the same types of objects for each field. Thus, we enable specialization of the value check for each field separately, which in the ideal case means that per field a simple type check or read of a flag is sufficient to determine whether the constructed object is a legal value object. Like all of the optimizations discussed here, this optimization relies on a lexical stability of program behavior, which for a majority of programs is given or can be reached by method splitting based on the usage context. With the correctness check on construction, value objects can be recognized based on a flag that is set in these objects without having to traverse the object graph.

To ensure isolation, we optimize asynchronous sends to far references, handler registration and asynchronous sends to promises that are already resolved, as well as explicit promise resolution. For all these cases there is a concrete lexical element in the program, and consequently the AST can contain a node that can specialize itself based on the observed values. For the asynchronous sends, this means for each argument

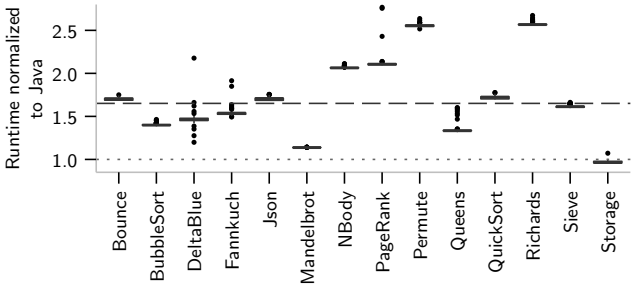


Figure 1: Peak performance comparison between SOM_{NS} and Java on classic sequential benchmarks. The dotted line is the Java performance based on the Graal compiler. The dashed line is the geometric mean over all SOM_{NS} benchmarks.

to the message send, a wrapper node is inserted into the AST that can specialize itself to one of the various cases that need to be handled. The assumption is that checking a guard such as the type of an object and whether sender and receiver actor are different is a faster operation than having to process all different cases repeatedly.

For the case that a handler is registered on an already resolved promise, or an asynchronous message send is performed, the same optimization applies and the operations are specialized directly in the AST corresponding to the interaction with the promise. However, for unresolved promises, this only applies to the arguments of the asynchronous message send. For the value to which the promise is resolved, we cannot use the same specialization. Since promises are normal objects, they can also be resolved explicitly by calling the `resolve()` method on the corresponding resolver object. For this specific case, the specialization is again applicable since it can be done as part of the AST element that does the call to the `resolve()` method. For the resolution of the promise that is the result of an asynchronous message send, this optimization applies as well. As discussed before, for each asynchronous send, we construct a root node that contains the actual synchronous send, i. e., method invocation done on the receiver side. We use this root node also to perform the promise resolution with the return value of the method invocation. Here, the send-site based specialization again provides the necessary context for the specialization.

5. PRELIMINARY RESULTS

For each benchmark, we measured 100 consecutive iterations within the same VM after 150 warmup iterations. The results represent SOM_{NS} peak performance. The benchmarks are executed on a system with two quad-core Intel Xeons E5520 processors at 2.26 GHz with 8 GB of memory and runs Ubuntu Linux with kernel 3.13, and Java 1.8.0_60.

To give an intuition of SOM_{NS}' performance, we compare it with Java. Since Truffle relies on the Graal compiler, we chose to also use Graal for the Java benchmarks to avoid cross comparison between compilers. On this benchmark set, Graal is about 10.9% slower than HotSpot's C2 compiler, which we consider more than acceptable. The results in fig. 1 show that SOM_{NS} is 1.65x (min. −3%, max. 2.6x) slower than Java on our set of benchmarks.

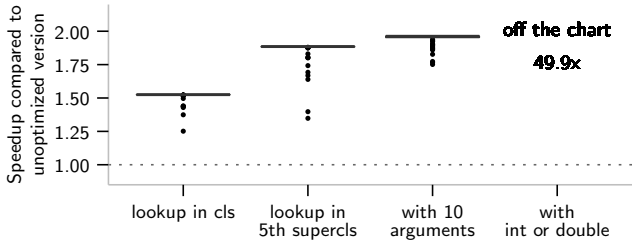


Figure 2: Speedup of asynchronous send operation compared to unoptimized SOM_{NS}. Microbenchmarks focus on lookup caching, ensuring of isolation, and preservation of calling context.

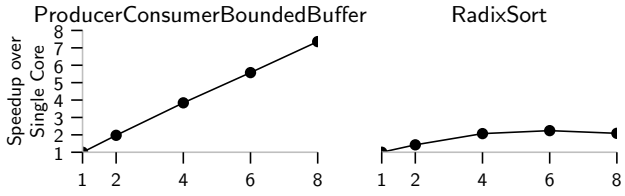


Figure 3: Results for two Savina benchmarks [10] to demonstrate scalability on multiple cores.

The microbenchmarks compare SOM_{NS} with and without the discussed optimizations. As depicted in fig. 2, the caching of lookups and the optimizations to reduce the run-time checks for the argument handling give a speedup of 1.5x to 2x on these microbenchmarks. The preservation of the calling context to enable optimizations can give even more speedup of 49.9x, which unfortunately does not fit onto the chart. As an initial verification that the parallel execution leads to speedup, we chose two of the Savina benchmarks [10] that have potential for parallelism. Figure 3 indicates that an increased number of actors indeed increases performance.

6. CONCLUSION

This first exploration investigates optimization challenges for communicating event-loop languages. With SOM_{NS}, a Newspeak implementation based on Truffle, we show that they can be implemented efficiently with Truffle. SOM_{NS} is only 1.65x slower than Java. Furthermore, we show that send-site caching reduces the lookup overhead, cost of ensuring isolation, and enables the use of the calling context for optimization. On microbenchmarks, we see speedups of 1.5x to 2x, while the use of the calling context for optimization can give a speedup of 49.9x. Finally, we also show that SOM_{NS} can realize parallel speedup on two benchmarks. While these are only preliminary results, some of the ideas are applicable to other types of languages. Since asynchronous message reception rarely leads to monomorphic behavior, send-site-based optimizations could also be beneficial for statically typed languages.

Nonetheless, much work remains to be done. For instance, we do not yet have a solution of handling complex promise dependencies efficiently and we did not yet verify the benefit of these optimizations on larger actor programs. We however hope, SOM_{NS} is an interesting platform for future research not only of optimization techniques but also for safe concurrent programming models beyond classic actors.

References

[1] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[2] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as Objects in Newspeak. In *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 405–428. Springer, 2010. ISBN 978-3-642-14106-5. doi: 10.1007/978-3-642-14107-2_20.

[3] J. B. Dennis and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Commun. ACM*, 9(3):143–155, Mar. 1966. ISSN 0001-0782. doi: 10.1145/365230.365252.

[4] T. Desell and C. A. Varela. SALSA Lite: A Hash-Based Actor Runtime for Efficient Local Concurrency. In G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, editors, *Concurrent Objects and Beyond*, volume 8665 of *LNCS*, pages 144–166. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44470-2. doi: 10.1007/978-3-662-44471-9_7.

[5] Ecma International. *ECMAScript 2015 Language Specification*. Geneva, 6th edition, June 2015.

[6] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI'73: Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[7] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. ISSN 0001-0782. doi: 10.1145/359576.359585.

[8] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9. doi: 10.1145/143095.143114.

[9] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91: European Conference on Object-Oriented Programming*, volume 512 of *LNCS*, pages 21–38. Springer, 1991. ISBN 3-540-54262-0. doi: 10.1007/BFb0057013.

[10] S. M. Imam and V. Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, pages 67–80. ACM, 2014. ISBN 978-1-4503-2189-1. doi: 10.1145/2687357.2687368.

[11] R. K. Karmani, A. Shali, and G. Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-598-7. doi: 10.1145/1596655.1596658.

[12] S. Marr and S. Ducasse. Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. In *Proceedings of the 2015 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '15. ACM, 2015. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660194.

[13] S. Marr, T. Pape, and W. De Meuter. Are We There Yet? Simple Language Implementation Techniques for the 21st Century. *IEEE Software*, 31(5):60–67, September 2014. ISSN 0740-7459. doi: 10.1109/MS.2014.98.

[14] M. S. Miller, E. D. Tribble, and J. Shapiro. Concurrency Among Strangers: Programming in E as Plan Coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, April 2005. doi: 10.1007/11580850_12.

[15] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *LNCS*, pages 275–299, Berlin, 2010. Springer. ISBN 978-3-642-14106-5. doi: 10.1007/978-3-642-14107-2_13.

[16] T. Van Cutsem, E. Gonzalez Boix, C. Scholliers, A. Lombide Carreton, D. Harnie, K. Pinte, and W. De Meuter. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures*, 40(3–4):112–136, 2014. ISSN 1477-8424. doi: 10.1016/j.cl.2014.05.002.

[17] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward!'13, pages 187–204. ACM, 2013. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509581.

[18] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Dynamic Languages Symposium*, DLS'12, pages 73–82, October 2012. ISBN 978-1-4503-1564-7. doi: 10.1145/2384577.2384587.

Connect.js

A cross mobile platform actor library for multi-networked mobile applications

Elisa Gonzalez Boix Christophe Scholliers Nicolas Larrea Wolfgang De Meuter
Vrije Universiteit Brussel
{egonzale,cfscholl,nlarrea,wdmeuter}@vub.ac.be

Abstract

Developing mobile applications which communicate over multiple networking technology is a difficult task. First, developers usually have to maintain a different version of the application for each mobile platform they target. Recent trends in mobile cross-platform solutions may alleviate this issue. However, developers still need to program a variation of the application for each different network interface. In addition, the APIs for communicating over ad-hoc networking technologies (eg. wifi direct), are very different from the cloud APIs. Finally, developers need to write highly asynchronous code for communication. This is often written with callbacks which invert the control flow of the application leading to code which is hard to debug and maintain. This paper introduces Connect.js, a JavaScript library for writing multi-networked cross-platform mobile applications. Applications consists of distributed objects which communicate with one another by means of asynchronous messages via a special kind of reference which is transparent for the underlying network technology used. Connect.js also provides dedicated language constructs for structuring asynchronous code by means of future combinators.

1. Introduction

Today we are witnessing a convergence in mobile technology and cloud computing trends. One the one hand, mobile devices have become ubiquitous. Many of them have more computing power than high end (fixed) computers developed 15 years ago. Moreover, they are equipped with multiple wireless network capabilities such as cellular network (3G/4G), wifi, bluetooth, wifi-direct, and NFC. As to be expected with any new technology, multiple mobile platforms are currently available (being the most relevant ones Android, iOS, and Windows Mobile). Important for the programmer is that each of these platforms have a radically different programming environment (e.g., Java in Android, Objective C in iOS, etc).

On the other hand, with the advanced on mobile broadband internet access, the web has evolved from data presentation layer to a data sharing and computation platform, to wit the cloud. Implementing mobile applications employing web-based technologies (like HTML and JavaScript) has the potential benefit of running in multiple mobile environments. However, mobile web-based applications usually perform worse, and do not provide a consistent look and feel than their native counterparts.

In order to minimize the software development costs, mobile cross-platform tools allow to develop one application for multiple mobile platforms [4]. Specially relevant are interpreted tools, like Appcelerator Titanium[1] or Xamarin[2], where developers write applications in a specific language (e.g., JavaScript or C++) and the tool builds a native application for the different targeted mobile platforms. They provide a number of built-in APIs for constructing native GUI and accessing the underlying hardware without requiring detailed knowledge of the targeted platform. The resulting rich mobile applications (RMAs) contribute to the intersection of mobile and cloud computing [1].

In this paper, we focus on a new breed of rich mobile applications which make use of both P2P communication and centralised wireless network access to coordinate and share data. Such *multi-networked* RMAs enable communication over both infrastructure-less networks of mobile devices, and the cloud. Developing such multi-networked applications burdens developers with the following tasks:

- Programmers need to implement a variation of the application for each network interface, and write complex failure handling code to support for reliable communication over multiple networking interfaces. Moreover, the API's for communicating over mobile ad hoc networking

[1] http://www.appcelerator.com

[2] http://xamarin.com

technologies like wifi-direct or bluetooth, are very different from the cloud API's. While the cloud typically requires a client-server communication model, the lack of infrastructure in ad hoc networking technologies requires a peer-to-peer communication model, in which services can be directly discovered in proximate devices.

- Programmers need to write highly asynchronous code for the network communication. This is often written with callbacks. However, these callbacks invert the control flow of the application leading to code which is hard to debug and maintain.

To overcome these issues we propose Connect.js, a mobile cross-platform development library for multi-networked mobile applications. In order to be able to communicate over multiple networking technologies, Connect.js introduces a novel kind of remote object reference which abstract over the kind of network interface being used, called network transparent references (NTR). As a result, applications can seamlessly communicate over the cloud or using an infrastructureless mobile network depending on the underlying available networking technology. NTRs offer reliable communication and as such, programmers do not need to manually verify the delivery of each message sent over multiple network interfaces. In order to mitigate the negative effects of callbacks, Connect.js provides dedicated language constructs for structuring asynchronous code by means of future combinators. Future combinators treat futurized messages as monads and provide a number of operators to combine futurized message passing while avoiding deep nesting associated with callbacks. We believe that the combination of NTRs and future combinators eases programming multi-networked rich mobile applications.

2. Connect.js

Connect.js is a mobile cross-platform library integrated in Appcelerator Titanium which allows programmers to write their distributed mobile applications in JavaScript and deploy it on several mobile platforms, namely iOS and Android. Figure 1 shows the general architecture of Connect.js. Programmers write mobile applications in JavaScript importing the Connect.js library in their Titanium project for distributed programming. The abstractions provided by the Connect.js API are based on the ambient-oriented programming model from AmbientTalk[2] which treats network partitions as a normal mode of operation. As in AmbientTalk, every device hosts at least one actor which encapsulates one or more objects. Objects can communicate with objects in another actor system by means of sending asynchronous messages via a special remote reference called a far-reference. To this end, Connect.js incorporates a built-in service discovery mechanism which allows to discover services in devices accessible under the same ad hoc network or via the cloud, independently of the mobile platform of the device (explained in the next section). From an implementation point of view, Connect.js contains two different native modules (also called plugins) for service discovery on an ad hoc network based on zero-configuration networking technologies specific to the mobile platform. The iOS plugin uses Bonjour [3], and the Android plugin employs NDS [4] We then rely on the JavaScript - Java/ObjectiveC bridge from Titanium to transform the JavaScript code into native applications in the targeted mobile platform.

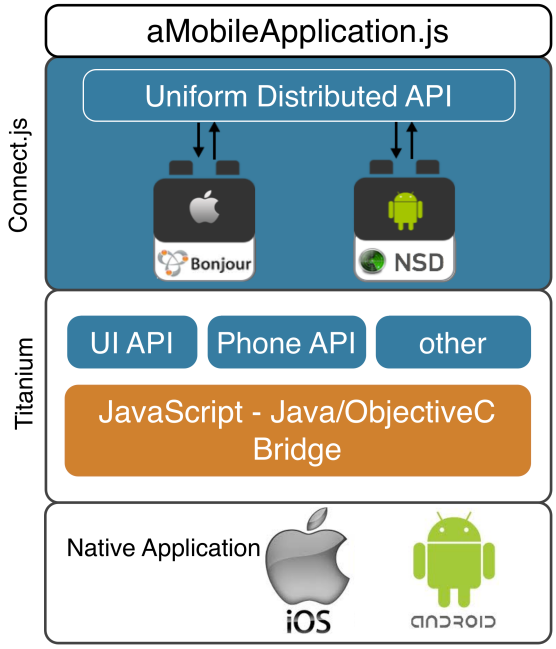


Figure 1. Architectural Overview of Connect.js.

2.1 Network Transparent References (NTRs)

Connect.js considers actors as the unit of distribution, and as such two objects are said to be remote when they are owned by different actors. The only type of communication allowed on remote object references is asynchronous message passing. Any messages sent via a remote reference to an object are enqueued in the message queue of the owner of the object and processed by the owner itself.

In Connect.js actors communicate with one another over wireless links or mobile broadband access. As such, remote references in Connect.js abstract the underlying networking technology being used for communication. Such *network transparent references* (NTRs) are resilient to network fluctuations by default. When a network technology (e.g. wifi) is not available, the NTR attempts to transmit messages sent to it using another networking technology, i.e. 3G. If all networking interfaces are down, the remote reference starts buffering all messages sent to it. When the network partition is restored at a later point in time, the far reference flushes

[3] https://developer.apple.com/bonjour/

[4] http://developer.android.com/reference/android/net/nsd/NsdManager.html

all accumulated messages to the remote object in the same order as they were originally sent. As such, temporary network failures or fluctuations on the availability of the different network interfaces does not have an immediate impact on the applications' control flow.

To illustrate NTRs and the different distributed programming constructs in Connect.jsconsider the following code snippet from a chat application:

```
var buddyList = {};
var Ambient = require("js/connectjs/ConnectJS");
function initializeMessenger(name) {
 //....
  Ambient.wheneverDiscovered("MESSENGER",
   function(ntr){
     var msg = Ambient.createMessage(getName,[]);
     var future = ntr.asyncSend(msg,"twoway");
     future.whenBecomes(function(reply) {
        buddyList[reply] = ntr;
        // send a salute message
     });
   });
}
```

Listing 1: Example use of asynchronous message passing over NTRs

The `wheneverDiscovered` function takes as arguments a string representing the service type and a function serving as callback. Whenever an actor is encountered in the ad hoc network that exports a matching object, the callback function is executed. The `ntr` parameter of the function is bound to a network transparent reference to the exported messenger object of another device.

In order to define objects exporting service to the network, the `exportAs` function is employed. Code snippet below shows how to create an object which implements a service corresponding to the chat application using the string `MESSENGER` as service type.

```
var remoteObj = Ambient.createObject({
  "getName": function ()  { return myName; },
  "talkTo": function (msg) { displayMessage(msg);}
});
Ambient.exportAs(remoteObj, "MESSENGER");
```

With the code provided two phones can already communicate with each other when they are within direct communication range. However, when the phones are not in direct communication range, Connect.js allows the phones to communicate with each other through a centralised node server. To this end, the programmer needs to configure a node server so that service objects are allowed to be exported as well via in an intermediary server in the cloud. The code to configure the node server is shown below.

```
var Ambient=require("js/connectjs/ConnectJS"),
  express = require("express"),
  nodeServer = require("http").Server(express());
nodeServer.listen(3000);
Ambient.initServer(nodeServer);
```

2.2 Future as a Monad

.

The use of asynchronous communication has proven to be beneficial to build distributed mobile applications because it mitigates the negative effects of frequent network failures. However, asynchronous primitives suffers from similar problems as traditional callbacks. In this section we give an overview of how defining futures in terms of a monad allows programmers to better structure their asynchronous code. We also present a number of combinators which turn out to be useful but do not follow the monadic structure.

2.2.1 The Future Monad

The basic constructs of futures can be formulated as a monad [3]. Listing 2 shows the unit and bind type signature of the monad typeclass in Haskell. A monad has two operations, `bind` (>>=) and `unit`. Bind takes a monad of `x` and a function which takes a value of type `x` and returns a monad `m y`. Unit lifts a regular value of type `x` into a monad of `x`.

```
class Monad m where
 (>>=)  :: m x -> (x -> m y) -> m y
 unit :: x -> m x
```

Listing 2: Monad typeclass in Haskell

While we can not reap the advantages of the Haskell type system to enforce monadic behaviour we can still implement the basic bind an unit operators over futures in Javascript. Pseudo code [5] for the bind operator as defined in the future prototype in Connect.js is defined in listing 3. When `bind` is applied on a future `F`, a new future `R` is created (line 2). This new future is also the result of applying `bind` (line 9). Bind registers an anonymous function on the future `F`. This anonymous function resolves the future `R` with the result of applying the function `f` to the value where the future `F` resolves to (line 4-8).

More easily the return operator lifts a normal value into a future value. Return simply creates a new future and immediately resolves the future with the given value.

2.3 Future Combinators

The basic monadic operators together with the lift construct provide a very useful set of abstractions for composing asynchronous computations. However, in our context programmers often need some more high-level abstractions. Our li-

[5] For clarity we omitted the code for future pipelining here.

```
// bind :: (x-> future[y]) -> future[y]
function bind(f) {
  var R = new Future();
    self.register( function(v) {
      f(v).register( function(res){
          R.resolve(res);
        });
    });
    return R;
}
```

Listing 3: Monad bind in the Future Prototype of Connect.js

brary therefore defines a set of operators implemented on top of these basic monadic combinators. An overview of these operators is shown in tabel 1. Note that the in the type signature `f [a]` should be read as: A future which will resolve to an array of a's.

Conditionals	
if	`f a -> (a -> Bool) -> f a`
or	`f a -> f b -> (f a | f b)`
Simple synchronisation	
first	`f a -> f b -> (f a | f b)`
last	`f a -> f b -> (f a | f b)`
Group synchronisation	
many	`f[a]->(a->Bool)->f[a]`
some	`f[a]->(a->Bool)->f[a]`
all	`f[a]->(a->Bool)->f[a]`
Array operators	
filter	`f[a]->(a->Bool)->f[a]`
map	`f[a]->(a->b)->f[b]`

Table 1. Overview of the basic future combinators.

2.3.1 Future Combinator: Chat example

In this section we give a small example to showcase how the asynchronous nature of the communication can be hidden by the use of our future combinators. Consider the implementation of sending a friendly message to your two best buddies. We first define a (synchronous) function which checks that the user is either Christophe or Elisa (lines 1-4). Next we define a function which sends out a friendly message to each user in a given array of users (lines 5-7). Finally we first get the list of all our friends from the server (line 8), we filter this list (line 10), we transform the list of users to a list of user objects (line 11), and send a message to each of them with the function sendFriendlyMsgs (line 12).

3. Validation

In order to compare our approach to existing approaches we have implemented two example applications: a chat application and a distributed unit test suite. For both applications we have compared the percentage of lines which are attributed

```
var filterBuddy = function(user) {
   return "Christophe" == user.firstname
            || "Elisa" == user.firstname;
};
var sendFriendlyMsgs = function(users) {
        users.map(sendFriendlyMsg)
};
var msg = Ambient.createMessage("getFriendList");
server.asyncSend(msg, "twoway")
      .filter(filterBuddy)
      .reduce(makeBudObj,{})
      .whenBecomes(sendFriendlyMsgs);
```

Listing 4: Example use of future combinators

to the application logic compared to the lines of code for communication. The chat application has been implemented twice in Connect.js, once with network aware references and once without. For the unit testing framework we also implemented the tests twice, once with normal futures and once with future combinators. The results for both the chat application and the unit testing framework are shown in figure 2. As shown in the figure in both cases there is a clear shift from the percentage of lines spent for the application logic compared to the lines of code attributed to the communication logic. While not shown in the figure the absolute lines of code for each application also reduced.

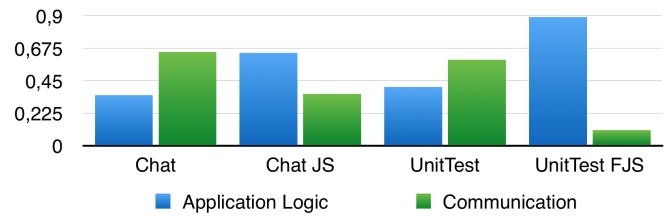


Figure 2. Comparison in usage distribution

4. Conclusion

In this paper we reported on initial work on Connect.js : a cross mobile-platform actor library for multi-networked RIA applications. With Connect.js the programmer does no longer need to write complex network code in order to exploit the use of both P2P communication and communication over a centralised wireless network. To this end, Connect.js provides the programmers with a new kind of distributed object reference called network transparant references. When one network interface fails these reference automatically and transparently tries to send the message over another network interface. Finally, in order to minimise the negative effects of writing asynchronous code Connect.js provides a set of future combinators. Initial validation of our artefact on two small applications shows promising results.

References

[1] S. Abolfazli, Z. Sanaei, A. Gani, F. Xia, and L. T. Yang. Review: Rich mobile applications: Genesis, taxonomy, and open issues. *J. Netw. Comput. Appl.*, 40:345–362, April 2014. ISSN 1084-8045. .

[2] T. V. Cutsem, E. G. Boix, C. Scholliers, A. L. Carreton, D. Harnie, K. Pinte, and W. D. Meuter. Ambienttalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures*, 40(34):112 – 136, 2014. ISSN 1477-8424. .

[3] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press. ISBN 0-8186-1954-6.

[4] S. Xanthopoulos and S. Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13, pages 213–220, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1851-8.

Extended Abstract: Towards Verified Privacy Policy Compliance of an Actor-based Electronic Medical Record Systems

Tom MacGahan Claiborne Johnson Armando L. Rodriguez Mark Appleby Jianwei Niu Jeffery von Ronne

Department of Computer Science
The University of Texas at San Antonio
vonronne@acm.org

1. Introduction

Many organizations use information systems to store, process, and communicate personal information about the individuals with which the organization interacts. These organizations needs to ensure that these information systems do not disclose information when not allowed by applicable privacy policies. One important instance of this are the electronic medical records systems utilized by health care providers. In the United States, these organizations are required to comply with the Privacy Rule of the Health Insurance Portability and Accountability Act and their own privacy policies. (In the rest of these papers, we will refer to these simply as "the privacy policy".) Verifying, whether a complex software system can cause an organization to violate the privacy policy, however, is not an easy task.

We have been developing a framework for building actor-based privacy policy compliant software systems. In this framework, the software system is decomposed into collections of actors instantiated from class and component definitions written in the domain-specific History Aware Programming Language (HAPL). Along the lines previously outlined in [2]. Each actor class definition and each component definition is also associated with formal specification written of relevant parts of its behavior. Actor code is verified against individual actor class specifications, and individual class specifications are shown to entail the specifications of larger component entities. The specification for the entire system is shown to entail the privacy policy. In this way, the system implementation is shown to implement the privacy policy.

We are currently in the process of creating the framework's build tools and runtime systems. We are also applying the framework to build a prototype electronic medical record system.

2. Framework Design and Implementation

Our framework can be viewed as facilitating two processes: creating a information system, and verifying its compliance with the privacy policy. The two are not unrelated, however, since the tools and techniques used to build the system are constrained so that the system can be more easily verified.

2.1 Language, Build Tools, and Runtime Support

In our framework, an information system is written in the History Aware Programming Language (HAPL), which provides facilities for defining *actor classes* and *component definitions*. In HAPL, an actor class is a static template from which actors can be instantiated at runtime. A component definition is a static description of a configuration of actors that can be instantiated at runtime. Component definitions can be defined in terms of actors instantiated from actor classes or subcomponents instantiated from other component definitions. A component definition is used to describe the system as whole.

A HAPL compiler translates actor classes and component definitions into Scala code using the Akka actor library. HAPL actor class code can make use a novel "dynamic history query" to make the execution of certain code dependent, at runtime, on the truth of first-order LTL formulas involving the messages in which the actor participates. This is supported through a runtime mechanism that dynamically tracks information necessary to answer potential queries. Special UI actor classes in the HAPL code generate Lift framework code to create actors that can communicate with a user through a web browser.

2.2 Verification Tools and Techniques

Our verification techniques aim to verify complete information systems against a formalization of the privacy policy in a first-order temporal logic.

Furthermore, our framework is built around a set of assume-guarantee specifications written in a first-order temporal logic. These specifications are associated with each actor class and each component definition. Specifications associated with component definitions describe not only the behavior of the actors in the initial configuration of components instantiated out of a definition but also describes the externally visible behavior of any new actors that are created as the component evolves from those in the initial configu-

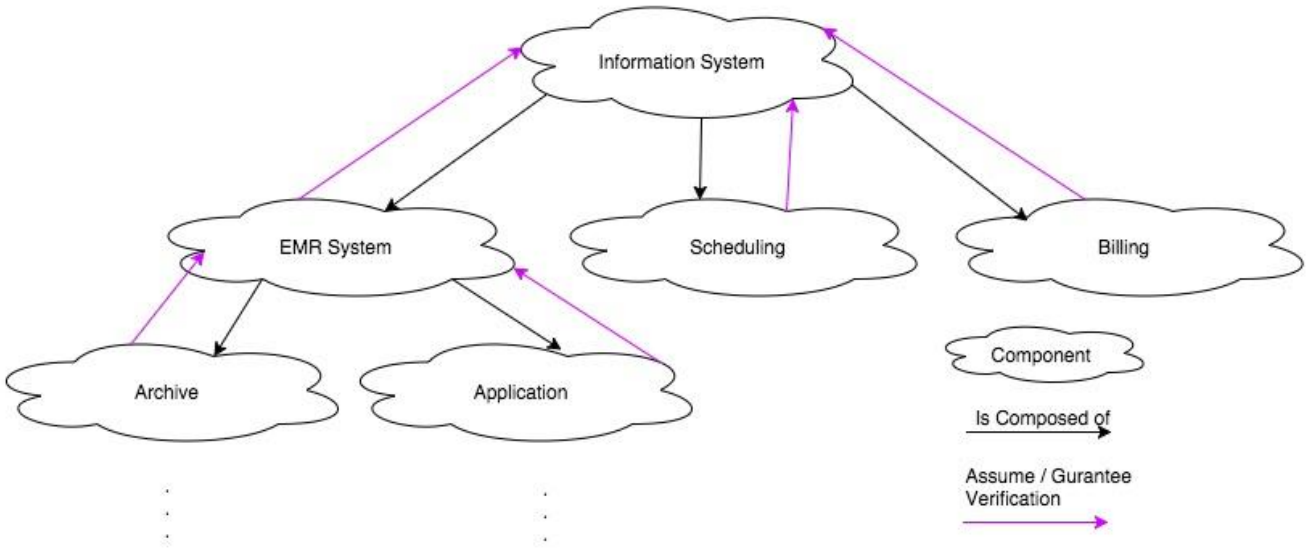


Figure 1. Component Structure of Prototype Medical Record System

ration. The specification associated with the "outer-most" whole-system component is taken as system specification.

Individual actor classes are verified against their specifications using an abstract interpretation that abstracts program traces according to the specification subformulas that are true when execution reaches a particular program point.

A variety of formal methods can be used to show that the refinement of a system specification into the specifications of the component definitions that realize it, as well as the specification of the component definitions into the subcomponent definition and actor class definitions. We are currently looking into utilizing model checking and Coq theorem proving for this purpose.

3. Prototype Medical System

In order to evaluate our framework, we have been using it to develop a prototype electronic medical record system. It consists of a patient and doctor user interface, an medical record archive, a scheduling subsystem and a billing subsystem. Patients are allowed to view their own record. Doctors can view and edit records of any patient for whom they claim to be giving care. The medical record archive is presumed to be persistent and is acts as a database.

The hierarchical component structure of the prototype medical system is shown if Figure 1. The "Information System" component is comprises of the "EMR System", "Scheduling", and "Billing" subcomponents. Our framework associates each of these components has an associated assume-guarantee specification. It is then necessary to show that the supercomponents specification is entailed by the specifications of the subcomponents and actors that comprise the supercomponent.

4. Future Work and Outlook

Although our policy language is well developed [1], many aspects of our framework are still incomplete. We are in the process of completing our runtime system implementation, our verification tools, and verifying our medical system prototype.

We are also still investigating the best method for verifying that the subcomponent assume/guarantee specifications compose to entail the supercomponent assume/guarantee specification; we are currently looking at using model checking tools or the Coq assisted theorem prover.

Our experience so far, however, leads us to believe in the appropriateness of our framework for building privacy-protecting actor-based information systems.

Acknowledgments

This work is dedicated to the memory of William Winsborough. The authors also thank Omar Chowdhuri, Shamim Ashik, and Sam Miller who have contributed to this project. This work was supported by the National Science Foundation under grant CNS-0964710.

References

[1] O. Chowdhury, A. Gampe, J. Niu, J. von Ronne, J. Bennatt, A. Datta, L. Jia, and W. H. Winsborough. Privacy promises that can be kept: A policy analysis method with application to the hipaa privacy rule. In *Proceedings of the 18th ACM Symposium on Access Control Models and Technologies*, SACMAT '13, pages 3–14, 2013.

[2] J. von Ronne. Leveraging actors for privacy compliance. In *Proceedings of the 2Nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*, AGERE! 2012, pages 133–136, 2012.