HAL open science

Simple Isolation for an Actor Abstract Machine

Benoit Claudel, Quentin Sabah, Jean-Bernard Stefani

To cite this version:

Benoit Claudel, Quentin Sabah, Jean-Bernard Stefani. Simple Isolation for an Actor Abstract Machine. 35th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE), Jun 2015, Grenoble, France. pp.213-227, 10.1007/978-3-319-19195-9_14 . hal-01767336

HAL Id: hal-01767336

https://inria.hal.science/hal-01767336v1

Submitted on 16 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Simple Isolation for an Actor Abstract Machine

Benoit Claudel[1], Quentin Sabah[2], and Jean-Bernard Stefani[3]

[1] The MathWorks, Grenoble, France
[2] Mentor Graphics, Grenoble, France
[3] INRIA Grenoble-Rhône-Alpes, France

Abstract. The actor model is an old but compelling concurrent programming model in this age of multicore architectures and distributed services. In this paper we study an as yet unexplored region of the actor design space in the context of concurrent object-oriented programming. Specifically, we show that a purely run-time, annotation-free approach to actor state isolation with reference passing of arbitrary object graphs is perfectly viable. In addition, we show, via a formal proof using the Coq proof assistant, that our approach indeed enforces actor isolation.

1 Introduction

Motivations. The actor model of concurrency [1], where isolated sequential threads of execution communicate via buffered asynchronous message-passing, is an attractive alternative to the model of concurrency adopted e.g. for Java, based on threads communicating via shared memory. The actor model is both more congruent to the constraints of increasingly distributed hardware architectures – be they local as in multicore chips, or global as in the world-wide web –, and more adapted to the construction of long-lived dynamic systems, including dealing with hardware and software faults, or supporting dynamic update and reconfiguration, as illustrated by the Erlang system [2]. Because of this, we have seen in the recent years renewed interest in implementing the actor model, be that at the level of experimental operating systems as in e.g. Singularity [9], or in language libraries as in e.g. Java [24] and Scala [13].

When combining the actor model with an object-oriented programming model, two key questions to consider are the exact semantics of message passing, and its efficient implementation, in particular on multiprocessor architectures with shared physical memory. To be efficient, an implementation of message passing on a shared memory architecture ought to use data transfer by reference, where the only data exchanged is a pointer to the part of the memory that contains the message. However, with data transfer by reference, enforcing the share-nothing semantics of actors becomes problematic: once an arbitrary memory reference is exchanged between sender and receiver, how do you ensure the sender can no longer access the referenced data ? Usual responses to this question, typically involve restricting the shape of messages, and controlling references (usually through a reference uniqueness scheme [19]) by various means, including run-time support, type systems and other static analyses, as in Singularity [9], Kilim [24], Scala actors [14], and SOTER [21].

Contributions. In this paper, we study a point in the actor model design space which, despite its simplicity, has never, to our knowledge, been explored before. It features a very simple programming model that places *no restriction* on the shape and type of messages, and *does not require* special types or annotations for references, yet still enforces the share nothing semantics of the actor model. Specifically, we introduce an actor abstract machine, called Siaam. Siaam is layered on top of a sequential object-oriented abstract machine, has actors running concurrently using a shared heap, and enforces strict actor isolation by means of run-time barriers that prevent an actor from accessing objects that belong to a different actor. The contributions of this paper can be summarized as follows. We formally specify the Siaam model, building on the Jinja specification of a Java-like sequential language [18]. We formally prove, using the Coq proof assistant, the strong isolation property of the Siaam model. We describe our implementation of the Siaam model as a modified Jikes RVM [16]. We present a novel static analysis, based on a combination of points-to, alias and liveness analyses, which is used both for improving the run-time performance of Siaam programs, and for providing useful debugging support for programmers. Finally, we evaluate the performance of our implementation and of our static analysis.

Outline. The paper is organized as follows. Section 2 presents the Siaam machine and its formal specification. Section 3 presents the formal proof of its isolation property. Section 4 describes the implementation of the Siaam machine. Section 5 presents the Siaam static analysis. Section 6 presents an evaluation of the Siaam implementation and of the Siaam analysis. Section 7 discusses related work and concludes the paper. Because of space limitations, we present only some highlights of the different developments. Interested readers can find all the details in the second author's PhD thesis [22], which is available online along with the Coq code [25].

2 Siaam: model and formal specification

Informal presentation. Siaam combines actors and objects in a programming model with a single shared heap. Actors are instances of a special class. Each actor is equipped with at least one mailbox for queued communication with other actors, and has its own logical thread of execution that runs concurrently with other actor threads. Every object in Siaam belongs to an actor, we call its *owner*. An object has a unique owner. Each actor is its own owner. At any point in time the ownership relation forms a partition of the set of objects. A newly created object has its owner set to that of the actor of the creating thread.

Siaam places absolutely *no restriction* on the references between objects, including actors. In particular objects with different owners may reference each other. Siaam also places *no constraint* on what can be exchanged via messages: the contents of a message can be an arbitrary object graph, defined as the graph of objects reachable (following object references in object fields) from a root object specified when sending a message. Message passing in Siaam has a zero-copy semantics, meaning that the object graph of a message is not copied from

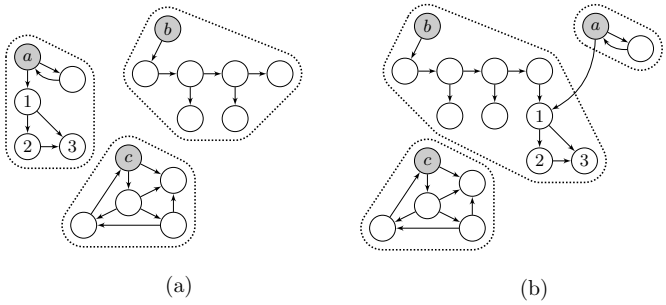

Fig. 1. Ownership and ownership transfer in Siaam

the sender actor to the receiver actor, only the reference to the root object of a message is communicated. An actor is only allowed to send objects it owns[4], and it cannot send itself as part of a message content.

Figure 1 illustrates ownership and ownership transfer in Siaam. On the left side (a) is a configuration of the heap and the ownership relation where each actor, presented in gray, owns the objects that are part of the same dotted convex hull. Directed edges are heap references. On the right side (b), the objects $1, 2, 3$ have been transferred from a to b, and object 1 has been attached to the data structure maintained in b's local state. The reference from a to 1 has been preserved, but actor a is no longer allowed to access the fields of $1, 2, 3$.

To ensure isolation, Siaam enforces the following invariant: an object o (in fact an executing thread) can only access fields of an object that has the same owner than o; any attempt to access the fields of an object of a different owner than the caller raises a run-time exception. To enforce this invariant, message exchange in Siaam involves twice changing the owner of all objects in a message contents graph: when a message is enqueued in a receiver mailbox, the owner of objects in the message contents is changed atomically to a null owner ID that is never assigned to any actor ; when the message is dequeued by the receiver actor, the owner of objects in the message contents is changed atomically to the receiver actor. This scheme prevents pathological situations where an object passed in a message m may be sent in another message m' by the receiver actor without the latter having dequeued (and hence actually received) message m.

[4] Siaam enforces the constraint that all objects reachable from a message root object have the same owner – the sending actor. If the constraint is not met, sending the message fails. However, this constraint, which makes for a simple design, is just a design option. An alternative would be to consider that a message contents consist of all the objects reachable from the root object which have the sending actor as their owner. This alternate semantics would not change the actual mechanics of the model and the strong isolation enforced by it.

$$\textsc{Read}\ \frac{hp\ s\ a = \texttt{Some}\ (C, fs) \qquad fs(F, D) = \texttt{Some}\ v}{P, w \vdash \langle a.F\{D\}, s\rangle - \texttt{OwnerCheck}\ a\ True \rightarrow \langle \texttt{Val}\ v, s\rangle}$$

$$\textsc{ReadX}\ P, w \vdash \langle a.F\{D\}, s\rangle - \texttt{OwnerCheck}\ a\ False \rightarrow \langle \texttt{Throw}\ OwnerMismatch, s\rangle$$

$$\textsc{Global}\ \frac{\begin{array}{c} acs\ s\ w = \texttt{Some}\ x \qquad P, w \vdash \langle x, shp\ s\rangle - wa \rightarrow \langle x', h'\rangle \qquad \texttt{ok_act}\ P\ s\ w\ wa \\ \texttt{upd_act}\ P\ s\ w\ wa = (xs', ws', ms', _) \qquad s' = (xs'[w \mapsto x'], ws', ms', h') \end{array}}{P \vdash s \rightarrow s'}$$

Fig. 2. Siaam operational semantics: sample rules

Since Siaam does not modify object references in any way, the sender actor can still have references to objects that have been sent, but any attempt from this sender actor to access them will raise an exception.

Siaam: model and formal specification. The formal specification of the Siaam model defines an operational semantics for the Siaam language, in the form of a reduction semantics. The Siaam language is a Java-like language, for its sequential part, extended with special classes with native methods corresponding to operations of the actor model, e.g. sending and receiving messages. The semantics is organized in two layers, the *single-actor* semantics and the *global* semantics. The single-actor semantics deals with evolutions of individual actors, and reduces actor-local state. The global semantics maintains a global state not directly accessible from the single-actor semantics. In particular, the effect of reading or updating object fields by actors belongs to the single-actor semantics, but whether it is allowed is controlled by the global semantics. Communications are handled by the global semantics.

The single actor semantics extends the Jinja formal specification in HOL of the reduction semantics of a (purely sequential) Java-like language [18] [5]. Jinja gives a reduction semantics for its Java-like language via judgments of the form $P \vdash \langle e, (lv, h)\rangle \rightarrow \langle e', (lv', h')\rangle$, which means that in presence of program P (a list of class declarations), expression e with a set of local variables lv and a heap h reduces to expression e' with local variables lv' and heap h'.

We extend Jinja judgments for our single-actor semantics to take the form $P, w \vdash \langle e, (lv, h)\rangle - wa \rightarrow \langle e', (lv', h')\rangle$ where $\langle e, lv\rangle$ corresponds to the local actor state, h is the global heap, w is the identifier of the current actor (owner), and wa is the actor action requested by the reduction. Actor actions embody the Siaam model per se. They include creating new objects (with their initial owner), including actors and mailboxes, checking the owner of an object, sending and receiving messages. For instance, succesfully accessing an object field is governed by rule Read in Figure 2. Jinja objects are pairs (C, fs) of the object class name

[5] Jinja, as described in [18], only covers a subset of the Java language. It does not have class member qualifiers, interfaces, generics, or concurrency.

C and the field table fs. A field table is a map holding a value for each field of an object, where fields are identified by pairs (F, D) of the field name F and the name D of the declaring class. The premisses of rule READ retrieve the object referenced by a from the heap ($hp\ s\ a = \texttt{Some}\ (C, fs)$ – where hp is the projection function that retrieves the heap component of a local actor state, and the heap itself is an association table modelled as a function that given an object reference returns an object), and the value v held in field F. In the conclusion of rule READ, reading the field F from a returns the value v, with the local state s (local variables and heap) unchanged. The actor action `OwnerCheck` $a\ True$ indicates that object a has the current actor as its owner. Apart from the addition of the actor action label, rule READ is directly lifted from the small step semantics of Jinja in [18]. In the case of field access, the rule READ is naturally complemented with rule READX, that raises an exception if the owner check fails, and which is specific to Siaam. Actor actions also include a special *silent* action, that corresponds to single-actor reductions (including exception handling) that require no access to the global state. Non silent actor actions are triggered by object creation, object field access, and *native calls*, i.e. method calls on the special actor and mailbox classes.

The global semantics is defined by the rule GLOBAL in Figure 2. The judgment, written $P \vdash s \rightarrow s'$, means in presence of program P, global state s reduces to global state s'. The global state (xs, ws, ms, h) of a Siaam program execution comprises four components: the actor table xs, an ownership relation ws, the mailbox table ms, and a shared heap h. The projection functions acs, ows, mbs, shp return respectively the actor table, the ownerships relation, the mailbox table, and the shared heap component of the global state. The actor table associates an actor identifier to an actor local state consisting of a pair $\langle e, lv \rangle$ of expression and local variables. The rule GLOBAL reduces the global state by applying a single step of the single-actor semantics for actor w. In the premises of the rule, the shared heap $shp\ s$ and the current local state x (expression and local variables) for w are retrieved from the global state. The actor can reduce to x' with new shared heap h' and perform the action wa. `ok_act` tests the actor action precondition against s. If it is satisfiable, `upd_act` applies the effects of wa to the global state, yielding the new tuple of state components $(xs', ws', ms', _)$ where the heap is left unchanged. The new state s' is assembled from the new mailbox table, the new ownership relation, the new heap from the single actor reduction and the new actor table where the state for actor w is updated with its new local state x'. We illustrate the effect of actor actions in the next section.

3 Siaam: Proof of isolation

The key property we expect the Siaam model to uphold is the strong isolation (or share nothing) property of the actor model, meaning actors can only exchange information via message passing. We have formalized this property and proved it using the Coq proof assistant (v8.4) [8]. We present in this section some key elements of the formalization and proof, using excerpts from the Coq code. The

formalization uses an abstraction of the operational semantics presented in the previous section. Specifically, we abstract away from the single-actor semantics. The local state of an actor is abstracted as being just a table of local variables (no expression), which may change in obvious ways: adding or removing a local variable, changing the value held by a local variable. The formalization (which we call Abstract Siaam) is thus a generalization of the Siaam operational semantics.

Abstract Siaam: Types. The key data structure in Abstract Siaam is the *configuration*, defined as an abstraction of the global state in the previous section. A configuration `conf` is a tuple comprising an actor table, an ownership relation, a mailbox table and a shared heap. In Coq:

```
Record conf : Type := mkcf { acs : actors; ows : owners; mbs : mailboxes; shp : heap }.
```

Actor table, ownership relation, mailbox table and heap are all defined as simple association tables, i.e. lists of pairs $\langle i, d \rangle$ of identifiers i and data d:

```
Definition actors := table aid locals.          Definition actor := prod aid locals.
Definition owners := table addr (option aid).   Definition mailboxes := table mid mbox.
Definition heap := table addr object.
```

Identifiers `aid`, `addr`, and `mid` correspond to actor identifiers, object references, and mailbox identifiers, respectively. The data `locals` is a table of local variables (with identifier type `vid`), an actor is just a pair associating an actor identifier with a table of local variables, and a mailbox `mbox` is a list of messages associated with an actor identifier (the actor receiving messages via the mailbox):

```
Definition locals := table vid value.    Definition message := prod msgid addr.
Definition queue := list message.        Record mbox : Type := mkmb { own : aid ; msgs : queue}.
```

A message is just a pair consisting of a message identifier and a reference to a root object. A value can be either the null value (`vnull`), the *mark* value (`vmark`), an integer (`vnat`), a boolean (`vbool`), an object reference, an actor id or a mailbox id. The special *mark* value is simply a distinct value used to formalize the isolation property.

Abstract Siaam: Transition rules. Evolution of a Siaam system are modeled in Abstract Siaam as transitions between configurations, which are in turn governed by transition rules. Each transition rule in Abstract Siaam corresponds to an an instance of the Global rule in the Siaam operational semantics, specialized for dealing with a given actor action. For instance, the rule governing field access, which abstracts the global semantics reduction picking the `OwnerCheck` $a\ True$ action offered by a Read reduction of the single-actor semantics (cf. Figure 2) carrying the identifier of actor e, and accessing field f of object o referenced by a is defined as follows:

```
Inductive redfr : conf → aid → conf → Prop :=
| redfr_step : ∀ (c1 c2 : conf)(e : aid)(l1 l2 : locals)(i j : vid)(v w : value)(a: addr)
                 (o : object)(f: fid),
  set_In (e,l1) (acs c1) → set_In (i, w) l1 → set_In (j,vadd a) l1 →
    set_In (a,o) (shp c1) → set_In (f,v) o → set_In (a, Some e) (ows c1) →
      v_compat w v → l2 = up_locals i v l1 →
          c2 = mkcf (up_actors e l2 (acs c1)) (ows c1) (mbs c1) (shp c1) →
              c1 =fr e ⇒ c2
where " t '=fr' a '⇒' t' " := (redfr t a t').
```

The conclusion of the rule, `c1 =fr e ⇒ c2`, states that configuration `c1` can evolve into configuration `c2` by actor `e` doing a field access `fr`. The premises of the rule are the obvious ones: `e` must designate an actor of `c1`; the table `l1` of local variables of actor `e` must have two local variables `i` and `j`, one holding a reference `a` to the accessed object (`set_In (j,vadd a) l1`), the other some value `w` (`set_In ( i, w) l1`) compatible with that read in the accessed object field (`v_compat w v`); `a` must point to an object `o` in the heap of `c1` (`set_In (a,o) (shp c1) )`, which must have a field `f`, holding some value `v` (`set_In (f,v) o)` ; and actor `e` must be the owner of object `o` for the field access to succeed (`set_In (a, Some e) (ows c1)`). The final configuration `c2` has the same owernship relation, mailbox table and shared heap than the initial one `c1`, but its actor table is updated with new local state of actor `e` (`c2 = mkcf (up_actors e l2 (acs c1)) (ows c1) (mbs c1) (shp c1)`), where variable `i` now holds the read value `v` (`l2 = up_locals i v l1`).

Another key instance of the Abstract Siaam transition rules is the rule presiding over message send:

```
Inductive redsnd : conf → aid → conf → Prop :=
 | redsnd_step : ∀ (c1 c2 : conf)(e : aid) (a : addr) (l : locals) (ms: msgid)(mi: mid)
                  (mb mb': mbox)(owns : owners),
    set_In (e,l) (acs c1) →
     set_In (vadd a) (values_from_locals l) →
      trans_owner_check (shp c1) (ows c1) (Some e) a = true →
       set_In (mi,mb) (mbs c1) →
        not (set_In ms (msgids_from_mbox mb)) →
         Some owns = trans_owner_update (shp c1) (ows c1) None a →
          mb' = mkmb (own mb) ((ms,a)::(msgs mb)) →
           c2 = mkcf (acs c1)  owns  (up_mboxes mi mb' (mbs c1))  (shp c1) →
            c1 =snd e ⇒ c2
where " t '=snd' a '⇒' t' " := (redsnd t a t').
```

The conclusion of the rule, `c1 =snd e ⇒ c2`, states that configuration `c1` can evolve into configuration `c2` by actor `e` doing a message send `snd`. The premises of the rule expects the owner of the objects reachable from the root object (referenced by `a`) of the message to be `e`; this is checked with function `trans_owner_check` : `trans_owner_check (shp c1) (ows c1) (Some e) a = true`. When placing the message in the mailbox `mb` of the receiver actor, the owner of all the objects reachable is set to `None`; this is done with function `trans_owner_update`: `Some owns = trans_owner_update (shp c1) (ows c1) None a`. Placing the message with id `ms` and root object referenced by `a` in the mailbox is just a matter of queuing it in the mailbox message queue: `mb' = mkmb (own mb) ((ms,a)::(msgs mb))`.

The transition rules of Abstract Siaam also include a rule governing silent transitions, i.e. transitions that abstract from local actor state reductions that elicit *no* change on other elements of a configuration (shared heap, mailboxes, ownership relation, other actors). The latter are just modelled as transitions arbitrarily modifying a given actor local variables, with no acquisition of object references that were previously unknown to the actor.

Isolation proof. The Siaam model ensures that the only means of information transfer between actors is message exchange. We can formalize this isolation property using mark values. We call an actor `a` *clean* if its local variables do not hold a mark, and if all objects *reachable* from `a` and belonging to `a` hold no

mark in their fields. An object `o` is reachable from an actor `a` if `a` has a local variable holding `o`'s reference, or if, recursively, an object `o'` is reachable from `a` which holds `o`'s reference in one of its fields. The isolation property can now be characterized as follows: a clean actor in any configuration remains clean during an evolution of the configuration if it never receives any message. In Coq:

```
Theorem ac_isolation : ∀ (c1 c2 : conf) (a1 a2: actor),
  wf_conf c1 → set_In a1 (acs c1) → ac_clean (shp c1) a1 (ows c1) →
      c1 =@ (fst a1) ⇒∗ c2 → Some a2 = lookup_actor (acs c2) (fst a1) →
         ac_clean (shp c2) a2 (ows c2).
```

The theorem states that, in any well-formed configuration `c1`, an actor `a1` which is clean (`ac_clean (shp c1) a1 (ows c1)`), remains clean in any evolution of `c1` that does not involve a reception by `a1`. This is expressed as `c1 =@ (fst a1) ⇒∗ c2` and `ac_clean (shp c2) a2 (ows c2)`, where `fst a1` just extracts the identifier of actor `a1`, and `a2` is the descendant of actor `a1` in the evolution (it has the same actor identifier than `a1`: `Some a2 = lookup_actor (acs c2) (fst a1)`). The relation `=@ a ⇒∗`, which represents evolutions not involving a message receipt by actor `a`, is defined as the reflexive and transitive closure of relation `=@ a ⇒`, which is a one step evolution not involving a receipt by `a`. The isolation theorem is really about transfer of information between actors, the mark denoting a distinguished bit of information held by an actor. At first sight it appears to say nothing about about ownership, but notice that a clean actor a is one such that all objects that belong to a are clean, i.e. hold no mark in their fields. Thus a corollary of the theorem is that, in absence of message receipt, actor a cannot acquire an object from another actor (if that was the case, transferring the ownership of an unclean object would result in actor a becoming unclean).

A well-formed configuration is a configuration where each object in the heap has a single owner, all identifiers are indeed unique, where mailboxes hold messages sent by actors in the actor table, and all objects referenced by actors (directly or indirectly, through references in object fields) belong to the heap. To prove theorem `ac_isolation`, we first prove that well-formedness is an invariant in any configuration evolution:

```
Theorem red_preserves_wf : ∀ (c1 c2 : conf), c1 ⇒ c2 → wf_conf c1 → wf_conf c2.
```

The theorem `red_preserves_wf` is proved by induction on the derivation of the assertion `c1 ⇒ c2`. To prove the different cases, we rely mostly on simple reasoning with sets, and a few lemmas characterizing the correctness of table manipulation functions, of the `trans_owner_check` function which verifies that all objects reachable from the root object in a message have the same owner, and of the `trans_owner_update` function which updates the ownership table during message transfers. Using the invariance of well-formedness, theorem `ac_isolation` is proved by induction on the derivation of the assertion `c1 =@ (fst a1) ⇒∗ c2`. To prove the different cases, we rely on several lemmas dealing with reachability and cleanliness.

The last theorem, `live_mark`, is a liveness property that shows that the isolation property is not vacuously true. It states that marks *can* flow between actors during execution. In Coq:

```
Theorem live_mark : ∃ (c1 c2 : conf)(ac1 ac2 : actor),
  c1 ⇒∗ c2 ∧ set_In ac1 (acs c1) ∧ ac_clean (shp c1) ac1 (ows c1)
           ∧ Some ac2 = lookup_actor (acs c2) (fst ac1) ∧ ac_mark (shp c2) ac2 (ows c2).
```

4 Siaam: Implementation

We have implemented the Siaam abstract machine as a modified Jikes RVM [16]. Specifically, we extended the Jikes RVM bytecode and added a set of core primitives supporting the ownership machinery, which are used to build trusted APIs implementing particular programming models. The Siaam programming model is available as a trusted API that implements the formal specification presented in Section 2. On top of the Siaam programming model, we implemented the ActorFoundry API as described in [17], which we used for some of our evaluation. Finally we implemented a trusted event-based actor programming model on top of the core primitives, which can dispatch thousand of lightweight actors over pools of threads, and enables to build high-level APIs similar to Kilim with Siaam's ownership-based isolation.

Bytecode. The standard Java VM instructions are extended to include: a modified object creation instruction `New`, which creates an object on the heap and sets its owner to that of the creating thread; modified field read and write acess instructions `getfield` and `putfield` with owner check; modified instructions load and store array instructions `aload` and `astore` with owner check.

Virtual machine core. Each heap object and each thread of execution have an owner reference, which points to an object implementing the special `Owner` interface. A thread can only access objects belonging to the `Owner` instance referenced by its owner reference. Core primitives include operations to retrieve and set the owner of the current thread, to retrieve the owner of an object, to withdraw and acquire ownership over objects reachable from a given root object. In the Jikes RVM, objects are represented in memory by a sequence of bytes organized into a leading header section and the trailing scalar object's fields or array's length and elements. We extended the object header with two reference-sized words, `OWNER` and `LINK`. The `OWNER` word stores a reference to the object owner, whereas the `LINK` word is introduced to optimize the performance of object graph traversal operations.

Contexts. Since the JikesRVM is fully written in Java, threads seamlessly execute application bytecode and the virtual machine internal bytecode. We have introduced a notion of execution context in the VM to avoid subjecting VM bytecode to the owner-checking mechanisms. A method in the *application context* is instrumented with all the isolation mechanisms whereas methods in the *VM context* are not. If a method can be in both context, it must be compiled in two versions, one for both contexts. When a method is invoked, the context of the caller is used to deduce which version of the method should be called. The decision is taken statically when the invoke instruction is compiled.

Owernship transfer. Central to the performance of the Siaam virtual machine are operations implementing ownership transfer, `withdraw` and `acquire`. In the formal specification, owner-checking an object graph and updating the owner of objects in the graph is done atomically (see e.g. the message send transition rule in Section 3). However implementing the `withdraw` operation as an atomic operation would be costly. Furthermore, an implementation of ownership transfer must minimize graph traversals. We have implemented an iterative algo-

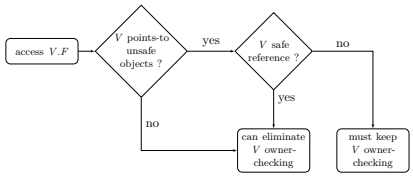

Fig. 3. Owner-check elimination decision diagram

rithm for `withdraw` that chains objects that are part of a message through their `LINK` word. The list thus obtained is maintained as long as the message exists so that the `acquire` operation can efficiently traverse the objects of the message. The algorithm leverages specialized techniques, initially introduced in the Jikes RVM to optimize the reference scanning phase during garbage collection [10], to efficiently enumerate the reference offsets for a given base object.

5 Siaam: Static Analysis

We describe in this section some elements of Siaam static analysis to optimize away owner-checking on field read and write instructions. The analysis is based on the observation that an instruction accessing an object's field does not need an owner-checking if the object accessed belongs to the executing actor. Any object that has been allocated or received by an actor and has not been passed to another actor ever since, belongs to that actor. The algorithm returns an under-approximation of the owner-checking removal opportunities in the analyzed program.

Considering a point in the program, we say an object (or a reference to an object) is *safe* when it always belongs to the actor executing that point, regardless of the execution history. By opposition, we say an object is *unsafe* when sometimes it doesn't belong to the current actor. We extend the denomination to instructions that would respectively access a safe object or an unsafe object. A safe instruction will never throw an `OwnerException`, whereas an unsafe instruction might.

Analysis. The Siaam analysis is structured in two phases. First the safe dynamic references analysis employs a local must-alias analysis to propagate owner-checked references along the control-flow edges. It is optionally refined with an inter-procedural pass propagating safe references through method arguments and returned values. Then the safe objects analysis tracks safe runtime objects along call-graph and method control-flow edges by combining an inter-procedural points-to analysis and an intra-procedural live variable analysis. Both phases depend on the transfered abstract objects analysis that propagates unsafe abstract objects from the communication sites downward the call graph edges.

By combining results from the two phases, the algorithm computes conservative approximations of unsafe runtime objects and safe variables at any control-flow point in the program. The owner-check elimination for a given instruction

s accessing the reference in variable V proceeds as illustrated in Figure 3. First the unsafe objects analysis is queried to know whether V may points-to an unsafe runtime object at s. If not, the instruction can skip the owner-check for V. Otherwise, the safe reference analysis is consulted to know whether the reference in variable V is considered safe at s, thanks to dominant owner-checks of the reference in the control-flow graph.

The Siaam analysis makes use of several standard intra and inter-prodedural program analyses: a call-graph representation, an inter-procedural points-to analysis, an intra-procedural liveness analysis, and an intra-procedural must-alias analysis. Each of these analyses exists in many different variants offering various tradeoffs of results accuracy and algorithmic complexity, but regardless of the implementation, they provide a rather standard querying interface. Our analysis is implemented as a framework that can make use of different instances of these analyses.

Implementations. The intra-procedural safe reference analysis which is part of the Siaam analysis has been included in the Jikes RVM optimizing compiler. Despite its relative simplicity and its very conservative assumptions, it efficiently eliminates about half of the owner-check barriers introduced by application bytecode and the standard library for the benchmarks we have tested (see Section 6). The safe reference analysis and the safe object analyses from the Siaam analysis have been implemented in their inter-procedural versions as an offline tool written in Java. The tool interfaces with the Soot analysis framework [23], that provides the program representation, the call graph, the inter-procedural pointer analysis, the must-alias analysis and the liveness analysis we use.

Programming assistant. The Siaam programming model is quite simple, requiring no programmer annotation, and placing no constraint on messages. However, it may generate hard to understand runtime exceptions due to failed owner-checks. The Siaam analysis is therefore used as the basis of a programming assistant that helps application developers understand why a given program statement is potentially unsafe and may throw an owernship exception at runtime. The Siaam analysis guarantees that there will be no false negative, but to limit the amount of false positives it is necessary to use a combination of the most accurate standard (points-to, must-alias and liveness) analyses. The programming assistant tracks a program P backward, starting from an unsafe statement s with a non-empty set of unverified ownerhip preconditions (as given by the `ok_act` function in Section 2), trying to find every program points that may explain why a given precondition is not met at s. For each unsatisfied precondition, the assistant can exhibit the shortest execution paths that result in an exception being raised at s. An ownership precondition may comprise requirements that a variable or an object be safe. When a requirement is not satisfied before s, it raises one or several questions of the form "why is x unsafe before s?". The assistant traverses the control-flow backward, looks for immediate answers at each statement reached, and propagates the questions further if necessary, until all questions have found an answer.

6 Evaluation

Siaam Implementation. We present first an evaluation of the overall performance of our Siaam implementation based on the DaCapo benchmark suite [3], representative of various real industrial workloads. These applications use regular Java. The bytecode is instrumented with Siaam's owner-checks and all threads share the same owner. With this benchmark we measure the overhead of the dynamic ownership machinery, encompassing the object owner initialization and the owner-checking barriers, plus the allocation and collection costs linked to the object header modifications.

We benchmarked five configurations. `no siaam` is the reference Jikes RVM without modifications. `opt` designates the modified Jikes RVM with JIT owner-checks elimination. `noopt` designates the modified Jikes RVM without JIT owner-checks elimination. `sopt` is the same as `opt` but the application bytecode has safety annotations issued by the offline Siaam static analysis tool. Finally `soptnc` is the same as `sopt` without owner-check barriers for the standard library bytecode. We executed the 2010-MR2 version of the DaCapo benchamrks, with two workloads, the `default` and the `large`. Table 1 shows the results for the Dacapo 2010-MR2 runs. The results were obtained using a machine equipped with an Intel Xeon W3520 2.67Ghz processor. The execution time results are normalized with respect to the `no-siaam` configuration for each program of the suite: *lower is better*. The geometric mean summarizes the typical overhead for each configuration. The `opt` figures in Table 1 show that the modified virtual machine including JIT barrier elimination has an overhead of about 30% compared to the not-isolated reference. The JIT elimination improves the performances by about 20% compared to the `noopt` configuration. When the bytecode is annotated by the whole-program static analysis the performance is 10% to 20% better than with the runtime-only optimization. However, the DaCapo benchmarks use the Java reflection API to load classes and invoke methods, meaning our static analysis was not able to process all the bytecode with the best precision. We can expect better results with other programs for which the call graph can be entirely built with precision. Moreover we used for the benchmarks a context-insensitive, flow-insensitive pointer analysis, meaning the Siaam analysis could be even more accurate with sensitive standard analyses. Finally the standard library bytecode is not annotated by our tool, it is only treated by the JIT elimination optimization. The `soptnc` configuration provides a good indication of what the full optimization would yield. The results show an overhead (w.r.t. application) with a mean of 15%, which can be considered as an acceptable price to pay for the simplicity of developing isolated programs with Siaam.

The Siaam virtual machine consumes more heap space than the unmodified Jikes RVM due to the duplication of the standard library used by both the virtual machine and the application, and because of the two words we add in every object's header. The average object size in the DaCapo benchmarks is 62 bytes, so our implementations increases it by 13%. We have measured a 13% increase in the full garbage collection time, which accounts for the tracing of the two additional references and the memory compaction.

Benchmark	opt	noopt	sopt	soptnc	opt	noopt	sopt	soptnc
workload	*default*				*large*			
`antlr`	1.20	1.32	1.09	1.11	1.21	1.33	1.11	1.10
`bloat`	1.24	1.41	1.17	1.05	1.40	1.59	1.14	0.96
`hsqldb`	1.24	1.36	1.09	1.06	1.45	1.60	1.29	1.10
`jython`	1.52	1.73	1.41	1.24	1.45	1.70	1.45	1.15
`luindex`	1.25	1.46	1.09	1.05	1.25	1.43	1.09	1.03
`lusearch`	1.31	1.45	1.17	1.18	1.33	1.49	1.21	1.21
`pmd`	1.32	1.37	1.29	1.24	1.34	1.44	1.39	1.30
`xalan`	1.24	1.39	1.33	1.35	1.29	1.41	1.38	1.40
geometric mean	1.28	1.43	1.20	1.16	1.34	1.50	1.25	1.15

Table 1. DaCapo benchmarks

Siaam Analysis. We compare the efficiency of the Siaam whole-program analysis to the SOTER algorithm, which is closest to ours. Table 2 contains the results that we obtained for the benchmarks reported in [21], that use ActorFoundry programs. For each analyzed application we give the total number of owner-checking barriers and the total number of message passing sites in the bytecode. The columns "Ideal safe" show the expected number of safe sites for each criteria. The column " Siaam safe" gives the result obtained with the Siaam analysis. The analysis execution time is given in the third main colum. The last column compares the result ratio to ideal for both SOTER and Siaam. Our analysis outperforms SOTER significantly. SOTER relies on an inter-procedural live-analysis and a points-to analysis to infer message passing sites where a by-reference semantics can applies safely. Given an argument a_i of a message passing site s in the program, SOTER computes the set of objects passed by a_i and the set of objects transitively reachable from the variables live after s. If the intersection of these two sets is empty, SOTER marks a_i as eligible for by-reference argument passing, otherwise it must use the default by-value semantic. The weakness to this pessimistic approach is that among the live objects, a significant part won't actually be accessed in the control-flow after s. On the other hand, Siaam do care about objects being actually accessed, which is a stronger evidence criterion to incriminate message passing sites. Although Siaam's algorithm wasn't designed to optimize-out by-value message passing, it is perfectly adapted for that task. For each unsafe instruction detected by the algorithm, there is one or several guilty dominating message passing sites. Our diagnosis algorithm tracks back the application control-flow from the unsafe instruction to the incriminated message passing sites. These sites represent a subset of the sites where SOTER cannot optimize-out by-value argument passing.

7 Related Work and Conclusion

Enforcing isolation between different groups of objects, programs or threads in presence of a shared memory has been much studied in the past two decades. Although we cannot give here a full survey of the state of the art (a more in depth analysis is available in [22]), we can point out three different kinds of

	Ownercheck			Message Passing				ratio to Ideal	
		Ideal	Siaam		Ideal	Siaam	Time		
	Sites	safe	safe	Sites	safe	safe	(sec)	Siaam	SOTER
ActorFoundry									
`threadring`	24	24	24	8	8	8	0.1	100%	100%
(1) `concurrent`	99	99	99	15	12	10	0.1	98%	58%
(2) `copymessages`	89	89	84	22	20	15	0.1	91%	56%
`performance`	54	54	54	14	14	14	0.2	100%	86%
`pingpong`	28	28	28	13	13	13	0.1	100%	89%
`refmessages`	4	4	4	6	6	6	0.1	100%	67%
Benchmarks									
`chameneos`	75	75	75	10	10	10	0.1	100%	33%
`fibonacci`	46	46	46	13	13	13	0.2	100%	86%
`leader`	50	50	50	10	10	10	0.1	100%	17%
`philosophers`	35	35	35	10	10	10	0.2	100%	100%
`pi`	31	31	31	8	8	8	0.1	100%	67%
`shortestpath`	147	147	147	34	34	34	1.2	100%	88%
Synthetic									
`quicksortCopy`	24	24	24	8	8	8	0.2	100%	100%
(3) `quicksortCopy2`	56	56	51	10	10	5	0.1	85%	75%
Real world									
`clownfish`	245	245	245	102	102	102	2.2	100%	68%
(4) `rainbow_fish`	143	143	143	83	82	82	0.2	99%	99%
`swordfish`	181	181	181	136	136	136	1.7	100%	97%

Table 2. ActorFoundry analyses.

related works: those relying on type annotations to ensure isolation, those relying on run-time mechanisms, and those relying on static analyses.

Much work has been done on controlling aliasing and encapsulation in object-oriented languages and systems, in a concurrent context or not. Much of the works in these areas rely on some sort of reference uniqueness, that eliminates object sharing by making sure that there is only one reference to an object at any time, e.g. [5, 14, 15, 19, 20]. All these systems restrict the shape of object graphs or the use of references in some way. In contrast, Siaam makes no such restriction. A number of systems rely on run-time mechanisms for achieving isolation, most using either deep-copy or special message heaps for communication, e.g. [7, 9, 11, 12]. Of these, O-Kilim [12], which builds directly on the PhD work of the first author of this paper [6], is the closest to Siaam: it places no constraint on transferred object graphs, but at the expense of a complex programming model and no programmer support, in contrast to Siaam. Finally several works develop static analyses for efficient concurrency or ownership transfer, e.g. [4, 21, 24]. Kilim [24] relies in addition on type annotations to ensure tree-shaped messages. The SOTER [21] analysis is closest to the Siaam analysis and has been discussed in the previous section.

With its annotation-free programming model, which places no restriction on object references and message shape, we believe Siaam to be really unique compared to other approaches in the literature. In addition, we have not found an equivalent of the formal proof of isolation we have conducted for Siaam. Our evaluations demonstrate that the Siaam approach to isolation is perfectly viable: it suffers only from a limited overhead in performance and memory consumption, and our static analysis can significantly improve the situation. The one drawback

of our programming model, raising possibly hard to understand runtime exceptions, is greatly alleviated by the use of the Siaam analysis in a programming assistant.

References

1. G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.
2. J. Armstrong. Erlang. *Commun. ACM*, 53(9), 2010.
3. S.M. Blackburn, R. Garner, and C. Hoffman et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*. ACM, 2006.
4. R. Carlsson, K. F. Sagonas, and J. Wilhelmsson. Message analysis for concurrent programs using message passing. *ACM Trans. Program. Lang. Syst.*, 28(4), 2006.
5. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP '03*, volume 2743 of *LNCS*. Springer, 2003.
6. B. Claudel. *Mécanismes logiciels de protection mémoire*. PhD thesis, Université de Grenoble, 2009.
7. G. Czajkowski and L. Daynès. Multitasking without compromise: A virtual machine evolution. In *OOPSLA 2001*. ACM, 2001.
8. Coq development team. `http://coq.inria.fr`.
9. M. Fahndrich and M. Aiken et al. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *1st EuroSys Conference*. ACM, 2006.
10. R.J. Garner, S.M. Blackburn, and D. Frampton. A comprehensive evaluation of object scanning techniques. In *10th ISMM*. ACM, 2011.
11. N. Geoffray, G. Thomas, and G. Muller et al. I-JVM: a java virtual machine for component isolation in osgi. In *DSN 2009*. IEEE, 2009.
12. O. Gruber and F. Boyer. Ownership-based isolation for concurrent actors on multi-core machines. In *ECOOP 2013*, volume 7920 of *LNCS*. Springer, 2013.
13. P. Haller and M. Odersky. Actors that unify threads and events. In *9th Int. Conf. COORDINATION*, volume 4467 of *LNCS*. Springer, 2007.
14. P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *24th ECOOP*, volume 6183 of *LNCS*. Springer, 2010.
15. J. Hogg. Islands: aliasing protection in object-oriented languages. *SIGPLAN Not.*, 26(11), 1991.
16. `http://jikesrvm.org`.
17. R.K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: a comparative analysis. In *7th. PPPJ*. ACM, 2009.
18. G. Klein and T. Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4), 2006.
19. N. H. Minsky. Towards alias-free pointers. In *ECOOP'96*, volume 1098 of *LNCS*. Springer, 1996.
20. P. Müller and A. Rudich. Ownership transfer in universe types. *SIGPLAN Not.*, 42(10), 2007.
21. S. Negara, R. K. Karmani, and G. A. Agha. Inferring ownership transfer for efficient message passing. In *16th PPOPP*. ACM, 2011.
22. Q. Sabah. *SIAAM: Simple Isolation for an Abstract Actor Machine*. PhD thesis, Université de Grenoble, December 2013.
23. `http://sable.github.io/soot/`.
24. S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *22nd ECOOP*, volume 5142 of *LNCS*. Springer, 2008.
25. `https://team.inria.fr/spades/siaam/`.