Towards an Actor-based Concurrent Machine Model

Hans Schippers
Universiteit Antwerpen
Antwerpen, Belgium
hans.schippers@ua.ac.be

Tom Van Cutsem*
Vrije Universiteit Brussel
Brussels, Belgium
tvcutsem@vub.ac.be

Stefan Marr†
Vrije Universiteit Brussel
Brussels, Belgium
stefan.marr@vub.ac.be

Michael Haupt
Hasso Plattner Institute
Potsdam, Germany
michael.haupt@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

In this position paper we propose to extend an existing delegation-based machine model with concurrency primitives. The original machine model which is built on the concepts of objects, messages, and delegation, provides support for languages enabling multi-dimensional separation of concerns (MDSOC). We propose to extend this model with an actor-based concurrency model, allowing for both true parallelism as well as lightweight concurrency primitives such as coroutines. In order to demonstrate its expressiveness, we informally describe how three high-level languages supporting different concurrency models can be mapped onto our extended machine model. We also provide an outlook on the extended model's potential to support concurrency-related MDSOC features.

1. INTRODUCTION

The delegation-based *delMDSOC* machine model [7] has good support for modularizing crosscutting concerns and hence for generally supporting programming languages enabling multi-dimensional separation of concerns (MDSOC). Its implementation, the *delMDSOC kernel*[1] [15] achieves most of the features required to implement MDSOC programming languages. An important feature still missing is explicit concurrency support. This, however, is required to support thread-local behavior, which in turn is needed to implement several crosscutting constructs (e. g., `cflow`).

Adding concurrency support to delMDSOC involves three main tasks. First, the machine model has to be extended with concurrency primitives in a way compatible with the already present primitives for (mostly non-concurrent) MDSOC support. Second, it needs to be verified that the resulting machine model is sufficiently expressive in order to support different concurrency mechanisms as exhibited by a range of high-level languages. Finally, the model should be sufficiently powerful to allow for concurrency-related MDSOC features such as thread-local aspects and `cflow`.

This position paper presents a first attempt to add concurrency support to the delMDSOC machine model. As the latter is object-based and relies on late-bound message dispatch (cfr. Sec. 2), a concurrency model that is based on messages rather than shared state is preferable. Hence, the actors model [1] can serve as a source of inspiration. A more traditional model based on threads and locks does not seem to be a good choice: it is deceivingly simple—problems like starvation, deadlocks and race conditions frequently occur. Moreover, an actor-based model using message passing is conceptually closer to the delMDSOC model as it provides natural isolation of state in actors—a concept that is in line with the recent success of multi-core architectures with processor-local memory.

Compared to machine models proposed earlier, implementation in hardware is not an issue. A parallel machine interface consisting of primitives sufficient for dataflow, shared memory, and message passing models is discussed in [5]. It was designed keeping the tradeoffs for being realized in hardware in mind. Today, this is not an adequate restriction any more. Instead, an implementation as part of a high-level language virtual machine is desirable. Thus, we concentrate on a conceptually clean model which fits to the delMDSOC model and disregard implementation complexity as an important design consideration.

Throughout this paper, we first introduce the delMDSOC model in Sec. 2 before we present the concurrency model we propose for integration with delMDSOC in Sec. 3. Next, in Sec. 4, we discuss a possible translation process to the extended delMDSOC for several high-level languages with support for different kinds of concurrency models. Sec. 5 provides an outlook on the model's suitability concerning support for concurrency-related MDSOC features. Finally, the paper is concluded and future work discussed in Sec. 6.

*Postdoctoral Fellow of the Research Foundation, Flanders (FWO)

†Funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT)

[1] `http://www.hpi.uni-potsdam.de/swa/projects/delmdsoc/`

ICOOOLPS'09 July 6, 2009, Genova, Italy

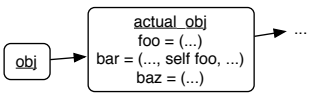

Figure 1: An object is represented as a combination of a proxy and the actual object.

2. A MACHINE MODEL FOR MDSOC

In this section we briefly describe the *delMDSOC* machine model originally introduced in [7]. Based on the notions of objects, messages and delegation, its original application domain is to serve as a compilation target for high-level languages dealing with the modularization of *crosscutting concerns*. Such concerns cannot be modularized in a program's dominant decomposition, and hence appear scattered throughout several unrelated modules. Examples of MDSOC paradigms trying to address this issue are *aspect-oriented programming* (AOP) [12] and *context-oriented programming* (COP) [9].

Core features of the model pertain to the representation of application entities and that of *join points*, which are points in the execution of a program where functionality dispatch occurs. The latter are consistently regarded as *loci of late binding* organized along multiple dimensions. Each dimension represents one possible way to choose a particular binding of a piece of functionality to a join point, e. g., the current object, the target of a method call, the invoked method, or the current thread.

The model assumes a prototype-based object-oriented environment in which objects are consistently represented as "seas of fragments" [14]: each object is visible to others via a *proxy* determining the object's identity. Messages sent to an object are received by its proxy and *delegated* to the actual object, as displayed in Fig. 1. The model allows dynamic modification of this initial configuration by inserting and removing additional proxy objects in between the proxy and its delegate object. This results in a chain of proxy objects (fragments) organized in a delegation chain, which collectively constitute the whole object. A crucial property of delegation is that *self* remains bound to the original receiver object, i. e., the proxy at the head of the delegation chain.

Classes are represented likewise: each class is a pair of a proxy and an object representing the actual class. Each object references its class by delegating to the class proxy.

The granularity of the supported join point model is that of message receptions. This granularity exists only at the level of the execution model, where member field access is also mapped to messages. Language implementations on top of the model map their join point model to the one defined by the machine model if appropriate.

A join point's nature as a locus of late binding is realized by means of inserting additional proxy objects in between the proxy and the actual object, or in-between the class object's proxy and the actual class-representing object. This way, a message passed along the delegation chain can be interpreted differently by several proxies, establishing late binding of messages to functionality.

3. AN ACTOR-BASED MACHINE MODEL

Our proposed machine model with explicit concurrency support is based on the well-known actor model of computation [1]. In contrast to multithreading, actors are a message-passing-based concurrency model that is more in line with the principles of object-oriented computing.

Actors were originally formulated as an extension of a functional programming language with three primitives to create new actors, send (asynchronous) messages to them, and to change the actor's behavior in how it will respond to future messages. Messages are buffered in an actor's mailbox. This model of "functional" actors is still used in well-known languages such as Erlang [3], but has also been reconciled in many ways with stateful object-oriented programming also known as "active objects" as used in ProActive [4].

The pure actor model makes no distinction between actors and regular objects. In a sense, all objects are actors. In practice, forcing all objects to be active is not always desirable, neither from a programming point-of-view nor from an implementation point-of-view. More concurrency leads to more non-determinism in a system. If it can be avoided, it should be for the sake of simplicity.

The E language (with its predecessors) [13] was the first to reconcile actors with objects. In E, actors are not "active objects" but rather *vats* (containers) of regular objects. An actor *contains* any number of regular objects, shielding them from unwanted concurrent modifications. Each actor still has its own message queue and its own thread of control. Messages that arrive in such message queue are not directed at the actor itself but rather to a particular object contained within the actor. All objects in an actor share the actor's message queue. Objects within the same actor can communicate using familiar synchronous method invocations. Objects within different actors can only communicate asynchronously, as in the actor model.

An important feature of this model is that actors *can* share mutable state (objects), but only the actor that *contains* the state (object) can access it synchronously. All other actors can only manipulate it asynchronously, by sending an asynchronous message to the object. This message is then executed by the actor containing the object. Shared state is thus supported without introducing locks avoiding many race condition and deadlock issues of traditional multithreading.

Our concurrent machine model is based on E's model of concurrency, extended with lightweight concurrency support in the form of coroutines. The abstract model is depicted in Fig. 2.

As in E, programs are composed of a number of actors. The threads of control of these actors can execute in parallel, so they can exploit multiprocessor concurrency. Similar to threads in a JVM, threads of control of actors are scheduled preemptively and are a relatively heavyweight unit of concurrency. Actors process messages sequentially from their message queue.

Different from E, however, every message is processed in its own coroutine. Each actor runs a "main" coroutine that encodes an infinite message processing loop. When a message arrives in the actor's queue, the main coroutine dequeues it, spawns a new coroutine to execute the message and then yields to this coroutine. Coroutines are scheduled cooperatively and may yield explicitly to hand over control to another coroutine within the same actor. If a coroutine does not yield, it runs to completion without interruption.

Messages are never sent to an actor but rather to an individual object contained by an actor. Each individual object

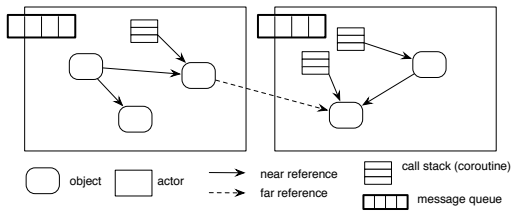

Figure 2: Actor-based concurrent machine model

is still represented as a sea of fragments, that is, it is represented by a single proxy object that may delegate to other objects as explained in Sec. 2. This way, our extended model retains its support for MDSOC features not related to concurrency.

Objects within the same actor refer to each other via so-called *near* references. Objects contained by other actors can be referred to via *far* references instead.

Messages sent over far references are asynchronous by default. On the other hand, messages sent over near references are synchronous by default, although they may be asynchronous upon explicit specification. In the latter case, they end up in the message queue of the containing actor. If a synchronous message is sent across a far reference, the coroutine processing that message is *paused* until the return value is available. A paused coroutine is not scheduled for execution until it is explicitly resumed. This mechanism can be used to implement more high-level synchronization mechanisms such as futures.

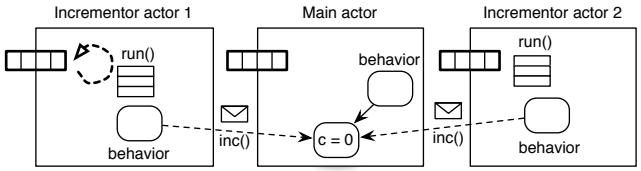

Figure 3: Running example: incrementing a shared counter object

We illustrate our model by means of a running example. Consider a counter object that can be incremented concurrently. While this is a simple example, it is an instance of a more general class of problems where two concurrent activities must synchronize their access to some mutable shared state. Fig. 3 illustrates how this problem can be expressed in our concurrency machine model. It shows three actors: the actor executing the main program (which created the counter object) and two incrementor actors whose task it is to increment the shared counter. Both incrementor actors have a far reference to the counter, so their `inc()` messages will be sent asynchronously and then processed sequentially by the main actor.

4. HIGH-LEVEL LANGUAGE MAPPINGS

In this section, we informally describe mappings from three high-level languages featuring different concurrency mechanisms onto our machine model to show that the model is sufficiently expressive and hence a suitable compilation target for a range of languages. The three languages under study are Java, Salsa [16], and Io [6], because they are representative of a multi-threading scheme, an actor-based concurrency model, and coroutines respectively.

For each language we show by means of our running example that it is possible to map its concurrency mechanisms onto our machine model in such a way that the ordering of instructions between concurrent activities is preserved. This mapping is not to be taken as a formal proof that the language's entire concurrency model can be straightforwardly mapped onto our concurrency model. Rather, it is a thought experiment to investigate the feasibility of our approach.

4.1 Java

Java features a prototypical concurrency model based on threads and monitors. Threads are preemptively scheduled and may share state without restrictions. Critical sections can be defined using `synchronized` statements. Below is a translation of our running example to Java:

```
class Counter {
  int c;
  Counter(int i) { c = i; }
  synchronized void inc() {
     this.c = this.c + 1;
  }
}
class Incrementor extends Thread {
  Counter cnt;
  Incrementor(Counter c) { cnt = c; }
  void run() { cnt.inc(); }
}
static void main(String[] args) {
  Counter cnt = new Counter(0);
  new Incrementor(cnt).start();
  new Incrementor(cnt).start();
}
```

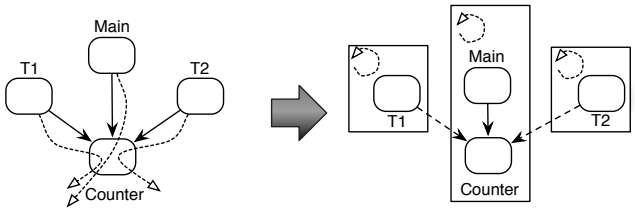

Figure 4: Language mapping for Java

A general mapping from Java's model to our actor-based model is possible by employing available techniques for distributed shared-memory systems. Here, we will only give a brief description of this idea. Implementation details and optimization techniques have been discussed in the general setting of distributed systems [17, 10].

We map each thread onto a separate actor. Actors, like threads, execute in parallel. Unlike threads, actors do not have synchronous access to shared state, so the objects from the original Java program must be partitioned across actors. A straightforward way to do this is to assign each object to the thread (actor) that created it.

Figure 4 depicts two threads `T1` and `T2` that share the counter object created by the `Main` thread. The mapping to the actor-based machine model is depicted on the right. Note that `T1` and `T2` now refer to the `Counter` instance via

a far reference. If these threads want to invoke a method on the counter, they can only do so by sending an asynchronous message to it. Such messages will be processed sequentially by the `Main` thread and cause the thread of the sender object to block until the message has been processed. The mapping thus implicitly makes methods on shared objects `synchronized`.

While the above mapping maintains the semantics of the `synchronized` method, programs usually exhibit more complex patterns of data sharing. Advanced techniques for synchronizing access to objects created by different threads and certain optimizations (e.g. for objects only used in a thread in which it was not created) are not discussed here, but are covered in the literature about distributed shared memory.

It should be noted that techniques like object migrations between actors, for instance using the *Arrow* [8] protocol, integrate very naturally with the underlying delegation mechanism of our machine model. The benefits of this correlation are to be investigated in future work.

4.2 Salsa

Salsa [16] is an actor extension to Java. It features a prototypical "active object" model: active objects have their own thread of control and a message queue from which they process messages sequentially. Active objects may send each other asynchronous messages. Active objects cannot share state. They each have their own set of "passive" Java objects, but may not directly refer to each other's objects. Below is a translation of our running example to Salsa:

```
behavior Counter {
  int c;
  Counter(int i) { c = i; }
  void inc() { this.c = this.c + 1; }
}
behavior Incrementor {
  Counter cnt;
  Incrementor(Counter c) { cnt = c; }
  void run() { cnt<-inc(); }
}
static void act(String[] args) {
  Counter cnt = new Counter(0);
  new Incrementor(cnt)<-run();
  new Incrementor(cnt)<-run();
}
```

The `behavior` keyword is similar to Java's `class` keyword, except that instances of this class are active objects. Within methods declared on active objects (such as the `inc` method in the above example) there is no need for `synchronized` statements because active objects process their incoming messages one at a time. The operator `<-` denotes an asynchronous send. Because active objects cannot share passive objects, the counter object must be modelled as a separate active object such that it can be safely shared by the two incrementors.

A Salsa active object is mapped onto an actor in our concurrency machine model. Fig. 5 depicts the mapping of four Salsa active objects to actors. Note that, because actors are not objects in our model, the actors contain a "behavior" object that corresponds to the Salsa active objects. The reference from `AO1` to `CounterAO` is now represented as a far reference from the behavior object of `AO1` to the behavior object of the counter.

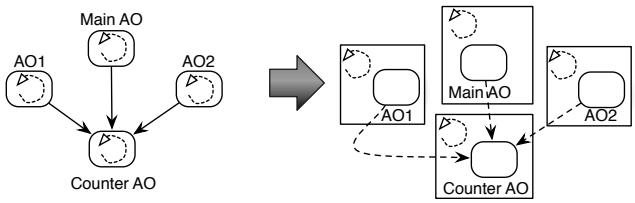

Figure 5: Language mapping for Salsa

In Salsa, each active object has a single thread of control that processes messages from a message queue. In our mapping, the actor activates only a single coroutine, which never yields. Instead of yielding, the coroutine is a loop that processes messages sequentially from the message queue.

4.3 Io

The Io language [6] features no real parallelism, but rather offers user level cooperative threads in the form of coroutines. All coroutines are scheduled sequentially in one OS-level thread. Control is transferred from one coroutine to another by means of the *yield* instruction. When a coroutine yields, it reclaims control after all other active coroutines have had a chance to process some instructions. In between processing different messages, coroutines yield implicitly.

In Io, by default, messages are processed synchronously and within one coroutine. Prefixing a message with `@` turns it into an asynchronous send, and results in that message being stored in the receiver object's message queue, waiting to be processed by a coroutine dedicated for that object. The coroutine from which the message was sent, does not block, however. Instead, it immediately receives a future. It is only upon accessing this future that the coroutine will block until the result becomes available.

Our running example, in Io syntax, looks like the following:

```
cnt := Object clone
cnt N := 0
cnt inc := method(N = N + 1)
cnt read := method(N)

incrementor := Object clone
incrementor run := method(cnt @inc)

f1 := incrementor @run
f2 := incrementor clone @run
```

Since Io is object-based, new objects are created by *cloning* existing ones. As it is moreover a dynamic language, fields and methods can be added to objects on the fly. `f1` and `f2` have futures assigned to them until they are accessed later on. At that point, the future transparently turns into the actual result when the latter becomes available.

Fig. 6 depicts the mapping of the Io example to our concurrency machine model. In Io, each object to which an asynchronous message has been sent has its own message queue. Each of the depicted objects has its own coroutine, but there is only a single thread of control that executes these coroutines in turn.

Mapping the Io concurrency model to our machine model is rather straightforward. We use a single actor in which all application objects reside. The message queues of objects

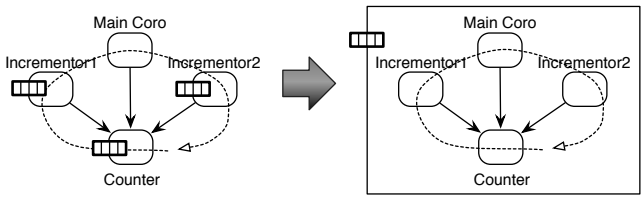

Figure 6: Language mapping for Io

are all aggregated into a single message queue at the actor-level. The main coroutine is a loop that takes a message from the actor queue and starts a new machine-level coroutine to process the message. Hence our mapping maps a single Io coroutine per object to a single machine-level coroutine per incoming message. Blocking on futures can be simulated by pausing the coroutine which accesses a future, and resuming it when the corresponding result becomes available.

5. CONCURRENT MDSOC FEATURES

In the previous section, we discussed a number of languages that provide concurrency mechanisms, yet do not offer modularization mechanisms exceeding those available in traditional object-orientation. As a result, the MDSOC capabilities offered by the delMDSOC model are never called upon. As described in Sec. 2, the model's MDSOC support is a result of the representation of application-level entities as seas of fragments, combined with late-bound message dispatch. As these properties have been retained in the extended machine model (cfr. Sec. 3), it should be clear that support for high-level MDSOC languages without concurrency mechanisms is still intact.

However, some high-level MDSOC languages, such as AspectJ [11] and CaesarJ [2], do exhibit concurrency-related MDSOC features. On the one hand, AspectJ introduced the `cflow` construct, which restricts the application of advice to the dynamic control flow of certain join points. Implementing `cflow` implies taking into account in which concrete thread a join point constituting such a control flow occurred. On the other hand, CaesarJ aspects can be dynamically deployed thread-locally, meaning that any advice belonging to that aspect is only ever executed within a certain thread.

In its original introduction, the delMDSOC model included an outline on how to deal with this. More specifically, if an object's delegate is not determined by a simple pointer, but is instead calculated by an actual function, the delegate could differ depending on the current context, including the current thread. This is illustrated in Fig. 7, where the `asp_c_proxy` proxy only intercepts `foo` messages provided the current thread of execution is `T1`.

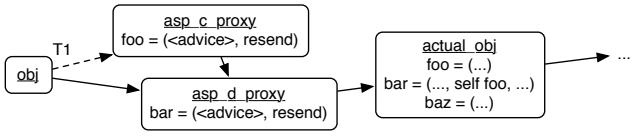

Figure 7: Thread-local proxy deployment

At the machine level, however, threads are not first class entities, as explained in Sec. 3. Rather, true concurrency is achieved by using multiple actors. The object from Fig. 7 (which consists of the complete delegation chain) would belong to one specific actor. Thread-local access at the high level would then correspond to access from within the same actor, i. e., over a near reference. Messages from other threads (actors) would be sent asynchronously over a far reference. As messages over near references can be sent both synchronously and asynchronously, the only suitable criterion in order to determine thread-local access hence seems to be the sender object. This suggests the delegation function should now take into account the identity of the sender (and its owning actor) instead of the current thread as in Fig. 7.

To the authors' best knowledge, no high-level actor-based MDSOC languages or actor-based MDSOC patterns exist. Hence, no validation can be done in this area, and reflection on such languages and patterns is considered future work.

6. CONCLUSION

In this paper, we propose the addition of actor-based concurrency support to a delegation-based machine model [7]. Our new model is based on the actor model of E [13] where actors are containers of objects. Our actors directly support true concurrency. Objects within actors can also act concurrently by means of user-level cooperative scheduling via coroutines.

For validation purposes, we informally describe how the concurrency mechanisms of three high-level languages can be mapped onto our machine model. This serves to motivate that our proposed extensions are sufficiently expressive in order to support a whole range of high-level concurrency models.

Although our machine model is inherently object-based and hence fits high-level object-oriented languages more naturally, future work includes considering other interesting concurrency models such as that of Erlang [3]. Moreover, the proposed extensions should be supported by formal semantics and implemented in a research prototype. This will allow for a deeper investigation of potential performance implications of the presented mappings.

Acknowledgements

This research is funded by the Belgian Science Policy (Belspo) under the project on 'Modeling, Verification and Evolution of Software' (MoVES) as part of the IAP-Phase VI Interuniversity Attraction Poles Programme.

7. REFERENCES

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

[2] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of caesarj. *Transactions on AOSD*, LNCS 3880, 2006.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[4] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[5] W. J. Dally and D. S. Wills. Universal mechanisms for concurrency. In *PARLE '89 Parallel Architectures and*

Languages Europe, volume 365 of *LNCS*, pages 19–33. Springer, 1989.

[6] S. Dekorte. Io: a small programming language. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 166–167, New York, NY, USA, 2005. ACM Press.

[7] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 501–524. Springer, 2007.

[8] M. Herlihy and M. P. Warres. A tale of two directories: Implementing distributed shared objects in java. *Concurrency: Practice and Experience*, 12(7):555–572, 2000.

[9] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology (JOT)*, 7(3):125–151, March-April 2008.

[10] R. Kaiabachev and B. Richards. Java-based dsm with object-level coherence protocol selection. In *Proc. of Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 648–653, Marina del Ray, CA, USA, November 2003.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *ECOOP '97: Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[13] M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In R. D. Nicola and D. Sangiorgi, editors, *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, April 2005.

[14] H. Ossher. A direction for research on virtual machine support for concern composition. In *Proc. Workshop VMIL '07*. ACM Press, 2007.

[15] H. Schippers, M. Haupt, R. Hirschfeld, and D. Janssens. An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns. In *Proceedings of ACM Symposium on Applied Computing (SAC), PSC Track, Honolulu, HI (USA), March 09 - March 12, 2009*. ACM, 2009.

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

[17] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, C. J. H. Jacobs, and H. E. Bal. Source-level global optimizations for fine-grain distributed shared memory systems. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 83–92, New York, NY, USA, 2001. ACM.