Scoped Synchronization Constraints for Large Scale Actor Systems

Peter Dinges and Gul Agha

Department of Computer Science
University of Illinois at Urbana–Champaign, USA
pdinges@acm.org, agha@illinois.edu

Abstract. Very large scale systems of autonomous concurrent objects (Actors) require coordination models to meet two competing goals. On the one hand, the coordination models must allow Actors to dynamically modify protocols in order to adapt to requirement changes over the, likely extensive, lifetime of the system. On the other hand, the coordination models must enforce protocols on potentially uncooperative Actors, while preventing deadlocks caused by malicious or faulty Actors. To meet these competing requirements, we introduce a novel, scoped semantics for *Synchronizers* [7,6]—a coordination model based on declarative synchronization constraints. The mechanism used to limit the scope of the synchronization constraints is based on capabilities and works without central authority. We show that the mechanism closes an attack vector in the original Synchronizer approach which allowed malicious Actors to intentionally deadlock other Actors.

1 Introduction

A well-understood lesson from the design of the Internet helps to build scalable software systems: having autonomous, loosely coupled components avoids central bottlenecks that limit system growth. However, another lesson taught by the Internet is often neglected: *every* component in a large system cannot be trusted. The principle that components cannot be trusted not only holds in systems that execute other users' code. Even if all components are under a central trusted regimen, the probability of having a faulty or compromised component increases with the system size.

Coordination models for large systems must therefore take into account that components may be uncooperative or even malicious [14,12]. Consequently, coordination protocols must be enforced to fulfill their guarantees. For example, ignoring a replication protocol can result in inconsistent state of the participating databases. Furthermore, coordination models for large systems must support dynamic adaptation. Restarting is rarely an option for large systems and the specification of a system is likely to change over its lifetime. Naively addressing these two requirements severely hampers the system's stability: allowing faulty components to impose protocols on all other components can easily result in a deadlock.

The contribution of this article is a novel, scoped semantics for *Synchronizers* [7,6] that is better adapted to the requirements of large scale systems. Synchronizers are declarative synchronization constraints that model coordination by enforcing restrictions on the interaction patterns between components. Following the capability approach to security—but with a twist— we propose to limit the scope of synchronization constraints. The central idea behind our approach is that synchronization constraints restrict not the *targets*, but the *sources* of interactions. Thus, every component may install constraints on other components, but the constraints will affect only interactions originating from components for which the installing component holds the required capabilities.

Without these capabilities, malicious components cannot intentionally deadlock their acquaintances by imposing impossible constraints. Consequently, our scoped semantics close this attack vector. Scoping also mitigates accidental deadlocks of a component from interfering constraints because the scoping requires the interfering constraints to hold overlapping capabilities. However, as we explain later, scoping cannot completely prevent accidental deadlocks.

The questions of constraint inheritance and implementation performance are not addressed in this article. In both cases, however, we believe the new semantics to maintain the characteristics of the conventional Synchronizer semantics, meaning that it does not impose an extra burden.

In the remainder of this article, we briefly introduce Synchronizers (section 2), discuss the challenges of coordination in large systems (section 3), and—from this motivation—develop a scoping mechanism for Synchronizers to adapt them to the requirements of large systems (section 4). Next, we provide the exact semantics of our solution (section 5). The conclusion (section 7) follows after a discussion of related work (section 6).

2 Synchronization Constraints

This section gives a brief overview of *Synchronizers* and their conventional semantics [7,6]. The examples are taken from Frølund and Agha's ECOOP'93 article [7]. The term *Actor* takes the place of the generic *system component* because Actors are precisely defined in their properties [2]: they are concurrently executing mobile objects with perfectly encapsulated state that communicate via asynchronous messages. Actors are autonomous by design; the scalability of the Actor model to large systems is well established [10,15].

Synchronizers are declarative synchronization constraints that can be imposed on groups of Actors. The constraints express under which conditions an Actor is able to handle a message. Until the conditions are met, the message stays in the Actor's message queue. The constraints have a global effect and affect all messages an Actor receives. Conceptually, a Synchronizer can be seen as a special kind of Meta-Actor [11,21] that observes and limits the message dispatch of other Actors. The conventional form of Synchronizers supports disabling and atomicity constraints:

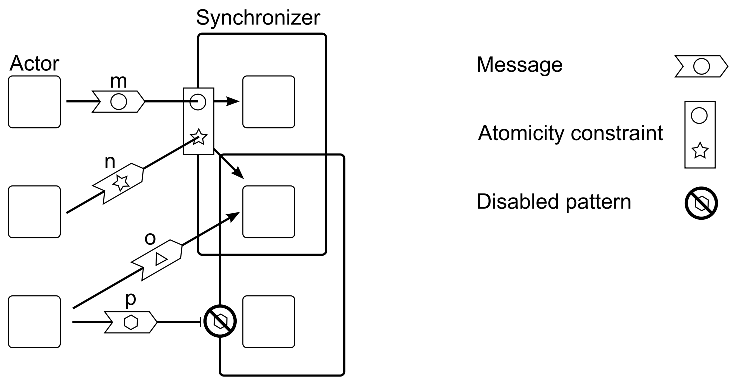

Fig. 1. *Constraints Enforced by Conventional Synchronizers.* Synchronizers support (combinations of) atomicity and disabling constraints. Atomicity constraints ensure that a set of messages is dispatched as a whole and without temporal (happened before [9]) ordering. Messages m and n satisfy the atomicity constraint together and are therefore dispatched at their target Actors. Message p matches a disabling pattern in the lower Synchronizer and therefore cannot be dispatched. Synchronizers can overlap. If message n matched the same disabling pattern as message p, then the atomicity constraint would have to remain unsatisfied, preventing the delivery of message m. Message o matches no pattern and thus is unconstrained.
Under the conventional semantics, Sychronizers observe and control all messages an Actor receives. The synchronization constraints therefore form a conceptual *membrane* around every Actor in a constrained group. Despite Synchronizers being drawn as a single membrane around the whole group, messages sent between two Actors coordinated by the same Synchronizer still have to satisfy the synchronization constraints.

Disabling constraints prevent the constrained Actor from handling messages that match a given pattern. For example, by disabling the handlers for all but the initialization message, a disabling constraint ensures that an Actor dispatches (starts to process) the initialization message before it dispatches any other message.

Atomicity constraints coordinate groups of Actors by bundling messages into indivisible sets. A constraint enforces that either all the messages in a set are dispatched, or none of them are (there is no partial delivery). The constraint provides *spatial* atomicity. An atomicity constraint can, for example, implement a simple online music payment scheme by fusing the *deduct money from credit card* message with the *enable download* message.

Programmers declare Synchronizers as templates. Similar to classes or Actor behaviors, these templates are dynamically instantiated at run-time with concrete values filled in for the parameters. Thus, Synchronizers can adapt the system to meet new specifications during execution. Actors may install Synchronizers at any of their acquaintances. Synchronizers can have local state that changes with the observed messages. They may also overlap, that is, multiple Synchronizers can constrain the same Actor. Figure 1 shows example effects of

$$
\begin{array}{rcl}
\langle\text{Synchronizer}\rangle & ::= & \langle\text{Id}\rangle\ (\ \langle\text{List}\{\text{Id}\}\rangle\ )\ \{\ [\textsf{\textbf{init}}\ \langle\text{Binding}\rangle]\ \langle\text{Relation}\rangle\ \} \\
\langle\text{Relation}\rangle & ::= & \langle\text{Pattern}\rangle\ \textsf{\textbf{updates}}\ \langle\text{Binding}\rangle \\
 & | & \langle\text{BExp}\rangle\ \textsf{\textbf{disables}}\ \langle\text{Pattern}\rangle \\
 & | & \textsf{\textbf{atomic}}\ (\ \langle\text{List}\{\text{Pattern}\}\rangle\ ) \\
 & | & \langle\text{Pattern}\rangle\ \textsf{\textbf{stops}} \\
 & | & \langle\text{Relation}\rangle\ ,\ \langle\text{Relation}\rangle \\
\langle\text{Pattern}\rangle & ::= & \langle\text{Id}\rangle\ .\ \langle\text{Id}\rangle \\
 & | & \langle\text{Id}\rangle\ .\ \langle\text{Id}\rangle\ (\ \langle\text{List}\{\text{Id}\}\rangle\ ) \\
 & | & \langle\text{Pattern}\rangle\ \textsf{\textbf{or}}\ \langle\text{Pattern}\rangle \\
 & | & \langle\text{Pattern}\rangle\ \textsf{\textbf{where}}\ \langle\text{BExp}\rangle \\
\langle\text{Binding}\rangle & ::= & \langle\text{Id}\rangle\ :=\ \langle\text{Exp}\rangle \\
 & | & \langle\text{Binding}\rangle\ ;\ \langle\text{Binding}\rangle
\end{array}
$$

Fig. 2. *Abstract Syntax for Synchronizer Declarations.* Names in angle brackets denote syntactic categories; $\langle\text{List}\{\cdot\}\rangle$ stands for a comma-separated list of elements in the given category. We assume the category of identifiers, $\langle\text{Id}\rangle$, to range over alpha-numeric strings. The categories $\langle\text{Exp}\rangle$ and $\langle\text{BExp}\rangle$ denote expressions and Boolean expressions respectively.
Relations define the constraints that a Synchronizer enforces. The patterns are matched against observed messages: using the customary dot-syntax, the identifier before a dot specifies the name of the target Actor (a variable holding an Actor address), and the identifier after the dot specifies the message type. The list of identifiers in parentheses is a list of variable names that get bound to the message arguments.
Variables have a unique binding for every observed message. Thus, using the same variable in two different places means that the same value must appear in these places for the pattern to match. All expressions are free of side-effects.

Synchronizers. In this article, we employ the abstract syntax of Frølund and Agha [7] for Synchronizer declarations given in Figure 2.

2.1 Example: Cooperating Resource Administrators

Consider a system that provides two kinds of resources for its users, for example disk drives and optical drives. There are multiple instances of both drive types and each of these resource kinds is governed by an administrating Actor that limits the number of instances that can be used at the same time. Suppose that the disks and optical drives are accessed over the same network connection. To ensure that drive accesses stay within the bandwidth limit, the administrating Actors have to restrict the *total* allocations made of both drive types.

The Synchronizer below implements the necessary coordination pattern using disabling constraints. It stores the total number of allocated drives in the system in an internal counter alloc. Observing requests and releases at the resource administrators updates the counter (lines 5 and 6). When the maximum number of drives has been requested, the Synchronizer disables the request handlers of both administrators (line 4). Thus, neither administrator can process further allocation requests. These pending requests can be processed only after one of them releases a drive.

```
1  AllocationPolicy(adm1, adm2, max) {
2    init alloc := 0
3
4    alloc >= max disables (adm1.request or adm2.request)
5    (adm1.request or adm2.request) updates alloc := alloc + 1,
6    (adm1.release or adm2.release) updates alloc := alloc − 1
7  }
```

2.2 Example: Dining Philosophers

In the classic problem of the dining philosophers, a group of philosophers (processes) must coordinate their behavior to access a number of chopsticks (resources). Typically, five philosophers sit at a round table and a chopstick is placed between each of them. Thus, there as many chopsticks as there are philosophers. To eat (make progress), every philosopher must pick up both, the left and right neighboring chopstick. Without coordination, for instance if every philosopher starts picking up the left chopstick, the system can deadlock and philosophers can starve.

Suppose philosophers and chopsticks are modeled as Actors, and chopsticks implement an allocation policy such that pick messages can be dispatched only if the chopstick is currently lying on the table (free). Philosophers can then use atomicity constraints to prevent deadlocks. By ensuring that every philosopher's two pick requests are either dispatched together, or not at all, every philosopher is guaranteed to always pick up both neighboring chopsticks—given that the constraints are installed following the neighborhood relation. Under certain fairness assumptions about the implementation, this prevents the system from deadlocking. A Synchronizer implementing this approach could look as follows:

```
PickUpConstraint(c1, c2, phil) {
  atomic( (c1.pick(sender) where sender = phil),
          (c2.pick(sender) where sender = phil) )
}
```

3 Coordination in Large Scale Systems

This section discusses the challenges of coordination in large scale Actor systems and demonstrates the semantic problems of Synchronizers in this context.

3.1 Properties of Large Systems

Scalable coordination models must not only use additional resources efficiently, but also address the inherent requirements of large systems:

Support of dynamic reconfiguration and adaptation. Large systems, for instance a cloud computing service, are expensive to reboot. Nevertheless, the environment and specifications of the system are likely to change over the system lifetime, for example when new services are introduced. A scalable coordination model must therefore support dynamic adaptation.

Robustness against misbehaving Actors. The chance of having a faulty, compromised, or malicious Actor in a system increases with the system size. A scalable coordination model must therefore be able to cope with uncooperative Actors and gracefully degrade in the presence of failures. It must also guard its reconfiguration mechanisms against abuse.

The second requirement implies that, in general, Actors in large systems cannot rely on the good intentions of other Actors. We therefore think of Actors as being mutually suspicious, that is, they do not trust each other. Consequently, Actors try to give others as little control over themselves as possible and follow the principle of least authority [12]. In particular, Actors try to avoid making their—eventual—progress in computation dependent on others.

3.2 Problems of Globally Scoped Constraints

Mutual suspicion conflicts with the global scope of synchronization constraints defined in the conventional Synchronizer semantics [6]. Under these semantics, Synchronizers observe and affect all messages a constrained Actor receives. Any Actor may install Synchronizers on acquaintances, which opens the door to malicious Actors causing intentional deadlocks on other Actors, effectively resulting in a denial of service at the target.

For example, suppose that an Actor A can handle messages of type message1, message2, and so on, up to messageN. A malicious Actor M can prevent A from receiving any further messages by installing a Synchronizer that disables all message handlers in A:

```
DisablingAttack(a) {
  true disables (a.message1 or a.message2 or ... or a.messageN)
}
```

Similar problems arise from atomicity constraints. If M forces A to only dispatch messages in unison with an anonymous Actor that never receives any messages, then A will deny all service. A Synchronizer achieving this effect could look as follows:

```
AtomicityAttack(a, anonymous) {
  atomic( (a.message1 or a.message2 or ... or a.messageN),
           anonymous.message )
}
```

Malice is not the only source of problems. Even if the access to Synchronizer installation is limited and only legitimate Actors may install Synchronizers, incompatible constraints may cause deadlocks. Consider the case where two independent Actors, originating in different libraries and unaware of each other, impose the BigEndianConstraint and LittleEndianConstraint on a common acquaintance G. The argument of any enjoyEgg message sent to G is either *big* or *little*, which prevents G from enjoying any of them.

```
BigEndianConstraint(a) {
  endianness(e) != "big" disables a.enjoyEgg(e)
}

LittleEndianConstraint(a) {
  endianness(e) != "little" disables a.enjoyEgg(e)
}
```

4 Scoped Constraints

The previous section demonstrated that allowing Synchronizers to constrain *all* messages an Actor receives is problematic in large systems. In this section, we introduce a scoping mechanism for synchronization constraints that restricts their effects to a subset of messages. The exact semantics of this approach are the topic of the next section.

The central idea behind our approach is that synchronization constraints restrict not the receivers, but the sources of messages. Consequently, a constraint installed on Actor A by Actor I should not apply to all messages that A receives. Instead, the constraints should only apply to messages received by A if they were sent by Actors that are under control of I. Thus, the constraints should only apply if the installing Actor I has the capability to impose constraints on the sending Actors.

4.1 Synchronization-Capabilities

Synchronization constraints, and thus Synchronizers, therefore work in the opposite direction of object-capabilities [12]. Object-capability security is the natural security model of Actor systems. Its defining notion is that once an Actor address—the capability for this Actor—is known, any message may be sent to it. Access to services hence depends on the knowledge of Actor addresses; security can be implemented through their careful distribution. The underlying assumptions are that addresses are unique across the system and cannot be guessed. For Actors, the only ways of obtaining knowledge of other Actors' addresses are (1) *initialization*: the system starts with this knowledge distribution; (2) *parenthood*: creating a new Actor yields an address; and (3) *introduction*: addresses are values and can be propagated inside messages.

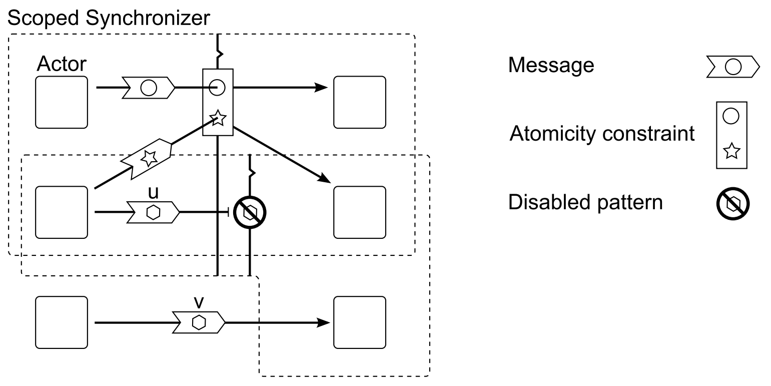

Fig. 3. *Constraints Enforced by Scoped Synchronizers.* Scoped Sychronizers (dashed frames) constrain only messages sent by Actors for which they hold the synchronization-capability. These Actors are placed in the left part of the Synchronizer. Their sent messages must satisfy the constraints before they can be dispatched at the recipients (placed right). Since message u matches a disabling pattern of the lower Synchronizer, it cannot be dispatched. However, the respective Synchronizer lacks control over the sender of message v, so v can be dispatched despite having the same shape as u.

In addition to object-capabilities, we introduce *synchronization-capabilities* that determine the scope of synchronization constraints. Synchronizers can constrain messages only if they hold the synchronization-capability to the message source. They receive their synchronization-capabilities from the installing Actor. Figure 3 shows the scoping effects of synchronization-capabilities.

As with object-capabilities, we assume that synchronization-capabilities are unique across the system and cannot be guessed. Their distribution follows similar rules. Actors can obtain synchronization-capabilities through *initialization* and *introduction*. However, the *parenthood* rule is transitive: creating a new Actor yields a synchronization-capability *for this Actor and all its children*. The transitivity of synchronization-capabilities prevents Actors from escaping synchronization constraints by transferring their behavior to a new Actor, thereby changing their identity. Synchronization-constraints hence grant control over families of Actors, including future members whose identities are yet unknown.

The two types of capabilities are separate; a capability of one type cannot be used in places that require the other. This separation allows Actors to send messages to other, potentially untrusted Actors, without submitting to the synchronization constraints of the recipient Actors. In contrast to the conventional Synchronizer semantics, the semantics of scoped Synchronizers ensures that the reply address contained inside a message can be used solely for communication.

4.2 Scoped Synchronization Constraints

With Synchronizers only constraining messages for which they hold the synchronization-capabilities, it becomes unnecessary to restrict access to the Synchronizer installation primitive. Any Actor may therefore install Synchronizers on all its acquaintances. The imposed constraints will simply stay without effect for most messages.

Synchronization-capabilities thus prevent the intentional deadlock scenarios discussed in section 3. Revisiting the DisablingAttack and AtomicityAttack Synchronizer examples, we see that with scoping the situation is similar to that of the lower right Actor in Figure 3: unless the Synchronizers hold some relevant synchronization-capability, all messages will remain unaffected—as is the case for message v in the figure. Hence, the malicious installing Actor poses no threat if none of the other Actor in the system supplies it with a synchronization-capability. However, even in this case, the deadlock concerns only parts of the system.

Synchronization-capabilities cannot completely prevent deadlocks that arise from incompatible constraints as in the endian example. However, the scoping of constraints mitigates the problem. If the Actors imposing the BigEndianConstraint and LittleEndianConstraint on G possess disjoint synchronization-capabilities, then each Actor's constraints have no effects on messages from the *other* parts of the system. Thus, accidental interference of constraints becomes less likely.

5 Semantics

This section describes in detail the semantics of the synchronization-capabilities introduced in section 4. The semantics are defined in the context of a toy programming language called IMPACT-S. While IMPACT-S embodies some design choices, the general principles behind the design of scoped Synchronizers can be easily extracted from its description.

IMPACT-S adds Actor primitives—message sending and Actor creation—to IMP, a pedagogical example of an imperative language [22]. Furthermore, it adds Synchronizers and synchronization-capabilities. We limit the discussion of IMPACT-S's semantics to the parts relevant to synchronization-capabilities without the distracting bookkeeping and infrastructure necessary for complete semantics. A technical report that is currently being prepared defines the formal semantics of IMPACT-S in the $\mathbb{K}$ rewriting logic framework [18].

5.1 Synchronization-Capabilities

In section 4, we introduced synchronization-capabilities as scoping mechanism for synchronization constraints: the idea is to let Synchronizers control only those messages for whose sender they hold the synchronization-capability. Unlike object-capabilities, synchronization-capabilities are transitive; granting control over the messages sent by an Actor and all its children prevents Actors from escaping their constraints by transferring their behavior and state to a new Actor. Thus, the set of synchronization-constraints $\mathcal{S}$ is partially ordered by this hierarchy of control. For $S_1, S_2 \in \mathcal{S}$, write

$$\text{controls}(S_1, S_2) \quad \text{iff} \quad S_1 = S_2 \text{ or actor}(S_1) \text{ is an ancestor of actor}(S_2),$$

where $\mathrm{actor}(S_i)$ denotes the Actor to which the capability S_i belongs. An Actor A is an ancestor of another Actor B if either A created B, or A created an ancestor of B.

5.2 Actor Creation

IMPACT-S implements the $\mathrm{controls}(\cdot,\cdot)$ relation through prefix comparison. Internally, synchronization-capabilities are lists of integers. The list for a new Actor is derived by extending the creating Actor's list with the count of child Actors created thus far. Assuming that all lists are distinct when the system starts, this method yields a unique list for every new Actor. Furthermore, the derivation of new lists is distributed and works without communication.

To avoid the redundancy of having every Actor store its own synchronization-capability, IMPACT-S gives Actor addresses—that is, object-capabilities—the same integer-list representation. This way, every Actor has to store only one list of integers that doubles as its address and synchronization-capability. When used as values, the system keeps the two kinds of capabilities separate by tagging the lists with `addr` and `syncap` labels.

Suppose an Actor with address $\mathtt{addr}(i_1;\ldots;i_n)$ creates an Actor with behavior B by executing

$$\mathtt{new}\ B(a_1,\ldots,a_l),$$

$a_1,\ldots a_l$ being the arguments to the behavior's constructor. Let the new Actor be the k-th child. Then the new Actor's address is $\mathtt{addr}(i_1;\ldots;i_n;k)$, its synchronization-capability is $\mathtt{syncap}(i_1;\ldots;i_n;k)$. Using prefix comparison, we clearly have

$$\mathrm{controls}\big(\mathtt{syncap}(i_1;\ldots;i_n),\mathtt{syncap}(i_1;\ldots;i_n;k)\big).$$

Since the capabilities are separate, both are returned to the creating Actor. Thus, creating an Actor in IMPACT-S yields not only the address of the new Actor, but a pair of capabilities.

5.3 Message Sending and Dispatching

Synchronization constraints determine whether a message can be dispatched (processed) at the receiving Actor. Because communication is asynchronous, the sending Actor cannot answer this question as the state of the recipient Actor may change while the message is in transit. Synchronizers therefore reside at the receiving Actors; they can be regarded as *constraint servers* that are queried by the message dispatch mechanism. This remains true despite the scoping mechanism's focus on message senders. The only change is that Synchronizers now have to possess the right synchronization-capability to control a message.

An Actor's scheduler can dispatch a message only if the message is not disabled by a Synchronizer. The scheduler identifies applicable Synchronizers by matching the message against the patterns declared by installed Synchronizers.

The scoped semantics requires not only that the pattern matches (as in conventional Synchronizer semantics), but also that the Synchronizer's synchronization-capability $S_{\text{Sync}} \in \mathcal{S}$ gives it control over the message. Thus, for a message sent by an Actor with synchronization-capability $S_{\text{Act}} \in \mathcal{S}$, the scheduler checks whether

$$\text{controls}(S_{\text{Sync}}, S_{\text{Act}}).$$

To have all matching information available, messages in IMPACT-S are therefore stamped with the sender's synchronization-capability. Thus, the command

$$\texttt{send}\ m(a_1, \ldots, a_l)\ \texttt{to}\ r$$

executed at an Actor with address $\texttt{addr}(i_1; \ldots; i_n)$ creates a message

$$\texttt{msg}\big(r; \texttt{syncap}(i_1; \ldots; i_n); m; a_1; \ldots; a_n\big).$$

For applicable Synchronizers, the dispatcher then queries whether the constraint is active. This happens *synchronously*; if communication with other dispatchers is necessary, as is the case with atomicity constraints, the dispatcher employs a suitable protocol such as atomic two-phase commitment. The message can be dispatched if

1. *all* disabling patterns allow dispatching the message;
2. *any* of the matching atomic patterns allows dispatching.

For both, disabling and atomic patterns, no match means that the message is enabled.

5.4 Synchronizer State Updates

When a message is dispatched, all Synchronizers belonging to matching update patterns receive a notice. This includes Synchronizers that lack the required synchronization-capability. Making the dispatch of messages public guarantees a consistent view on the system; it allows Synchronizers to take into account the actions of the *uncontrolled* part of the environment.

For example, consider the cooperating resource administrators of subsection 2.1. If the AllocationPolicy Synchronizer was blind to the requests and release messages of some users, then it could not enforce the intended limit on the total number of drive allocations on the users it controls.

However, a globally visible message dispatch is a trade-off. While it allows a consistent view on the system, it enables malicious Actors to spy on other Actors; see Figure 4.

6 Related Work

Actor-Based Coordination. Much of the prior work on Actor-based coordination models ignores the question of trust between Actors. The models either

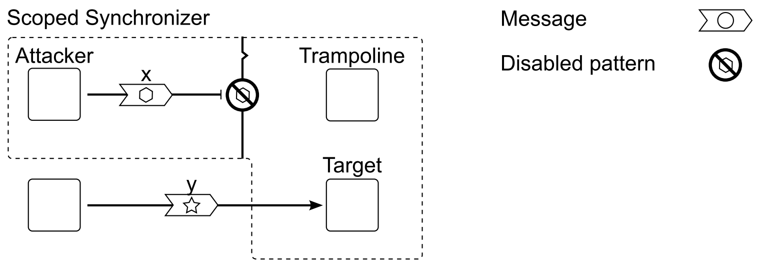

Fig. 4. *Information Leak through Updates.* Scoping only limits the constraining power of Synchronizers. To guarantee a consistent view on the system, Synchronizers can observe all messages that an Actor dispatches—regardless of the synchronization-capabilities the Synchronizer holds. The Attacker Actor exploits this fact to gather information about the Target Actor: First, the Attacker creates a Trampoline Actor and installs a Synchronizer on the Target and the Trampoline. The Synchronizer disables the dispatch of message x at the Trampoline until it observes message y at the Target. Then, the Attacker sends message x to the Trampoline. Once the Trampoline dispatches x, it bounces a message back to the Attacker, providing the Attacker with the knowledge that the Target dispatched message y.

assume cooperative behavior, trustworthy protocols, or—if dynamic reconfiguration is supported—do not clearly specify who has access to coordination primitives and how they are installed. These factors make them less suited for large scale systems.

Like Synchronizers, regulated coordination policies [14] are declarative coordination constraints for autonomous agents. However, the policies have a purely local effect so that agents can be subject to multiple policies without interference. Policies are enforced by trusted agents, which directly translate to proxy Actors. The strict separation between policies prevents the modular composition supported by (scoped) Synchronizers, which allows, for instance, combining allocation policies (subsection 2.1) for single chopsticks with the philosophers' coordination policies (subsection 2.2).

The *Directors* coordination model [20] organizes Actors into trees. Messages sent between two Actors are delivered to the closest common ancestor and have to be forwarded by all Actors along the branch leading to the recipient. Actors higher in the tree can therefore determine what messages Actors in their sub-tree receive. Unlike Synchronizers, Directors do not support arbitrarily overlapping constraints. Furthermore, the model does not provide semantics for dynamic reconfiguration: Actors are inserted into the tree when they are constructed.

The middleware architectures proposed by Astley [3] and Sturman [19] display problems similar to conventional Synchronizers. Using Meta-Actors [11,21] as foundation, protocols in these frameworks have global effects, which leads to the problems described in section 3.

In the *Actor-Role-Coordinator* (ARC) model [17], coordination is transparent to base-level Actors; coordination tasks are divided into intra-role and interrole communication. While this hierarchical design provides load-balancing for highly dynamic systems, the coordination structure itself is static. ARC systems therefore avoid security issues through reconfiguration, but require a restart to adapt to changing specifications.

Transactors [5] extend the Actor model with distributed checkpointing as a method for coordination. The goals are fault-tolerance and consistency. In contrast to the assumptions made in this article, Transactors rely on cooperation.

Tuple-Spaces. The anonymous communication provided by tuple-spaces [8] has been proposed as a good fit for open agent systems: writing information tuples on a conceptual global blackboard, agents can coordinate their behavior without knowing each other. This raises robustness concerns in the presence of faulty agents because any agent may remove any tuple from the space. Several mixed static–dynamic [16,23] and dynamic solutions [13] mitigate the problem by limiting the access to tuples. (See these articles for references to many more approaches.)

A recurring goal of security policies in tuple-spaces is secure message passing. The Actor model provides this primitive without the overhead of first sharing, and then enforcing limits on tuples. A tuple-space can be implemented as an Actor, or the Actor model can be extended to include group messaging [1]; if global access to the space is desired, the name of the space can always be provided to any Actor joining the system. The contributions of policy enforcement in tuple-spaces directly apply to the implementation of these tuple-space Actors. We therefore think that tuple-spaces are a valuable communication concept, but are subsumed by the Actor model.

7 Conclusion

We proposed a novel, scoped semantics for Synchronizers that better meets the requirements of coordination in large scale systems. We started with a brief overview of Synchronizers, then demonstrated that the global scope of their constraints allow malicious Actors to intentionally deadlock other Actors, and resolved this challenge by introducing synchronization-capabilities—informally and formally—as a scoping mechanism. While scoping cannot completely prevent accidental deadlocks as sketched at the end of section 3, it still mitigates the problem. The central idea behind our approach was that synchronization constraints should only affect messages originating from Actors to which the constraint-installing Actor holds the capabilities.

Future Work. Declarative synchronization constraints offer a powerful method for describing coordination patterns. However, in their current form, Synchronizers are limited in their expressiveness through their choice to offer but a functional core consisting of two constraint types. An interesting opportunity for future research is extending the selection of available constraints. For instance, syntactic sugar like ordering constraints allows programmers to express their intentions more naturally, and thus make less mistakes. Other concepts like non-interleaving of message sequences cannot be expressed at all.

Another opportunity concerns the robustness of Synchronizers against network partitions and crash failures. Augmenting the semantics with failure detectors [4] appears to be a promising approach. A further interesting direction are methods for handling the information leak discussed in subsection 5.4.

Finally, implementing Synchronizers in a modern Actor framework and conducting a large case study would give interesting insights into the (programmer and computational) performance of Synchronizers.

Acknowledgments. The authors wish to thank Minas Charalambides, Juan Fernando Mancilla Caceres, and the anonymous reviewers for their insightful comments on earlier versions of this article.

This material is based on research sponsored by the Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Gul Agha and Christian J. Callsen. ActorSpaces: An open distributed programming paradigm. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 1993*, pages 23–32, 1993.
2. Gul A. Agha. *ACTORS — A Model of Concurrent Computation in Distributed Systems.* MIT Press series in artificial intelligence. MIT Press, 1986.
3. Mark Astley and Gul Agha. Customizaton and compositon of distributed objects: Middleware abstractions for policy management. In *SIGSOFT FSE*, pages 1–9, 1998.
4. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43:225–267, March 1996.
5. John Field and Carlos A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, pages 195–208. ACM, 2005.
6. Svend Frølund. *Coordinating distributed objects - an actor-based approach to synchronization.* MIT Press, 1996.
7. Svend Frølund and Gul Agha. A language framework for multi-object coordination. In *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern*, volume 707 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 1993.
8. David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
9. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
10. Eugene Letuchy. Facebook chat. Blog entry, May 2008. `http://www.facebook.com/note.php?note_id=14218138919&id=9445547199&index=9` (retrieved on 2011/09/25).

11. José Meseguer and Carolyn L. Talcott. Semantic models for distributed object reflection. In *ECOOP 2002 - Object-Oriented Programming, 16th European Conference*, volume 2374 of *Lecture Notes in Computer Science*, pages 1–36. Springer, 2002.
12. Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.
13. Naftaly H. Minsky, Yaron Minsky, and Victoria Ungureanu. Safe tuplespace-based coordination in multiagent systems. *Applied Artificial Intelligence*, 15(1):11–33, 2001.
14. Naftaly H. Minsky and Victoria Ungureanu. Regulated coordination in open distributed systems. In *Coordination Languages and Models, Second International Conference, COORDINATION '97*, volume 1282 of *Lecture Notes in Computer Science*, pages 81–97. Springer, 1997.
15. Waiming Mok. How twitter is scaling. Blog entry, June 2009. `https://waimingmok.wordpress.com/2009/06/27/how-twitter-is-scaling/` (retrieved on 2011/09/25).
16. Rocco De Nicola, Daniele Gorla, René Rydhof Hansen, Flemming Nielson, Hanne Riis Nielson, Christian W. Probst, and Rosario Pugliese. From flow logic to static type systems for coordination languages. In *Coordination Models and Languages, 10th International Conference, COORDINATION 2008*, volume 5052 of *Lecture Notes in Computer Science*, pages 100–116. Springer, 2008.
17. Shangping Ren, Yue Yu, Nianen Chen, Kevin Marth, Pierre-Etienne Poirot, and Limin Shen. Actors, roles and coordinators - a coordination model for open distributed and embedded systems. In *Coordination Models and Languages, 8th International Conference, COORDINATION 2006*, volume 4038 of *Lecture Notes in Computer Science*, pages 247–265. Springer, 2006.
18. Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebr. Program.*, 79(6):397–434, 2010.
19. Daniel Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.
20. Carlos A. Varela and Gul Agha. A hierarchical model for coordination of concurrent activities. In *Coordination Languages and Models, Third International Conference, COORDINATION '99*, volume 1594 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 1999.
21. Nalini Venkatasubramanian and Carolyn L. Talcott. Reasoning about meta level activities in open distributed systems. In *PODC*, pages 144–152, 1995.
22. Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, MA, USA, 1993.
23. Fan Yang, Tomoyuki Aotani, Hidehiko Masuhara, Flemming Nielson, and Hanne Riis Nielson. Combining static analysis and runtime checking in security aspects for distributed tuple spaces. In *Coordination Models and Languages - 13th International Conference, COORDINATION 2011*, volume 6721 of *Lecture Notes in Computer Science*, pages 202–218. Springer, 2011.