Actors for Engineering Scalable Multiagent Systems

Amit K. Chopra[1] and Munindar P. Singh[2]

[1] Lancaster University, Lancaster, UK
[2] North Carolina State University, USA
{amit.chopra@lancaster.ac.uk, mpsingh@ncsu.edu}

Abstract. Despite much progress on engineering multiagent systems (MAS), current approaches do not tackle scalability. Indeed, popular MAS frameworks and programming models implicitly limit an agent to running on a single physical computer. Without addressing scalability, MAS engineering approaches stand little chance of being widely adopted.

Interaction protocols model multiagent systems. Recent years have witnessed significant advances in programming models based on protocols. Actors is an influential model of shared-nothing concurrency. A major benefit of actors is scalability. In this contribution, we outline a vision for realizing scalable MAS that synthesizes ideas from information protocols and actors. Specifically, we outline an approach and attendant challenges in realizing a protocol-based agent as a MAS of actors, each handling an independent computation and executing somewhere in a cluster of machines.

1 Introduction

Virtually every application domain for information technology, e.g., business, health, and smart cities, involves interactions between autonomous real-world principals. Therefore, a natural approach to addressing such applications is to represent each principal by an *agent* who encodes the principal's decision making and interacts with other agents on its behalf. To promote loose coupling, the interactions between the agents are realized via asynchronous messaging. This *multiagent systems (MAS)* conception contrasts with traditional approaches such as Web services by respecting the autonomy of their member agents and decoupling their implementations [12].

Scalability is the idea that an operational software system can cope with unbounded growth in its usage. Scalability is ultimately bounded by the computing hardware (cores, memory, storage, etc.) that the software system runs on. Being able to cope with unbounded growth therefore requires a software system to run on ever larger computers (possibly virtual, such as clusters) and utilize them effectively. Moreover, scalability mechanisms should be transparent to application programmers, enabling them to focus on the business logic. Scalability

is obviously a crucial practical concern and a motivation for modern paradigms such as cloud computing, including innovations such as serverless [22].

Distribution makes scalability challenging. Programmers rely on application state to write the application's business logic. A distributed architecture implies that the state would be distributed, which may produce unintended application-level consequences. In the context of a Web service that is replicated for purposes of scalablity, a classic example is the double booking of a resource such as a hotel room. The crux of the challenge is to devise programming abstractions that hide the distribution but give correctness guarantees. Additionally, the abstractions must not unduly hit performance. Distributed transactions [5] is one such classic abstraction; however, it is also known to not be suitable to settings of autonomous principals [26, 17]. Current serverless platforms support stateless applications; abstractions for stateful applications are of considerable interest [13].

Research on software abstractions for engineering MAS has paid little attention to scalability. Over the years diverse agent programming models and frameworks have emerged. JADE [3], Jason [7], JaCaMo [6], and Kiko [15], and Orpheus [9] reflect this diversity. None addresses scalability. Although each agent in a MAS can run on a separate physical computer, current approaches limit an agent to running on a single computer. Most MAS engineering methodologies entertain the idea that an agent could itself be implemented as a MAS. As each agent can run on a separate machine, such an recursive architecture would in principle be conducive to scalability. However, current approaches (most notably [24], whose *raison d'être* is to exploit this architectural possibility) are geared toward explicitly modeling the internal structure of agents, not automated, transparent scalability mechanisms.

In recent years, *actors* [19, 1, 32] has come to be widely appreciated as a basis for building highly scalable, concurrent systems. Akka is a popular, mainstream actor-based programming model and platform (http://akka.io). Erlang [2], which embodies the spirit of the actor model, has witnessed a resurgence in interest. Microsoft Orleans [8, 4] introduces *virtual actors* with the aim of making actor-based programming native to the cloud.

Notably though, despite conceptual affinities, especially in their emphasis on building systems out of asynchronously communicating components, the actor model has not seen much uptake in MAS research. Although some claim inspiration, e.g., [23, 15], none of the aforementioned MAS programming models apply the actor model.

To address the challenge of MAS scalability, we propose a novel synthesis of declarative information-based protocols [28, 30, 29, 31] and the actor model. An information protocol models a MAS by specifying the communication between agents. Agents enact information protocols in a decentralized manner by sending and receiving messages. Our synthesis rests on the discovery of a complementarity between information protocols and the actor. Exploiting scalability via actors rests on identifying logically independent computations. Specifically, an actor could be assigned to each such computation. The enactments of an

information protocol are logically independent computations. This suggests the possibility of realizing an agent as a set of actors, one for each enactment it is engaged in.

In this paper, we elaborate upon our proposition. Section 2 explains information protocols and their value as a general-purpose abstraction for engineering multiagent systems. Section 3 explains the value proposition of the actor model and recent advances such as *virtual actors*. Section 4 explains our proposed MAS architecture, highlighting how an agent could potentially be realized as a distributed entity using virtual actors. It also poses some research challenges in realizing this vision. Section 5 discusses some related work and summarizes key points.

2 Information Protocols

Several approaches for specifying interaction protocols exist in the literature. None matches the information protocols approach when it comes to specifying flexible, decentralized enactments [10]. Below, we highlight the novelty and expressiveness of this approach.

2.1 Specifying Protocols

The listing below gives a protocol *Ebusiness*. It specifies messages, with their senders, receivers, and information parameters. Parameter ID is key: it identifies enactments; messages with the same binding for ID belong to the same enactment. Think of an enactment as a *business transaction* (as opposed to a database transaction). Adornments ⌜in⌝, ⌜out⌝, and ⌜nil⌝ specify *causality* and are interpreted relative to enactments. A message instance has bindings for the ⌜in⌝ and ⌜out⌝ but not the ⌜nil⌝ parameters. An agent may emit a message (instance) if it *knows* all ⌜in⌝ parameters (that is, their bindings exist in the agent's *local state*, its communication history) and does not know any ⌜out⌝ or ⌜nil⌝ parameter (that is, their bindings do not exist in its local state). Sending the message adds it to the agent's history (making its ⌜out⌝ parameters known). Receiving a message adds it to the receiver's history (making all its parameters known). Notably, messages can be received in any order.

Listing 1: A three-party protocol for conducting e-business transactions.

```
Ebusiness {

 roles Buyer, Seller, Bank
 parameters out ID key, out item, out price, out amount, out
     status

 //Messages
 Seller -> Buyer: Offer[out ID key, out item, out price]
 Buyer  -> Seller: Accept[in ID key, in item, in price, out
     decision]
```

```
 Buyer  -> Bank: Instruct[in ID key, in price, out details]
 Bank   -> Seller: Transfer[in ID key, in price, in details,
     out amount]
 Seller -> Buyer: Shipment[in ID key, in item, in price, out
     status]
}
```

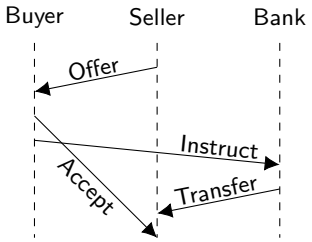

Fig. 1: A flexible enactment of *Ebusiness* in which messages are reordered in the communication infrastructure.

Applying the above reasoning to *Ebusiness*, after sending *Offer*, SELLER knows ID, item, and price, and therefore may send *Shipment* anytime thereafter. By analogous reasoning, BANK may send *Transfer* anytime after receiving *Instruct*. Figure 1 shows an *Ebusiness* enactment that traditional approaches cannot handle: *Transfer* is received by SELLER before *Accept*. To get a sense of *Ebusiness*'s "exponential" flexibility, consider that any enactment can progress to completion via 630 distinct operational paths of sends and receives.

2.2 Programming Protocol-Based Agents

Given an information protocol, Kiko enables implementing Python agents that play roles in the protocol. To make agent development easy, Kiko includes an adapter (middleware) that can sit directly atop the Internet and exposes an event-driven, information-based interface for implementing agents.

An agent's Kiko adapter maintains its local state and based on that and the protocol specification keeps track of information-enabled *forms*. The forms are necessarily partial message instances that would be legal to send if completed. Specifically, a form's ⌜in⌝ parameters are bound (from the local state) and the ⌜out⌝ parameters are unbound (because they don't exist in the local state).

Table 1 gives a possible local state for a SELLER agent and the forms available to it in that state.

To write a Kiko agent, an agent writes a set of *decision makers*. A decision maker is an event-triggered piece of code that gets the set of enabled forms and completes some subset via some logic. The completed forms are emitted by the adapter as messages and added to the local state. Listing 2 shows a decision

Local state	Enabled forms
Offer(1, fig, 10) Offer(2, jam, 100) Accept(2, jam, 100, thanks) Transfer(1, fig, urgent, 10)	Shipment(1, fig, 10, status) Shipment(2, jam, 100, status)

Table 1: A possible local state for a SELLER agent and the enabled forms in that state.

maker for a SELLER agent. Its logic is to complete *Shipment* in those enactments for which *Accept* has been received, regardless of whether *Transfer* has been received. The completed *Shipment* forms are sent by the adapter as messages. The decision maker is triggered at 1700 hours every day; other trigger specifications are possible. Throw in a decision maker for completing *Offer* messages and that is all the messaging-related code a programmer need write to implement a SELLER. Notably, the programmer never writes code to receive messages.

Listing 2: A SELLER agent's decision maker that sends *Shipment* only in those enactments (as identified by ID) in which *Accept* has been received.

```
@adapter.schedule_decision(00 17 * * *)
def shipment(enabled, state):
  shipments = enabled.messages(Shipment)
  for s in shipments:
     if(next(state.messages(Accept, system=s.system,
         ID=s["ID"])))
        s.bind(status="firstclass '')
```

3 Actors

The actor model is notable for standing in contrast to shared-something models of concurrency such as threads (which share memory) and communicating sequential processes (which, being synchronous, share clocks).

3.1 The Model

The actor model is a model of shared-nothing, concurrent computation. In this model, the computation is performed by logically distributed entities called *actors* and they may communicate only via *asynchronous* messaging.

Briefly, an actor is an addressable, stateful unit of computation. The actor encapsulates its state: Instead of other actors modifying its state directly, they send messages to it reflecting the operations they want performed. Each actor has a mailbox into which messages addressed to it arrive nondeterministically. The actor runs an infinite loop in each iteration of which it picks up a message

from its mailbox for processing. In processing the message, the actor may change its state, spawn new actors, and send messages to other actors. In this manner, the actor serializes the operations on its state and supports correctness.

3.2 Scalability

To see how the actor model can support scalability, imagine a system's resources to be *sharded*, that is, divvied up, such that each is managed by a unique actor. The actor represents its state and serializes operations on it. We make these ideas concrete via an ebusiness example inspired from the classic bank account example. Let's say a seller has several warehouses. Items may be deposited or withdrawn from a warehouse. We could represent each warehouse by an actor with operations *deposit* and *withdrawal*. Such a system model would guarantee that *withdrawal* operations on a warehouse do not violate the (classic) integrity constraint that the number of available items in a warehouse never becomes negative. Moreover, as many operations can run concurrently as the number of actors in the system. Depending on the numbers of cores in the system, the operations may even run in parallel. Scalability is supported in principle because as the number of warehouses (actors) grow, the system can scale by simply adding more cores. In a nutshell, actors accommodate true concurrency and resource sharding enables exploiting it.

Deposits and withdrawals are single-actor operations. However, often operations span actors, e.g., *transfers* between two warehouses. Again, following the classic example of transfers in banking, we might wish to impose the integrity constraint that the sum of the items in the involved warehouses before the operation must equal that after the operation. To accommodate this constraint, following traditional approaches, we would model a *transfer* as an *atomic* operation. Database (not business) transactions guarantee atomicity in a shared-memory setting. Since actors are distributed, work has specifically looked at how to extend actors-based programming models with distributed transactions [25, 16]. We have more to say on multiactor operations in Section 4.2.

3.3 Implementation in Orleans

Actor-based programming models such as Akka and Orleans are supported by middleware (platform) that abstracts over the underlying cluster of computers and offers services such as actor supervision and (state) persistence. We focus here on Orleans because it automates more of the actor management (at least for our present purposes).

The Orleans middleware abstracts over a cluster of machines. Actors have unique identifiers, which also double as their addresses. In Orleans, programmers don't create or destroy actors. An actor is spun up whenever a message is addressed it and it is spun down whenever it has been inactive for a while. An actor's state is saved to persistent storage, so when it is spun up, it is resurrected with its state. Orleans keeps track of where in a cluster an actor lives and does the message routing. Such a model supports fault tolerance and load

balancing. If an actor dies, when a message is addressed to it, it spins it up on some relatively lighted-loaded machine in the cluster. Orleans supports both request-response and one-way messages. Computers can be dynamically added or removed from the underlying cluster.

4 Mapping Agent Computations to Actors

4.1 Insight

An agent implements information protocols and may participate in several enactments of these protocols with other agents. Every enactment is identified by the bindings of the relevant key parameters and represents a distinct social object [31]. For example, in Table 1, the seller's local state maps to two social objects with identifiers ID=1 and 2, respectively, as shown in Figure 2. Enactments are independent computations; they may therefore proceed in parallel.

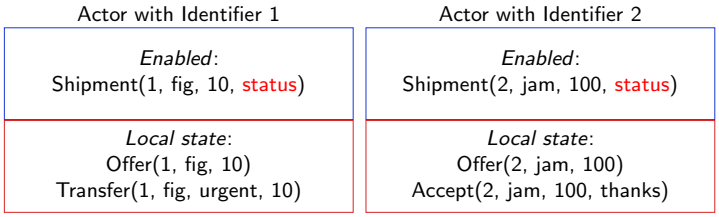

Fig. 2: Proposed agent architecture. Each Agent is realized via virtual actors, each of whom handles a protocol enactment. We refer to such actors as *e-actors*. Basically, an e-actor sends and receives messages in its enactment, maintains and persists the local state as its state, and offers a messaging-based interface for computing enabled messages and sending completed forms.

Each social object corresponds to an Orleans actor whose identifier is the object's identifier. This opens up the possibility, as shown in Figure 2, of realizing the seller agent via a set of Orleans virtual actors, one for each enactment. We refer to such actors as e-actors (for enactment actors).

4.2 Challenges

The challenges relate to figuring out the details of the agent architecture with the aim of supporting a simple agent programming interface and high performance. We list some below.

Programming Model. From the point of view that agents implement protocols, agent programmers need to be aware of the use of Orleans. A Kiko-inspired programming model that lets programmers query for enabled forms

and complete them based on whatever the relevant business logic happens to be is appealing if we want to *let programmers focus on the business logic*. How can we achieve this goal of a programming model that supports scalability via e-actors but abstracts it away from the programmer?

Nested Enactments. In general, protocols may feature composite keys to capture the fact that a business transaction may have subtransactions. For example, in an alternative ebusiness protocol, an enactment, identified by tID, may support multiple shipments, each identified by sID. How we do relate the e-actors in a manner that captures these relationships?

Reasoning about Resources. An agent may reason about *internal* resources, e.g., items available in its warehouse (as discussed above, actors enable maintaining the states of such resources). In reasoning about whether it should make an *Offer*, a seller may reason check availability in the warehouse and modify it. Such reasoning is commonplace, e.g., in making hotel and flight bookings. What kind of software abstractions facilitate such reasoning?

Meaning. Communications between agents has normative meaning [27], which is critical to business reasoning. For example, *Offer* may mean a commitment from the seller agent to the buyer agent to perform *Shipment* upon *Transfer*. Meaning relates to reasoning about resources. For example, a prudent seller may not want to make more *Offer* commitments than it can reasonably expect to fulfill, given the stock in its warehouses. How can we represent meaning and use it to inform decision making by agents?

Multienactment Reasoning. Agents often engage reasoning that spans multiple e-actors. For example, a BUYER agent may want to determine the best *Offer* before accepting any. To capture the meaning of protocols involving composite keys, the relevant norms would normally involve aggregate conditions [11], for example, that multiple shipments cover the promised quantity of items. How can the programming model support such aggregate reasoning?

Fault Tolerance. Addressing the challenges outlined above will require organizing the e-actors themselves as MAS. For example, to get the set of all enabled *Shipment*s, the seller's business logic must send a query message to each *Ebusiness* e-actor. In other words, what were synchronous, local procedure calls in Kiko have to be realized via asynchronous messaging. Moreover, the messages may be arbitrarily delayed. Recent work has studied failure handling for protocol-based MAS [14]. Can we apply similar techniques to handle failures in e-actor MAS?

Performance. Given the foregoing, could the e-actors be so chatty that performance will be severely impacted? What optimizations could reduce the chattiness? What hardware innovations does actor-based computation motivate?

Consistency. What consistency guarantees can the programming model offer? We can rely on Orleans to (more or less) guarantee that for any enactment, there exists at most one e-actor in the cluster at any time. This will guarantee consistency of an e-actor's state. However, multienactment reasoning raises some challenges. For example, if by the time an agent's instruction

to send a completed enabled form reaches the relevant e-actor, the form may no longer be legal for emission (maybe because the e-actor received a conflicting message in the meantime). It would be tempting to introduce distributed transactions, as discussed above, to avoid such scenarios; however, distributed transactions are incompatible with scalability [17].

Formalization. Can we formalize the theoretical ideas behind the programming model with the aim of introducing higher-level abstractions than actors and connecting both with the distributed systems and the programming languages community?

5 Summary

Information protocols represent a foundational abstraction for MAS. Our proposed approach is a middleware-supported programming model that enables automatically realizing an agent as a multiagent system of actors that is distributed over a cluster of machines. Our proposal automatically realizes an agent as a multiagent system of actors for purposes of scalability. We discussed some of the challenges that must be addressed to realize this vision, but the overarching one is a programming model that accords with flexibility, performance, and scalability.

Our approach addresses a perceived limitation of actors—that they are low-level. The reason for this perception is that "actors comes in systems" [20]. Modeling systems requires a focus on modeling the interactions between its components. We think information protocols and actors fit hand in glove due to their shared emphasis on loose coupling, asynchronous messaging, and reliance on higher-level social abstractions [18, 21].

References

1. Agha, G.A.: Actors. MIT Press, Cambridge, Massachusetts (1986). https://doi.org/10.7551/mitpress/1086.001.0001
2. Armstrong, J.: Erlang. Communications of the ACM **53**(9), 68–75 (2010)
3. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley-Blackwell (2007)
4. Bernstein, P.A., Bykov, S.: Developing cloud services using the orleans virtual actor model. IEEE Internet Computing **20**(5), 71–75 (2016)
5. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading, Massachusetts (1987)
6. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Science of Computer Programming **78**(6), 747–761 (Jun 2013)
7. Bordini, R.H., Hübner, J.F., Wooldridge, M.J.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons, Chichester, United Kingdom (2007). https://doi.org/10.1002/9780470061848
8. Bykov, S., Geller, A., Kliot, G., Larus, J.R., Pandya, R., Thelin, J.: Orleans: Cloud computing for everyone. In: Proceedings of the ACM Symposium on Cloud Computing. No. 16 (2011)

9. Chopra, A.K., Baldoni, M., Christie V, S.H., Singh, M.P.: Orpheus: Engineering multiagent systems via communicating agents (2025), https://drive.google.com/file/d/18cFBUUykuxdPZ3NB1NM4gzU9JODJnmey/view
10. Chopra, A.K., Christie V, S.H., Singh, M.P.: An evaluation of communication protocol languages for engineering multiagent systems. Journal of Artificial Intelligence Research **69**, 1351–1393 (2020)
11. Chopra, A.K., Singh, M.P.: Custard: Computing norm states over information stores. In: Proceedings of the 15th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 1096–1105. IFAAMAS, Singapore (May 2016). https://doi.org/10.5555/2936924.2937085
12. Chopra, A.K., Singh, M.P.: From social machines to social protocols: Software engineering foundations for sociotechnical systems. In: Proceedings of the 25th International World Wide Web Conference. pp. 903–914. ACM, Montréal (2016)
13. Christie V, S.H., Chopra, A.K., Singh, M.P.: Deserv: Decentralized serverless computing. In: Proceedings of the 19th IEEE International Conference on Web Services (ICWS). pp. 51–60. IEEE Computer Society, Virtual (Sep 2021). https://doi.org/10.1109/ICWS53863.2021.00020
14. Christie V, S.H., Chopra, A.K., Singh, M.P.: Mandrake: Multiagent systems as a basis for programming fault-tolerant decentralized applications. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **36**(1), 16:1–16:30 (Apr 2022). https://doi.org/10.1007/s10458-021-09540-8
15. Christie V, S.H., Chopra, A.K., Singh, M.P.: Kiko: Programming agents to enact interaction protocols. In: Proceedings of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS). p. to appear. IFAAMAS, London (May 2023)
16. Eldeeb, T., Burckhardt, S., Bond, R., Cidon, A., Yang, J., Bernstein, P.A.: Cloud actor-oriented database transactions in orleans. Proceedings of the VLDB **17**(12), 3720–3730 (2024)
17. Helland, P.: Autonomous computing. ACM Queue **20**(1), 80–104 (2022)
18. Hewitt, C.: ORGs for scalable, robust, privacy-friendly client cloud computing. Internet Computing **12**(5), 96–99 (2008)
19. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI). pp. 235–245. William Kaufmann, Stanford (Aug 1973), http://ijcai.org/Proceedings/73/Papers/027B.pdf
20. Hewitt, C., Meijer, E., Szyperski, C.: The actor model (everything you wanted to know). https://www.youtube.com/watch?v=7erJ1DV_Tlo
21. Jamali, N., Agha, G.A.: CyberOrgs: A model for decentralized resource control in multi-agent systems. In: Proceedings of the AAMAS Workshop on Representations and Approaches for Time-Critical Decentralized Resource/Role/Task Allocation (2003)
22. Jangda, A., Pinckney, D., Brun, Y., Guha, A.: Formal foundations of serverless computing. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 149:1–149:26 (Oct 2019). https://doi.org/10.1145/3360575
23. Ricci, A., Santi, A.: From actors and concurrent objects to agent-oriented programming in simpAL. In: Agha, G.A., Igarashi, A., Kobayashi, N., Masuhara, H., Matsuoka, S., Shibayama, E., Taura, K. (eds.) Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday. LNCS, vol. 8665, pp. 408–445. Springer (2014)

24. Rodriguez, S., Hilaire, V., Gaud, N., Galland, S., Koukam, A.: Holonic multi-agent systems. In: Self-organising Software - From Natural to Artificial Adaptation, pp. 251–279. Natural Computing Series, Springer (2011)
25. Shah, V., Salles, M.A.V.: Reactors: A case for predictable, virtualized actor database systems. In: Proceedings of the International Conference on Management of Data (SIGMOD). pp. 259–274. ACM (2018)
26. Singh, M.P.: Multiagent systems as spheres of commitment. In: Proceedings of the International Conference on Multiagent Systems (ICMAS) Workshop on Norms, Obligations, and Conventions. pp. 1–13. Kyoto, Japan (Dec 1996)
27. Singh, M.P.: Agent communication languages: Rethinking the principles. IEEE Computer **31**(12), 40–47 (Dec 1998)
28. Singh, M.P.: Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language. In: Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems. pp. 491–498. IFAAMAS (2011)
29. Singh, M.P.: LoST: Local State Transfer—An architectural style for the distributed enactment of business protocols. In: Proceedings of the 9th IEEE International Conference on Web Services (ICWS). pp. 57–64. IEEE Computer Society, Washington, DC (2011)
30. Singh, M.P.: Semantics and verification of information-based protocols. In: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 1149–1156. IFAAMAS, Valencia, Spain (Jun 2012)
31. Singh, M.P.: Bliss: Specifying declarative service protocols. In: Proceedings of the 11th IEEE International Conference on Services Computing (SCC). pp. 235–242. IEEE Computer Society, Anchorage, Alaska (Jun 2014). https://doi.org/10.1109/SCC.2014.39
32. Varela, C.A., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. ACM SIGPLAN Notices **36**(12), 20–34 (2001)