

Dreams: a framework for distributed synchronous coordination*

José Proença Dave Clarke
IBBT-DistriNet, KUL
Leuven, Belgium
{jose.proenca,dave.clarke}@cs.kuleuven.be

Erik de Vink
TUE, Eindhoven,
The Netherlands
evink@win.tue.nl

Farhad Arbab
CWI, Amsterdam,
The Netherlands
farhad.arbab@cwi.nl

ABSTRACT

Synchronous coordination systems, such as *Reo*, exchange data via indivisible actions, while distributed systems are typically asynchronous and assume that messages can be delayed or get lost. To combine these seemingly contradictory notions, we introduce the *Dreams* framework. Coordination patterns in *Dreams* are described using a synchronous model based on the *Reo* language, whereas global system behaviour is given by the runtime composition of autonomous actors communicating asynchronously. *Dreams* also exploits the use of actors in the composition of coordination patterns to allow asynchronous communication whenever possible, increasing the scalability of the implementation.

Keywords

Reo, synchronous coordination, actor model, distributed systems

1. INTRODUCTION

Synchronous languages, such as *Reo* [4] and Esterel [9], are useful for programming reactive systems, but they seem less suited for coordinating distributed systems. For example, existing implementations based on *Reo* [6] do not scale up and have expensive reconfiguration costs, mainly due to the global synchronisation constraints imposed by *Reo*.

To remedy this situation, the GALS model [12] has been adopted, GALS abbreviating globally asynchronous and locally synchronous, wherein local computation is synchronous but communication between actors running on different machines is asynchronous. Our work contributes to the field of coordination, in particular to *Reo*, by incorporating the ideas underlying GALS into our approach to execute synchronisation models, thereby increasing scalability and allowing inexpensive reconfiguration.

*This research is supported by FCT grant 22485 – 2005, Portugal, and by the K.U.Leuven BOF-START project STRT1/09/031 DesignerTypeLab, Belgium.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'12 March 25-29, 2012, Riva del Garda, Italy.
Copyright 2011 ACM 978-1-4503-0857-1/12/03 ...\$10.00.

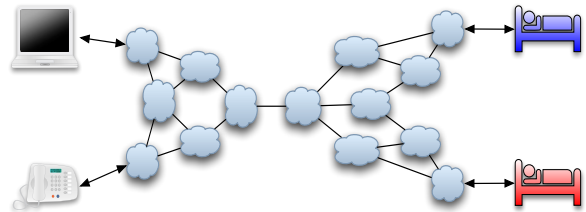


Figure 1: Distributed coordination of web services.

This paper introduces *Dreams*, an acronym for distributed runtime evaluation of atomic multiple steps), a decentralised framework that supports the execution of multiple concurrent threads, each describing part of the global coordination behaviour. Furthermore, we statically partition the concurrent threads performing coordination into smaller sets, called *synchronous regions*. This partitioning allows communication among threads within a synchronous region to be handled synchronously, having asynchronous communication between synchronous regions only. In short, the main contributions of this paper are:

- support for *decoupled execution* of *Reo*,
- improved *scalability*, and
- possibility of inexpensive *reconfiguration*.

Dreams uses an underlying actor model [1, 2]. Primitive coordinators, which we simply call *actors*, exchange asynchronous messages. In the *Reo* setting, an actor of *Dreams* manages one or more *Reo* channels, nodes, or connectors. The communication among actors follows the graph structure of the corresponding *Reo* connector, which restricts the potential communication partners of each actor to its neighbours. Figure 1 illustrates the main idea behind *Dreams*. It depicts the coordination of four different services, connecting a phone-based and an Internet-based service to book hotels. Each hotel provides its own reservation service. A cloud represents an actor, an independent thread of execution of the coordination layer which cooperates with its neighboring actors via asynchronous messages.

Each actor has an associated behavioural automaton [23] that describes its behaviour. The global coordination behaviour is given by the composition of the automata of all actors. The *Dreams* framework supports the distributed execution of the coordination mechanism. The network is represented by the net of clouds in Figure 1.

In this paper we focus on the key design decisions in the development of *Dreams* and evaluate its performance, but do

not describe the protocol that the actors use to impose the global synchronous constraints, which can be found in [23]. Moreover, at present the *Dreams* framework does not address failure, and assumes that the exchange of messages between actors always succeeds in finite time. However, we envisage failure addressed orthogonally at a later stage, as suggested by existing techniques for distributed programming [15].

We provide an overview of *Reo* in §2. The *Dreams* framework is explained in §3, and the current state of its implementation is briefly described in §4. We include also a small benchmark to compare our implementation with another existing *Reo* engine. We compare *Dreams* with related work in §5, and conclude in §6.

2. REO OVERVIEW

Reo is a channel-based coordination language with a graphical notation, introduced by Arbab in 2001 [3], wherein complex connectors are built out of an open set of *primitive connectors*, also simply called primitives. *Reo* is synchronous, exogenous, and composable, yielding an expressive and intuitive coordination model. Channels are special primitives with two ports, which can either send or receive data. For example, the Sync channel ‘ $a \longrightarrow b$ ’ is a primitive that receives data through port a and sends it through port b atomically, that is, it only receives data if it can also send it. The LossySync channel ‘ $a \dashrightarrow b$ ’ can either forward data from a to b atomically, as the Sync channel, or discard data received through port a . The SyncDrain channel ‘ $a \dashleftarrow b$ ’ has two input ports, and requires both ports to have dataflow atomically, or no dataflow is possible. The FIFO₁ ‘ $a \square b$ ’ is a stateful channel: when empty it can receive a data token through the port a , becoming full, and when full it can send the previously received data through the port b .

Connectors are composed via *Reo* nodes. A *Reo* node, depicted as \bullet , is a primitive with a set of input ports and a set of output ports. Semantically, a *Reo* node receives a data item from only one of its input ports at a time, and atomically, replicates this data item to all of its output ports. The left side of Figure 2 depicts a sequencer connector, used as a running example throughout this paper. This connector has three input ports, a , b , and c , which send data atomically to the three output ports d , e and f , respectively. The SyncDrain channels and the loop of FIFO₁ channels enforce the alternation between the atomic sending of data. Initially, only a can send data to d , and only in a following step can b send data to e .

The *Dreams* framework uses behavioural automata [23] to give semantics to *Reo*. We do not present here behavioural automata, and describe only the main concepts behind this model. Each label of a behavioural automaton describes an atomic step that the corresponding connector can perform. A label in this model represents, among other things, a set P of known ports and a set $F \subseteq P$ of ports with dataflow. When composing two connectors, the labels of each automata are composed in a pairwise fashion via a partial function \otimes , which is undefined when the actions in the two labels cannot be performed in the same atomic step. Furthermore, the actions in some labels can be performed without being composed with another label, according to a predicate associated with each of the states of a behavioural automaton.

3. COORDINATION VIA ACTORS

The distribution mechanism introduced by the *Dreams* framework addresses the limitations of centralised approaches for implementing synchronous languages. Primitive entities in *Dreams* are *actors* [1, 2]. An actor is an active entity that runs concurrently with other actors and communicates with them using a reliable, order-preserving asynchronous message passing mechanism. Each actor has an associated behavioural automata, used to describe the synchronous semantics of the coordination layer.

We now briefly summarise the protocol used to impose the global synchronous constraints. *Dreams* evolves in rounds, in a manner similar to other *Reo* engines [7, 14], where each round consists of four distinct phases. In the *request* phase a subset of actors, defined later in this section, sends a message to each of its neighbours asking for the part of their behavioural automata relevant for the current round. In turn, the neighbours will also ask the behavioural automata of their own neighbours, and so on. In the *atomic commitment* phase a single actor succeeds in collecting the automata of all actors, and selects a transition to be executed in the current round. The selected transition is propagated to all actors along with the appropriated data in the *propagation* phase. Finally, each actor updates the state of its behavioural automaton according the selected transition in the *update* phase.

This paper focuses on choosing the actors involved and on the properties of the exchanged behavioural automata, carving out the essence of the protocol used to coordinate the actor system [23]. Intuitively, actors propagate requests for the behavioural automaton of their neighbours, starting by at least one actor, until all the system is covered. Requests have an associated rank value, based on the actor that started to propagate the request, which is used to resolve race conditions. After collecting and deciding the atomic step of the global system, actors propagate this step, possibly including also the data value flowing through the associated port or a data request. After all actors perform their associated steps, the state of their behavioural automata is updated, and a new round starts.

3.1 Reo as a system of actors

The existing implementation of the *Dreams* framework currently incorporates two concrete incarnations of behavioural automata: one based on connector colouring [13] and one on constraint-based models of *Reo* [14]. Each *Reo* channel is mapped directly to an actor with a concrete incarnation of the behavioural automaton given by its *Reo* semantics. Merging and replicating of dataflows are performed by *Reo* nodes with multiple input and output ports, each of which is also mapped to a single actor.

We depict a possible encoding of the sequencer *Reo* connector into the *Dreams* framework in Figure 2, according to the extreme scenario where every node and channel is mapped to a separate actor. The sequencer connector forwards data from a to d , then from b to e , and finally from c to f , returning to its original state. Alternatively, we can also define a single actor with the combined behavioural automaton of the full connector, resulting in a centralised implementation of this connector. The sequencer connector is a variation of some of van Der Aalst’s workflow patterns [24]. Following the ideas behind our motivating hotel example, a

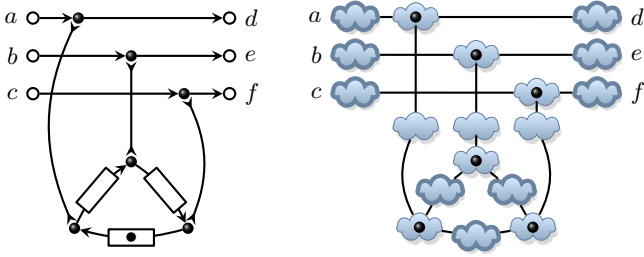



Figure 2: Sequencer connector in Reo (left) and its encoding into Dreams (right).

possible usage of this connector is to alternate the connection to different receptionists in a single hotel.

Proactive actors

A *proactive actor*, depicted with thick borders  in Figure 2, is a special actor that can produce or receive data through one of its ports independently of any of its other ports. Intuitively, an actor is proactive if it has a port, the proactive port, that, in a particular reachable state, sends or receives data that is independent of the data at any of the other ports. Proactive actors are responsible for starting to send requests in each round, and for selecting the transition to be performed.

Example. In the case of the empty $FIFO_1$ channel, its input port is proactive because for any possible value flowing through a , this atomic step does not depend on the data flowing on b . In turn, its output port is not proactive when the $FIFO_1$ channel is empty because it cannot have dataflow.

The Dreams framework makes two additional assumptions to avoid the scenario where proactive actors constantly try to send or receive data without success. First, a transition without dataflow, i.e., labeled with an empty set of ports, does not change the state of a behavioural automaton. Second, during the atomic commitment phase a transition without dataflow can be selected only if there is no transition with dataflow possible.

3.2 Synchronous regions

Scalability in Dreams is achieved via a true decoupling of the execution. The specification of a Dreams configuration as a set of connected actors that can execute concurrently already provides a basic decoupling of the execution. We go beyond this basic decoupling by analysing the behavioural automata of the actors involved and identifying links between actors that require only asynchronous communication. We depict these truly asynchronous connections using dotted lines, as shown between some of the clouds in Figure 3.

The presence of asynchronous communication yields what we call *synchronous regions*, depicted in Figure 3 by a grey background grouping actors from the same region. Actors in the same synchronous region must reach consensus in each round of communication, but actors from different synchronous regions can communicate asynchronously over several rounds. Reducing the number of actors involved in the search for a consensus reduces the complexity of this search, which is confirmed by our benchmarks.

Synchronous regions communicate with each other only using asynchronous messages. In Figure 3 we divided the se-

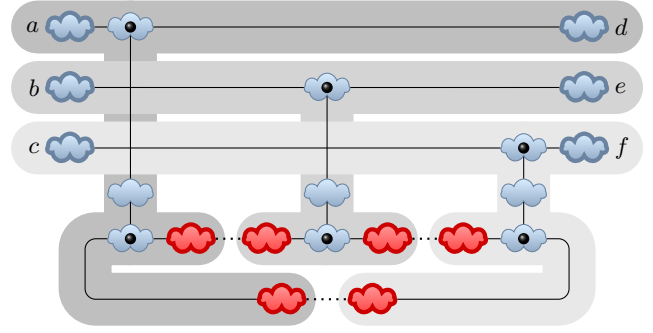


Figure 3: Division of the sequencer connector into synchronous regions.

quencer connector, introduced in Figure 2, by splitting each actor associated with a $FIFO_1$ channel into two new actors that communicate only asynchronously. The *splitting* of actors is performed at compile time, based on the behavioural automata of the actors.

Splitting actors

The behavioural automata of an actor may execute asynchronously at their ports. For example, the behavioural automaton of the $FIFO_1$ channel can evolve each of its ports asynchronously: if the source and sink ports can perform steps with labels ℓ_1 and ℓ_2 , respectively, then performing the composed step with label $\ell_1 \otimes \ell_2$ is equivalent to perform ℓ_1 and ℓ_2 one at a time. The criteria for deciding if ports behave asynchronously is formalised by Proença [23], and omitted in this paper. When ports from an actor exhibit asynchronous behaviour we can *split* the actor into two smaller actors, each with its own set of ports, that communicate with each other using asynchronous messages only.

We split the behavioural automaton of an actor with ports P by duplicating it, and by restricting each copy of the automaton to two disjoint sets of ports X and Y , such that $X \cup Y = P$. Each split actor communicates with the other split actor only by sending it an asynchronous message whenever its state is updated.

4. IMPLEMENTATION

We developed a prototype Dreams engine to experiment with our approach, using an actor library for the Scala language [16]. The current version of the source code of the prototype is available in the Reo repository.¹ Deploying a Reo connector creates an actor for each Reo primitive and node, which run in concurrent threads. Deployed $FIFO_1$ channels are split, as explained in §3.2. Note that although we deploy split actors, we did not yet automate the splitting process. We also implemented data producers and data consumers of a given number of data values. Data consumers, data producers, and $FIFO_1$ channels are deployed as proactive actors, while the other Reo primitives and nodes are non-proactive.

The actor library for Scala allows actors to execute on different machines connected through IP. Using this functionality, our implementation allows the deployment of the

¹reoproject.cwi.nl/cgi-bin/trac.cgi/reo/browser/reo-engine

actors of a single connector onto different machines. Furthermore, we developed a graphical deployment plug-in for Eclipse. It uses the *Reo* editor included in the ECT framework² [6, 19] to extract information from the connectors under construction. The user, the developer of a *Reo* application, can then deploy and run actors associated with primitives in the editor using a graphical interface. However, in our benchmark all actors of a connector execute concurrently on a single machine as the *Reo* engine that we use for comparison is centralised.

Evaluation

We evaluate the performance of our implementation of the *Dreams* framework by considering the time to create a *Reo* connector and to perform the communication to pass a sequence of data values from some writers to some readers. Each of the writers in our experiments produce exactly four small strings. For comparison, we use the constraint automata-based implementation of *Reo* [7], which we call the *CA* engine. To the best of our knowledge this is the most complete and well supported *Reo* engine, and no other distributed engine for *Reo* exists. Our main concern is the performance of a connector that evolves in time, acknowledging the time to deploy a connector and the time to send data through the connector. We present a generalisation of our running example of the sequencer connector, and two simpler examples with best- and worse-scenarios for each engine. More details about the two engines used in our benchmark follow below.

CA engine. The *CA* engine is a code generator and interpreter of *Reo* connectors included in the suite of Eclipse³ plug-ins for *Reo* [6].⁴ It uses a context independent semantics of *Reo*, following the ideas of port automata [18] extended with memory and data transfer functions.

Dreams engine. The *Dreams* engine implements the framework using the behavioural automata of *Reo* primitives and nodes. We use the constraint-based approach with context dependency [14]. In this benchmark we consider the extreme (and less efficient) case where every node, channel, and component is deployed as an independent actor. The *Dreams* engine uses specific data constraints that describe how data should be routed within a channel or node, similar to the *CA* engine.

By associating each node, channel, and component to an independent actor we emphasise the differences between the *CA* and the *Dreams* engines. More optimal scenarios for the *Dreams* engine would deploy one actor for a group of connected *Reo* primitives, using their combined behavioural automaton. For example, we can group actors from the same synchronous region, as we will discuss later in this section.

We present three test cases. The first is a chain of n synchronous channels, *Syncs*, the second a chain of $FIFO_1$ channels, *Fifos*, and the third is our running example of the sequencer connector generalised to a sequencer with n $FIFO_1$ channels, *Seq*. Note that the sequence of synchronous channels is the optimal scenario for the *CA* engine and the worst

scenario for the *Dreams* engine, due to its stateless nature. In turn, the sequence of $FIFO_1$ channels is the optimal and worst scenarios for the *Dreams* and *CA* engines, respectively. The third test case is a more realistic, although still simple example, that favours the *Dreams* engine due to the presence of synchronous regions.

Results

All benchmarks were executed on a PC with an Intel[®] Core 2 Quad CPU Q9550 processor at 2.83GHz and with 7.8GB of RAM, running Fedora release 10. For each value of n , we performed 10 different executions and used the averages of the measurement values. For each test case we evaluate two different aspects, the time to *build* a connector and the time to *exchange* data.

Build time. The creation of the connector is performed once, after which the connector can be executed multiple times. In the *CA* engine this comprises joining the automata representations of all channels and nodes, deploying a centralised engine, and connecting the engine to the components. In the *Dreams* engine, creating a connector consists of deploying each actor and establishing connections between them. Note that measuring the build time in the *Dreams* engine requires the use of a variable shared by all actors that is locked whenever updated, introducing an unfair handicap necessitated only by our need for measurement.

Exchange time. After a connector is deployed and connected, the exchange time consists of the time required to exchange a sequence of messages between the components until no more data can be exchanged. The *exchange time* is calculated in both engines by measuring the time when the first message is sent, and the time when the last message is processed.

The results for our small benchmark are presented in Figure 4. In the *CA* engine, the sequence of synchronous channels achieves its best results, with a linear growth of the build time, taking less than 3 seconds to compose 3000 channels with constant send time. The constant time is easily explained by the fact that the composition of the automata of two *Sync* channels is again the automaton of a *Sync* channel. The main problem is shown in the build times of the $FIFO_1$ channels in the *Fifos* connector. An automaton with 6 $FIFO_1$ channels already takes around 35 seconds to generate. An automaton resulting from the composition of 6 $FIFO_1$'s (without hiding any intermediate node) has $2^6 = 64$ states and 274 transitions, increasing the time to compose the automata exponentially.⁵ Note that, in the case of the sequencer, once the loop of n $FIFO_1$ channels is closed, the number of states drops to n .

Discussion

The main conclusion of our benchmarks is that the centralised engine cannot handle connectors even with a relatively small number of $FIFO_1$ channels. The *CA* engine builds the entire state space before perming any of the transitions, while the *Dreams* engine simply calculates how each pair of $FIFO_1$'s communicate in runtime. This gives the

⁵Since our evaluation several optimisations have been made to the *CA* engine, improving its performance, but not affecting our conclusions.

²reoproject.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools

³www.eclipse.org

⁴reoproject.cwi.nl/cgi-bin/trac.cgi/reo/browser/ea-codegen

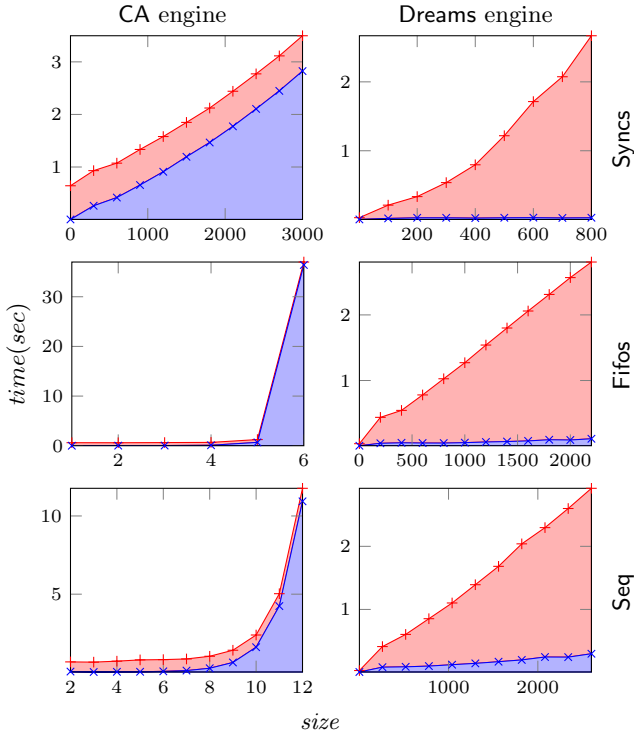


Figure 4: Accumulative graph with results of the evaluation of the build and send times for the Syncs, Fifos and Seq connectors, using the CA and Dreams engines. The top part represents the exchange time, and the bottom part represents the build time.

Dreams engine a huge advantage over connectors that perform asynchronous communication. Furthermore, the Dreams engine takes very little time to build and deploy its components, making it more suitable for systems that are frequently reconfigured. Observe also that reconfiguration of part of a connector does not affect the runtime or even concurrent execution of the rest of the connector, because of their decoupled execution.

Centralised implementations have an advantage over distributed implementations with respect to the runtime coordination overhead. For some scenarios, the lower overhead is more relevant than considerations for scalability or easy reconfiguration. This suggests that some complex scenarios may benefit from a *hybrid* deployment, where some parts of a connector can be compiled using a centralised approach (e.g., constraint automata), and deployed in a different location as an actor. Ideally, a connector would be partitioned automatically into its synchronous regions by an automatic splitting mechanism, and each region would be compiled separately to run as a single actor. Note that although we call this an *ideal scenario*, the developer of a system may still desire to force a single synchronous region to be distributed across a network, for example, because of hardware or software requirements.

5. RELATED WORK

We now compare Dreams with other approaches to implement *Reo*, we refer to some of the most popular formalisms used in industry, and we then look at some other languages

and tools that involve distributed coordination.

We distinguish five existing implementation approaches for *Reo*, plus our approach. The *speculative* approach consists of trying to send data through the channels and rolling back when an inconsistency arises. The *automata-based* approach [20] pre-computes all future behaviour at compile time. Implementations based on *connector colouring* [13] compute all solutions for the behaviour of each round, described as colouring tables, and deal with data transfer orthogonally. Existing *search-based* implementations are based on SOS models and implemented either in Maude or in Alloy [22, 17], and can also be based on the Tile models [5]. Finally, the *constraint-based* approach [14] utilises SAT solving techniques to search for single solutions in each round.

The speculative approach is the only approach prior to Dreams that aims to achieve a distributed implementation of *Reo*. However, this approach was never successfully implemented. Hence, Dreams is the first successful implementation of a distributed engine for *Reo*, and is based on a *commit and send* approach.

The industry standard for coordination of web services, the business process execution language (BPEL) [10], is a block-structured language that uses a centralised execution model, much like the automata-based implementations of *Reo*. *Reo* has also been used for the composition of web services. The mashup environment SABRE [20] built using *Reo*, provides tools to combine, filter and transform web services and data sources like RSS and ATOM feeds.

The two phase commit protocol, as well as some of its variants, is well known in the context of fault tolerance in distributed systems. Usually a *centralised* coordinator is involved that exchanges asynchronous messages with a set of participants. The protocol checks if all participants agree to perform some transaction, or if there is any that aborts, and communicates this decision back to the participants. The *distributed two phase commit (d2pc)* protocol [11], introduced by Bruni *et al.* as an extension of the 2pc protocol, can be compared to the distributed agreement protocol in the Dreams framework. Both approaches try to achieve a *global consensus*, which consists of a commit or abort in the case of the d2pc protocol and a description of a step in the case of Dreams. Their work served as inspiration for the development of the Dreams framework. Baragati *et al.* developed a prototype application [8] for two different platforms based on the d2pc protocol, using as a case study a rescue unit composed of a central base and several teams. Experiments in the Dreams framework involving real case studies have not been performed yet.

Minsky and Ungureanu developed the Law-Governed Interaction (LGI) mechanism [21], implemented by the Moses toolkit. This mechanism coordinates distributed heterogeneous agents, using a policy that enforces extensible laws. Agents execute events that are regulated by some controllers that enforce the laws. Laws are specified in a Prolog-like language, but as opposed to *Reo*, they reflect local properties only and do not require non-local synchronisation. The authors emphasise the need to replace a centralised controller imposing the laws of the full system by certified controllers, one for each connection. As with the 2pc protocol, this reflects a need to decentralise coordination, which is the main concern of the Dreams framework.

6. CONCLUSIONS

This paper introduces *Dreams*, a framework that provides a distributed implementation based on the Actor Model for coordination models that can be encoded with behavioural automata, and how it is used in the context of *Reo*. We summarise below our analysis of how the *Dreams* framework meets the goals set out in the above.

Decoupling The basic building blocks of the *Dreams* framework are actors, which execute concurrently. *Dreams* takes advantage of this concurrency by identifying parts of the connector that can be executed independently. We call these parts synchronous regions. We propose a simple approach to exploit synchronous regions, yielding a true decoupling of execution.

Scalability As a consequence of decoupling the execution of the instances of synchronous models such as *Reo* connectors, no global consensus is required. Furthermore, the behaviour is computed per step, avoiding the state space explosion that would result from computing all possible future behaviour. These two factors form the basis for a scalable implementation. The implementation of *Dreams* also allows different parts of a connector to execute across physical machine boundaries.

Reconfiguration The single-step semantics provides a very low deployment overhead, reducing the cost of reconfiguration. Furthermore, reconfiguring a connector affects only the synchronous regions that it modifies, while the rest of the connector continues to execute.

Currently we have a functioning implementation of the *Dreams* framework. The distributed implementation benefits from specific implementation optimisations offered by centralised schemes, in particular the compilation of connectors into the automata of its synchronous regions. The details of the implementation are described elsewhere [23].

7. REFERENCES

- [1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. The MIT Press, 1986.
- [2] G. Agha and P. Thati. An algebraic theory of actors and its application to a simple object-based language. In O. Owe, S. Krogdahl, and T. Lyche, editors, *Essays in Memory of Ole-Johan Dahl*, pages 26–57. LNCS 2635, 2004.
- [3] F. Arbab. Coordination of mobile components. *Electronic Notes in Theoretical Computer Science*, 54:1–16, 2001.
- [4] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
- [5] F. Arbab, R. Bruni, D. Clarke, I. Lanese, and U. Montanari. Tiles for Reo. In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques*, pages 37–55. LNCS 5486, 2009.
- [6] F. Arbab, C. Koehler, Z. Maraïkar, Y.-J. Moon, and J. Proença. Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. In *Proceedings of FACS*, 2008.
- [7] C. Baier, M. Sirjani, F. Arbab, and J. J. M. M. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.
- [8] A. Baragatti, R. Bruni, H. Melgratti, U. Montanari, and G. Spagnolo. Prototype platforms for distributed agreements. *Electronic Notes in Theoretical Computer Science*, 180(2):21–40, 2007.
- [9] G. Berry. The foundations of Esterel. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction*, pages 425–454. The MIT Press, 2000.
- [10] BPEL4WS. *Business Process Execution Language for Web Services*, May 2003.
- [11] R. Bruni, C. Laneve, and U. Montanari. Orchestrating transactions in Join calculus. In L. Brim, P. Jancar, M. Kretínský, and A. Kucera, editors, *CONCUR*, pages 321–337. LNCS 2421, 2002.
- [12] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, 1984.
- [13] D. Clarke, D. Costa, and F. Arbab. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming*, 66(3):205–225, 2007.
- [14] D. Clarke, J. Proença, A. Lazovik, and F. Arbab. Channel-based coordination via constraint satisfaction. *Science of Computer Programming*, 76, 2011.
- [15] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.
- [16] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.
- [17] R. Khosravi, M. Sirjani, N. Asoudeh, S. Sahebi, and H. Iravanchi. Modeling and analysis of Reo connectors using Alloy. In D. Lea and G. Zavattaro, editors, *COORDINATION*, volume 5052, pages 169–183. LNCS 5052, 2008.
- [18] C. Koehler and D. Clarke. Decomposing port automata. In *Proc. SAC '09*, pages 1369–1373, New York, NY, USA, 2009. ACM.
- [19] C. Krause. *Reconfigurable component connectors*. PhD thesis, LIACS, Faculty of Mathematics and Natural Sciences, Leiden University, 2011.
- [20] Z. Maraïkar, A. Lazovik, and F. Arbab. Building mashups for the enterprise with SABRE. In A. Bouguettaya, I. Krüger, and T. Margaria, editors, *ICSOC*, pages 70–83. LNCS 5364, 2008.
- [21] N. H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, 2000.
- [22] M. Mousavi, M. Sirjani, and F. Arbab. Formal semantics and analysis of component connectors in Reo. *Electronic Notes in Theoretical Computer Science*, 154(1):83–99, 2006.
- [23] J. Proença. *Synchronous Coordination of Distributed Components*. PhD thesis, LIACS, Faculty of Mathematics and Natural Sciences, Leiden University, 2011.
- [24] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed Parallel Databases*, 14(1):5–51, 2003.