Automatic Distributed Code Generation from Formal Models of Asynchronous Processes Interacting by Multiway Rendezvous

Hugues Evrard, Frédéric Lang

Inria
Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
CNRS, LIG, F-38000 Grenoble, France

Abstract

Formal process languages inheriting the concurrency and communication features of process algebras are convenient formalisms to model distributed applications, especially when they are equipped with formal verification tools (e.g., model checkers) to help hunting for bugs early in the development process. However, even starting from a fully verified formal model, bugs are likely to be introduced while translating (generally by hand) the concurrent model —which relies on high-level and expressive communication primitives— into the distributed implementation —which often relies on low-level communication primitives. In this paper, we present DLC, a compiler that enables distributed code to be generated from models written in a formal process language called LNT, which is equipped with a rich verification toolbox named CADP, and where processes interact by value-passing multiway rendezvous. The generated code uses an elaborate protocol to implement rendezvous, and can be either executed in an autonomous way (i.e., without requiring additional code to be defined by the user), or connected to external software through user-modifiable C functions. The protocol itself is modeled in LNT and verified using CADP. We present several experiments assessing the performance of DLC, including the Raft consensus algorithm.

Keywords: Multiway Rendezvous, Compilation, Process Algebras, Distributed Systems

Email addresses: `Hugues.Evrard@inria.fr` (Hugues Evrard), `Frederic.Lang@inria.fr` (Frédéric Lang)

1. Introduction

Distributed systems often consist of several concurrent processes, which interact to achieve a global goal. Programming concurrent and interacting processes is recognized as complex and error-prone. One way to detect bugs early is to (a) produce a model of the system in a language with well-defined semantics, and to (b) use formal verification methods (e.g., model checking) to hunt for bugs in the model. However, formal models of distributed systems must eventually be translated into a distributed implementation. If this translation is done by hand then semantic discrepancies may appear between the model and the final implementation, possibly leading to bugs. In order to avoid such discrepancies, an automatic translator, i.e., a compiler, can be used.

Such a compiler takes a formal model as input and generates a runnable program, which behaves according to the model semantics. In the case of distributed systems, we want to produce several programs, which can be executed on distinct machines, from a single model of a distributed system. We identified several challenges related to this kind of compilation.

First, formal models generally rely on concurrency theory operators to express complex interactions between processes, whereas implementation languages often offer only low-level communication primitives. Hence, the complex interactions have to be implemented by non-trivial protocols built upon the low-level primitives, which may be hard to master by (even experienced) programmers. As a brief example, the synchronization of n distributed processes may be expressed by a single rendezvous primitive (high-level), while it requires a protocol between the n processes when only message passing primitives (low-level) are available. For any process interaction specified in the high-level model, the compiler must be able to automatically instantiate such protocols in the generated code.

Second, the generated programs should be able to interact with their environment. Such interactions are often abstracted away in the formal models, while a real interaction is required in the final implementation. For instance, consider a distributed system where some process deals with a database. In the formal model, the database may be abstracted away by read and write operations. However, we want the implementation of these processes to actually connect to an external database which is developed independently from the distributed system under study. The compiler should provide a mechanism to define interactions with the external environment and embed them in the final implementation.

Third, the generated implementation must take benefit of the distributed nature of the system to achieve reasonable performances for rapid prototyping. Performance not only depends on the speed of each process, but also on how process interaction is implemented. Naive implementations can lead to very inefficient executables, due to unforeseen bottlenecks. For instance, a compiler implementing a naive protocol that consists in acquiring a unique global lock to proceed on process interaction would be extremely inefficient as processes would mostly waste time waiting for the lock while they often could safely execute concurrently. An efficient and decentralized protocol is therefore required to enable decent execution times. Even though the aim is not to compete with hand-crafted optimized implementations, a too important performance penalty would make the rapid prototyping approach irrelevant.

In this paper, we consider models written in LNT [12], a process language with formal semantics. LNT combines a user-friendly syntax, close to mainstream imperative languages, together with communication and concurrency features inherited from process algebras, in particular the languages LOTOS [31] and E-LOTOS [32]. Its semantics are formally defined in terms of an LTS (*Labeled Transition System*): the observable events of an LNT process are *actions* (possibly parametrized with data) on *gates* (which represent ports of interaction between processes, and also with the environment), which label the transitions between states of the process.

LNT models can be formally verified using software tools available in the CADP[1] (*Construction and Analysis of Distributed Processes*) [26] tool box, which provides simulation, model checking, and test generation tools, among others.

LNT enables a high-level description of nondeterministic concurrent processes that run asynchronously (i.e., at independent speeds, as opposed to synchronous processes driven by a global clock), and that interact by *value-passing rendezvous* (or *synchronization*) on actions. The value-passing rendezvous mechanism of LNT is expressive and general:

- A rendezvous may involve any number of processes (*multiway* ren-

[1] `http://cadp.inria.fr`

dezvous), i.e., it is not restricted to binary synchronizations. LNT even features n-among-m synchronization [27], in which a rendezvous may involve any subset of n processes out of a larger set of m.

- Due to nondeterminism (select statement), every process may be ready for several actions at the same time. Different rendezvous may thus involve one or more common processes, in which case we say that the rendezvous are *conflicting*. Therefore, for a rendezvous between processes to occur actually, it is not enough that all processes are ready; they must also all simultaneously agree to take that rendezvous instead of conflicting ones.

- Processes may exchange data during the rendezvous (*value-passing* rendezvous). Each data exchange may involve an arbitrary number of senders and receivers, and a given process may simultaneously send and receive different pieces of data during the same rendezvous.

The research problem we tackle here is how to automatically generate a distributed implementation from an LNT model of a distributed system. To our knowledge, there does not exist an automatic distributed code generation tool for a formal language that not only features such a general rendezvous mechanism, but is also equipped with powerful verification tools. We introduce DLC[2] (*Distributed LNT Compiler*), a new tool that achieves automatic generation of a distributed implementation in C from an LNT model. We focus on LNT since we think its roots in process algebra offer a well-grounded basis for formal study of concurrent systems [22], and because it is already equipped with the numerous verification features of our team's toolbox CADP, which however still lacks distributed rapid prototyping. Nonetheless, our approach should be relevant to any language whose inter-process communication and synchronization primitive is value-passing multiway rendezvous. DLC meets the three challenges stated earlier:

- DLC transforms each concurrent process of the distributed system model into a sequential program, and instantiates an elaborate protocol to handle value-passing multiway rendezvous. We designed a

[2] `http://hevrard.org/DLC`

rendezvous protocol that combines ideas from the literature into an efficient solution, that we formally verified. The generated programs can run on several distinct machines.

- Interactions with the external environment are made possible through calls to user-defined external procedures. With DLC, the user can define *hook functions* that are integrated in the final implementation and called upon actions in the system. Hook functions are written in C, and they provide a convenient way to interact with other systems.

- DLC generates programs with reasonable performances, which qualify for rapid prototyping. Although generated programs execution speed may not be on par with an implementation in a classic programming language, DLC makes it possible to easily produce a validated prototype, which can be deployed and run on a cluster, from a distributed system modeled and verified using LNT and CADP.

We provide a formal model, written in LNT, of the multiway rendezvous protocol used by DLC. This model has been verified using CADP, following the approach depicted in [20]. The protocol model and its verification approach were developed before the compiler. To obtain the protocol eventually used by DLC, we started from the protocol proposed by Parrow and Sjödin [53], and we iteratively brought several enhancements to make it more general, in order to handle LNT synchronizations, and also more efficient, for better performances. At each step of this iteration, we relied on our verification approach to check that the protocol remained correct. Henceforth, we have a high confidence on the protocol correctness.

This paper is structured as follows: Section 2 explores related work. Section 3 illustrates how we can model a distributed system in LNT. Section 4 details the multiway rendezvous protocol, and Section 5 covers how hook functions enable interactions with the external environment. Section 6 exposes how a distributed implementation is automatically generated. Section 7 presents experimental results, including a non-trivial application, the Raft [51] consensus algorithm. Section 8 concludes and suggests future work.

2. Related Work

Several programming languages offer useful primitives or libraries for interaction between *distant* processes, i.e., processes on separate machines connected by a network. The most common mechanisms are: message passing, where processes can send messages to each other, e.g., POSIX sockets in C, or Erlang built-in messaging; and RPC (*Remote Procedure Call*), where a process can invoke a procedure executed by another distant process, e.g, Java RMI (*Remote Method Invocation*), or the "net/rpc" package of the Golang[3] standard library. However, we are not aware of a library for popular programming languages that would implement LNT-like value-passing multiway rendezvous.

Modeling Languages Equipped with both Formal Verification and Code Generation Tools

The formal study of concurrent processes is a rich field of research, and several formalisms exist to model such systems. For synchronous models, where all processes share a unique clock, a good illustration is the Esterel language, which comes with a suite of verification tools and compilers [7].

As regards asynchronous systems, i.e., the domain in which lies the language LNT, the Topo [42] tool set for LOTOS features code generation in either C or Ada, and enables environment interactions via LOTOS annotations. However, the generated implementation is sequential, and Topo is not maintained anymore. LOTOS is also the historical formal language of CADP, which provides the EXEC/CÆSAR [28] tool to generate C code with interface functions that must be user-defined. Once again, this code is sequential, and our DLC tool builds upon EXEC/CÆSAR (which also accepts LNT as input) for generating the code corresponding to sequential processes. UPPAAL [4] provides a framework to operate on networks of timed automata, including formal verification tools. The associated Times tool [1] generates C code from UPPAAL models, but the final program is sequential.

In the framework of SPIN [30], Promela is a modeling language which uses channels rather than multiway rendezvous for process interactions. A Promela to distributed C compiler has been proposed [41], relying on a client-server approach, still the user must explicitly specify by hand which process is server or client. More recently, a refinement calculus to obtain C from Promela has been presented [59], but this time the generated code is not distributed.

The Chor [11] language enables programming of distributed systems as choreographies, and has verification features based on behavorial types. Chor

[3] Golang is a programming language made public in 2009, see `https://golang.org`

adopts a "correct-by-construction" approach, by checking for instance deadlock freedom at the choreography level, and providing automated generation of distributed implementations. The Chor authors also study composition of choreographies [46], which is a desirable feature in "correct-by-construction" approaches. Another choreographic language with tool support is Scribble[4], which has recently been extended to parametrized protocols in Pabble [48] and relies on parametrized session types for verification features. Still, neither Chor nor Pabble offer value-passing multiway rendezvous as a primitive, since in these languages, processes interact through message passing.

The BIP framework describes a system in three layers: Behaviors, Interactions and Priorities. Interactions between behaviors correspond to value-passing multiway synchronizations. In addition, priorities may differentiate interactions: when several interactions are possible, the one with highest priority must occur, preempting others (when interaction have the same priority, any of them may occur). To our knowledge, BIP verification features are now limited to a deadlock detection tool [5], while CADP offers several model checkers [43, 44, 45], equivalence checkers [6], tools for compositional verification [23, 38, 25], test case generation [33], performance evaluation [14], and even more[5]. Nonetheless, a distributed code generation tool is available for BIP [9]; it instantiates a multiway rendezvous protocol to handle interaction in a distributed way—the protocol presented in this paper improves over the one used in BIP. BIP priorities, which is not a built-in concept in LNT, is handled in the rendezvous protocol by requiring a centralized knowledge to resolves them, thus limiting the parallel execution of the generated implementation.

A recent paper [17] estabishes a formal relation between BI(P) (i.e., BIP without the priority layer) and the Reo [2] coordination language, thus paving the way to interoperability between their tools. Besides, the Dreams [56] framework provides a methodology to generate, from Reo programs, distributed applications running on Java Virtual Machines.

Both BIP and Reo distributed code generators create a program for each process present in the formal specification, and also extra programs required to implement interactions between the specification processes. When running on a cluster of machines, one must decide how to *partition*, i.e., dispatch, all

[4]http://www.scribble.org/

[5]For an overview of CADP tools, see `http://cadp.inria.fr/tools.html`

these programs on the available nodes. This seems to be a non-trivial problem: the BIP distributed code generator requires the end user to explicitely provide this partition, while specific techniques [34] are needed in Reo. The parallel composition operator of LNT provides a, if not optimal, at least relevant partition of the generated programs, such that the end user does not have to think about partitioning.

Distributed Implementation of Multiway Rendezvous

Since the process interaction mechanism is a key challenge in a distributed system, we also briefly review protocols that implement the multiway rendezvous in a distributed manner. As soon as 1983, works on the distributed implementation of Petri nets lead to propositions [64, 63]. Each transition of a Petri net can be considered as a rendezvous between its preceding places, and transitions are in conflict when they share common preceding places. To ensure the mutual exclusion of transitions in conflict, a transition must lock a token in each preceding place. There are several approaches to avoid deadlocks during this locking phase: either elect a winner among transitions that lock the same tokens [64], or always lock the tokens in the same order [29, 63].

Multiway rendezvous can be considered as a variation of the committee coordination problem, stated by Chandy and Misra [13], where professors (processes) must schedule committee meetings (rendezvous), with every professor being a member of several committees. Bagrodia [3] lists classical solutions to this problem and presents the event manager algorithm, based on a token ring approach, which is also explored by Kumar [36].

At the same period, various studies on the distributed implementation of LOTOS led to several protocol proposals [8, 61, 62, 53], and a protocol based on ordered broadcast was later designed [65]. In a previous study [20], we used LNT and CADP to model and verify three protocols, and we spotted previously undetected deadlocks, under asynchronous communication hypothesis, in the one designed by Parrow and Sjödin [53]. The current work is based on a corrected version we suggested and on which we verified the absence of deadlocks.

Out of the LOTOS context, Pérez *et al.* [54] presented the "α-core" protocol, but the original specification contains a bug documented by Katz and Peled [35]. More recently, work on the hardware implementation of CSP programs required the design of a protocol [49], which however imposes a restriction on the number of processes that can send data during an interaction. Theoretical studies on the encoding of interactions in the π-calculus also refer to rendezvous implementation techniques [47, 55]. All the works presented in [8, 61, 62, 53, 65, 54] focus on the protocol rather than on the compiler implementation.

At last, this paper comes after a series of other papers that are directly related with DLC. A first paper [20], already mentioned above, deals with formal verification of rendezvous protocols using CADP. A second paper [21], of which the current paper is an extended version, presents the implementation of the protocol into the DLC tool. The extension mainly consists of a new section that provides details about the multiway rendezvous protocol, and an appendix containing the LNT formal model of the protocol. Moreover, the related work section has been enriched, and we present additional experiments to assess performance of the generated code. A third paper [19] demonstrates the usage of DLC on a pedagogical toy example. Finally, the PhD thesis of the first author [18] (in French) presents a comprehensive description of the protocol, its verification, the DLC tool, and case studies achieved using DLC.

3. Modeling Distributed Systems in LNT

LNT provides several levels of abstraction and structuration, namely modules, types, functions, and processes. We consider distributed systems to be composed of several *tasks*, which interact with each others. The behaviour of each task is defined by an LNT process and the interactions between tasks are described by parallel composition of the corresponding processes, synchronized by value-passing multiway rendezvous on gates.

We give an informal introduction to LNT using an example; for a formal and full definition of LNT syntax and semantics, see [12]. We model a simplified version of the leader election phase of the Raft [51] consensus algorithm, which consists of a set of servers that have to elect a leader among them. The servers either run correctly or they crash and terminate (as opposed to erratic "Byzantine" behaviors). Since the leader can crash, several elections may happen as time goes by. Time is divided in *terms*, each server maintaining a term index, which increases monotonically. A term represents a logical period of time during which at most one leader may emerge from the group of servers, and it is also possible that no leader is elected during a term before the next is started.

In each term, servers may be in either *follower*, *candidate* or *leader* state. All servers start as followers, then some of them eventually become candidate after a timeout. A candidate increases its term index, votes for itself and asks other servers for their vote. A server grants its vote only if its term is equal to the candidate one and if it has not voted for someone else earlier in the current term. When a candidate has received a majority of votes, it becomes the leader for this term. Whenever servers communicate, they provide their current term, and when a server receives a term higher than its own, it updates its own term and resigns to the follower state. Moreover, servers may crash and stop. In the context of Raft, the leader election is more elaborate, e.g., the leader prevents timeouts of other servers with a heartbeat mechanism; we do not model these features here for the sake of brevity.

Figure 1 illustrates the LNT model of a server. LNT syntax is close to mainstream implementation languages, and most code should be understandable for someone with a programming background. After initialization, a server enters its main loop where the nondeterministic choice operator `select`, reminiscent of Dijkstra [16], is used to enumerate several possible behaviors, separated by "`[]`". The server will execute one branch of the select operator, depending on its current state and the possible actions in the system.

The observable events of an LNT process are actions on gates; gates are declared between the square brackets in the process header. For instance, a server indicates that it performs a timeout or a crash, or announces its leadership with an action on either gates `TIMEOUT`, `CRASH` or `LEADER`, respectively. Actions on these three gates are used to make the related events observable from the environment, they are not used to synchronize servers (any server can make an action on one of these three gates on its own). Servers deal votes through an abstracted RPC mechanism: a request for vote is queried by an action on `RVOTE` (lines 43 and 61), followed by an answer on `AVOTE` (lines 54 and 62). Actions on these two gates will synchronize two servers to enable communication between them.

A process can send or receive data using *data offers* on an action. Each data offer may have one of two forms: either a value-expression (optionally preceded by the symbol "**!**"), corresponding to the emission of the corresponding data value; or a variable preceded by the symbol "**?**", corresponding to the reception of a data value, which is stored in the variable. For instance, a server sends its identifier and its current term when it announces its leadership on `LEADER` (line 72) and when a server is requested for vote on `RVOTE`, the caller identifier is stored in the `rpcId` variable (line 43) that is

```
1 -- Data types
2 type state is follower, candidate, leader end type
3 type abool is array [0 .. 2] of bool end type
4
5 -- Global parameters (constants declared as functions)
6 function majority : nat is return 2 end function
7 function maxId : nat is return 3 end function
8 function maxTerm : nat is return 2 end function
9
10 function resign (out state: state, out votedId: abool,
11                  out voteCount: nat, out voted: bool) is
12   state     := follower;
13   votedId   := abool(false); (* set all array to false *)
14   voteCount := 0;
15   voted     := false
16 end function
17
18 process SERVER [LEADER,CRASH,TIMEOUT,RVOTE,AVOTE: any]
19            (selfId : nat) is
20 var state: state,
21     selfTerm, voteCount, rpcId, rpcTerm: nat,
22     votedId: abool,
23     voted, voteGranted: bool
24 in
25   (* initialization *)
26   selfTerm := 0;
27   eval resign(?state, ?votedId, ?voteCount, ?voted);
28   (* main loop *)
29   while selfTerm < maxTerm loop
30     select (* possible behaviors delimited by "[]" *)
31       (* timeout, become candidate *)
32       case state in
33         follower | candidate ->
34           TIMEOUT(selfId, selfTerm);
35           selfTerm          := selfTerm + 1;
36           votedId[selfId]   := true;
37           state             := candidate;
38           voteCount         := 1;
39           voted             := true
40       | leader -> stop (* leader cannot become candidate *)
41       end case
42     [] (* receive vote request *)
43       RVOTE(?rpcId, selfId, ?rpcTerm);
44       if rpcTerm > selfTerm then
45         selfTerm := rpcTerm;
46         eval resign(?state, ?votedId, ?voteCount, ?voted)
47       end if;
48       if (selfTerm == rpcTerm) and (not(voted)) then
49         voteGranted := true;
50         voted       := true
51       else
52         voteGranted := false
53       end if;
54       AVOTE(selfId, rpcId, selfTerm, voteGranted)
55     [] (* send vote request *)
56       case state in
57         candidate ->
58           rpcId := any nat where rpcId < maxId;
59           (* Don't send request if rpcId already voted *)
60           if (votedId[rpcId]) then stop end if;
61           RVOTE(selfId, rpcId, selfTerm);
62           AVOTE(rpcId, selfId, ?rpcTerm, ?voteGranted);
63           if rpcTerm > selfTerm then
64             selfTerm := rpcTerm;
65             eval resign(?state, ?votedId, ?voteCount, ?voted)
66           else
67             votedId[rpcId] := true;
68             if voteGranted then
69               voteCount := voteCount + 1;
70               if voteCount >= majority then
71                 state := leader;
72                 LEADER(selfId, selfTerm)
73               end if
74             end if
75           end if
76       | follower | leader -> stop (* do not request vote *)
77       end case
78     [] (* fail stop *)
79       CRASH(selfId, selfTerm); stop (* server halts *)
80     end select
81   end loop
82 end var
83 end process
```

Figure 1: LNT specification of a server for the leader election algorithm.

used later in the answer action on `AVOTE` (line 54). Note that both emission and reception data offers may occur mixed on the same gate (see e.g., action `AVOTE` at line 62), and that a rendezvous may involve an arbitrary number of senders and receivers. LNT follows the *value-matching* semantics adopted by process algebras such as LOTOS and CSP, in which a condition for a rendezvous to take place is that the values taken by the data offers match (similarly to pattern-matching) during rendezvous.

Figure 2 illustrates a parallel composition of servers. The `par` operator defines which processes must synchronize on which gates. Here for example, we use n-among-m synchronization to indicate that processes must synchronize by pair ($n = 2$) on gates `RVOTE` and `AVOTE`. Thus, an action on one of these two gates consists of a binary rendezvous of two processes with data

```
par RVOTE #2, AVOTE #2 in
   SERVER [LEADER, CRASH, TIMEOUT, RVOTE, AVOTE] (0 of nat)
|| SERVER [LEADER, CRASH, TIMEOUT, RVOTE, AVOTE] (1 of nat)
|| SERVER [LEADER, CRASH, TIMEOUT, RVOTE, AVOTE] (2 of nat)
end par
```

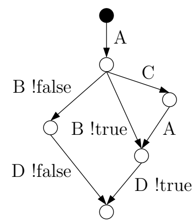
Figure 2: Parallel composition of server processes. "#2" indicates that actions on gates RVOTE and AVOTE must involve two processes among the three servers (n-among-m synchronization, where $n = 2$ and $m = 3$).

exchange. By default, actions on other gates only involve one process, i.e., they are not synchronized. Although not illustrated here, it is also possible to indicate, for each process, the list of gates it must synchronize on. Together with n-among-m synchronization and the possibility of nesting `par` operators, we can model complex interactions between an arbitrary number of processes. The possible interactions defined by a parallel composition can be represented internally with *synchronization vectors* [38] that denote, for each gate, which tuples of processes must synchronize their action. For instance, if we denote by S_0, S_1 and S_2 the three servers, the synchronization vectors for gate `LEADER` (and also `CRASH` and `TIMEOUT`) are $\{S_0\}$, $\{S_1\}$ and $\{S_2\}$; the ones for gate `RVOTE` (and also `AVOTE`) are $\{S_0, S_1\}$, $\{S_0, S_2\}$ and $\{S_1, S_2\}$. We say that two synchronization vectors (and the corresponding transitions in a given state) are *conflicting* if the intersection between their synchronization vectors is not empty (i.e., they have at least one task in common).

In this example of distributed system, servers represent task processes and possible interactions between tasks are set by the parallel composition. Before we dig into how we generate a distributed implementation from such a model, we briefly illustrate how formal verifications can be applied to it.

LNT semantics are defined formally in terms of an LTS (*Labeled Transition System*). Formally, an LTS is defined as a tuple (S, A, T, s_0) where S is the set of states, s_0 the initial state, A the set of observable events, called *actions*, and $T \subseteq S \times A \times S$ the transition relation between process states, labeled by actions. Non-observable (a.k.a. *hidden*) events can be modeled using a particular action written τ. To any LNT process corresponds an LTS whose observable actions consist of the gate name, followed by the exchanged data values (if any). When building the LTS, each state is built from the vector of variable values and control state of the LNT process. However, the state contents are dropped once the LTS construction is complete, and we consider LTSs modulo the strong bisimulation equivalence[6] [52], which allows to merge LTS states which have the same future (e.g., all deadlock states may be merged into a unique deadlock state). For instance, here is a small LNT process and its corresponding LTS, where the initial state is marked by a black disc:

```
process foo [A,B,C,D: any] is
   var b : bool in
      A ;
      select
         B(?b)
      [] C ; b := true ; A
      end select ;
      D (b)
   end var
end process
```



The LTS represents the LNT model state space, i.e., all its possible execution paths. Since it may be huge, models are often parametrized and parameters are assigned at low values to control the state space explosion. For instance, the election algorithm is approximated to a smaller state space by bounding server terms with a predefined `maxTerm`.[7]

The CADP tools can be used to perform formal verifications, e.g., model checking, on the LTS representation, either on-the-fly or after complete state space generation. For instance, EVALUATOR4 [45] can be used to check the safety property "there are not two leaders in the same term" expressed as the following MCL (*Model Checking Language*) [45] formula:

```
[ true* . { LEADER ?id1:Nat ?t1:Nat } .
  true* . { LEADER ?id2:Nat ?t2:Nat where t1 = t2 } ] false
```

This formula states that there must not be consecutive leader announcements (gate `LEADER`) for the same term. Similarly, we can verify other properties such as "if less than a majority of servers have crashed or reach the maximum term, then a leader can be elected". The interested reader may take a

[6]In an LTS (S, A, T, s_0), two states $s, t \in S$ are *strongly bisimilar* if there exists a symmetric relation R on $S \times S$ such that $R(s,t)$ and for each s', t' such that $R(s', t')$, if there exists a transition $(s', a, s'') \in T$, then there exists a transition $(t', a, t'') \in T$ such that $R(s'', t'')$ (the converse also holds by the symmetry condition).

[7]In Raft, terms are unbounded and overflow is not addressed; with a timeout of 150 ms, terms stored on 32 (resp. 64) bits take, in the worst case, more than 20 (resp. 80 billion) years to overflow.

look at [26] to know more about formal verification using CADP, which also features equivalence checking, simulation, and many other tools.

4. Multiway Rendezvous Protocol

Multiway rendezvous requires a protocol in order to be implemented in a distributed way. This protocol defines how tasks, and possibly other auxiliary processes, communicate in order to decide which actions are realized by the system with respect to the possible rendezvous defined by the parallel composition of tasks. We make the assumption that processes communicate using asynchronous messages over a reliable network (no message loss), and that, from a process to another, messages are received in the order they are sent.

Among the protocols of the literature (see Section 2), we selected the one designed by Parrow and Sjödin [53] as a basis, since it is extensible to the general synchronizations of LNT, and it requires few messages to achieve a rendezvous. In the sequel, we briefly present this protocol and our formal verification approach. We then identify the *offset synchronization* phenomenon, enhance the protocol in various ways to simplify it and make it more efficient, and add the *autolock* optimization. In order to keep the protocol correct in the presence of both autolock and offset synchronizations, we also present the *purge* mechanism that we have designed.

4.1. Parrow and Sjödin Protocol

The protocol designed by Parrow and Sjödin defines two kinds of auxiliary processes: *managers* conduct rendezvous negotiations for tasks, and *gates* represent the gates of the system.[8] Each task is associated with a manager, and each gate is represented by a gate process. Table 1 lists the different types of messages exchanged between tasks, managers and gates.

We can distinguish three phases in the protocol:

Announce phase When a task is ready on one or more actions, it sends these actions to its manager through a *request* message. Then, the manager dispatches these ready announces to all relevant gates, with *ready* messages.

[8] In the original paper [53], managers and gates are called *mediators* and *ports*, respectively.

Type	Description
request	A task sends its possible actions to its manager
ready	A manager forwards possible actions of its task to a gate
query	A gate starts a negotiation by sending a lock request to the first manager of the synchronization vector
lock	A manager forwards the lock request to another manager
yes	A manager alerts a gate that the negotiation is successful
commit	A manager alerts a manager that the negotiation is successful
no	A manager alerts a gate that the negotiation has failed
abort	A manager alerts a manager that the negotiation has failed
confirm	A manager sends to its task which action must be realized

Table 1: The nine types of messages in Parrow and Sjödin protocol.

Locking phase When a gate detects that all tasks of its synchronization vector are ready, it starts a negotiation with task managers. A negotiation consists in trying to lock all managers of the tasks involved in the synchronization in order to ensure the exclusion with other potentially conflicting rendezvous. Managers are then similar to shared resources between gates, and the protocol uses the classic *ordered locking* technique [29] to avoid deadlocks. To enable this technique, all gates consider the same order defined on managers. A gate starts a negotiation by sending a lock request, using a *query* message, to the first manager of the synchronization vector. A manager accepts at most one lock at a time, and when it does so, it forwards the lock request to the next manager of the synchronization vector by sending a *lock* message. Managers involved in a synchronization thus form an ordered chain that is called a *lock chain*.

Result phase If the last manager of the synchronization vector receives and accepts the lock request, then the negotiation is a success. This manager sends a *yes* message to the gate that started the negotiation, and a *commit* message that is forwarded along managers of the synchronization vector, in reverse order of the lock chain. Moreover, each concerned manager sends a *confirm* message to its task, which realizes the selected action accordingly and continues its execution.

When a negotiation succeeds, each manager in the lock chain discards each of its pending lock requests (if any) by sending a *no* message to the relevant gate, and an *abort* message to the manager that sent the lock request. Like a *commit* message, an *abort* message is forwarded back along locked managers of the failed negotiation, which are released from their lock. A locked manager that is released by an *abort* message can accept a new or a pending lock request, and can thus participate to another negotiation.

We illustrate this protocol on the following example, where the parallel composition imposes that actions on gates A or B must be synchronized between tasks T1 and T2, while actions on gate C can be realized by task T2 alone. Therefore, the synchronization vector for both A and B is {T1, T2}, and the synchronization vector for C is {T2}.

```
process T1 [A, B: any] is
   select
      A
   [] B
   end select
end process
```

```
process T2 [A, B, C: any] is
   select
      A
   [] B
   [] C; B
   end select
end process
```

```
-- composition
par A, B in
   T1 [A, B]
|| T2 [A, B, C]
end par
```

Figure 3 illustrates a possible execution of the protocol, where managers of tasks T1 and T2 are labeled M1 and M2, respectively. At the start, task readiness is signaled with *request* and *ready* messages. When gate A detects that enough tasks are ready for an action, it starts a negotiation with a *query* message. So do gates B and C. The first query to reach manager M1 is the one from gate A; the manager then forwards the lock query to manager M2. Manager M1 also receives a query from gate B, and stores it as a pending lock request. Meanwhile, manager M2 has successfully negotiated an action on gate C for its task, which is now ready for an action on gate B, solely. Therefore, manager M2 refuses the lock request for gate A received from manager M1, and sends an *abort* and a *no* message accordingly. Upon reception of the *abort* message, manager M1 releases itself, then accepts and forwards the pending lock request related to gate B. Manager M2 accepts this lock request and replies to gate B and manager M1 with *yes* and *commit* messages, respectively. Both managers also send *confirm* messages to their tasks.

A noticeable feature of this protocol is that the locking scheme requires only one message per task to be locked. For a comparison, the α-core protocol [54] also relies on an ordered locking of tasks, but gates centralize lock

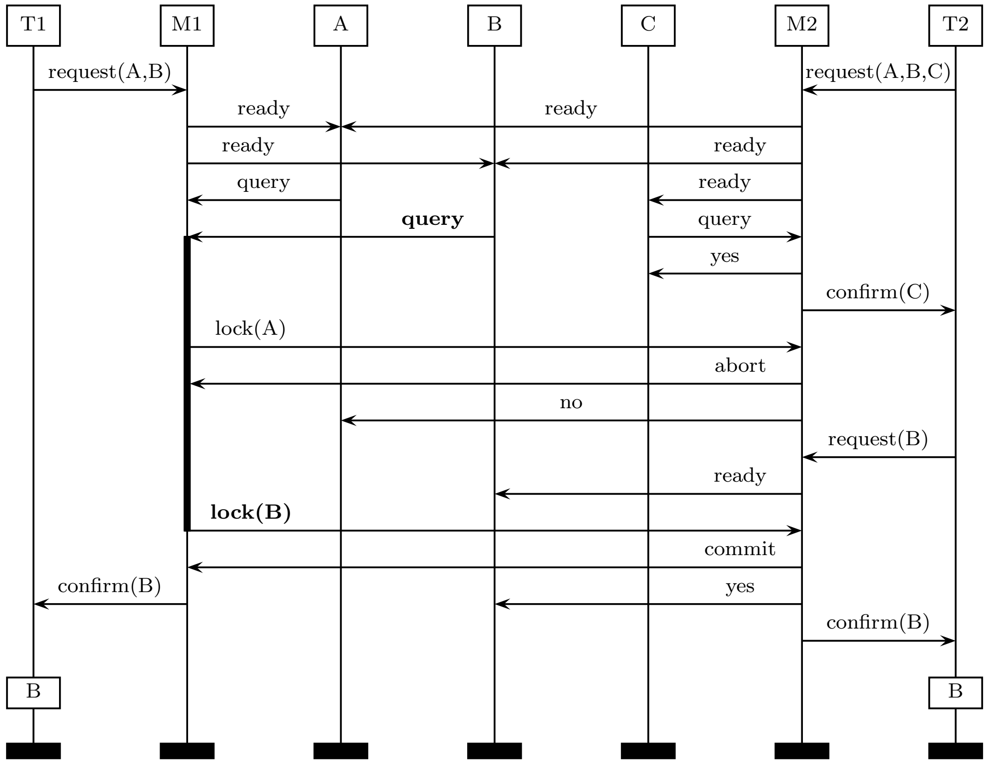


Figure 3: Illustration of Parrow and Sjödin protocol, the locking phase of an offset synchronization is bolded. The synchronization vector for both A and B is {T1, T2} and the synchronization vector for C is {T2}.

requests, hence the locking phase requires two messages per task. As illustrated in Figure 4, Parrow and Sjödin locking approach is more efficient.

The ordered locking technique may lead to overload of lower managers, which are likely to receive more lock requests than others. However, when a manager receives several lock requests while it is waiting on a negotiation answer, these lock requests correspond to negotiations for conflicting rendezvous. Lower managers act as filters for negotiations of conflicting rendezvous, by forwarding only one negotiation at a time to upper managers. Since only one of these negotiations will eventually succeed anyway, the ear-

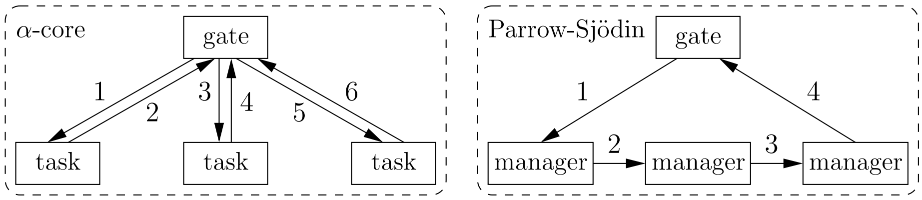


Figure 4: Parrow and Sjödin locking scheme requires less messages than the α-core one.

lier it is selected, the better. Therefore, the ordered locking technique enables the early selection of a negotiation among conflicting ones, while still allowing non-conflicting negotiations to occur in parallel since they lock different sets of managers.

Offset Synchronization

This protocol enables a particular phenomenon that we named *offset synchronization*. We expose this phenomenon since it appears in discussions on the correctness of the protocol.

In most protocols, when a rendezvous succeeds, then all negotiations dealing with conflicting rendezvous are aborted because the tasks that participated to the successful rendezvous have moved to a new state, whose set of ready actions may have changed. However, in Parrow and Sjödin protocol, a negotiation on a conflicting rendezvous may still succeed if the set of ready actions in the new states still contain the action concerned by the negotiation. The synchronization (which is valid) resulting from this negotiation is named *offset synchronization*, because there is an offset of some task state between the start of the negotiation and its ending. An offset synchronization can be seen as the result of a "short-cutting" negotiation, in the sense that the successful negotiation spans over a state update of at least one of the involved tasks, whereas in most protocols such state updates systematically invalidate ongoing negotiations.

This phenomenon is illustrated in Figure 3, where the bold path from messages *query* to *lock(B)* denotes such a negotiation. Gate B starts a negotiation by sending a *query* message to manager M1, in order to synchronize both tasks T1 and T2. Meanwhile, manager M2 concludes a negotiation for task T2, which realizes an action on gate C (message *confirm(C)* sent by M2 to T2) and reaches a new state —in which it is ready on gate B, again (message *request(B)* sent by T2 to M2). Therefore, when the negotiation started by gate B reaches manager M2 (message *lock(B)* sent by M1 to M2), this manager can accept it. Thus, task T2 has updated its state while the negotiation started by gate B was ongoing, and the negotiation still succeeds: the resulting rendezvous is an offset synchronization.

4.2. Protocol Correctness: Systematic Validation Approach

In order to gain confidence in the protocol correctness, we use the formal approach set up in our previous work [20]. In a nutshell, from the specification of a distributed system, we automatically generate the formal model of the system implementation, which includes the rendezvous protocol. In other words, from an LNT composition of tasks interacting by multiway rendezvous, we generate an LNT model of the implementation, which contains a model of tasks, managers, gates, and buffers for asynchronous message passing between processes, as illustrated in Appendix A.5. Using CADP, we then perform three formal verifications:

Livelock detection. We check in the implementation model that the protocol cannot conduct negotiations forever without reaching a result, i.e., there is no infinite loop of protocol messages without announces of a successful action.

Deadlock detection. We check in the implementation model that the protocol cannot get into a sink state before reaching an action, if any action is possible with respect to the specification.

Equivalence between specification and implementation. We check that the implementation model is behaviorally equivalent to the original system specification, with respect to an equivalence relation that abstracts away the actions of the protocol. To do so, we use safety equivalence[9] [10], the abstraction consisting in turning every action of the protocol into the invisible action τ. This guarantees that every action sequence of the model can be mimicked by the implementation.

[9]Two LTSs $(S_1, A_1, T_1, s_{(1,0)})$ and $(S_2, A_2, T_2, s_{(2,0)})$ are *safety equivalent* if their exists a $\tau^*.a$ preorder $\sqsubseteq$ on $(S_1 \times S_2) \cup (S_2 \times S_1)$ such that $s_{(0,1)} \sqsubseteq s_{(0,2)}$ and $s_{(0,2)} \sqsubseteq s_{(0,1)}$. A $\tau^*.a$ preorder is any relation that satisfies the following constraint: if $s \sqsubseteq t$ and s is the source of a (arbitrarily long, possibly null) sequence of transitions labeled by τ followed by a transition labeled by a visible action a and leading to a state s', then t is the source of a similar sequence (of possibly different length, but ended by the same visible action a) that leads to a state t' such that $s' \sqsubseteq t'$.

We performed these formal verifications on a test suite made of 1571 systems. Taking into account our knowledge of synchronization protocols, we wrote 63 tests by hand. These systems aim at pushing the protocol in its corners, and include intricate multiway synchronizations of three or more tasks. Nevertheless, we have a subjective vision of possible corner cases for the protocols, therefore we also generated other tests in an attempt to cover all basic cases. The remaining 1508 tests are automatically generated and represent parallel composition of tasks with two transitions.

Our verification approach may not be as complete as a formal proof of the protocol, but we underline that our approach led to the detection of possible deadlocks in Parrow and Sjödin protocol, despite that the correctness of this protocol had been proven manually [53]. Later, using the same approach, we also confirmed possible deadlocks (already identified by Katz and Peled [35]) in α-core, which had also been proven manually [54].

Moreover, since our verification approach is automated, it allowed us to perform a systematic validation of several protocol enhancements. Each time we modified the protocol, we could quickly verify whether the modification triggered bugs in any system of our test suite. Starting from the Parrow and Sjödin protocol model, we thus iterated to obtain the protocol eventually used in DLC, even before implementing the compiler.

In the sequel, we informally present our iterations from the Parrow and Sjödin protocol. In Appendix A, we give the LNT formal specification of the resulting protocol, which is the one used in DLC. This LNT specification is also available in the DLC distribution, since it is the one actually used for the protocol formal verification with CADP. The LNT specification was validated using our systematic validation approach. On our test suite, it never leads to a livelock or to a deadlock, and safety equivalence is preserved between the original system specification and the automatically generated implementation model. We thereby have a good confidence in the protocol correctness.

4.3. Protocol Enhancement

In order to improve the implementations generated by DLC, we enhanced the protocol. The enhancements are tagged with respect to their goal: correctness, simplification, expressiveness, or performance. For the reader interested in more formal details, we regularly make an explicit reference to lines of the LNT model given in Appendix A.

Supporting Asynchronous Communications (correctness). In one of our previous works [20], we showed that the Parrow and Sjödin protocol can lead to deadlocks when processes communicate asynchronously. To summarize, the issue may arise when a gate receives a *yes* message and removes all ready announces it has received so far —the idea being that since the negotiation succeeded, ready announces are not valid anymore. However, a task involved in the negotiation may have received a *commit* message, realized the action and transferred a new *ready* message before the *yes* message reaches the gate. In such a case, the gate erases the task from the set of ready tasks, possibly leading to a deadlock.

Our solution to fix this problem is to separate the ready announces that are received during a negotiation from those that were already there before the negotiation. When the gate receives the negotiation result, it updates the set of ready tasks. If the negotiation is successful (message *commit*), the gate removes the concerned tasks from the ready set, and then updates the ready set with ready announces received during the negotiation (lines 365–376). Otherwise, the gate removes the task that sent the *abort* message from the ready set, and still update the ready set with ready announces received during the negotiation (lines 378–387).

Merging Task and Manager (simplification). A task and its associated manager are merged into one process, where both task and manager behave as coroutines. Once a task has listed its possible actions, it yields the execution to its manager. The manager conducts negotiations, and yields back the execution to the task once a negotiation succeeded. This modification removes the need for *request* and *confirm* message types.

Reducing Message Types (simplification). Since *query* and *lock* messages have resembling semantics (i.e., a lock request), we unite these two types of messages into a single *lock* type. Similarly for the result messages, we unite *yes* and *commit* into a single *commit* message type, and *no* and *abort* into a single *abort* message type. Consequently, out of the original nine message types only four remain, namely *ready* for announces, *lock* for locking, and *commit* and *abort* for results (lines 73–80).

Broadcasting Results (performance). To avoid deadlocks, the locking phase respects the manager order. However, ordered transmission is not required for the result messages. Therefore, the manager that initiates a *commit* or *abort* chain might as well broadcast this message to all concerned managers (for instance, see lines 510–515 for the broadcast of *commit* messages by a manager). This modification does not reduce the total number of messages, but it avoids a sequence of messages by broadcasting results in parallel.

Supporting Multiple Synchronization Vectors per Gate (expressiveness). The Parrow and Sjödin protocol is specified for only one synchronization vector per gate. We extended the protocol to support several ones, such that all constructions using the LNT parallel composition, in particular n-among-m synchronization, can be handled. Prior to starting a negotiation, a gate selects any of its synchronization vectors for which all tasks are ready (lines 332–338). In addition, the synchronization vector is included in lock requests, such that each task knows which other tasks must be locked.

Supporting Internal Actions (expressiveness). A task can perform internal actions (traditionally noted τ in process algebras, or i in LNT), on which no rendezvous can be performed. Internal actions are decided at the task level, with respect to ongoing negotiations: a task can realize an internal action only if it is not currently locked by a negotiation for another action on a gate (lines 550–555). In practice, i.e. in the C implementation of the protocol, we let a task—ready for both internal actions and gate actions—wait for lock requests for some time, and then proceed to an internal action if no lock request has been received.

Adding Optional Gate Confirmation (expressiveness). The last task of the lock chain is the one that, if it accepts the lock, makes the synchronization happen. However, as we will see in Section 5, we sometimes need to decide at the gate level whether an action happens or not. We add the possibility for a gate to require the negotiation confirmation. When the gate wants to confirm a negotiation, it adds a *confirm* flag to the lock request (lines 355–356). When the last task of the lock chain accepts a lock request with a *confirm* flag, it forwards the *lock* message to the gate (lines 502–504), which must decide whether to confirm the negotiation or not and then accordingly broadcast a *commit* or *abort* message back to all involved tasks (lines 389–413). This protocol modification lets a gate know when all tasks are locked but still does not consider the negotiation as a success yet.

Supporting Data Offers (expressiveness). Although data offers may seem to be orthogonal with the synchronization problem, we actually discovered that a naive handling of offers can trigger deadlocks. Consider the following system:

```
process T [A: nat] is
   select
      A (1 of nat)
   [] i ; A (2 of nat)
   end select
end process
```

Figure 5 illustrates a possible protocol execution. We skip the detailed description of the start, in order to focus on the gate behavior when it receives the *abort* message. In the Parrow and Sjödin protocol, when the gate receives an abort message from a task, it considers this task as not ready anymore since it has just refused a lock request. Here however, task T is still ready on gate A, only with an offer incompatible with the one proposed for the lock request. If gate A had to consider task T as unready, the system would deadlock. Therefore, gate A must still consider task T as ready, even though the gate has just received an *abort* message from the task. To summarize, when a gate receives an *abort* message, it should consider the sending task to be still ready if the task has signaled itself as ready during the negotiation.

These possible deadlocks were not discovered by our formal verification approach, but by classical testing of implementations generated by DLC. This is due to the fact that when we generate the model of the implementation, we cannot take data offers of the original system into account. However, this limitation only concerns the generation of the implementation model, whereas the actual implementations generated by DLC can handle data offers. The correction was taken into account in the formal model (lines 381–382).

4.4. Autolock Optimization

The autolock optimization is a performance enhancement that aims at reducing the length of negotiations.

The locking phase ensures that no task commits to more than one action at a time. However, when a task is ready on only one gate, there is no necessity to lock this task since it will not accept locks from any other gate. Based on this observation, the α-core protocol [54] avoids unnecessary lock messages (see the *participate* message type of α-core).

We introduce a similar optimization that we name *autolock*: a task that is ready on only one gate automatically locks itself and signals it to the gate by a *ready(locked)* message (lines 363–370). The locking phase of a

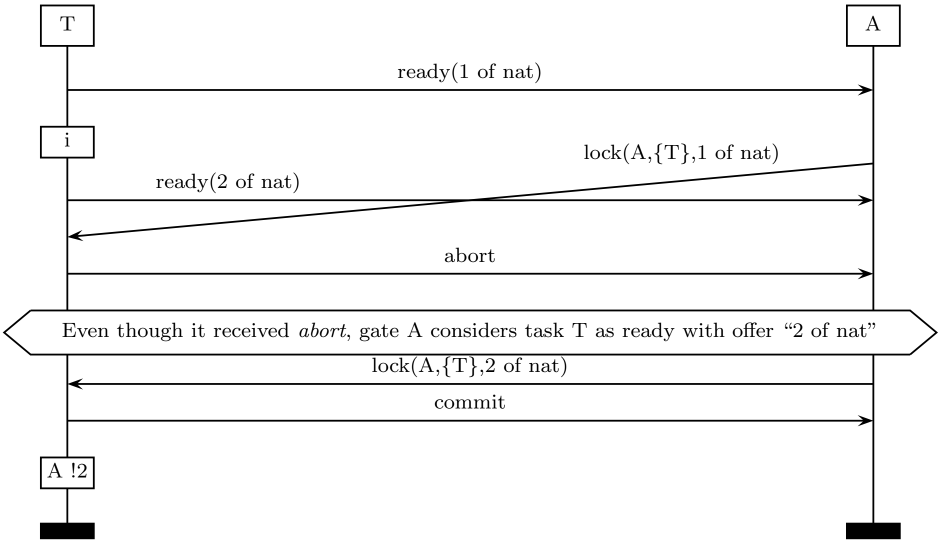


Figure 5: Data offer handling requires to modify the gate behavior.

subsequent negotiation from this gate can safely bypass the autolocked task, and therefore requires less messages.

We illustrate the autolock optimization on the following example, where gate A has a single synchronization vector {T1, T2} as specified by the parallel composition (on the right below):

```
process T1 [A: any] is
   select
      A
   [] i ; A
   end select
end process
```

```
process T2 [A: any] is
   select
      A ; A
   [] i ; A
   end select
end process
```

```
par A in
   T1 [A]
|| T2 [A]
end par
```

Figure 6 illustrates a possible execution of the protocol. Initially, both tasks are ready on gate A and on the internal action i. Task T1 executes the internal action, becomes ready only on gate A and announces it with a *ready(locked)* message. At this point, gate A considers both tasks as ready and T1 as autolocked. The dotted arrows indicate the locking phase that would be required in absence of autolock: the lock chain must pass through both tasks. Thanks to the autolock, this locking phase is reduced to only one lock request for T2.

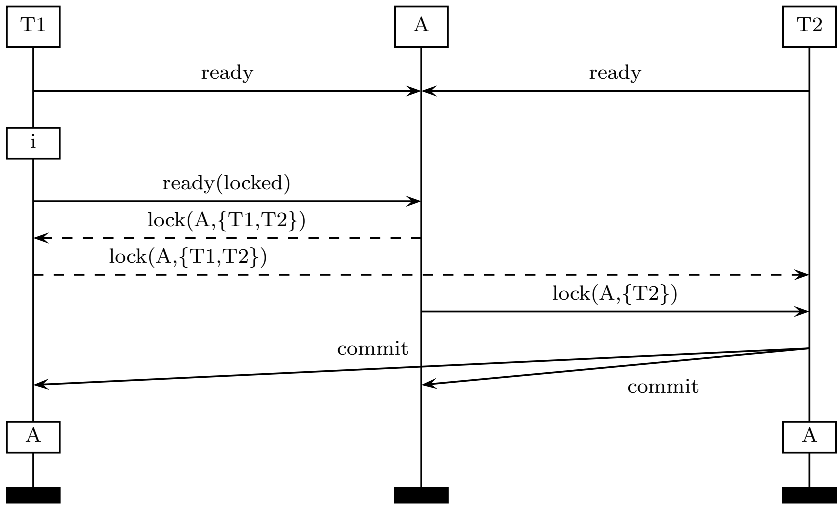


Figure 6: When T1 is ready only on gate A, it locks itself, and the subsequent locking phase is reduced.

4.5. Purge Mechanism

As soon as we added the autolock optimization to the protocol, our systematic validation approach allowed us to identify an error caused by the combination of autolock and offset synchronization. We first illustrate this problem, and then present the purge mechanism that allows to use the autolock optimization while preserving the protocol correctness.

Figure 7 illustrates the issue on the previous example, with a different protocol execution. Both tasks T1 and T2 send a *ready* message to gate A, which starts a negotiation by sending a *lock* message to task T1. Before the reception of this *lock* message, task T1 realizes an internal action, becomes ready only for an action on gate A and sends a *ready(locked)* message to gate A. Then, task T1 receives the lock request from gate A, accepts it and forwards it to task T2, which accepts the lock request and informs both gate A and task T1 of the negotiation success with a *commit* message: a first rendezvous on gate A between tasks T1 and T2 is achieved. At this point, gate A considers task T1 autolocked, since gate A has received the *ready(locked)* message after it has sent the lock request to task T1. Task T2 becomes ready for only an action on gate A, and signals itself as autolocked to gate A. Gate A now considers both tasks autolocked, and concludes that a second rendezvous on gate A is achieved. However, the specification of task T1 authorizes only one action on gate A, therefore this second rendezvous is invalid for task T1.

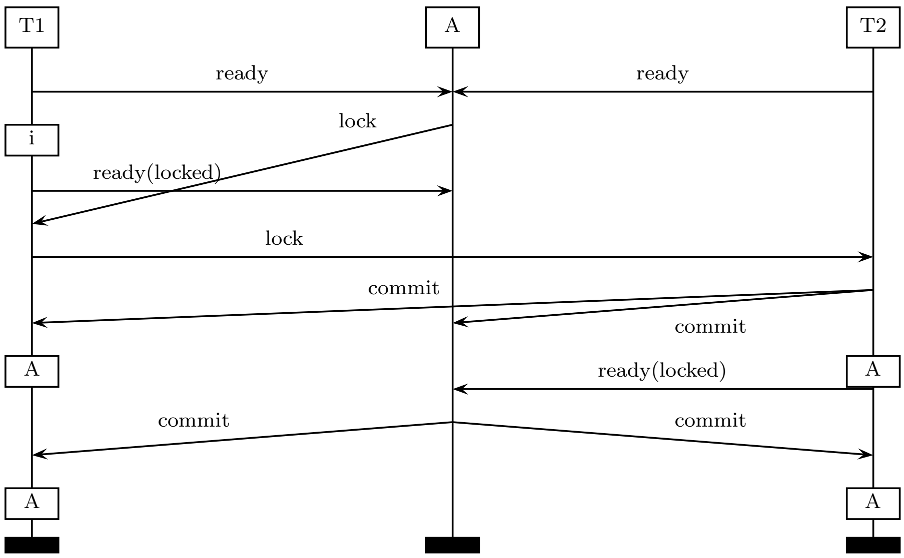


Figure 7: Autolock and offset synchronization lead to an invalid action.

The invalid action comes from the fact that gate A considers task T1 to be autolocked although it is not. To avoid such situations, we designed the purge mechanism that enables a task to purge, i.e., to cancel, an autolock message already sent to a gate. We describe this mechanism on the previous example. Figure 8 illustrates an execution of the protocol where the purge is implemented.

The beginning of the execution is similar to before. When task T1 is autolocked but receives a lock request from gate A, it knows that gate A started

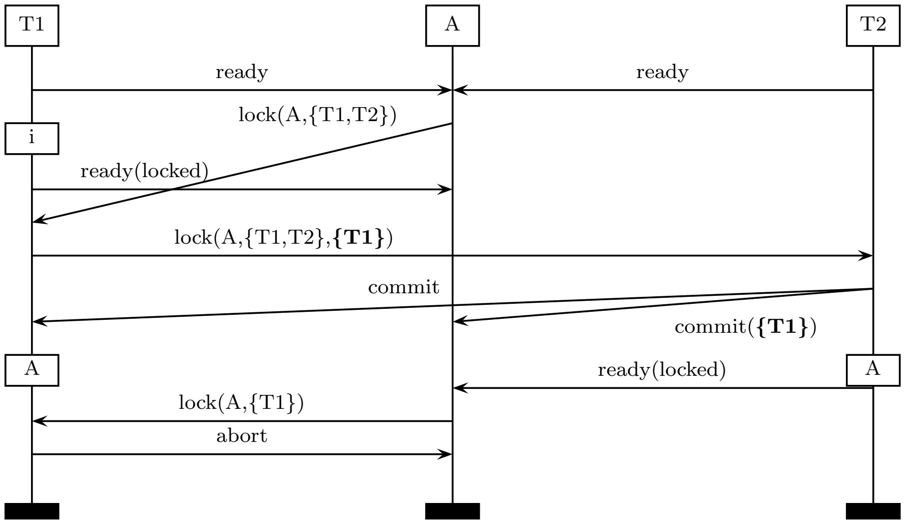


Figure 8: The purge mechanism avoids the invalid action.

the negotiation before receiving the *ready(locked)* message. In this case, task T1 adds itself to the new purge field of the lock message (lines 495–498), written in bold on Figure 8. This purge field is transmitted to gate A by the *commit* message from task T2. When gate A receives this message, it purges the *ready(locked)* message from T1: gate A now considers task T1 as ready, but *not* autolocked (see the call to function "update_purge" at line 374). Then, task T2 declares itself autolocked to gate A, which starts a new negotiation. Since gate A does not consider T1 as autolocked anymore, the negotiation starts with a lock request to task T1, which refuses it. Henceforth, the invalid action cannot occur, and the execution remains correct with respect to the system specification.

4.6. Protocol Complexity

We compare the complexity of the Parrow and Sjödin protocol, α-core and the one used in DLC. Table 2 summarizes the number of messages required to achieve a synchronization between n tasks including k autolocked tasks. Since the broadcast of messages is generally not much more costly in time that the transmission of a single message, we also give the length of the longest chain of messages sent in sequence during a successful negotiation. As explained below, numbers between brackets represent optional messages.

Protocol	Total messages	Longest sequence
Parrow-Sjödin [53]	$5n$	$2n+2$
α-core [54]	$4n-2k+2\sum_{i=1}^{n}(p_i-1)$	if $n=k$: 2 if $n>k$: $4+2(n-k)$
DLC	$3n-k$ [+2]	$2+n-k$ [+1]

Table 2: Summary of protocol complexity: total number of messages and length of the longest sequence of messages required to synchronize n tasks including k autolocked tasks. p_i represents the number of gates on which task i is ready ($p_i > 0$). For DLC, expressions between brackets indicate the message overhead when gate confirmation is required.

We briefly comment on how we computed these message counts:

Parrow-Sjödin. Each task sends a *request* message to its manager, which then sends a *ready* message to the gate. The lock chain consumes n messages to reach the last manager, which sends 1 *yes* message to the gate and starts a chain of $n-1$ *commit* messages. All involved managers also send a *confirm* message to their task. Therefore, $5n$ messages are required in total.

The *request*, *ready* and *confirm* messages can be transferred in parallel, whereas there exists an order due to causality in the transmission of *lock* and *commit* messages. Therefore, the longest sequence consists in 1 *request* message, 1 *ready* message, followed by n *lock* messages and $n-1$ *commit* messages, plus 1 *confirm* message. The *yes* message is not taken into account since it is sent in parallel with a *commit* message. Hence, the longest sequence is made of $2n + 2$ messages.

α-core. First, each task signals that it is ready to the gate, which then locks tasks in order. As illustrated in Figure 4, the locking scheme consumes 2 messages per task. Autolocked tasks need not to be locked, so the locking phase requires $2(n - k)$ messages, followed by n confirmation messages broadcasted by the gate to all tasks. At this point, each task that was also ready on other gates signals these other gates that it is not ready anymore, and waits for the acknowledgment of these gates: this disallows offset synchronizations, and requires extra messages. We denote by p_i the number of gates on which task i is ready, and here we assume $p_i > 0$ since when $p_i = 0$ the task i is not ready on any gates and therefore no negotiation occurs. When a task realizes an action on a gate, the protocol requires 2 messages for each other gate, for a total of $2\sum_{i=1}^{n}(p_i - 1)$ extra messages. Hence, the α-core protocol needs $4n - 2k + 2\sum_{i=1}^{n}(p_i - 1)$ messages in total.

When all tasks are autolocked, i.e. $n = k$, only readiness and confirmation messages are exchanged, both in parallel, so the longest sequence is made of 2 messages. Otherwise, lock requests are needed, and each non autolocked task also consumes extra messages, sent in parallel, to warn other gates that it is not ready anymore. The longest sequence then amounts to the 2 readiness and confirmation messages, plus $2(n - k)$ locks and 2 extra messages, for a total of $4 + 2(n - k)$ messages.

DLC. Each task sends a *ready* message, followed by $(n-k)$ lock requests, and then by n *commit* messages, for a total of $3n - k$ messages. Moreover, when the gate requires the confirmation, the extra *lock* and *commit* messages add 2 messages.

The longest sequence is made of 1 *ready* message followed by $(n - k)$ lock requests, and then by 1 *commit* message, for a total of $2 + n - k$ messages. In case of gate confirmation, 1 extra lock message is required.

To summarize, our protocol combines the locking phase of Parrow and Sjödin protocol with the autolock optimization. The α-core protocol has a similar optimization, but includes extra messages that disable offset synchronizations. Thanks to the purge mechanism, which is embedded in the payload of existing messages and does not require additional messages, our protocol can use the autolock optimization in presence of offset synchronizations.

5. Interaction with the Environment

DLC generates standalone programs, which do not require user-defined external code to run. However, the programs generated by DLC are of limited usage if they cannot perform side effect interactions with their external environment, such as writing data to a file, or prompting a user. Moreover, the end user may also want to influence which actions are selected at runtime, for instance to control the server crash rate in the leader election example of Section 3. To cover these cases, we designed a mechanism that permits user-defined external procedures written in C, named *hook functions*, to be integrated into the final implementation. Our goal is to make interaction with the external environment and control of actions as easy to program as possible, while keeping decent performance.

Hook functions are triggered upon actions, which are the observable events of an LNT distributed system. Three kinds of hook functions are introduced:

- When gate g is about to start a negotiation, it first executes a hook function named g`_pre_negotiation_hook`, which returns a boolean value indicating whether the negotiation is worth being started. The role of this hook is to prevent useless negotiations for actions that the user would not allow anyway. If the hook replies positively, the gate starts a negotiation for which it requires the confirmation, as discussed in Section 4.3.

- When a negotiation succeeds on a gate g, the gate executes a hook function named g`_post_negotiation_hook`, which returns a boolean value indicating whether the action can actually occur. Additionally, this function can be used to feed the system with data taken from the environment, as we will detail later.

- When an action occurs, i.e., when the gate program announces a commit to this action, each involved task t executes a local hook function named t`_hook`, which can be used for local monitoring.

When a pre-negotiation or a post-negotiation hook replies false, the gate program reacts similarly to a negotiation failure: it checks whether some new task messages arrived, then searches a possible action with respect to synchronization vectors, and, if one is detected, it calls the pre-negotiation hook and, accordingly, either starts the negotiation or not. Thus, a gate program loops on trying to perform an action, each time randomly selected among the currently possible ones.

The three of the hook functions take as argument a structure containing information about the action, including the gate, the merged data offers, and the involved tasks. A gate program executes its post-negotiation hook before it checks that all data offer variables are set. Therefore, the user can use the post-negotiation hook to detect unset variables, assign to them a value from the external environment, and flag them as set. This enables feeding data values from the external environment into the system at runtime.

We illustrate the usage of hook functions on a system with a unique task logger, which loops on getting the data associated to a key in a database and logging this data, until it receives an interruption. The task is specified as follows:

```
process logger [GET, LOG, INTERRUPT: any] (key: nat) is
var val : nat in
   loop (* get and log data, until interruption *)
      select
         GET(key, ?val) ; LOG(val)
      [] INTERRUPT ; stop
      end select
   end loop
end var
end process
```

Figures 9, 10, and 11 illustrate various usages of hooks. Figure 9 defines a hook function `logger_hook` for task logger. This function writes the data passed on LOG actions onto the local storage of the machine where the task program runs. Figure 10 defines pre- and post-negotiation hook functions for gate GET. There is no motivation to prevent actions on gate GET, so its pre-negotiation hook `GET_pre_negotiation_hook` always returns true. The GET post-negotiation hook `GET_post_negotiation_hook` retrieves the key from data offers, connects to an external database to fetch the corresponding value, and then provides this value to the logger task by setting the second data offer variable. At last, Figure 11 defines pre- and post-negotiation hooks for gate INTERRUPT. The pre-negotiation hook `INTERRUPT_pre_negotiation_hook` prevents useless negotiations if no interruption is detected. The post-negotiation hook `INTERRUPT_post_negotiation_hook` is executed only if the pre-negotiation hook gave its authorization earlier, so it blindly replies true. The gate INTERRUPT illustrates the purpose of pre-negotiation hooks: the user knows that an interruption is a rare event, so he checks it early in the pre-negotiation hook to prevent unnecessary negotiations for INTERRUPT, and thus does not hamper negotiations for GET.

With hooks, the user can prevent some actions, but cannot achieve actions that would not have been previously allowed by the protocol. Hence, since hooks can only restrict the system behavior, the execution path eventually walked is still within the original LNT model semantics. Nevertheless, users have to use hook functions carefully as preventing actions can obviously

```
void logger_hook(struct action *a) {
  switch(a->gate) {
  case GATE_GET:     break; // no local side effect
  case GATE_INTERRUPT: break; // no local side effect
  case GATE_LOG:
    uint val = a->offers[0].value;
    WriteLog(val); // write on task machine local storage
    break; }
}
```

Figure 9: Example of local hook function for task `logger`.

```
bool GET_pre_negotiation_hook(struct action *a) {
  return True; // no reason to prevent a GET action
}

// post-negotiation hook can feed data into the system
bool GET_post_negotiation_hook(struct action *a) {
  uint key = a->offers[0].value; // get key from offer
  uint val = DataBase_read(key); // external database call
  a->offers[1].value = val;   // set the value
  a->offers[1].set = True;    // mark the value as set
  return True;                // always allow the action
}
```

Figure 10: Example of pre-negotiation and post-negotiation hooks for gate `GET`.

introduce deadlocks.

The possibility that the system deadlocks does not question the safety properties (nothing bad will happen) checked on the model. As regards the liveness properties (something good will happen), as usual they assume that the environment will interact with the system in a way that the good things will effectively happen. For instance, it can be checked that a telecommunication protocol will transfer arriving data (which is a liveness property), but nothing guarantees that the environment will enable some data to arrive. In this respect, one should view the hook conditions, which are exactly at the interface between the system and the environment, as part of the environment rather than part of the system. For the verification of hooks themselves, we invite users to use traditional verification methods such as testing.

```
bool interruption = False; // record interruption detection

// Prevent useless negotiations
bool INTERRUPT_pre_negotiation_hook(struct action *a) {
  if (!interruption) {     // may be previously detected
    interruption = detect_interrupt();   // rarely true
  }
  return interruption;
}

bool INTERRUPT_post_negotiation_hook(struct action *a) {
  interruption = False;      // reset interruption flag
  return True;
}
```

Figure 11: Example of pre-negotiation and post-negotiation hooks for gate `INTERRUPT`.

6. Automatic Generation of Distributed Implementation

Figure 12 gives an overview of DLC architecture. The DLC tool takes a system specification given as an LNT parallel composition of tasks as input, optionally together with C hook functions, and produces a distributed implementation in C.

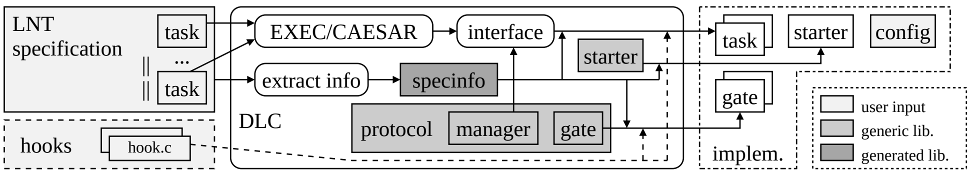


Figure 12: DLC architecture overview.

DLC first extracts information about the input specification and collects them into a C library named "specinfo", which is thus automatically generated for each system compiled by DLC. This library contains for instance the number of tasks and gates, the synchronization vectors, and the like.

DLC uses the EXEC/CÆSAR tool of CADP to obtain a sequential implementation, in C, for each task. A program generated by EXEC/CÆSAR is able to list possible actions from the current state of the task, but cannot decide which action is realized. DLC injects an interface into the C code produced by EXEC/CÆSAR in order to bind the task with the manager logic of the rendezvous protocol, which is responsible of conducting negotiations to determine which action should be realized. Moreover, each task is linked with the specinfo library in order to have access to the system information, such as synchronization vectors.

DLC produces a gate process for each gate of the system. The gate logic is implemented in a generic module, whose behavior is configured to match a gate of the current system thanks to information of the specinfo library.

Moreover, both tasks and gates use the "network" library of CADP (not represented in Figure 12) for communication between distant processes. This library is built upon TCP sockets, and thus satisfies the reliable and ordered communication hypothesis required for the protocol (as was shown in [40]). In addition, the network library provides a integrated deployment service through a "starter" program that is able to automatically distribute and start other programs on a cluster of machines. The starter program is configured with a simple text file (named "config" on Figure 12) that lists the names of machines available for deployement. The configuration file can be written by hand or generated by other scripts, thus making automatic cluster deployment easy. By default, DLC produces a configuration file where all tasks and gates run on the local host.

The user can define hook functions for tasks and gates in C source files, named `*task*.taskhook.c` and `*gate*.gatehook.c`. DLC automatically detects the presence of these files and embeds them into the generated implementation. DLC also provides a hook template creator, which can be used to obtain hook functions with empty bodies for any task or gate of the system.

In terms of program size, the code generator part of DLC is made of more than 1600 lines of C, and the runtime of generated implementations (i.e., mainly the protocol logic) represents more than 2000 lines of C. The amount of C code generated depends on the system given as input. For instance, on the Raft example of Section 7.3, DLC generates 2302 lines of C code for each server, and 84 lines of C code for the synchronization vector library.

Nondeterminism and Fairness in the Generated Implementation

If the input specification is nondeterministic, then the distributed implementation generated by DLC is also nondeterministic. The main source of nondeterminism is the variable delay of messages exchanged between programs. When several negotiations are concurrently started for conflicting rendezvous, the first negotiation that locks all tasks will succeed: this depends on the communication delay to transfer lock messages to tasks. If such delays are variable, then any of the started negotiations has a chance to succeed.

However, this is not enough to give a chance to all possible actions: when a gate receives enough ready messages to enable several synchronization vectors, if the gate always chooses to start a negotiation for the *same* synchronization vector among the enabled ones, then actions corresponding to other enabled synchronization vectors have no chance to happen. In order to avoid such a restriction of nondeterminism, a gate randomly chooses a synchronization vector (to start a negotiation for) among the enabled ones. Thus, when a gate detects several synchronization vectors enabled at the same time, a negotiation may be started for any of the enabled synchronization vectors.

Since a negotiation may be started for any enabled synchronization vector, and that all started negotiations have a chance to succeed, all possible actions of the system may be realized. Hence, the generated implementation keeps the same level of nondeterminism as the original specification. This is actually checked in the protocol verification method (discussed in Section 4.2): since the model of the implementation is at least safety-equivalent with the original specification, all actions possible in the original specification are reachable by the implementation.

A slightly more involved question is whether all conflicting rendezvous have the same probability of being executed by the implementation. We believe that it is not the case. Indeed, if ready messages are sent by a task to the ready gates always in the same order, then it is likely that the gate that is contacted first will achieve its rendezvous slightly more often than the next gates, because of the high probability that it will receive the ready message before its conflicting gates and will be the first to lock all tasks in its lock chain. This can easily be solved by choosing randomly the order in which gates are contacted by a task, but the complexity of the locking mechanism let us think that several other parameters can have an impact on the distribution of execution probabilities between conflicting rendezvous, such as the length of the respective lock chains, the order of tasks in lock chains, and the relative positions in the lock chains of those tasks that are in the intersection of the conflicting synchronization vectors. In the future, it would be interesting to study formally this aspect, for instance using the quantitative analysis tools available in CADP [14] after adding quantitative annotations in the LNT model. Such a study requires to have a realistic quantitative model of communication delays, which itself may depend on several parameters, but we believe that reasonable asumptions can be made, which would help to improve the fairness of the implementations generated by DLC.

Bootstrapping and Rendezvous Protocol Implementation

We do not have an LNT formal model of the whole DLC compiler, but it is in itself a collection of code generation procedures, which are sequential. We focused our effort on the formal specification and verification of the rendezvous protocol, which is at the heart of each distributed implementation generated by DLC.

Given that DLC is able to generate the LNT model of an implementation for verification purposes (see Section 4.2), we can think of a bootstrapping approach that consists in using EXEC/CÆSAR on this LNT model to eventually obtain a C implementation. However, this is currently impractical, essentially because the verification branch of DLC is limited to systems where rendezvous have no data exchange (whereas the implementation branch of DLC does support value-passing rendezvous). Therefore, we implemented the protocol by hand, strictly following the LNT specification for the synchronization logic. The hand-writing approach allowed us to directly integrate data offers and hook functions support, with minimal performance overhead.

The protocol implementation consists of two modules for the protocol logic of tasks and gates. These modules are written once and for all, and are subsequently reused in generated implementations, where their behavior is tailored to the current system through information from the specinfo library. The isolation of the protocol core logic in generic modules eases its debugging and maintenance, and raises the level of trust we have in its correctness.

As a comparison, the approach used to generate a distributed implementation in BIP is closer to the bootstrapping approach mentioned earlier: the protocol logic is inserted at the BIP level, to obtain a BIP specification where processes interact only by sending and receiving messages. Then, this model is compiled to a platform that provides message-passing primitives. This is a valid correct-by-construction approach when the equivalence of BIP models before and after protocol insertion can be demonstrated; however, the proof does not concern the protocol actually used in the implementation (namely α-core), but simplified protocols, which do not enforce progress, i.e., do not guarantee that possible rendezvous will eventually happen (see the discussion on "interoperability of reservation protocols" in Section 6 of [57]). Progress is checked in our approach using livelock and deadlock detection.

Current Limitations

We briefly list the main current limitations of DLC:

- DLC can handle data offers in rendezvous for simple types which values can fit on a 32-bit C integer, but it cannot handle data offers for more complex types such as arrays and lists. Complex types *can* be used in the specification, but they must not appear in rendezvous data offers, otherwise DLC emits an error during compilation. The support of complex types needs serialization and deserialization primitives for any user-defined type. We consider that such primitives should be generated by CADP tools which have the control on the C implementation of these types; we thus left complex type support for future work.

- DLC considers that the number of tasks is a constant defined by the (static) parallel composition of the input systems. In particular, a task cannot dynamically create other tasks at runtime. Although the dynamic creation of tasks is an interesting feature, it requires substantial modifications of the EXEC/CÆSAR tool, such that the generated C implementation of a task could fork itself into several tasks, which could be deployed at runtime. Moreover, the protocol would also need to modify the synchronization vectors at runtime, to take new tasks into account. For the moment, dynamic creation of tasks can be simulated in the specification by declaring a static pool of tasks, and by activating some tasks among this pool using specific actions at runtime.

- LNT allows *guarded actions*, i.e., actions which are authorized only if a condition, which may depend on a value received during the action, is verified. For instance, the following LNT code specifies an action on gate A that can be realized only if the value received in variable x is greater than the value stored in variable y:

  ```
  A (?x) where x > y
  ```

  DLC does not handle guarded actions yet because EXEC/CÆSAR does not give access to the guard condition. To support guarded actions, we need to modify EXEC/CÆSAR; this is left for future work.

7. Experimental Results

We conducted several experiments to evaluate the implementations generated by DLC. The first two experiments focus on the evalutation of the multiway rendezvous protocol. The last experiment is a case study on the Raft consensus algorithm. These experiments are performed on clusters provided by the distributed computing testbed *Grid'5000*[10]. Measures may have been impacted by other experiments of other researchers running at the same time.

7.1. Distributed Synchronization Barrier

This experiment evaluates the rendezvous protocol on a system with non-conflicting multiway rendezvous between a various number of tasks. The system is a classical distributed synchronization barrier between several deterministic processes. We measure the time required for distant processes to synchronize themselves several times on a barrier.

Implementing a distributed barrier in LNT is directly achieved by a multiway rendezvous between all workers on a single gate, as depicted in Figure 13. In order to compare the performances of the implementation generated by DLC with other possible solutions, we also implemented this system in C, Java and Erlang, using respectively sockets, Java RMI (*Remote Method Invocation*) and Erlang's built-in message passing as communication primitive between processes. Since these languages do not offer multiway rendezvous, we fall back on the classical implementation of a distributed barrier. For instance, Figure 14 illustrates the Java implementation: a distinct barrier process blocks workers until they have all invoked the SYNC method, and then let them continue. C and Erlang implementations follow the same idea, using message passing between workers and the barrier process.

```
process WORKER [SYNC: none] is
  var n : nat in
    for n := 0 while n < 1000 by n := n + 1 loop
      SYNC
    end loop
  end var
end process

-- Parallel composition: 5 workers
par SYNC in
    WORKER [SYNC]
 || WORKER [SYNC]
 || WORKER [SYNC]
 || WORKER [SYNC]
 || WORKER [SYNC]
end par
```

Figure 13: Implementation of a synchronization barrier in LNT: all worker processes synchronizes with a multiway rendezvous on gate SYNC.

Figure 15 illustrates the time required to perform a thousand synchronizations between several processes which are deployed on distinct machines. We observe that the implementations generated by DLC are slower than the

[10]http://www.grid5000.fr

```
public class Barrier implements BarrierInterface {
  private static int c = 0;
  private final static Object lock = new Object();
  private static int nb_worker = 5;

  public void SYNC() {
    synchronized (lock) {
      c++;
      if (c == nb_worker) {
        c = 0;
        lock. notifyAll ();
      } else {
        lock.wait ();
      }
    }
  }
  // main method: create RMI registry, register method SYNC
}

public class Worker {
  public static void main(String [] args) {
    // Retrieve RMI registry from host given as argument
    Registry registry = LocateRegistry. getRegistry (args [0]);
    // Get barrier stub
    BarrierInterface stub = ( BarrierInterface ) registry .lookup("SYNC");
    // Synchronize 1000 times
    for (int i = 0; i < 1000; i++) {
      stub.SYNC();
    }
  }
}
```

Figure 14: Implementation of a synchronization barrier in Java: each Worker invokes (through Remote Method Invocation) the SYNC method of the Barrier process, which makes workers wait until they have all invoked the method.

C programs, but faster that the Erlang and Java ones. All programs seem to scale linearly with the number of processes.

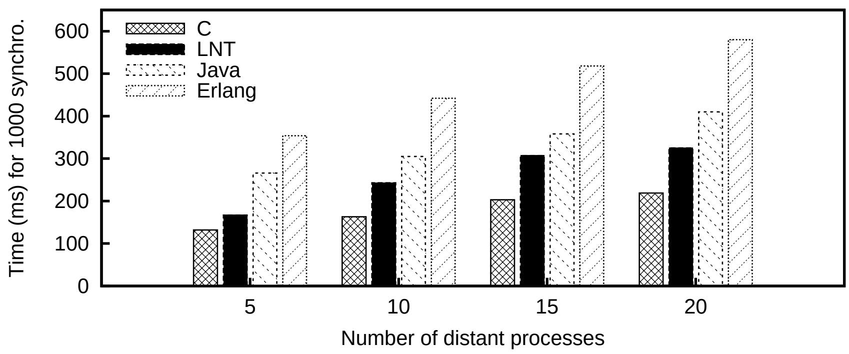


Figure 15: Distributed synchronization barrier: thanks to the autolock optimization, the code generated by DLC reaches the speed of regular programming languages.

The synchronization protocol appears to be as fast as native implementations in the situation of a distributed barrier, which can be explained by the autolock optimization. In the LNT implementation, task processes are always ready on only one gate (which corresponds to the barrier), therefore the autolock optimization is activated. With autolock, protocol negotiations are reduced to a *ready* and a *commit* message per task: this matches the classical implementation of a distributed barrier used in other implementations.

There are constant performance gaps between the implementations. On the one hand, we think that DLC generated implementations are slower than the native C ones because DLC generates C code that contains all the logic of the protocol, and that uses a library for message passing on top of sockets. On the other hand, we suppose that Java and Erlang solutions are slower than the DLC ones because of the overhead imposed by their respective virtual machines. This experiment shows that, in the absence of conflicts, the DLC protocol performance is similar to native implementations.

7.2. Dining Philosophers

The aim of this experiment is to evaluate the efficiency of the rendezvous protocol on a system containing many conflicting multiway rendezvous. We consider the dining philosophers problem [15], which is a classical problem of mutual exclusion when accessing shared resources. This example has the advantage of being simple and well-understood, so we consider it as an appropriate benchmark to evaluate DLC. It consists of several philosophers sitting at a round table to eat meals. In order to eat, a philosopher must take its two surrounding forks, which are shared with its neighbors. Forks correspond to resources that are shared between philosophers, and the problem is to guarantee the mutual exclusion of philosophers who want to access the same forks, without introducing deadlocks.

Most solutions are based on the hypothesis that a philosopher can only interact with one fork at a time. Thus, the solution is a protocol to ensure that both forks can be picked without leading the system into a deadlock. We revisit the problem in LNT, now equipped with the multiway rendezvous: a philosopher takes its both surrounding forks in one rendezvous where the three processes (the philosopher and the two forks) synchronize. An excerpt of the LNT code is given in Figure 16. Rendezvous on eating actions are conflicting for neighboring philosophers. These conflicts are resolved in the DLC-generated implementations by the synchronization protocol, which ensures the mutual exclusion of conflicting rendezvous.

For comparison, we wrote a distributed philosopher solution in Java, using RMI for process interactions. An excerpt of the Java code is given in Figure 17. Forks are objects with "take" and "release" methods, and philosophers are objects that call fork methods through RMI. In order to avoid deadlocks, we use the simple solution that consists in imposing a global order on fork picking.

In practice, we measure the amount of time required by a group of philosophers to eat a certain amount of meals each. Note that both LNT and Java implementations do not prevent the possible starvation of a philosopher.

However, in the context of this experiment, we do not focus on a starvation-free solution to the dining philosophers. We merely want to produce implementations with many interactions between distant processes. Moreover, since we bound the number of meals that each philosopher must eat, all philosophers eventually have the opportunity to finish all their meals. The execution times for both the LNT/DLC and Java versions of the dining philosophers example are presented in Figures 18 and 19 respectively. They show that both DLC and Java provide solutions with similar performance.

```
process PHILO [EAT: none] (nbmeals : nat) is
  while nbmeals > 0 loop
    EAT;
    nbmeals := nbmeals − 1
  end loop
end process

process FORK [EAT_LEFT, EAT_RIGHT: none] (nbmeals : nat) is
  nbmeals := nbmeals ∗ 2; −− a fork is used by 2 philo
  while nbmeals > 0 loop
    select
      EAT_LEFT
    [] EAT_RIGHT
    end select;
      nbmeals := nbmeals − 1
  end loop
end process

−− 3 philo and 3 forks, 1000 meals per philo
par
   EAT_0         −> PHILO [EAT_0] (1000)
|| EAT_0, EAT_1 −> FORK [EAT_0, EAT_1] (1000)
|| EAT_1         −> PHILO [EAT_1] (1000)
|| EAT_1, EAT_2 −> FORK [EAT_1, EAT_2] (1000)
|| EAT_2         −> PHILO [EAT_2] (1000)
|| EAT_2, EAT_0 −> FORK [EAT_2, EAT_0] (1000)
end par
```

Figure 16: LNT code for the dining philosophers example.

```
public class Fork implements ForkInterface {
  private static Lock l = new ReentrantLock(true);

  public void take()    { l.lock(); }
  public void release() { l.unlock(); }

  // main method: create RMI registry, register Fork
}

public class Philo {
  public static void main(String[] args) {
    // args: forkid1, host1, forkid2, host2, nbmeals
    int forkid1 = Integer.parseInt(args[0]);
    Registry r1 = LocateRegistry.getRegistry(args[1]);
    int forkid2 = Integer.parseInt(args[2]);
    Registry r2 = LocateRegistry.getRegistry(args[3]);
    int nbmeals = Integer.parseInt(args[4]);
    // Get Forks stub
    ForkInterface s1 = (ForkInterface) r1.lookup("Fork");
    ForkInterface s2 = (ForkInterface) r2.lookup("Fork");
    // sort to take forks in order
    if (forkid1 > forkid2) {
      ForkInterface tmp = s1; s1 = s2; s2 = tmp;
    }
    for (int i = 0; i < nbmeals; i++) {
      s1.take();
      s2.take();
      s1.release();
      s2.release();
    }
  }
}
```

Figure 17: Java code for the dining philosophers example.

7.3. Case Study: Raft Consensus

We modeled Raft [51] in LNT in order to demonstrate DLC on a non-trivial system. Raft, like the better known Paxos [37], is a consensus algorithm: it maintains a consistent log of entries replicated among a set of servers, while surviving the failure of some servers. It thus enables fault tolerant services to be built using the replicated state machine technique [58].

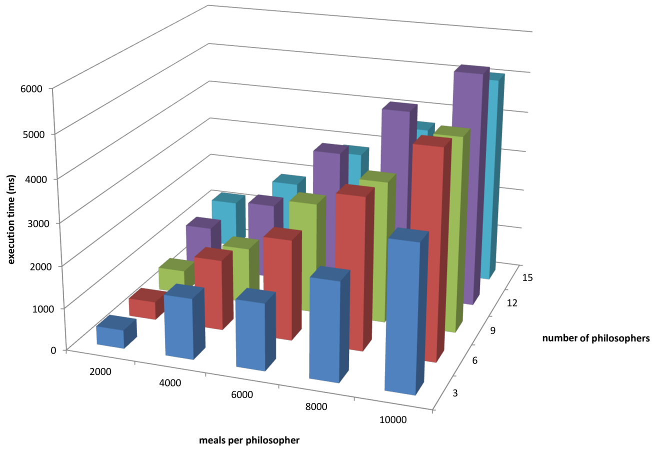


Figure 18: Execution time for the dining philosophers example using LNT and DLC. Varying parameters are the number of phiosophers and the number of meals per philosopher.

Raft is used in several industrial-class fault tolerant key-value stores, such as Consul.[11]

A TLA+ formal specification of Raft core features (leader election and log replication) is available, upon which a hand-written safety proof is built [50]. Our LNT model includes a basic key-value store made fault tolerant using Raft: every client request to the store is first committed on a majority of servers before the answer is sent back to the client. We use hook functions to implement (a) the timeout mechanism needed in Raft, (b) the control of server crashes, and (c) a socket interface to the key-value store, such that external client programs can be implemented in any language. We managed to implement the core of Raft in approximately 500 lines of LNT plus 300 lines of C for hook functions (mainly boilerplate for sockets); for comparison, the Consul Raft library alone represents approximately 4000 lines of Golang.

The generated distributed programs successfully run on a cluster of machines. We first experimented with server crashes to validate that the key-value store remains available as long as a majority of servers are running. Then, for different cluster sizes, we made several runs of a thousand write requests to the key-value store, with crashes disabled. Figure 20 compares the performances of DLC with those of Consul.

[11]Consul: `www.consul.io`, and its Raft library: `github.com/hashicorp/raft`

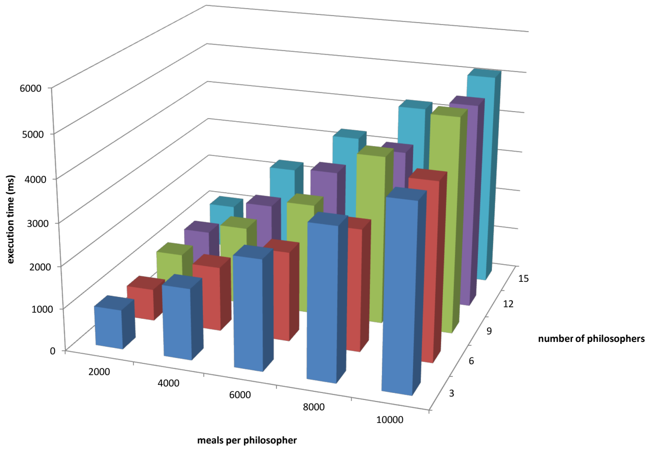


Figure 19: Execution time for the dining philosophers example using Java.

We measure throughput with requests coming from many clients in parallel (see left of Figure 20). In this case, Consul implementation is up to ten times faster than our solution, and seems to be only slightly impacted by the cluster size. After a discussion with Consul developers, we realized that Consul uses a Raft-level optimization: when the leader server receives a client request, it waits 50ms to gather other client requests in order to replicate the group of requests among Raft servers in only one round of log replication, whereas the LNT implementation triggers a log replication for each client request. We cannot easily implement the Consul strategy since DLC does not yet handle arrays or lists in rendezvous.

Nonetheless, Consul latency, measured with sequential requests from a single client (see right of Figure 20), suffers from the optimization. Indeed, the leader server pauses 50ms for each requests, thus the proceeding time for 1000 serial requests reaches 50 seconds. The LNT implementation is not impacted since its leader server treats requests sequentially anyway, and presents a latency which increased with the size of the Raft cluster, as expected. For the 7 servers configuration, our solution proceeds 1000 requests in 5469ms (in average), i.e., a little bit more than 5ms per request replication

While DLC does not pretend to generate implementations that compete with hand-crafted programs, we consider that the performance achieved so far still qualify for rapid prototyping, with all the benefits that formal verifications brought on. Moreover, hook functions enable to model and prototype

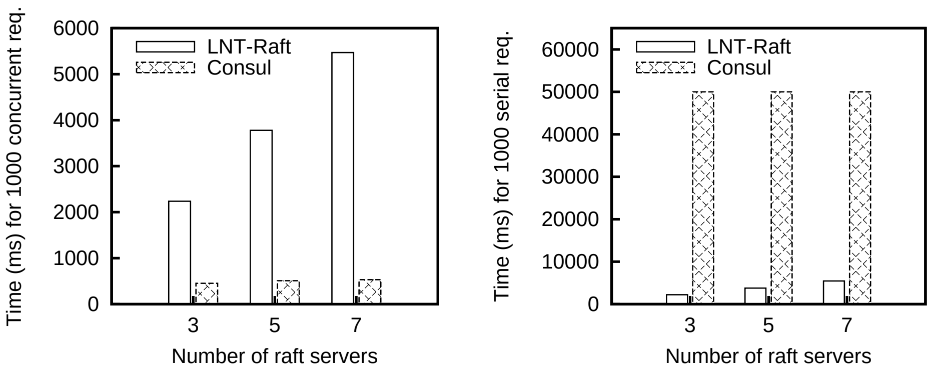


Figure 20: Raft consensus: comparison with Consul, throughput (left) and latency (right).

only a part (e.g., the safety critical part) of a larger system while still interacting with the rest of the system through hook functions.

8. Conclusion and Future Work

A distributed system made of asynchronous concurrent processes can be formally modeled in LNT, using powerful primitives such as value-passing multiway rendezvous. An LNT model can be formally verified thanks to the numerous and mature tools of CADP. The tool DLC, presented in this paper, now also enables rapid prototyping by automatically generating a distributed implementation in C. We think the combination of LNT, CADP and DLC provides a featureful framework for the formal verification and rapid prototyping of distributed systems.

We presented the protocol used to implement value-passing multiway rendezvous, which allows offset synchronizations together with the autolock optimization, made correct thanks to the purge mechanism. We incrementally developed this protocol thanks to an automatic verification approach which relies on the formal techniques that our team has been working on for years. We provide the LNT formal specification of this protocol in Appendix A. In order to let the end-user have some control on the generated programs and define interactions with the external world, we introduced hook functions, which enable user-defined C procedures to be integrated into the final implementation. The hook functions can only restrict the system behavior, therefore they should not be able to make it behave incorrectly with respect to the original specification semantics. We covered how DLC proceeds to generate distributed programs, and we exposed DLC internal architecture. We presented three experiments made with DLC, including an implementation of the non-trivial Raft algorithm. The measured performances reveal that even if DLC generated programs may be currently slower than solutions written in general programming languages, we consider that they still qualify for rapid prototyping.

As future work, we plan to make DLC handle complex types, such as lists and arrays, in data offers. We also think the protocol negotiations can be shortened in some special cases (such as binary rendezvous) which could lead to better performances. Moreover, it would be useful to implement timing mechanisms (such as timeouts) as primitives of LNT, as already suggested in [60]. Currently, DLC communication relies on TCP sockets, which is a uniform communication mean but not necessarily the most efficient in all situations. A new track of research could be to investigate how DLC could generate code specialized to specific computing architectures (multi-core or distributed, communication through a local network or through internet, etc.), for instance by adding options in the network configuration file, or DLC-specific annotations in the LNT model. Finally, a way to raise the trust in the correctness of DLC could be to bootstrap the compiler from LNT sources, for instance using our team compiler construction framework [24]. We can also consider using CADP tools on the source LNT model to perform co-simulation of the distributed program execution, in a way similar to what Garavel *et al.* [28] and Lantreibecq *et al.* [39] have already explored using EXEC/CÆSAR.

Acknowledgments

The authors warmly thank Lucas Cimon for suggesting Raft as a case study, and the Inria/CONVECS team members, in particular Wendelin Serwe and Hubert Garavel, for useful discussions. This work was partly funded by the French *Fonds national pour la Société Numérique* (FSN), Pôles Minalogic, Systematic and SCS (project OpenCloudware). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Appendix A. LNT Model of the Multiway Rendezvous Protocol

This appendix presents the LNT model of the multiway rendezvous protocol in five parts. Appendix A.1 lists the data types, the functions defined on these types, and the communication channels used in the specification. Some standard functions, such as set-related operations (member, insert, diff, etc), are predefined in LNT, see [12] for more details. Appendix A.2 presents the generic model of a gate process and Appendix A.3 presents the generic model of a manager process. Appendix A.4 presents a buffer process, which is a bounded FIFO buffer used to model asynchronous communications between gates and managers. Finally, Appendix A.5 presents a small system specification and the implementation model generated by DLC from this specification, which uses instances of the generic models of gate, manager and buffer.

This LNT model is the one actually used for the formal verification of the protocol with CADP. Therefore, it is also present in the DLC distribution, available at `http://hevrard.org/DLC`.

Appendix A.1. Data Types, Functions and Channels

```
-- TYPES

type nat_set is
   set of nat
   with "length", "access", "member"
end type

type id_set is
   sorted set of DLC_ID
   with "head", "length", "access", "member", "diff", "union", "remove", "empty", "inter"
end type

type id_list is
   list of DLC_ID
   with "union", "empty", "head", "member", "delete", "tail"
end type

type sync_vect_list is
   list of id_set
   with "head", "access", "length"
end type

type sync_map_entry is
   sync_map_entry (gate : DLC_ID, vect_list : sync_vect_list)
   with "get"
end type

type sync_map is
   list of sync_map_entry
```

```
   with "access", "length"
end type

type dlc_action is
   action (gate : DLC_ID)
   with "get", "=="
end type

type action_set is
   set of dlc_action
   with "length", "access", "member"
end type

type transition is
    nil_transition ,
    transition (action : dlc_action, next_states : nat_set)
   with "get", "=="
end type

type transition_list is
    list of transition
end type

type state is
    nil_state ,
    state ( id : nat, transitions : transition_list )
   with "get"
end type

type state_list is
    list of state
end type

type lock is
   lock (action : dlc_action, index : nat, path : id_set, confirm : bool, purge : id_list )
   with "get", "set"
end type

type lock_list is
    list of lock
   with "empty", "append", "head", "length", "access", " tail "
end type

type message is
   READY (autolocked : bool),
   LOCK (lock : lock),
   COMMIT,
   COMMIT (purge : id_list),
   ABORT,
   ABORT (purge : id_list)
end type

type message_list is
    list of message
   with "append", "head", " tail ", "length", "empty"
end type

```

```
type arrival is
    arrival (action : dlc_action, arrival : nat)
   with "get"
end type

type arrival_list is
    list of arrival
   with "access", "length"
end type

type gate_state is
    idle,
    dealing
   with "=="
end type

type manager_state is
    free,
    locked,
    autolock_free,
    autolock_locked
   with "==", "!="
end type

-- FUNCTIONS

function find_state (space : state_list, id : nat) : state is
   case space in
   var i : nat, tra : transition_list, tail : state_list in
      {} -> return nil_state
   | cons(state (i, tra), any state_list) where i == id ->
         return state (i, tra)
   | cons (any state, tail) ->
         return find_state (tail, id)
   end case
end function

function find_transition (tl : transition_list, act : dlc_action) : transition is
   case tl in
   var a : dlc_action, nl : nat_set, tail : transition_list in
      {} -> return nil_transition
   | cons (transition (a, nl), any transition_list) where a == act ->
         return transition (a, nl)
   | cons (any transition, tail) ->
         return find_transition (tail, act)
   end case
end function

function get_next(space : state_list, id : nat, action : dlc_action) : nat_set is
   var t : transition in
      t := find_transition (get_transitions (find_state (space, id)), action) ;
      if t == nil_transition then
         return {}
      else
         return get_next_states (t)
      end if
   end var
```

```
end function

function collect_action ( tl : transition_list , al : action_set ) : action_set is
   case tl in
   var act : dlc_action , tail : transition_list in
      {} -> return al
   |  cons ( transition (act , any nat_set) , tail ) ->
         return collect_action ( tail , insert (act , al ))
   |  cons ( nil_transition , tail ) ->
         -- should never happen, remove compiler warning
         return collect_action ( tail , al )
   end case
end function

function possible_actions (space : state_list , id : nat) : action_set is
   return collect_action ( get_transitions (find_state (space, id )), {})
end function

function extract_gate (al : action_set , gl : id_set) : id_set is
   case al in
   var g : DLC_ID, tail : action_set in
      {} -> return gl
   |  cons (action (g), tail ) ->
         return extract_gate ( tail , insert (g, gl ))
   end case
end function

function arrival_state (dl : arrival_list , act : dlc_action ) : nat
    raises action_not_found : none
is
   var n : nat in
      for n := 1 while n <= length (dl) by n := n+1 loop
         if get_action (access (dl , n)) == act then
            return get_arrival (access (dl , n))
         end if
      end loop;
      raise action_not_found
   end var
end function

function isin (vect , rdytask : id_set) : bool is
   var n : nat in
      for n := 1 while n <= length (vect) by n := n+1 loop
         if not (member (access (vect, n), rdytask )) then
            return false
         end if
      end loop ;
      return true
   end var
end function

function possible_rdv (rdytask : id_set , vectors : sync_vect_list ) : bool is
   var vect : id_set , n : nat in
      for n := 1 while n <= length (vectors) by n := n+1 loop
         vect := (access (vectors , n ));
         if isin (vect , rdytask ) then
            return true
```

```
        end if
      end loop ;
      return false
  end var
end function

function list_rdv_index (rdytask : id_set, vectors : sync_vect_list) : nat_set is
  var vect : id_set, n : nat, result : nat_set in
      result := {};
      for n := 1 while n <= length (vectors) by n := n+1 loop
        vect := (access (vectors, n));
        if isin (vect, rdytask) then
          result := insert (n, result)
        end if
      end loop ;
      return result
  end var
end function

function lock_state (in out manager : manager_state) raises invalid_state : none is
  case manager in
      free          -> manager := locked
    | autolock_free -> manager := autolock_locked
    | any           -> raise invalid_state
  end case
end function

function get_sync_vect (lock : lock, gsm : sync_map) : id_set is
  var g : DLC_ID, n, index : nat in
      g := get_gate (get_action (lock));
      index := get_index (lock);
      for n := 1 while n <= length (gsm) by n := n+1 loop
        if get_gate (access (gsm, n)) == g then
          return access (get_vect_list ( access (gsm, n)), index)
        end if
      end loop ;
      return {} of id_set
  end var
end function

function next_task ( task : DLC_ID, vect : id_set) : DLC_ID is
  var n : nat in
      for n := 1 while n < length (vect) by n := n+1 loop
        if task == access (vect, n) then
          return access (vect, n+1)
        end if
      end loop ;
      return DLC_NULL_ID
  end var
end function

function update_purge (in out purgel : id_list, purge : id_list, in out autolock : id_set) is
  var id : DLC_ID, newpurge : id_list in
      purgel := union (purgel, purge);
      newpurge := {};
      while not (empty (purgel)) loop
        id := head (purgel);
```

```
          if member (id, autolock) then
             autolock := remove (id, autolock)
          else
             newpurge := cons (id, newpurge)
          end if;
          purgel := tail (purgel)
       end loop;
       purgel := newpurge
    end var
end function

-- CHANNELS

channel com is
   (DLC_ID, message)
end channel

channel annonce is
   (DLC_ID, id_set)
end channel
```

Appendix A.2. Generic model of the Gate Process

```
process GATE [SEND, RECV : com, ACTION, HOOK_REFUSE : annonce]
             (gate : DLC_ID, vectors : sync_vect_list)
is
   var
      state       : gate_state,
      readyset    : id_set,     -- ready tasks
      autolock    : id_set,     -- autolocked tasks
      dealreadyset : id_set,    -- tasks ready during a negotiation
      dealautolock : id_set,    -- tasks autolocked during a negotiation
      dealvect    : id_set,     -- current negotiation synchro vector
      dealindex   : nat,        -- current negotiation synchro vector index
      dealpath    : id_set,     -- current negotiation lock chain
      purgelist   : id_list,    -- tasks to purge
      -- temporary variables
      n           : nat,
      task        : DLC_ID,
      lock        : lock,
      confirm     : bool,
      purge       : id_list,
      autolocked  : bool,
      vectindexes : nat_set
   in
      -- initialization
      state       := idle;
      readyset    := {};
      autolock    := {};
      dealreadyset := {};
      dealautolock := {};
      dealvect    := {};
      purgelist   := {};
      dealpath    := {};

      -- main loop
      loop
```

```
      select
         -- Receive READY message
         RECV (?task, ?READY (autolocked));
         if member (task, purgelist ) and (autolocked) then
            -- purge : ignore the autolock field
            purgelist := delete (task, purgelist );
            autolocked := false
         end if;
         if state == dealing then
            dealreadyset := insert (task, dealreadyset );
            if autolocked then
               dealautolock := insert (task, dealautolock)
            end if
         else
            readyset := insert (task, readyset );
            if autolocked then
               autolock := insert (task, autolock)
            end if
         end if
      []
         -- Start a negotiation
         only if (state == idle) and (possible_rdv (readyset, vectors )) then
            vectindexes := list_rdv_index (readyset, vectors );
            -- Choose randomly among possible synchronizations
            dealindex := any nat where member (dealindex, vectindexes );
            dealvect  := access (vectors, dealindex );
            dealpath  := diff (dealvect, autolock );
            if empty (dealpath) then
               -- All tasks are autolocked
               select
                  -- Post-negotiation hook may refuse the action
                  HOOK_REFUSE (gate, dealvect)
               []
                  ACTION (gate, dealvect);
                  for n := 1 while n <= length (dealvect) by n := n+1 loop
                     SEND (access (dealvect, n), COMMIT)
                  end loop;
                  readyset := diff (readyset, dealvect );
                  autolock := diff (autolock, dealvect)
               end select
            else
               -- Lauch a lock request
               task := head (dealpath );
               -- Simulate hook presence: randomly require confirmation
               confirm := any bool;
               SEND (task, LOCK (lock (action(gate), dealindex, dealpath,
                                       confirm, {})));
               dealreadyset := {};
               dealautolock := {};
               state := dealing
            end if
         end if
      []
         -- Receive a COMMIT message
         only if state == dealing then
            RECV (?task, ?COMMIT (purge) of message);
            readyset := diff (readyset, dealvect );
```

```
                readyset := union (readyset, dealreadyset);
                readyset := remove (task, readyset);
                autolock := diff (autolock, dealvect);
                autolock := union (autolock, dealautolock);
                autolock := remove (task, autolock);
                eval update_purge (!?purgelist, purge, !?autolock);
                state := idle
            end if
         []
            -- Receive an ABORT message
            only if state == dealing then
                RECV (?task, ?ABORT (purge) of message);
                readyset := remove (task, readyset);
                readyset := union (readyset, dealreadyset);
                autolock := remove (task, autolock);
                autolock := union (autolock, dealautolock);
                eval update_purge (!?purgelist, purge, !?autolock);
                state := idle
            end if
         []
            -- Receive a LOCK message
            only if state == dealing then
                RECV (?task, ? LOCK (lock) of message);
                select
                    HOOK_REFUSE (gate, dealvect);
                    for n := 1 while n <= length (dealpath) by n := n+1 loop
                        SEND (access (dealpath, n), ABORT)
                    end loop;
                    readyset := union (readyset, dealreadyset);
                    autolock := union (autolock, dealautolock)
                []
                    ACTION (gate, dealvect);
                    for n := 1 while n <= length (dealvect) by n := n+1 loop
                        SEND (access (dealvect, n), COMMIT)
                    end loop;
                    readyset := diff (readyset, dealvect);
                    readyset := union (readyset, dealreadyset);
                    readyset := remove (task, readyset);
                    autolock := diff (autolock, dealvect);
                    autolock := union (autolock, dealautolock);
                    autolock := remove (task, autolock)
                end select;
                eval update_purge (!?purgelist, lock.purge, !?autolock);
                state := idle
            end if
         end select
      end loop
   end var
end process
```

Appendix A.3. Generic Model of the Manager Process

```
process MANAGER [SEND, RECV : com, ACTION : annonce]
                (task : DLC_ID, statespace : state_list , map : sync_map)
is
   var
      manager      : manager_state,
```

```
      actions      : action_set ,     -- task currently possible actions
      arriv_list   : arrival_list ,   -- list of (action, state destination)
      taskstate    : nat,             -- current state of task
      waitlock     : lock_list ,      -- pending locks
      lock         : lock,            -- active lock
      action       : dlc_action,      -- next action to realize
      internal     : bool,            -- task can do an internal action
      sigpurge     : bool,            -- must add ourself to the purge
      -- temporary variables
      n            : nat,
      l            : lock,
      to, gate     : DLC_ID,
      vect         : id_set
   in
      -- initialization
      taskstate := 0;
      waitlock  := {};

      -- main loop
      loop
         -- Manager setup w.r.t. task current state
         manager  := free;
         internal := false;
         action   := action (DLC_NULL_ID);
         sigpurge := false;
         actions  := possible_actions (statespace, taskstate);

         -- For equivalence relation reasons, when a task can reach
         -- different state with the same action, the destination state
         -- must be decided before the negotiation
         arriv_list  := {};
         for n := 1 while n <= length(actions) by n := n+1 loop
            var dest_set : nat_set, dest : nat, act : dlc_action in
               act := access (actions, n);
               dest_set := get_next (statespace, taskstate, act);
               -- Choose randomly a destination state
               dest := any nat where member (dest, dest_set);
               arriv_list := cons ( arrival (act, dest), arriv_list )
            end var
         end loop;
         if (length (actions) == 1)
            and ((get_gate (access (actions, 1))) != DLC_GATE_I)
         then
            -- autolock
            action := access (actions, 1);
            SEND (action.gate, READY (true));
            manager := autolock_free;
            sigpurge := true
         else
            for n := 1 while n <= length (actions) by n := n+1 loop
               gate := get_gate (access (actions, n));
               if (gate == DLC_GATE_I) then
                  internal := true
               else
                  SEND (gate, READY (false))
               end if
            end loop
```

```
    end if;

    loop NEGOTIATION in
        select
            -- Receive a LOCK message
            RECV (? any DLC_ID, ?LOCK (l) of message);
            waitlock := append (l, waitlock)
        []
            -- Treat oldest pending lock
            only if not (empty (waitlock))
                and ((manager == free) or (manager == autolock_free))
            then
                lock        := head (waitlock);
                waitlock    := tail (waitlock);
                if member (lock.action, actions) then
                    if (manager == autolock_free) and (sigpurge) then
                        lock := lock.{purge => cons (task, lock.purge)};
                        sigpurge := false
                    end if;
                    action := lock.action;
                    if task == access (lock.path, length (lock.path)) then
                        -- We are the last task of the lock chain
                        if lock.confirm then
                            SEND (lock.action.gate, LOCK (lock));
                            eval lock_state (!?manager)
                        else
                            -- Conclude negotiation
                            vect := get_sync_vect (lock, map);
                            ACTION (lock.action.gate, vect);
                            SEND (lock.action.gate, COMMIT (lock.purge));
                            for n := 1 while n <= length(vect) by n := n+1 loop
                                to := access(vect, n);
                                if to != task then
                                    SEND (to, COMMIT)
                                end if
                            end loop;
                            break NEGOTIATION
                        end if
                    else
                        -- Forward lock request
                        to := next_task (task, lock.path);
                        SEND (to, LOCK (lock));
                        eval lock_state (!?manager)
                    end if
                else
                    -- Reject lock request
                    SEND (lock.action.gate, ABORT (lock.purge));
                    for n := 1 while n <= length (lock.path) by n := n+1 loop
                        to := access (lock.path, n);
                        if to < task then
                            SEND (to, ABORT)
                        end if
                    end loop
                end if
            end if
        []
            -- Receive a COMMIT message
```

```
                only if manager != free then
                    RECV (? any DLC_ID, COMMIT);
                    break NEGOTIATION
                end if
            []
                -- Receive an ABORT message
                only if (manager == locked) or (manager == autolock_locked) then
                    RECV (? any DLC_ID, ABORT);
                    if manager == locked then
                        manager := free
                    elsif manager == autolock_locked then
                        manager := autolock_free
                    end if
                end if
            []
                -- Realize an internal action
                only if (manager == free) and (internal) then
                    ACTION (DLC_GATE_I, {task} of id_set);
                    action := action (DLC_GATE_I);
                    break NEGOTIATION
                end if
            end select
        end loop; -- NEGOTIATION
        -- Reject pending locks
        while not (empty (waitlock)) loop
            l := head (waitlock);
            waitlock := tail (waitlock);
            SEND (l.action.gate, ABORT (l.purge));
            for n := 1 while n < length (l.path) by n := n+1 loop
                to := access (l.path, n);
                if to < task then
                    SEND (to, ABORT)
                end if
            end loop
        end loop;
        -- Task moves to next state
        taskstate := arrival_state ( arriv_list , action)
    end loop -- MAIN
  end var
end process
```

Appendix A.4. Generic Model of a Communication Buffer

```
-- Buffer size is a parameter
function BUFSIZE : nat is
    return 3
end function

-- Buffer acts as a FIFO (models TCP)
process BUFFER [GETFROM, SENDTO : com] (from, to : DLC_ID) is
    var
        msg : message,
        mq : message_list
    in
        mq := {};
        loop
            select
```

```
            only if length (mq) < BUFSIZE then
               GETFROM (to, ?msg);
               mq := append (msg, mq)
            end if
         []
            only if not (empty (mq)) then
               SENDTO (from, head (mq));
               mq := tail (mq)
            end if
         end select
      end loop
   end var
end process
```

Appendix A.5. Example of LNT Implementation Model Generated from a System Instance

Consider the following system:

```
process T1 [A,B: any] is
   A ;
   B
end process
```

```
process T2 [A,B: any] is
   select
      A
   [] B
   end select
end process
```

```
par A in
   T1[A,B]
|| T2[A,B]
end par
```

Our validation approach can automatically generate the LNT model of the implementation of this system. First, the system characteristics (identifiers, task state space, and synchronization vectors) are defined:

```
type DLC_ID is
   DLC_TASK_0_T1,
   DLC_TASK_1_T2,
   DLC_GATE_A,
   DLC_GATE_B,
   DLC_NULL_ID
   with "==", "!=", "<"
end type

function task_T1_state_space : state_list is
return {
  state (0, { transition ( action(DLC_GATE_A), {1})}),
  state (1, { transition ( action(DLC_GATE_B), {2})}),
  state (2, {} of transition_list (* deadlock *))
}
end function

function task_T2_state_space : state_list is
return {
  state (0, { transition ( action(DLC_GATE_A), {1}),
              transition ( action(DLC_GATE_B), {1})}),
  state (1, {} of transition_list (* deadlock *) )
}
```

```
end function

function gate_A_sync_vect : sync_vect_list is
  return {{ DLC_TASK_0_T1, DLC_TASK_1_T2 }}
end function

function gate_B_sync_vect : sync_vect_list is
  return {{ DLC_TASK_0_T1 },
          { DLC_TASK_1_T2 }}
end function

function global_sync_map : sync_map is
  return {
  sync_map_entry (dlc_gate_A, gate_A_sync_vect),
  sync_map_entry (dlc_gate_B, gate_B_sync_vect)
}
end function
```

Then, the implementation consists of managers, gates, and FIFO buffers running in parallel. The main process of the implementation model is thus:

```
process MAIN [TASK_0_T1_SEND, TASK_0_T1_RECV,
              TASK_1_T2_SEND, TASK_1_T2_RECV,
              GATE_A_SEND, GATE_A_RECV,
              GATE_B_SEND, GATE_B_RECV: com,
              ACTION, HOOK_REFUSE: annonce]
is
  par TASK_0_T1_SEND, TASK_0_T1_RECV,
      TASK_1_T2_SEND, TASK_1_T2_RECV,
      GATE_A_SEND, GATE_A_RECV,
      GATE_B_SEND, GATE_B_RECV,
  in
    par
         BUFFER [TASK_0_T1_SEND, TASK_1_T2_RECV] (DLC_TASK_0_T1, DLC_TASK_1_T2)
      || BUFFER [TASK_1_T2_SEND, TASK_0_T1_RECV] (DLC_TASK_1_T2, DLC_TASK_0_T1)
      || BUFFER [TASK_0_T1_SEND, GATE_A_RECV] (DLC_TASK_0_T1, DLC_GATE_A)
      || BUFFER [GATE_A_SEND, TASK_0_T1_RECV] (DLC_GATE_A, DLC_TASK_0_T1)
      || BUFFER [TASK_0_T1_SEND, GATE_B_RECV] (DLC_TASK_0_T1, DLC_GATE_B)
      || BUFFER [GATE_B_SEND, TASK_0_T1_RECV] (DLC_GATE_B, DLC_TASK_0_T1)
      || BUFFER [TASK_1_T2_SEND, GATE_A_RECV] (DLC_TASK_1_T2, DLC_GATE_A)
      || BUFFER [GATE_A_SEND, TASK_1_T2_RECV] (DLC_GATE_A, DLC_TASK_1_T2)
      || BUFFER [TASK_1_T2_SEND, GATE_B_RECV] (DLC_TASK_1_T2, DLC_GATE_B)
      || BUFFER [GATE_B_SEND, TASK_1_T2_RECV] (DLC_GATE_B, DLC_TASK_1_T2)
    end par
  ||
    par
       MANAGER [TASK_0_T1_SEND, TASK_0_T1_RECV, ACTION]
         (DLC_TASK_0_T1, task_T1_state_space, global_sync_map)
    || MANAGER [TASK_1_T2_SEND, TASK_1_T2_RECV, ACTION]
         (DLC_TASK_1_T2, task_T2_state_space, global_sync_map)
    || GATE [GATE_A_SEND, GATE_A_RECV, ACTION, HOOK_REFUSE]
         (DLC_GATE_A, gate_A_sync_vect)
    || GATE [GATE_B_SEND, GATE_B_RECV, ACTION, HOOK_REFUSE]
         (DLC_GATE_B, gate_B_sync_vect)
    end par
  end par
end process
```

[1] Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W., 2004. TIMES: A tool for schedulability analysis and code generation of real-time systems. In: Formal Modeling and Analysis of Timed Systems. Springer, pp. 60–72.

[2] Arbab, F., 2004. Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science 14 (3), 329–366.

[3] Bagrodia, R., 1989. Process synchronization: Design and performance evaluation of distributed algorithms. IEEE Transactions on Software Engineering 15 (9), 1053–1065.

[4] Behrmann, G., Larsen, K. G., Moller, O., David, A., Pettersson, P., Yi, W., 2001. Uppaal — present and future. In: Proceedings of the 40th IEEE Conference on Decision and Control. Vol. 3. IEEE, pp. 2881–2886.

[5] Bensalem, S., Griesmayer, A., Legay, A., Nguyen, T., Sifakis, J., Yan, R., 2011. D-Finder 2: Towards efficient correctness of incremental design. In: Proceedings of the 3rd NASA International Symposium on Formal Methods (NFM'2011), Pasadena, CA, USA. pp. 453–458.

[6] Bergamini, D., Descoubes, N., Joubert, C., Mateescu, R., Apr. 2005. Bisimulator: A modular tool for on-the-fly equivalence checking. In: Halbwachs, N., Zuck, L. (Eds.), Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2005 (Edinburgh, Scotland, UK). Vol. 3440 of Lecture Notes in Computer Science. Springer, pp. 581–585.

[7] Berry, G., 2007. SCADE: Synchronous design and validation of embedded control software. In: Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems. Springer, pp. 19–33.

[8] Bochmann, G., Gao, Q., Wu, C., 1989. On the distributed implementation of LOTOS. In: Proceedings of the 2nd International Conference on Formal Description Techniques (FORTE'89). pp. 133–146.

[9] Bonakdarpour, B., Bozga, M., Quilbeuf, J., 2013. Model-based implementation of distributed systems with priorities. Design Autom. for Emb. Sys. 17 (2), 251–276.

[10] Bouajjani, A., Fernandez, J.-C., Graf, S., Rodríguez, C., Sifakis, J., 1991. Safety for branching time semantics. In: Proceedings of 18th ICALP. Springer.

[11] Carbone, M., Montesi, F., 2013. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In: Giacobazzi, R., Cousot, R. (Eds.), Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13), Rome, Italy. ACM, pp. 263–274.

[12] Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., McKinty, C., Powazny, V., Lang, F., Serwe, W., Smeding, G., 2015. Reference manual of the LNT to LOTOS translator (version 6.2), INRIA/VASY and INRIA/CONVECS, 130 pages.

[13] Chandy, K. M., Misra, J., 1988. Parallel program design: A foundation. Addison-Wesley.

[14] Coste, N., Garavel, H., Hermanns, H., Lang, F., Mateescu, R., Serwe, W., 2010. Ten years of performance evaluation for concurrent systems using CADP. In: Margaria, T., Steffen, B. (Eds.), Proceedings of the 4th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation ISoLA 2010 (Amirandes, Heraclion, Crete), Part II. Vol. 6416 of Lecture Notes in Computer Science. Springer, pp. 128–142.

[15] Dijkstra, E. W., 1965. Solution of a problem in concurrent programming control. Commun. ACM 8 (9), 569–570.

[16] Dijkstra, E. W., 1975. Guarded commands, non-determinacy and formal derivation of programs. Communication of the ACM 18 (8), 453–457.

[17] Dokter, K., Jongmans, S. T. Q., Arbab, F., Bliudze, S., 2015. Relating BIP and reo. In: Knight, S., Lanese, I., Lluch-Lafuente, A., Vieira, H. T. (Eds.), Proceedings of the 8th Interaction and Concurrency Experience (ICE'2015), Grenoble, France. Vol. 189 of EPTCS. pp. 3–20.

[18] Evrard, H., 2015. Génération automatique d'implémentation distribuée à partir de modèles formels de processus concurrents asynchrones. Thesis, Université Grenoble Alpes.
URL `https://hal.inria.fr/tel-01215634`

[19] Evrard, H., 2016. DLC: Compiling a Concurrent System Formal Specification to a Distributed Implementation. In: Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2016. Lecture Notes in Computer Science. Springer-Verlag.
URL `https://hal.inria.fr/hal-01250925`

[20] Evrard, H., Lang, F., 2013. Formal verification of distributed branching multiway synchronization protocols. In: Beyer, D., Boreale, M. (Eds.), Proceedings of the IFIP Joint International Conference on Formal Techniques for Distributed Systems (FORTE/FMOODS'2013), Florence, Italy. Vol. 7892 of Lecture Notes in Computer Science. IFIP, Springer, pp. 146–160.

[21] Evrard, H., Lang, F., 2015. Automatic distributed code generation from formal models of asynchronous concurrent processes. In: Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'2015), Turku, Finland. IEEE.
URL `https://hal.inria.fr/hal-01086522`

[22] Garavel, H., 2008. Reflections on the future of concurrency theory in general and process calculi in particular. In: Palamidessi, C., Valencia, F. D. (Eds.), Proceedings of the LIX Colloquium on Emerging Trends in Concurrency Theory (Ecole Polytechnique de Paris, France), November 13–15, 2006. Vol. 209 of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, pp. 149–164, also available as INRIA Research Report RR-6368.

[23] Garavel, H., Lang, F., Aug. 2001. SVL: a scripting language for compositional verification. In: Kim, M., Chin, B., Kang, S., Lee, D. (Eds.), Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea). IFIP, Kluwer Academic Publishers, pp. 377–392, full version available as INRIA Research Report RR-4223.

[24] Garavel, H., Lang, F., Mateescu, R., 2002. Compiler construction using LOTOS NT. In: Horspool, N. (Ed.), Proceedings of the 11th International Conference on Compiler Construction (CC'2002), Grenoble, France. Vol. 2304 of Lecture Notes in Computer Science. Springer, pp. 9–13.

[25] Garavel, H., Lang, F., Mateescu, R., Apr. 2015. Compositional Verification of Asynchronous Concurrent Systems Using CADP. Acta Informatica 52 (4), 337–392.

[26] Garavel, H., Lang, F., Mateescu, R., Serwe, W., 2013. CADP 2011: A toolbox for the construction and analysis of distributed processes. Springer International Journal on Software Tools for Technology Transfer (STTT) 15 (2), 89–107.

[27] Garavel, H., Sighireanu, M., 1999. A graphical parallel composition operator for process algebras. In: Wu, J., Gao, Q., Chanson, S. T. (Eds.), Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'99), Beijing, China. IFIP, Kluwer Academic Publishers, pp. 185–202.

[28] Garavel, H., Viho, C., Zendri, M., 2001. System design of a CC-Numa multiprocessor architecture using formal specification, model-checking, co-simulation, and test generation. Springer International Journal on Software Tools for Technology Transfer 3 (3), 314–331, also available as INRIA Research Report RR-4041.

[29] Havender, J., 1968. Avoiding deadlock in multitasking systems. IBM systems journal 7 (2), 74–84.

[30] Holzmann, G. J., 2004. The SPIN model checker: Primer and reference manual. Vol. 1003. Addison-Wesley Reading.

[31] ISO/IEC, 1989. LOTOS — A formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Geneva.

[32] ISO/IEC, 2001. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, Geneva.

[33] Jard, C., Jéron, T., Aug. 2005. Tgv: Theory, principles and algorithms — a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. Springer International Journal on Software Tools for Technology Transfer (STTT) 7 (4), 297–315.

[34] Jongmans, S. T. Q., Santini, F., Arbab, F., 2014. Partially-distributed coordination with Reo. In: Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'2014), Torino, Italy. pp. 697–706.

[35] Katz, G., Peled, D., 2010. Code mutation in verification and automatic code correction. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2010). Springer, pp. 435–450.

[36] Kumar, D., 1990. An implementation of n-party synchronization using tokens. In: Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS'1990), Paris, France. pp. 320–327.

[37] Lamport, L., 2001. Paxos made simple. ACM Sigact News 32 (4), 18–25.

[38] Lang, F., 2005. EXP.OPEN 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In: van de Pol, J., Romijn, J., Smith, G. (Eds.), Proceedings of the 5th International Conference on Integrated Formal Methods (IFM'2005), Eindhoven, The Netherlands. Vol. 3771 of Lecture Notes in Computer Science. Springer, pp. 70–88, full version available as INRIA Research Report RR-5673.

[39] Lantreibecq, E., Serwe, W., 2011. Model checking and co-simulation of a dynamic task dispatcher circuit using CADP. In: Salaün, G., Schätz, B. (Eds.), Proceedings of the 16th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2011), Trento, Italy. Vol. 6959 of Lecture Notes in Computer Science. Springer, pp. 180–195.

[40] Lockefeer, L., Williams, D. M., Fokkink, W. J., 2016. Formal specification and verification of TCP extended with the window scale option. Science of Computer Programming 118, 3–23.

[41] Löffler, S., 1996. From specification to implementation: A PROMELA to C compiler. Project Report Ecole Nationale Supérieure des Télécommunications.

[42] Mañas, J. A., de Miguel, T., Salvachúa, J., Azcorra, A., 1993. Tool support to implement LOTOS formal specifications. Computer Networks and ISDN Systems 25 (7), 815–839.

[43] Mateescu, R., Garavel, H., Jul. 1998. Xtl: A meta-language and tool for temporal logic model-checking. In: Margaria, T. (Ed.), Proceedings of the International Workshop on Software Tools for Technology Transfer STTT'98 (Aalborg, Denmark). BRICS, pp. 33–42.

[44] Mateescu, R., Sighireanu, M., Mar. 2003. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Sci. Comput. Programming 46 (3), 255–281.

[45] Mateescu, R., Thivolle, D., 2008. A model checking language for concurrent value-passing systems. In: Cuellar, J., Maibaum, T., Sere, K. (Eds.), Proceedings of the 15th International Symposium on Formal Methods (FM'08), Turku, Finland. Vol. 5014 of Lecture Notes in Computer Science. Springer, pp. 148–164.

[46] Montesi, F., Yoshida, N., 2013. Compositional choreographies. In: D'Argenio, P. R., Melgratti, H. C. (Eds.), Proceedings of the 24th International Conference on Concurrency Theory (CONCUR'13) Buenos Aires, Argentina. Vol. 8052 of Lecture Notes in Computer Science. Springer, pp. 425–439.

[47] Nestmann, U., Pierce, B. C., 1996. Decoding choice encodings. In: Montanari, U., Sassone, V. (Eds.), Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96), Pisa, Italy. Vol. 1119 of Lecture Notes in Computer Science. Springer, pp. 179–194.

[48] Ng, N., Yoshida, N., 2014. Pabble: Parameterised scribble for parallel programming. In: 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'2014), Torino, Italy. IEEE Computer Society, pp. 707–714.

[49] Oliveira, M. V. M., de Medeiros Júnior, I. S., Woodcock, J., 2013. A verified protocol to implement multi-way synchronisation and interleaving in CSP. In: Hierons, R. M., Merayo, M. G., Bravetti, M. (Eds.), Proceedings of the 11th International Conference on Software Engineering and Formal Methods (SEFM'2013), Madrid, Spain. Vol. 8137 of Lecture Notes in Computer Science. Springer, pp. 46–60.

[50] Ongaro, D., Ousterhout, J., 2013. Safety proof and formal specification for Raft.
URL `http://ramcloud.stanford.edu/~ongaro/raftproof.pdf`

[51] Ongaro, D., Ousterhout, J., 2014. In search of an understandable consensus algorithm. In: Proceedings of the USENIX Annual Technical Conference (USENIX ATC'2014). USENIX Association, Philadelphia, PA, pp. 305–319.
URL https://www.usenix.org/conference/atc14/technical-sessi%ons/presentation/ongaro

[52] Park, D., 1981. Concurrency and automata on infinite sequences. In: Deussen, P. (Ed.), Theoretical Computer Science. Vol. 104 of Lecture Notes in Computer Science. Springer, pp. 167–183.

[53] Parrow, J., Sjödin, P., 1996. Designing a multiway synchronization protocol. Computer communications 19 (14), 1151–1160.

[54] Pérez, J. A., Corchuelo, R., Toro, M., 2004. An order-based algorithm for implementing multiparty synchronization. Concurrency - Practice and Experience 16 (12), 1173–1206.

[55] Peters, K., Nestmann, U., Goltz, U., 2013. On distributability in process calculi. In: Felleisen, M., Gardner, P. (Eds.), Proceedings of the 22nd European Symposium on Programming (ESOP'2013), Rome, Italy. Vol. 7792 of Lecture Notes in Computer Science. Springer, pp. 310–329.

[56] Proença, J., Clarke, D., de Vink, E., Arbab, F., 2012. Dreams: A framework for distributed synchronous coordination. In: Proceedings of the 27th International Symposium on Applied Computing (SAC'2012), Trento, Italy. ACM.

[57] Quilbeuf, J., 2013. Distributed implementations of component-based systems with prioritized multiparty interactions. Ph.D. thesis, Université de Grenoble.

[58] Schneider, F. B., 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys 22 (4), 299–319.

[59] Sharma, A., 2013. A refinement calculus for PROMELA. In: Proceedings of the 18th International Conference on Engineering of Complex Computer Systems (ICECCS'2013). IEEE, pp. 75–84.

[60] Sighireanu, M., 1999. Contribution à la définition et à l'implémentation du langage "Extended LOTOS". Thèse de Doctorat, Université Joseph Fourier, Grenoble.

[61] Sisto, R., Ciminiera, L., Valenzano, A., 1991. A protocol for multirendezvous of LOTOS processes. IEEE Transactions on Computers 40 (4), 437–447.

[62] Sjödin, P., 1991. From LOTOS specifications to distributed implementations. Ph.D. thesis, Department of Computer Science, University of Uppsala (Sweden).

[63] Taubner, D., 1987. On the implementation of Petri nets. In: Rozenberg, G. (Ed.), Proceedings of the 8th European Workshop on Applications and Theory of Petri Nets, Zaragoza, Spain. Vol. 340 of Lecture Notes in Computer Science. Springer, pp. 418–434.

[64] Winkowski, J., 1983. A distributed implementation of Petri nets. Tech. Rep. 518 (1983), Polish Academy of Science, Institute of Computer Science, Warsaw.

[65] Yasumoto, K., Higashino, T., Taniguchi, K., 2001. A compiler to implement LOTOS specifications in distributed environments. Computer Networks 36 (2), 291–310.