Formal Verification of Distributed Branching Multiway Synchronization Protocols

Hugues Evrard and Frédéric Lang

CONVECS Team, Inria Grenoble Rhône-Alpes and LIG
(*Laboratoire d'Informatique de Grenoble*), Montbonnot, France*

Abstract. Distributed systems are hard to design, and formal methods help to find bugs early. Yet, there may still remain a semantic gap between a formal model and the actual distributed implementation, which is generally hand-written. Automated generation of distributed implementations requires an elaborate multiway synchronization protocol. In this paper, we explore how to verify correctness of such protocols. We generate formal models, written in the LNT language, of synchronization scenarios for three protocols and we use the CADP toolbox for automated formal verifications. We expose a bug leading to a deadlock in one protocol, and we discuss protocol extensions.

1 Introduction

Concurrent systems are hard to design, in particular distributed systems whose processes potentially run asynchronously, i.e., at independent speeds, possibly on remote machines. Formal methods, applied to formal system specifications, help to detect design flaws early in the development process. However, implementations are often hand-written, and a semantic gap may appear between a specification and its implementation. This can be palliated by tools which automatically generate a correct implementation from a formal specification.

We consider distributed systems consisting of several *tasks* that interact by *synchronization*. The specification of such systems will describe each task behavior as a nondeterministic process and the possible synchronizations between tasks through a parallel composition operator. As a particular specification language, we have in mind LOTOS NT (LNT for short) [5], a successor of LOTOS [16] and a variant of the E-LOTOS standard [17]. LNT has a general parallel composition operator [12] that enables *multiway synchronization* (also available in LOTOS), where a set of two or more tasks can synchronize altogether, and *m-among-n synchronization* (not available in LOTOS), where any subset of m tasks among a set of n tasks can synchronize altogether.

LNT is already equipped with formal verification tools packaged in the CADP toolbox [11]. In a close future, we would also like to automatically generate from an LNT specification a distributed implementation consisting of one sequential

* This work was partly funded by the French *Fonds national pour la Société Numérique* (FSN), Pôles Minalogic, Systematic and SCS (project OpenCloudware).

process per task plus a synchronization protocol, as much distributed as possible to avoid the obvious bottleneck that a centralized synchronizer would represent in large distributed systems. Preserving the semantics of the specification is of major importance. We need elaborate protocols since classical synchronization barriers [8] cannot handle branching synchronizations, i.e., the situation where a task is ready to synchronize on several gates nondeterministically.

Several distributed synchronization protocols exist (see Section 2), many of them handling branching multiway synchronization, but not m-among-n synchronization. Some of these protocols have been proven correct either by demonstrating by hand the satisfaction of some properties, or by verifying by hand the behavior equivalence with an ideal synchronizer. To our knowledge, none of them has been verified using computer-assisted tools yet. We explore how protocols correctness can be verified using computer-assisted verification tools, which would provide better confidence in their correctness.

The contribution of this paper is the following. We selected three protocols that seemed most appropriate to handle LNT synchronization, respectively proposed by Sjödin [27], Parrow & Sjödin [23] and Sisto, Ciminiera & Valenzano [26] (respectively referred as Sjödin's, Parrow's and Sisto's protocol for short). For each of these three protocols, we generate formal specifications and use model checking to verify absence of deadlocks and livelocks, and equivalence checking to verify synchronization consistency and characterize precisely the semantic relation between the specification and the implementation. We claim that, under the hypotheses stated at the time of its publication, Parrow's protocol can lead to a deadlock, which we illustrate by an example and for which we propose a fix. At last, we discuss the limitations of the three protocols to handle m-among-n synchronization, and we propose some enhancements.

Paper overview. Section 2 exposes the related work. Section 3 briefly presents the CADP toolbox. Section 4 introduces the three protocols under study. Section 5 explains how we generate formal specifications of protocols, and Section 6 lists the verifications we apply to these specifications. Section 7 discusses the results of protocol verifications, and describes the bug found in Parrow's protocol. Finally, Section 8 gives concluding remarks and directions for future work.

2 Related Work

There is an analogy between multiway synchronization and the Committee Coordination Problem [6] (CCP), where professors (tasks) may attend committees (synchronizations). A professor may attend any committee, a committee needs a predefined set of professors to be conveyed, and a professor can attend only one committee at a time. Committees sharing professors must be in mutual exclusion, and committees must be conveyed only if all professors are ready (readiness).

Chandy and Misra (C.&M.) propose a solution where the mutual exclusion is solved by mapping the problem to the Dining (or Drinking) Philosophers problem, and readiness is guaranteed by a shuffle of tokens [6]. Bagrodia presents the Event Manager (EM) algorithm, which uses a unique token cycling among

committees to ensure mutual exclusion, and counters (of professors ready announcements and committee attendances) to guarantee readiness [1]. In the same paper, Bagrodia also proposes the Modified Event Managers (MEM) algorithm using the Dining Philosophers for mutual exclusion.

Bonakdarpour *et al.* address distributed implementations for the BIP framework [2]. Multiway synchronization is handled by a software layer, in which theoretically any protocol can be fitted. Their implementations use either a central synchronizer, a token-ring protocol (inspired by the EM algorithm of Bagrodia) or a mapping to the Dining Philosophers. They discuss the correctness of the derived implementation, but not of the protocols themselves.

LNT multiway synchronization differs slightly from the CCP in two ways. First, a single committee may be conveyed with different sets of professors: this is not a big deal, since we can declare new committees for every such set of professors and fall back to the CCP. Note however that we might face combinatorial explosion of committees, e.g. in the case of m-among-n synchronization. Second, a professor may be ready on a different subset of committees, depending on its current state. This extension to the CCP is addressed by C.&M. and Bagrodia: professors alert only committees they are ready on, but these still require mutual exclusion from all possible conflicting committees.

C.&M. and Bagrodia's protocols are based on solutions for synchronization in concurrency problems. At the same period, attempts to derive an implementation from a LOTOS specification lead to other solutions. Sisto *et al.* suggest a synchronization-tree based protocol [26]. In his thesis, Sjödin introduces a solution where committees directly lock professors [27], and a few years later Parrow & Sjödin propose a variation [23]. Although not in the framework of LOTOS, Perez *et al.* explore a very similar approach more recently [25].

In this paper, our main focus is protocol correctness. The solutions of C.&M., Bagrodia, and Perez are proven correct by satisfaction of properties. Sisto *et al.* discuss complexity but not correctness of their protocol. Sjödin demonstrates the equivalence between an ideal coordinator and his distributed solution; Parrow & Sjödin adopt the same approach but give only an overview of the proof. All these verifications are manual. To our knowledge, there was no attempt at verifying such protocols using automated verification tools.

We selected Sisto's, Sjödin's and Parrow's protocols in our study because, as they were designed to coordinate LOTOS synchronizations, they seemed most appropriate to handle also the case of LNT synchronizations efficiently. Regarding correctness, we verify not only the absence of livelocks and deadlocks, but we also compare the protocols' behavior with the expected reference behavior, which is obtained using reliable verification tools of CADP.

3 The CADP Toolbox

CADP (*Construction and Analysis of Distributed Processes*) [11] is a toolbox for modeling and verifying asynchronous systems. The CADP toolbox provides, among others, the following languages, models, and tools .

High-level languages allow concurrent systems to be modeled as processes running asynchronously and communicating by rendezvous synchronization on communication actions. Historically, LOTOS [16] was the main language of CADP. It combines algebraic abstract data types to model types and functions in an equational style, and a process algebra inheriting from CCS [21] and CSP [15] to model processes. In recent years, LNT [5] was developed, providing an easier syntax closer to mainstream imperative and functional programming languages. Models written in LNT can be verified using CADP, via an automated translation into LOTOS. The semantics of a LOTOS or an LNT program are defined as an LTS (*Labeled Transition System*) [21], that is a graph whose transitions between states are labeled by actions denoting value-passing communications.

Intermediate-level models are representations of systems between high-level languages and low-level models. As such, the EXP.OPEN 2.0 [18] language for networks of communicating LTSs consists of LTSs composed using various operators, including LOTOS and LNT parallel composition. EXP.OPEN 2.0 is a key component of CADP for compositional verification.

Low-level models are representations of LTSs. CADP provides the BCG (*Binary Coded Graph*) format to represent an LTS explicitly (as a set of states and transitions), and the OPEN/CÆSAR environment [9] to represent an LTS implicitly (as a set of types and functions, including functions for enumerating the successor transitions of a given state), for on-the-fly verification.

Temporal logics allow behavioral properties to be defined. The MCL language [20] combines the alternation-free μ-calculus together with regular formulas, primitives to handle data, and useful fairness operators of alternation 2.

Model checkers and *equivalence checkers* are also available in CADP. The EVALUATOR 4.0 model checker [20] allows an MCL formula to be checked on the fly on a system modeled in any language or format available in CADP, through the OPEN/CÆSAR interface. The BISIMULATOR 2.0 equivalence checker [19] allows the equivalence of two systems to be checked on the fly, modulo several equivalence relations, including strong [22], branching [29], safety [3] or weak trace [4] equivalences.

At last, CADP allows complex verification scenarios to be described succinctly using the intuitive language SVL (*Script Verification Language*) [10]. An SVL script is translated by the SVL compiler into a Bourne Shell script, which invokes the appropriate CADP tools automatically.

4 Overview of Synchronization Protocols

We consider a distributed system to be specified as several *tasks* which interact with each others by synchronous rendezvous on *gates*. A task is defined by an LTS of which transition labels are gate identifiers. A parallel composition expression defines for each gate which sets of tasks are synchronizable on that gate. In this paper, we also name a parallel composition of tasks a *synchronization scenario*.

Figure 1 illustrates a distributed system made of four tasks $t1$, $t2$, $t3$, and $t4$, which synchronize on gates A, B and C. Each task is represented by an LTS, the

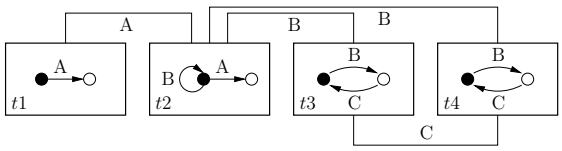


Fig. 1. A distributed system made of four tasks which synchronize on three gates.

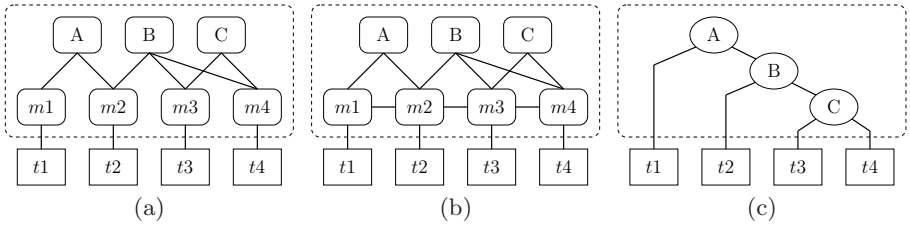


Fig. 2. Architecture of Sjödin's (a), Parrow's (b), and Sisto's (c) protocol.

black point denoting the initial state. Possible synchronizations are represented by lines labeled with a gate identifier. For instance, a synchronization on B involves either $t2$ and $t3$, or $t2$ and $t4$.

A synchronization protocol must guarantee mutual exclusion of synchronizations which involve common tasks, and that a synchronization happens only when all involved tasks are actually ready on it. For instance in Figure 1, $t2$ may synchronize on B with either $t3$ or $t4$, but cannot synchronize with both at the same time. Once $t2$ has synchronized on A, it will never be ready to synchronize on B again, so no other synchronization on B may occur.

In the sequel we briefly describe how the three protocols under study fulfill these requirements. For a complete and detailed explanation of their internals, we refer to original publications of the protocols [27, 23, 26]. Note that LNT also enables data exchange during rendezvous on gates, and guards on data values. We leave those aspects for future work, focusing here on synchronization. In addition, we assume that the composition of tasks is static, i.e., we do not consider the dynamic creation and deletion of tasks.

Sjödin's protocol overview. A *mediator* process is associated to each task, and a *port* process is associated to each gate. Ready tasks send a message to their mediator, which lets know the relevant ports. When a port has received enough ready messages, it tries to lock all mediators involved in a synchronization. If it succeeds, then the synchronization occurs and the port sends a confirmation to all locked mediators, which announce to their task on which gate the synchronization occurred. Otherwise, if one of the mediators has already been locked and confirmed by another port, then negotiation is aborted and the port releases all mediators it has locked so far. To avoid deadlocks, all ports lock mediators in the same order [14]. Figure 2 (a) illustrates the architecture of Sjödin's protocol on the example of Figure 1.

Parrow's protocol overview. Parrow's protocol is based on Sjödin's and adopts almost the same architecture, see Figure 2 (b). The locking process is different: a port starts by locking the first mediator, which is then responsible for locking the next one, etc. When the last mediator is locked, it announces synchronization success to the port and to other involved mediators, which inform their tasks. However, if a mediator refuses the lock, it directly informs the port, and tells the list of locked mediators to release themselves. Compared to Sjödin's, Parrow's protocol mediators communicate with each other, and the locking process is less centralized.

Sisto's protocol overview. The protocol is very tied to LOTOS since it is structured as a composition tree obtained from a LOTOS expression. For instance, the parallel composition of Figure 1 can be expressed as the LOTOS expression "`t1 |[A]| (t2 |[B]| (t3 |[C]| t4))`". Figure 2 (c) illustrates the composition tree obtained, where leaves are tasks. Tasks announce which gates they are ready on to their upper *node*. Nodes may control one or several gates, in which case they collect ready announcements for these gates; for other gates they propagate ready messages to their father node. If both children of a node are ready on a gate controlled by the node, it starts to lock both subtrees down to the tasks. If a synchronization already occurred in a subtree, then the lock refusal is propagated upward. When the node which started the negotiation receives a refusal, it aborts the negotiation and unlocks the other subtree. If both subtrees accept the lock, then the node sends a confirmation message to both subtrees and the synchronization is achieved.

In the example of Figure 1, if $t2$, $t3$ and $t4$ are all ready on B, then the B node sends a lock to $t2$ and to the C node. Here the C node must choose if it propagates the lock of B to either $t3$ or $t4$, but must not synchronize both of them since $t3$ and $t4$ are interleaving on B. So each node is characterized by the gates it *controls* (for which it starts negotiations), and the gates it *synchronizes* (for which both children must be ready to propagate ready upward, and both children must be locked). These two sets may be different, and a node always synchronizes a gate it controls.

5 Formal Specification of Protocols

The three of the above protocols are made of *protocol processes* (namely *nodes* for Sisto's protocol, and *mediators* and *ports* for Sjödin's and Parrow's) which interact with tasks. In this section, we explain how, from a synchronization scenario, we automatically generate a formal specification of tasks, protocol processes, and their interaction. Figure 3 gives an overview of our specification and verification approach. The approach is generic, and may be used to verify other synchronization protocols.

We assume that the *high-level* specification of a synchronization scenario consists of an LTS stored in BCG format for each task, and of an EXP.OPEN expression for the parallel composition of tasks. Because Sisto's protocol is tied to a LOTOS expression, for the time being we assume the composition expres-

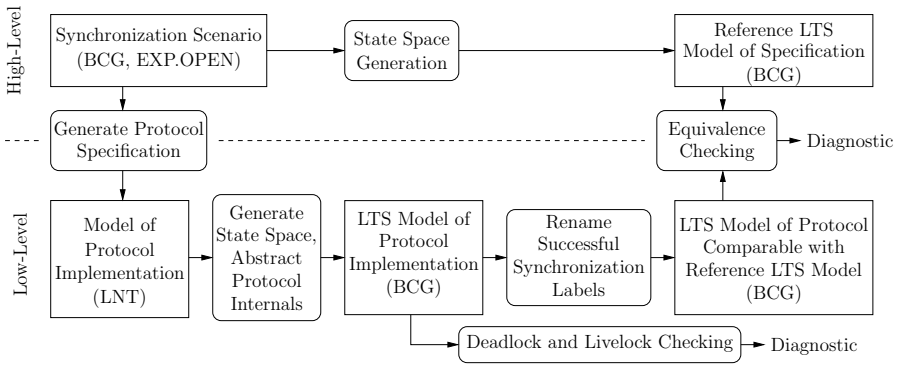


Fig. 3. Specification and verification steps in high and low level.

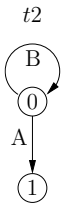
sion uses only LOTOS parallel composition[1]. This is our input to generate the *low-level* specification of the scenario, i.e., the model of the *implementation* of protocol processes, which manage synchronizations, and of tasks, which interact with protocol processes[2]. We write the low-level specification in LNT. We generate an LNT module for each task and for each protocol process. Moreover, a main module will compose tasks and protocol processes, along with LNT processes modeling the underlying network used in communications between tasks and protocol processes. Note that gates of high-level specification become data of message exchanges in low-level specification, and LNT gates in the low-level specification represent communication channels between low-level processes.

Low-level Tasks. When a task is ready to synchronize on one or more gates, it must exchange messages with some protocol processes until it receives the confirmation of a successful synchronization. Therefore, a synchronization transition in the high-level specification becomes a sequence of messages exchanged between task and protocol processes, as defined by the protocol interface. For each protocol, and each task, a different low-level specification is generated depending on the protocol interface. For instance, Figure 4 illustrates the low-level specification of $t2$ for Parrow's protocol interface. The task first sends a synchronization request on gate `M`, along with the list of high-level gates it is ready on. If a synchronization succeeds, then the synchronized gate is stored in variable `sync_gate`, and the state to go next is selected accordingly.

Protocol Processes. Each protocol process has a generic behavior which is precisely described in the protocol's original publication. We just transcript this behavior in an LNT module, once for all. These modules take arguments to specialize their behavior according to the synchronization scenario. For instance, in Sisto's protocol, arguments passed to nodes are the gates they control and the

[1] For instance, the EXP.OPEN composition expression corresponding to Figure 1 is: `"t1.bcg" |[A]| ("t2.bcg" |[B]| ("t3.bcg" |[C]| "t4.bcg"))`.

[2] Note that there is no relationship between the high and low levels of protocol models and the abstraction levels described in Section 3.




```
module task_t2 (data_types) is

process task_t2 [M: msg_channel] is    -- state 0
   var sync_gate: gate in
      M(request, {A, B});              -- send request to mediator
      M(confirm, ?sync_gate);          -- wait for confirmation
      case sync_gate in
         A -> task_t2_1[M]             -- synchro on A, go to state 1
      |  B -> task_t2[M]               -- synchro on B, go to state 0
      end case
   end var
end process

process task_t2_1 [M: msg_channel] is  -- state 1
   stop                                -- no outgoing transitions
end process

end module
```

Fig. 4. LNT code generated for a task using Parrow's protocol interface.

gates they synchronize. Moreover, in Sisto's protocol we introduce the `top_node` process which acts as a generic father for the root node.

Communications. The authors of the protocols assume that the underlying communication network is reliable (no messages are lost), and that tasks and protocol processes communicate via asynchronous message passing (i.e., sending and receiving the message are two distinct actions). Since LNT rendezvous is synchronous, we explicitly model communication buffers as LNT processes synchronizing with tasks and protocol processes.

Task and Protocol Process Composition. Finally, the main LNT process composes tasks, protocol processes and communication buffers in parallel. To model communication in a real network, this parallel composition uses only binary rendezvous between a communication buffer and either a task or a protocol process. Figure 5 illustrates the composition obtained from the example of Figure 1 for Sisto's protocol. For instance, a message from the `top_node` goes through a buffer via synchronization on `F0U` before reaching the destination node via a synchronization on `F0D`.

Tracing successful synchronization on gate EXT. In order to track which high-level synchronizations are achieved using the protocol, we represent the "external world" with a low-level gate called EXT. Protocol processes report successful synchronization on a high-level gate by sending a message on EXT.

6 Verification of Protocols

Figure 3 and the SVL script of Figure 6 summarize our verification approach. From the main module of the low-level specification, we generate a raw low-level LTS. In this LTS, a transition is labeled by either a protocol message or a synchronization announcement on gate EXT (e.g, "`EXT !A`" for a synchronization on gate A). For a given synchronization scenario and a given protocol, any pos-

```
module main_sisto (data_types, top_node, node, buffer, task_t1, task_t2,
                   task_t3, task_t4) is

process main [EXT, F0U, F0D, F1U, F1D, F2U, F2D, F3U, F3D, F4U, F4D,
              F5U, F5D, F6U, F6D: message] is
   par
      F0U            -> top_node[EXT, F0U]
   || F0U, F0D       -> buffer[F0U, F0D]
   || F0D, F1U, F2U -> node[EXT, F0D, F1U, F2U]({A}, nil of gate_set)
   || F1U, F1D       -> buffer[F1U, F1D]
   || F2U, F2D       -> buffer[F2U, F2D]
   || F1D            -> task_t1[F1D]
   || F2D, F3U, F4U -> node[EXT, F2D, F3U, F4U]({B}, nil of gate_set)
   || F3U, F3D       -> buffer[F3U, F3D]
   || F4U, F4D       -> buffer[F4U, F4D]
   || F3D            -> task_t2[F3D]
   || F4D, F5U, F6U -> node[EXT, F4D, F5U, F6U]({C}, nil of gate_set)
   || F5U, F5D       -> buffer[F5U, F5D]
   || F6U, F6D       -> buffer[F6U, F6D]
   || F5D            -> task_t3[F5D]
   || F6D            -> task_t4[F6D]
   end par
end process -- main

end module
```

Fig. 5. Main LNT process of Sisto's low-level specification for the example of Figure 1.

sible order of protocol message exchanges and synchronization announcements is represented by a path in this LTS.

Our first transformation is hiding all internal protocol messages. In the low-level LTS obtained, all protocol messages are now labeled "`i`", which is the convention label for *internal actions* in LNT. We then perform the following verifications.

Livelock detection. A livelock happens when low-level processes exchange messages indefinitely without agreeing on a synchronization, i.e., there exists somewhere in the low-level LTS a cycle of transitions which are only internal actions. Since this is the classical definition of a livelock, SVL comes with a built-in command to detect them (SVL actually calls the EVALUATOR4 tool of CADP with a predefined MCL formula that matches livelocks). If a livelock is detected, then a diagnostic, i.e., a path leading to a livelock, is stored in `diag_live.bcg`.

Deadlock detection. Generally, a deadlock is defined by a state which has no outgoing transitions. Note that this can be an expected behavior: for instance in Figure 1 once $t1$ has synchronized on A, it reaches a deadlock. Such kind of situations trigger deadlocks in the low-level LTS too, and these deadlocks are not due to protocol errors.

Nonetheless, a protocol may get stuck into a deadlock while a synchronization *could* have been reached. This is unacceptable as the protocol must be able to offer a synchronization as long as one exists in the high-level model. In the low-level LTS, such a situation is characterized by a state from which there exists both: a sequence of internal actions which leads to a deadlock state; and another sequence which eventually contains a synchronization announcement. The MCL formula falsified by such protocol deadlocks is "`[true*] ((< "i"* > [true] false) implies [true* . not("i")] false)`". MCL is a rich language, and for

```
(* Generate low-level LTS *)
"raw_lowlevel.bcg" = generation of "main.lnt";
(* Hide protocol messages *)
"lowlevel.bcg" = hide all but "EXT.*" in "raw_lowlevel.bcg";

(* Model checking: livelock and deadlock *)
"diag_live.bcg" = livelock of "lowlevel.bcg";
"diag_dead.bcg" = verify "deadlock.mcl" in "lowlevel.bcg";

(* Generate reference LTS from high-level spec *)
"reference.bcg" = generation of "composition.exp";
(* Rename synchronization announcements *)
"renamed.bcg" = total rename "EXT !\(.*\)" -> "\1" in "lowlevel.bcg";

(* Equivalence checking: branching, safety, weaktrace *)
"diag_branching.bcg" = branching comparison "renamed.bcg" == "reference.bcg";
"diag_safety.bcg" = safety comparison "renamed.bcg" == "reference.bcg";
"diag_weaktrace.bcg" = weak trace comparison "renamed.bcg" == "reference.bcg";
```

Fig. 6. Generic SVL script for verification operations

the sake of brevity we do not explain the semantics of MCL constructions used in this formula. For more details, please refer to [20].

This MCL formula is stored in file `deadlock.mcl`, and it is evaluated on the low-level LTS (again, using EVALUATOR4 underneath). If a deadlock is found, the diagnostic is stored in `diag_dead.bcg`.

Synchronization Consistency. A synchronization protocol must not only be deadlock and livelock free, but it must also synchronize tasks correctly, so we finally have to verify synchronization consistency. We naturally use the *high-level LTS* generated from the high-level specification (the EXP.OPEN and tasks' BCG files) as a reference, i.e., we consider this LTS to actually represent which synchronizations are possible for this scenario, and we compare this high-level LTS with the low-level LTS using equivalence checking. To do so, labels of high-level and low-level LTS must be comparable. We rename low-level LTS labels using a simple regular expression, such that for instance the label "`EXT !A`" is renamed to "`A`". Now both LTSs have the same labels for task synchronizations, and the low-level one also contains internal actions representing protocol messages.

Several equivalence relations correspond to different ways of abstracting away internal actions. We use, in decreasing order of strength, the branching [29], safety [3] and weak trace [4] equivalence relations. The SVL script calls the BISIMULATOR tool, which compares LTSs and, in case of differences, provides a diagnostic showing a behavior possible in an LTS and not in the other.

7 Analysis of Protocol Verifications

The goal of model checking is finding bugs, rather than proving correctness. To this aim, we wrote a bench of 51 synchronization scenarios, trying to cover a wide range of parallel compositions rather than a wide range of task behaviors. We thus focused on tasks containing up to no more than three states and ten transitions and we varied the number of concurrent tasks (from two up to four),

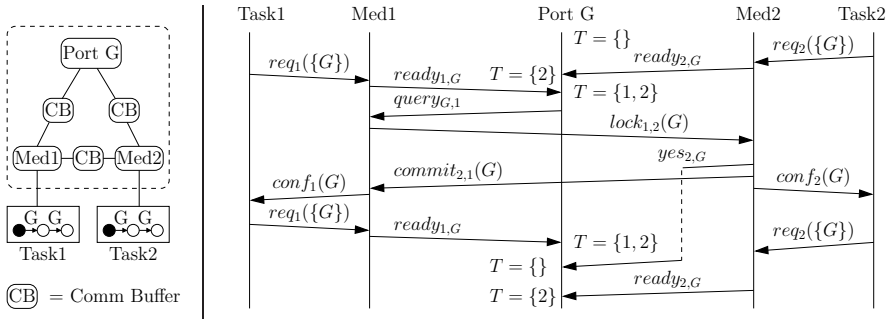


Fig. 7. Negotiation leading to a deadlock in Parrow's protocol.

the total number of gates (from zero up to four), the number of synchronized gates (from zero up to four), the number of gates simultaneously available in each task (from zero up to three), and the number of tasks synchronized on each gate (from one up to three). We obtain low-level LTSs with up to 500,000 states and 1,200,000 transitions.

Identification of Deadlocks in Parrow's Protocol. The test suite raised a design error that can lead to deadlocks. Figure 7 illustrates a scenario with two tasks Task1 and Task2 which synchronize two times on gate G. Figure 7 also exposes a negotiation leading to a deadlock after a first synchronization on G, whereas two synchronizations must occur. In the negotiation, we use the original notations of [23] for communication channels and data set names.

Each task notifies its own mediator with a request message to declare that it is ready on G. Mediators send ready messages to port G, which populates the set, called T, of tasks that are ready. When the port detects that both tasks are ready, it begins a negotiation by sending a *query* message to the mediator of Task1, called Med1. Med1 accepts the lock and sends a lock request to Med2. Med2 accepts the lock and sends a *yes* message to port G. We assume that this yes message is delayed (dashed line in Fig. 7), i.e., stored in a communication buffer and not consumed immediately.

Meanwhile, Med2 sends a *commit* message to Med1, and confirms successful synchronization on G to Task2. Med1 confirms to its task, and then both mediators receive new request messages. Med1 sends a ready message to port G, which accepts it. T is set to $T \cup \{1\}$, which actually leaves T unmodified.

Once the yes message from Med2 is received by port G, the set T is emptied so now $T = \{\}$. Med2 sends a ready notification, and T updates to $\{2\}$. In this situation, port G does not start a negotiation because T is not enough populated. However, ready requests of both mediators have already been received by the port. So we reach a deadlock, where a synchronization that could be successful (if the yes message had been received before the ready message of Med1) is not negotiated, and all protocol processes have stopped to communicate.

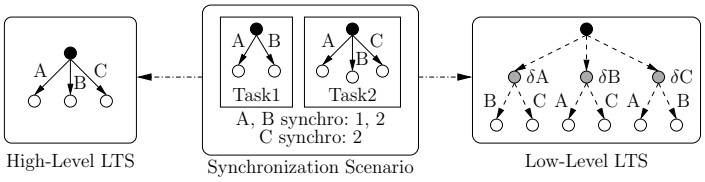


Fig. 8. High and low-level LTSs are not branching bisimilar.

Parrow's protocol can be fixed by separating the set which stores ready announcements (let's call it N) and the set which is used for a negotiation (we keep T). Every time a ready message is received by a port, the corresponding task is inserted in N. Before starting a negotiation, involved tasks are moved from N to T. If a yes message is received, T is emptied (ready messages received before the yes were stored in N). If a no message is received, the task refusing the lock is removed from T, and remaining tasks of T are inserted back in N before T is emptied. Using our test suite, we verified that this modification corrects the design flaw without triggering new issues.

Equivalence of High and Low Level Models. Comparison between high-level and low-level models gives information about their relations in terms of execution trees and execution sequences, modulo a transitive closure of internal actions. Weak trace equivalence indicates that every execution sequence of the high-level model is also an execution sequence in the low-level model, and conversely. Stronger relations, such as safety equivalence and branching bisimulation, give information about execution trees, i.e., not only about sequences of executed actions, but also on the choices of alternative actions that can be offered in the intermediate states.

In the three of these protocols, we observe that the models are equivalent modulo safety equivalence[3] (which obviously implies weak trace equivalence), but are not branching bisimilar. This indicates that every execution tree of the high-level model is also an execution tree of the low-level model, and conversely, but that some execution subtrees of the low-level model may be strictly contained in the high-level model.

This is illustrated by Figure 8: for the sake of brevity we consider synchronizations involving only one task (here on gate C), which is a limit case of synchronization. In the high-level LTS, the choice between all three possible synchronizations is made from a single state. The low-level LTS contains interleavings of protocol messages, represented by dashed arrows. During negotiation, the next synchronization may require several messages to be progressively selected, i.e., we may reach states where a synchronization on a particular gate cannot occur anymore (the gate has been "discarded", marked δ on the figure),

[3] In a manual proof, Sjödin and Parrow [24] use coupled simulation which, like safety equivalence, is a double simulation relation. We use the close but more standard safety equivalence, which is implemented in CADP.

but the choice remains between other synchronizations. Such intermediate states, grayed on the figure, have no bisimilar state in the high-level LTS.

For instance, consider Parrow's protocol on the scenario of Figure 8. If the first lock to happen is port A querying Med1, we reach a state where: if Med1 locks Med2 for A then A wins; or if port C locks Med2 then C wins (because both A and B need Med2, and C will not abort). Hence, we found a state where B will never happen but the choice between A or C remains.

Protocol Extension: m-among-n Synchronizations. So far, we considered high-level synchronizations to be specified by an EXP.OPEN expression using exclusively LOTOS parallel composition. In this section, we investigate how the protocols can manage m-among-n synchronizations offered by the LNT parallel composition operator. For instance, we write "`par A#2 in t1 || t2 || t3 end par`" to say that any group of 2 tasks among $t1$, $t2$ and $t3$ can synchronize on gate A. This cannot be directly expressed using LOTOS binary composition [12]. A way to implement this LNT operator is to flatten parallel composition by defining several sets of synchronizable tasks for each gate (*synchronization vectors*).

Sisto's protocol is so much tied to the tree structure of LOTOS expressions that it seems hard to make it manage m-among-n synchronizations without major design modifications.

Sjödin's protocol uses synchronization subtrees in its ready messages, and ports compose subtrees received from mediators to determine possible synchronizations. We replace these subtrees by simple lists of gates, and we give synchronization vectors as an argument to ports. Ports record ready announcements, and scan their synchronization vectors to detect possible synchronizations.

Parrow's protocol directly uses gate lists in its messages. However, each port is limited to only one synchronization vector, and in a locking sequence every mediator refers to this globally known vector to know what is the next mediator to lock. We extend port specifications to handle multiple synchronization vectors, and to send the relevant vector along lock requests. We also modify mediators to scan lock messages in order to know what is the next mediator to lock.

We verified that the modified versions of Sjödin's and Parrow's protocol still successfully handle synchronization scenarios of our test suite, plus a few more tests using m-among-n synchronizations.

Synchronization vectors are a direct and explicit way to express possible synchronizations. However, with nested parallel composition operators and m-among-n synchronizations, the number of synchronization vectors for a single gate can easily explode. To avoid this, we could use an equivalently expressive but more symbolic expression of possible synchronizations, such as the *synchronizers* proposed in the ATLANTIF intermediate model [28].

8 Conclusion and Future Work

In this paper, we presented how, for a given synchronization scenario and a given protocol, we can generate a formal LNT model of the implementation with asynchronous communication. Using the CADP verification toolbox, we

spotted previously undetected deadlocks in Parrow's protocol (illustrated by an example), whereas we found no bug in Sjödin's and Sisto's protocols. To our knowledge, this is the first attempt at verifying such synchronization protocols using automated verification tools.

In their original formulation, the three protocols under study cannot handle the full LNT synchronization semantics. We believe Sisto's protocol cannot be easily extended because its behavior is closely related to the binary nature of the LOTOS parallel composition operator. On the other hand, we modified Sjödin's and Parrow's protocols such that possible synchronizations are now specified by synchronization vectors. These extended versions can handle the generality of LNT synchronization, and we verified that no new bugs were introduced.

The formal models of protocols will help us to decide which protocol to use for implementation. Nevertheless, before making our final decision, this work should be continued in several directions. First, we will study how data exchanges can be added to the protocols. Second, we could use the protocol models to precisely measure how many messages are required in each protocol to agree on a synchronization, depending on the number of tasks and the possible synchronizations between them. Moreover, we could use the performance evaluation features of CADP [7] to simulate communication latency between remote sites, and measure protocol performances directly on the formal models.

Finally, we will be able to develop a stand-alone compiler to generate a prototype distributed implementation of an LNT composition of tasks, as a family of remote task and protocol processes. The code for each task could be obtained by extending the EXEC/CÆSAR framework [13] of CADP (which currently generates sequential code simulating a concurrent or sequential process) to make it fit the synchronization protocol interface.

Acknowledgments. The authors warmly thank the Inria/CONVECS team members for useful discussions.

References

1. R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Trans. on Software Engineering*, 15(9):1053–1065, 1989.
2. B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, J. Sifakis. From high-level component-based models to distributed implementations. *Proc. of the 10th ACM international conference on Embedded software*, 209–218, 2010.
3. A. Bouajjani, J.C. Fernandez, S. Graf, C. Rodríguez, J. Sifakis. Safety for Branching Time Semantics. *Proc. of 18th ICALP*, 1991.
4. S. D. Brookes, C. A. R. Hoare, A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, 1984.
5. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.8). Inria/CONVECS, 2013.
6. K. M. Chandy, J. Misra. Parallel Program Design: A Foundation. *Addison-Wesley*, 1988.

7. N. Coste, H. Garavel, H. Hermanns, F. Lang, R. Mateescu, W. Serwe. Ten Years of Performance Evaluation for Concurrent Systems Using CADP. *Proc. of ISoLA*, LNCS 6416, 2010.
8. E. W. Dijkstra. The Structure of the "THE"-Multiprogramming System. *Comm. of the ACM*, 1968.
9. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. *Proc. of TACAS*, LNCS 1384, 1998.
10. H. Garavel, F. Lang. SVL: a Scripting Language for Compositional Verification. *Proc. of FORTE*, Kluwer, 2001.
11. H. Garavel, F. Lang, R. Mateescu, W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 15(2):89–107, 2013.
12. H. Garavel, M. Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. *Proc. of FORTE/PSTV*, Kluwer, 1999.
13. H. Garavel, C. Viho, M. Zendri. System Design of a CC-NUMA Multiprocessor Architecture using Formal Specification, Model-Checking, Co-Simulation, and Test Generation. *STTT*, 3(3):314–331, 2001.
14. J.W. Havender. Avoiding deadlock in multitasking systems. *IBM systems journal*, 7(2):74–84, 1968.
15. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
16. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization, 1989.
17. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization, 2001.
18. F. Lang. EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. *Proc. of IFM*, LNCS 3771, 2005.
19. R. Mateescu, E. Oudot. Bisimulator 2.0: An On-the-Fly Equivalence Checker based on Boolean Equation Systems. *Proc. of MEMOCODE*, IEEE, 2008.
20. R. Mateescu, D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. *Proc. of FM*, LNCS 5014, 2008.
21. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
22. D. Park. Concurrency and Automata on Infinite Sequences. *Theoretical Computer Science*, LNCS 104, 1981.
23. J. Parrow, P. Sjödin. Designing a multiway synchronization protocol. *Computer communications*, 19(14):1151–1160, 1996.
24. J. Parrow, P. Sjödin. Multiway synchronization verified with coupled simulation. *CONCUR'92*, 518–533, 1992.
25. J. A. Pérez, R. Corchuelo, M. Toro. An order-based algorithm for implementing multiparty synchronization. *Concurrency and Computation: Practice and Experience*, 16(12):1173–1206, 2004.
26. R. Sisto, L. Ciminiera, A. Valenzano. A protocol for multirendezvous of LOTOS processes. *IEEE Trans. on Computers*, 40(4):437–447, 1991.
27. P. Sjödin. *From LOTOS Specifications to Distributed Implementations*. PhD thesis, Department of Computer Science, University of Uppsala (Sweden), 1991.
28. J. Stoecker, F. Lang, H. Garavel. Parallel Processes with Real-Time and Data: The ATLANTIF Intermediate Format. *Proc. of IFM*, LNCS 5423, 2009.
29. R. J. van Glabbeek, W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics. *Proc. of IFIP*, 1989.