Automatic Distributed Code Generation from Formal Models of Asynchronous Concurrent Processes

Hugues Evrard and Frédéric Lang
Inria
Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
CNRS, LIG, F-38000 Grenoble, France
Emails: Hugues.Evrard@inria.fr, Frederic.Lang@inria.fr

***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. The generated code 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. We present an experiment where DLC generates a distributed implementation from the LNT model of the Raft consensus algorithm.**

I. 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 experimented) programmers. 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 independently developed 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 achieve reasonable performances for rapid prototyping. 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 a distributed system, performance not only depends on the speed of each process, but also on how process interaction is implemented. As a classic illustration, a compiler implementing a naive protocol that consists in acquiring a unique global lock to proceed on an interaction would be extremely inefficient as processes would mostly waste time waiting for the lock while they often could safely execute concurrently.

In this paper, we consider models written in LNT [1], 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 [2] and E-LOTOS [3]. Its semantics are formally defined in terms of an LTS (*Labeled Transition System*): the observable events of an LNT process are *actions* on *gates* (possibly parametrized with data), 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*) [4] tool box, which provides simulation, model-checking, and test generation tools, among others.

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).

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

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

- A rendezvous may involve any number of processes (*multiway* rendezvous), i.e., it is not restricted to binary synchronizations. LNT even features n-among-m synchronization [5], in which a rendezvous may involve any subset of n processes out of a larger set of m.
- Due to nondeterminism, every process may be ready for several actions, and therefore several rendezvous, at the same time (nondeterministic choice). A rendezvous between processes may occur not only if all processes are ready, but also if they all simultaneously agree to take that rendezvous instead of other possibly concurrent ones.
- Processes may exchange data during the 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, where processes interact by rendezvous. 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 (*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 [6], 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 multiway value-passing 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 multiway rendezvous. The protocol we use [7] is not a new contribution, though it is extended to handle n-among-m rendezvous and data exchange. 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.

This paper is structured as follows: Section II explores related work. Section III illustrates how we can model a distributed system in LNT. Section IV covers how we proceed to generate a distributed implementation, and Section V focuses on how hook functions enable interactions with the external environment. Section VI presents experimental results, including a non-trivial application, the Raft [8] consensus algorithm. Section VII concludes and suggests future work.

II. 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 Golang standard library. However, for popular programming languages, we are neither aware of a library that would implement LNT-like multiway rendezvous, nor of a mature verification toolbox that would enable formal verification on the source code itself.

We explore 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 [9].

As regards asynchronous systems, the Topo [10] 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 [11] 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 [12] provides a framework to work on networks of timed automata, including formal verification tools. The associated Times tool [13] generates C code from UPPAAL models, but the final program is sequential. In the framework of SPIN [14], Promela is a modeling language which uses channels rather than multiway rendezvous for process interactions. A Promela to distributed C compiler has been proposed [15], where 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 [16], but again the generated code is sequential. Regarding the Reo [17] coordination language, the Dreams [18] framework provides a methodology to generate distributed applications running

on Java Virtual Machines. Finding a good partition of Reo connectors to distribute them accordingly is not trivial [19], while named synchronization points in process algebra derived languages, such as gates in LNT, naturally provide a —if not optimal, at least relevant— partition. The recent Chor [20] language enables programming of distributed system as choreographies, and can generate distributed implementation, but the process interaction primitive is message passing between two processes. At last, the BIP framework features a distributed code generation tool [21] that handles not only multiway synchronization but also priorities. However, the end user must provide, besides the BIP model, a partition of the distributed system, and to our knowledge only deadlock detection [22] is available for verification.

Since the process interaction mechanism is a key challenge in distributed implementations, we also briefly review how multiway rendezvous can be implemented in a distributed manner. Multiway rendezvous is a variation of the committee coordination problem, stated by Chandy and Misra [23], where professors (processes) must schedule committee meetings (rendezvous), with every professor being a member of several committees. Bagrodia [24] lists classical solutions for this problem and presents the event manager algorithm, based on a token ring approach. At the same period, various studies on the distributed implementation of LOTOS led to several protocol proposals [25], [26], [27]. In a previous study [28], we used LNT and CADP to model and verify three protocols [26], [27], [7], and we spotted previously undetected deadlocks, under asynchronous communication hypothesis, in the one designed by Parrow and Sjödin. 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, Perez *et al.* [29] presented the α-core protocol, but the original specification also contains a bug documented by Katz and Peled [30].

III. Modeling Distributed Systems in LNT

We consider distributed systems to be composed of several *tasks*, which interact with each other. Therefore, a distributed system is specified by the behavior of each of its tasks and by the possible interactions between them. In LNT, distributed systems are naturally mapped to a parallel composition of processes. Each process defines a task, and the parallel composition defines how tasks can interact by setting which rendezvous are allowed on each gate.

We give an informal introduction to LNT using an example; for a formal and full definition of LNT syntax and semantics, see [1]. We model a simplified version of the leader election phase of the Raft [8] 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* (numbered with consecutive integers) during which at most one leader shall be elected.

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 their vote. A server grants its vote only if its term is equal to the candidate

```
(* Not included for lack of space: definitions of majority,
 * maxId, maxTerm, types state and abool (array of bool) *)

function resign(out state: state, out votedId: abool,
                out voteCount: nat, out voted: bool) is
  state     := follower;
  votedId   := abool(false); (* set all array to false *)
  voteCount := 0;
  voted     := false
end function

process server [LEADER, CRASH, TIMEOUT, RVOTE, AVOTE: any]
               (selfId: nat) is
var state: state,
    selfTerm, voteCount, rpcId, rpcTerm: nat,
    votedId: abool,
    voted, voteGranted: bool
in
  (* initialization *)
  selfTerm := 0;
  eval resign(?state, ?votedId, ?voteCount, ?voted);
  (* main loop *)
  while selfTerm < maxTerm loop
    select (* possible behaviors delimited by "[]" *)
      (* timeout, become candidate *)
      case state in
        follower | candidate ->
          TIMEOUT(selfId, selfTerm);
          selfTerm        := selfTerm + 1;
          votedId[selfId] := true;
          state           := candidate;
          voteCount       := 1;
          voted           := true
      | leader -> stop (* leader cannot become candidate *)
      end case
    [] (* receive vote request *)
      RVOTE(?rpcId, selfId, ?rpcTerm);
      if rpcTerm > selfTerm then
        selfTerm := rpcTerm;
        eval resign(?state, ?votedId, ?voteCount, ?voted)
      end if;
      if (selfTerm == rpcTerm) and (not(voted)) then
        voteGranted := true;
        voted       := true
      else
        voteGranted := false
      end if;
      AVOTE(selfId, rpcId, selfTerm, voteGranted)
    [] (* send vote request *)
      case state in
        candidate ->
          rpcId := any nat where rpcId < maxId;
          (* Don't send request if rpcId already voted *)
          if (votedId[rpcId]) then stop end if;
          RVOTE(selfId, rpcId, selfTerm);
          AVOTE(rpcId, selfId, ?rpcTerm, ?voteGranted);
          if rpcTerm > selfTerm then
            selfTerm := rpcTerm;
            eval resign(?state, ?votedId, ?voteCount, ?voted)
          else
            votedId[rpcId] := true;
            if voteGranted then
              voteCount := voteCount + 1;
              if voteCount >= majority then
                state := leader;
                LEADER(selfId, selfTerm)
              end if
            end if
          end if
      | follower | leader -> stop (* do not request vote *)
      end case
    [] (* fail stop *)
      CRASH(selfId, selfTerm); stop (* server halts *)
    end select
  end loop
end var
end process
```

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

one and if it has not voted for someone else earlier in the

```
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
```

Fig. 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).

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 to 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 [31], 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, 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. Servers deal votes through an abstracted RPC mechanism: a request for vote is queried by an action on RVOTE, followed by an answer on AVOTE. A process can send or receive data using *data offers* on an action. For example, a server sends its identifier and its current term when it announces its leadership on LEADER. A variable prefixed by "?" indicates that the process wants to receive a data value —provided either by other processes (through rendezvous) or by the external environment. For instance, when a server is requested for vote on RVOTE, the caller identifier is stored in the `rpcId` variable that is used later in the answer action on AVOTE.

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 on gates RVOTE and AVOTE. Thus, an action on one of these two gates consists of a binary rendezvous of two processes with data 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 rendezvous and the possibility of nesting `par` operators, we can model complex interactions between an arbitrary number of processes.

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: to any LNT process corresponds an LTS where transitions represent actions and are labeled by the gate name, followed by the exchanged data values (if any). The LTS represents the LNT model state space; since it may be huge, models are often parametrized to control the state space explosion. For instance here, the election algorithm is approximated to a smaller state space by bounding server terms with a predefined `maxTerm`.[2]

CADP tools can perform formal verifications, e.g., model checking, on the LTS representation, either on-the-fly or after complete state space generation. For instance, EVALUATOR4 [32] can check the safety property "there is not two leaders in the same term" expressed as the following MCL (*Model Checking Language*) [32] formula:

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

which states that, there must not be consecutive leader announcements 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 refer to [4] to know more about formal verification using CADP, which also features equivalence checking, simulation, and many other tools.

IV. Generation of a Distributed Implementation

DLC takes an LNT parallel composition of processes as input and produces several executable programs (see Figure 3). Each process of the parallel composition becomes a specific program, called *task program*. In order to manage rendezvous, a synchronization protocol is instantiated through one program for each gate in the model, called *gate program*. DLC also generates a *main program*, which deploys other programs, possibly on distinct machines. This section covers how DLC proceeds to generate such a distributed implementation.

Every task program locally explores the task state space. When the program reaches a state, it lists all the actions representing the outgoing transitions in the corresponding LTS. In our example, when a server starts, it is in follower state with a term equal to zero: the possible transitions are actions on TIMEOUT, RVOTE and CRASH, each in a different branch of `select`. A task program does not decide on its own which transition to fire, because some actions may involve several tasks in a rendezvous. It offers an interface where it provides the list of current possible actions, and waits in response the one that must be accomplished. Once it receives this information, it accordingly proceeds to its next state and starts again to list possible actions.

The choice of the action to perform is resolved by a multiway synchronization protocol, instantiated in the implementation. The protocol works as follows: (1) it collects possible actions from each task; (2) it detects which synchronizations are possible (i.e., sufficiently many tasks are ready, with appropriate data offers); (3) it negotiates which of the

[2] 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.

actions will actually synchronize, ensuring that tasks that are ready on several actions commit to the negotiated one; and (4) it announces which action to commit to all tasks involved in this action.

Note that DLC software architecture does not rely on a specific protocol negotiation scheme, it only enforces the task-protocol interface, i.e., each task sends the list of current possible actions and waits in return for the action to realize. The protocol currently used in DLC is based on Parrow and Sjödin's proposal [7] because of (1) the small number of messages it needs to achieve synchronizations, (2) the formal analysis [28] that raises the level of trust we can put in its correctness and (3) the protocol extensibility. Indeed, we extended the original version to handle n-among-m synchronizations and data exchanges during rendezvous.

Without exposing the protocol details, we explain how data are managed by gate programs with respect to LNT semantics. When a task program lists its possible actions, each action is defined by a gate and a list of data offers. Each data offer is made of a data type, a data variable, and a flag which indicates whether the data variable is set (the task sends a value) or not (the task receives a value). Data offers are *compatible* if they have the same type, and if the variable is either not set by any task, set by only one task, or set by several tasks to the same value. Compatible data offers can be *merged* into one offer, whose variable is either set to the common value, or left unset if no task provided a value. For a rendezvous to be possible, all involved tasks must list the same number of data offers, which must be pairwise compatible. When a rendezvous is achieved, all tasks receive the merged data offers.

If some variables are still unset after data offers are merged, LNT semantics states that any value would fit. In DLC, gate programs, rather than generating a random value, always check that all data offer variables are set before announcing an action to commit, otherwise they signal a fatal error to the main program, which then stops the whole system. In Section V, we present an alternative way for the user to assign a value to unset variables using external code. Besides, DLC currently encodes each data offer value in a C 32-bit integer, and is therefore limited to simple types such as naturals, integers or enumerated types. More elaborate types like tuples, lists or arrays can be used in tasks, but must not appear in data offers, otherwise DLC will complain at compilation time. Supporting complex types in data offers is left for future work.

Figure 3 illustrates DLC internal architecture. In order to know which rendezvous are allowed by the parallel composition, we use a CADP tool to extract this information, which we store in the form of *synchronization vectors* [33], i.e., tuples of synchronizable tasks for each gate. Then, we automatically produce a C library that offers an interface to the synchronization vectors. A gate program has a generic behavior parametrized by the possible rendezvous in the system. We implemented this generic behavior in a C library, once and for all. For a given distributed system model, we link this generic module against the synchronization vector library to eventually obtain gate programs that coordinate tasks with respect to the parallel composition of the system. This approach allows us to isolate the synchronization core logic in a standalone module, for easier debugging and maintenance. Concerning tasks, we use the EXEC/CÆSAR [11] sequential code generator, already

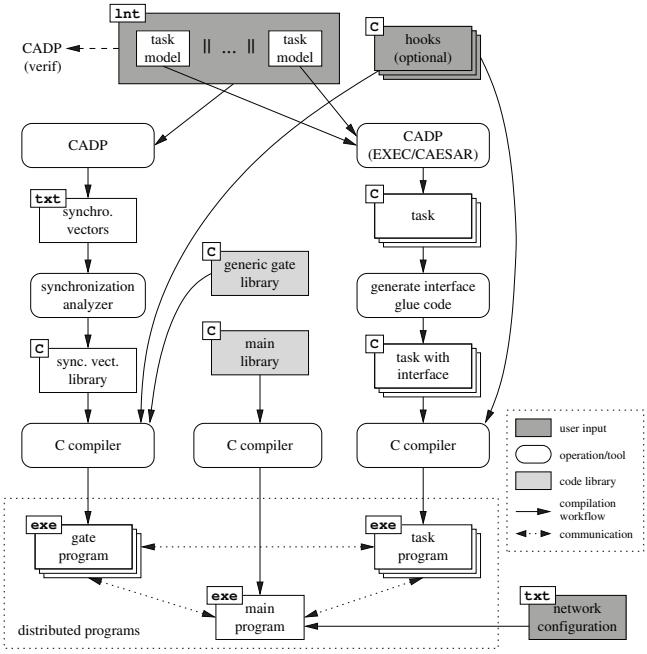

Fig. 3. Overview of DLC internal architecture.

available in CADP, to produce a C implementation of each task process. In addition, we automatically insert glue code in each task to offer the task-protocol interface described above.

Although we already studied the synchronization protocol correctness [28], ensuring the correctness of all DLC code generation is a much bigger effort. As an alternative, in a future work we plan to verify, at runtime, that actions realized by the distributed implementation do not violate the original LNT model. As a comparison, in BIP the synchronization protocol is inserted at the BIP model level, and the resulting model (whose only interactions are asynchronous message passing) is then compiled to C++. This is a nice correct-by-construction approach when the equivalence of BIP models before and after protocol insertion can be demonstrated; however we are aware of such a demonstration for only weak variations of two of the three protocols used in practice in BIP.

DLC is a command-line tool that takes as input a file containing an LNT parallel composition. In one command, DLC automatically generates C code for tasks, synchronization vectors library, glue code, and eventually calls a C compiler to produce the main, task, and gate programs. The protocol relies on asynchronous message passing on a reliable network that keeps messages ordered between two programs; we use TCP sockets to meet these requirements. Moreover, the main program uses a framework already employed in the context of distributed model checking [34] to deploy, start and monitor other programs on several machines. The main program parses a user-given, plain-text, network configuration file, which indicates on which machine each program should run, and then deploys the system using `ssh` (or equivalent tools) for distant access. A default configuration file where all programs are located on the local host is generated, enabling the implementation to be tested locally right after program generation. Scripts can be used to generate configuration

files for deployment on clusters or grids. The code generator comprises about 1500 lines of code (mainly shell scripts), and its internal libraries represent about 2200 lines of C code. On the example of Section III, DLC also generates 2302 lines of C code for each server, and 84 lines of C code for the synchronization vector library.

V. Interactions with the External 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. 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 performances.

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

- When sufficiently many tasks are ready for a (possibly synchronized) action on a gate g, the corresponding gate program starts a negotiation to decide whether the action can happen. To spare the cost of the negotiation when the environment would not allow the action anyway, the gate g program first executes a hook function named g_pre_negotiation_hook, which returns a boolean value indicating whether the negotiation is worth being started.
- When a negotiation succeeds on a gate g, its program 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, while preserving LNT semantics, which allow any value in such situations.

```
process logger[GET, LOG, CRASH: 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
```

```
/* ------------------ logger task hook ----------------- */
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; }
}
/* --------------------- GET hooks --------------------- */
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
}
/* ------------------ INTERRUPT hooks ------------------ */
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;
}
```

Fig. 4. Hook functions permit to interact with the external environment and to control system actions. Here, the `logger` task gets some value from an external database and logs it to the local storage, until it is interrupted. These operations are abstracted away in the LNT model, while the user-defined hook functions enable to actually perform side effects in the external environment, and to control when the interruption happens.

Figure 4 illustrates various usages of hooks. We consider a unique `logger` task which loops on getting the data associated to a key in a database and logging this data, until it receives an interruption. The task hook writes the data passed on `LOG` actions onto the local storage of the machine where the task program runs. There is no motivation to prevent actions on gate `GET`, so its pre-negotiation hook always returns true. The `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. The `INTERRUPT` pre-negotiation hook prevents negotiations if no interruption is detected. The `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 useless negotiations for `INTERRUPT`, and thus to favor negotiations for `GET`.

With hooks, the user can prevent some actions, but cannot achieve actions that would not have been previously stated as possible by the protocol. Hence, since hooks can only restrict the system behavior, the execution path eventually walked will still be within the original LNT model semantics.

VI. Experimental Results

We present two experiments that compare DLC with real-world software. First, we evaluate how much time a multiway rendezvous requires in comparison with mechanisms for synchronization of distant processes in other languages, namely C, Java and Erlang. Since the rendezvous is available neither as a primitive nor as a library in popular languages, as a workaround we set up for benchmark a simple distributed synchronization barrier using sockets in C, RMI in Java, built-in inter-process message passing in Erlang, and the rendezvous in LNT. A synchronization barrier is less complex than the general case of rendezvous since we know for sure that tasks are ready only on the barrier action; our synchronization protocol detects these cases and optimizes the synchronization negotiation accordingly. Figure 6 illustrates the time required to achieve a thousand synchronizations between several processes on different machines. LNT is slower than C but faster than Java and Erlang: this result indicates that the rendezvous implementation generated by DLC achieves relevant performances in a distributed context compared to languages used in industry. We do not focus on synchronization on a single machine: in our approach, if several LNT processes are known to be eventually run on the same computer, a single task can be made of the parallel composition of these processes. EXEC/CÆSAR will produce a single task process that handles the LNT processes interaction internally, calling the protocol only to cover synchronizations that may imply other task processes. Such hierarchical process composition is a powerful feature of process algebra based languages.

For our second experiment, we modeled Raft [8] in LNT in order to demonstrate DLC on a non-trivial system. Raft, like the better known Paxos [35], 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 [36]. A TLA+ formal specification of Raft core features (leader election and log replication) is available, upon which a hand written safety proof is built [37]. 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.

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 configurations, we made several runs of a thousand write requests to the key-value store, with crashes disabled. Figure 6 compares the performances of DLC with those of Consul[3], an industrial-class fault tolerant key-value store also based on Raft. The Consul implementation of Raft is up to ten times faster than ours, partly due to the fact that Consul can group several client requests together and thus requires only one round of log replication among Raft servers for a group of client requests, whereas the LNT implementation triggers a log replication for each client request. We cannot easily implement the Consul strategy since DLC does not handle arrays or lists in rendezvous yet. This also keeps our prototype implementation simpler, our Raft model fitting in approximately 500 lines of LNT plus 300 lines of C for hook functions (mainly boilerplate for sockets) while the Consul Raft library alone represents approximately 4000 lines of Golang.

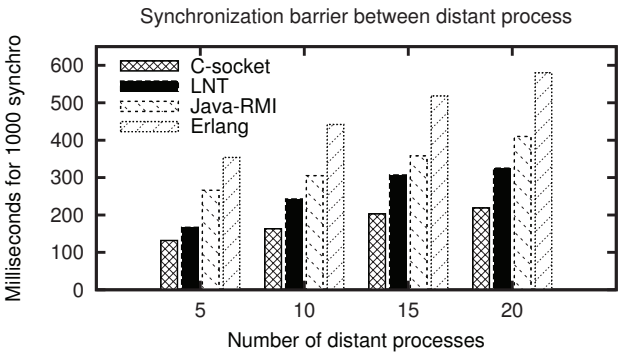

Fig. 5. Performances of the multiway rendezvous.

Time for 1000 requests to the replicated key-value store

Configuration	3 servers	5 servers	7 servers
Consul	453ms	509ms	533ms
LNT-Raft	2238ms	3779ms	5469ms

Fig. 6. Raft consensus: comparison with the industrial tool Consul.

While DLC does not pretend to generate implementations that directly compete with tools used in industry, we consider that the performances achieved so far still qualify for rapid prototyping, with all the benefits that formal verifications brought on. Moreover, hook functions enables to model and prototype only a part of a larger system while still interacting with the rest of the system.

VII. Conclusion

A distributed system made of asynchronous concurrent processes can be formally modeled in LNT, using powerful primitives such as 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.

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

We covered how DLC proceeds to generate distributed programs, and we exposed DLC internal architecture. Besides task and gate programs, DLC also generates a main program which eases the deployment on a cluster of machines. 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 model semantics. We presented two experiments made with DLC, including the implementation of the non-trivial Raft algorithm. The observed performances reveal that even if DLC generated programs are 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 elaborate types, such as lists and arrays, in data offers. Moreover, it would be useful to implement timing mechanisms (such as timeouts) as primitives of LNT, as already suggested in [38]. Finally, a way to raise the trust in the correctness of the generated implementation could be to monitor its execution. We can 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.* [11] and Lantreibecq *et al.* [39] had 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.

REFERENCES

[1] D. Champelovier, X. Clerc, H. Garavel *et al.*, "Reference manual of the LNT to LOTOS translator (version 6.0)", 2014.

[2] ISO/IEC, "LOTOS — a formal description technique based on the temporal ordering of observational behaviour," International Organization for Standardization, Standard 8807, 1989.

[3] ISO/IEC, "Enhancements to LOTOS (E-LOTOS)," International Organization for Standardization, Standard 15437:2001, 2001.

[4] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2011: A toolbox for the construction and analysis of distributed processes," *STTT journal*, vol. 15, no. 2, pp. 89–107, Springer, 2013.

[5] H. Garavel and M. Sighireanu, "A graphical parallel composition operator for process algebras," in *Proc. of FORTE/PSTV*. Kluwer, 1999.

[6] H. Garavel, "Reflections on the future of concurrency theory in general and process calculi in particular," in *Proc. of LIX Coll. on Emerging Trends in Concurrency Theory*. ENTCS vol. 209. Elsevier, 2008.

[7] J. Parrow and P. Sjödin, "Designing a multiway synchronization protocol," *Computer communications*, vol. 19, no. 14, pp. 1151–1160, 1996.

[8] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX ATC*, 2014.

[9] G. Berry, "SCADE: Synchronous design and validation of embedded control software," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Springer, 2007.

[10] J. Mañas, T. de Miguel, J. Salvachúa, and A. Azcorra, "Tool support to implement LOTOS formal specifications," *Computer Networks and ISDN Systems*, vol. 25, no. 7, pp. 815–839, 1993.

[11] H. Garavel, C. Viho, and M. Zendri, "System design of a CC-numa multiprocessor architecture using formal specification, model-checking, co-simulation, and test generation," *STTT journal*, vol. 3, no. 3, pp. 314–331, Springer, 2001.

[12] G. Behrmann, K. G. Larsen, O. Moller *et al.*, "UPPAAL-present and future," in *Decision and Control, 2001*, vol. 3., IEEE, 2001.

[13] T. Amnell, E. Fersman, L. Mokrushini *et al.*, "TIMES: a tool for schedulability analysis and code generation of real-time systems," in *Formal Modeling and Analysis of Timed Systems*. Springer, 2004.

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

[15] S. Löffler, "From specification to implementation: A Promela to C compiler," *Project Report Ecole Nat. Sup. Télécom.*, 1996.

[16] A. Sharma, "A refinement calculus for Promela," in *Proc. of ICECCS*. IEEE, 2013.

[17] F. Arbab, "Reo: a channel-based coordination model for component composition," *MFCS*, vol. 14, no. 3, pp. 329–366, 2004.

[18] J. Proenca, D. Clarke, E. de Vink, and F. Arbab, "Dreams: a framework for distributed synchronous coordination," in *Proc. of SAC*. ACM, 2012.

[19] S. T. Q. Jongmans, F. Santini, and F. Arbab, "Partially-distributed coordination with Reo," in *Proc. of PDP*, 2014.

[20] M. Carbone and F. Montesi, "Deadlock-freedom-by-design: multiparty asynchronous global programming," in *Proc. of POPL*, ACM, 2013.

[21] B. Bonakdarpour, M. Bozga, and J. Quilbeuf, "Model-based implementation of distributed systems with priorities," *Design Autom. for Emb. Sys.*, vol. 17, no. 2, pp. 251–276, 2013.

[22] S. Bensalem, A. Griesmayer, A. Legay, T. Nguyen, J. Sifakis, and R. Yan, "D-finder 2: Towards efficient correctness of incremental design," in *Proc. of NASA Formal Methods*, 2011.

[23] J. Misra and K. Chandy, "Parallel program design: a foundation," Addison-Wesley, 1988.

[24] R. Bagrodia, "Process synchronization: Design and performance evaluation of distributed algorithms," *IEEE Transactions on Software Engineering*, vol. 15, no. 9, pp. 1053–1065, 1989.

[25] G. Bochmann, Q. Gao, and C. Wu, "On the distributed implementation of LOTOS," in *Proc. of FORTE*, 1989.

[26] R. Sisto, L. Ciminiera, and A. Valenzano, "A protocol for multirendezvous of LOTOS processes," *IEEE Transactions on Computers*, vol. 40, no. 4, pp. 437–447, 1991.

[27] P. Sjödin, "From LOTOS specifications to distributed implementations," Ph.D. dissertation, Uppsala University, 1992.

[28] H. Evrard and F. Lang, "Formal verification of distributed branching multiway synchronization protocols," in *Proc. of FORTE/FMOODS*, LNCS vol. 7892, Springer Verlag, 2013.

[29] J. A. Pérez, R. Corchuelo, and M. Toro, "An order-based algorithm for implementing multiparty synchronization," *Concurrency - Practice and Experience*, vol. 16, no. 12, pp. 1173–1206, 2004.

[30] G. Katz and D. Peled, "Code mutation in verification and automatic code correction," in *Proc. of TACAS*. Springer, 2010.

[31] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Comm. of the ACM*, vol. 18, no. 8, 1975.

[32] R. Mateescu and D. Thivolle, "A model checking language for concurrent value-passing systems," in *Proc. of FM'08*, LNCS vol. 5014, Springer, 2008.

[33] F. Lang, "Exp.open 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods," in *Proc. of IFM*, LNCS vol. 3771, Springer, 2005.

[34] R. Mateescu and W. Serwe, "Model Checking and Performance Evaluation with CADP Illustrated on Shared-Memory Mutual Exclusion Protocols," *SCP*, vol. 78, no. 7, pp. 843–861, 2013.

[35] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[36] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[37] D. Ongaro and J. Ousterhout, "Safety proof and formal specification for Raft," *Draft of October*, vol. 7, 2013.

[38] M. Sighireanu, "Contribution à la définition et à l'implémentation du langage Extended LOTOS," Ph.D. dissertation, INP Grenoble, 1999.

[39] E. Lantreibecq and W. Serwe, "Model checking and co-simulation of a dynamic task dispatcher circuit using CADP," in *Proc. of FMICS*, LNCS vol. 6959, Springer, 2011.