DLC: Compiling a Concurrent System Formal Specification to a Distributed Implementation

Hugues Evrard

Team CONVECS – Inria Grenoble Rhône-Alpes & LIG

Abstract. Formal methods can verify the correctness of a concurrent system by analyzing its model. However, if the actual implementation is written by hand, subtle and hard to detect bugs may be unintentionally introduced, thus ruining the verification effort. In this paper, we present DLC (*Distributed LNT Compiler*), a tool that automatically generates distributed implementation of concurrent systems modeled in the LNT language, which can be formally verified using the CADP toolbox.

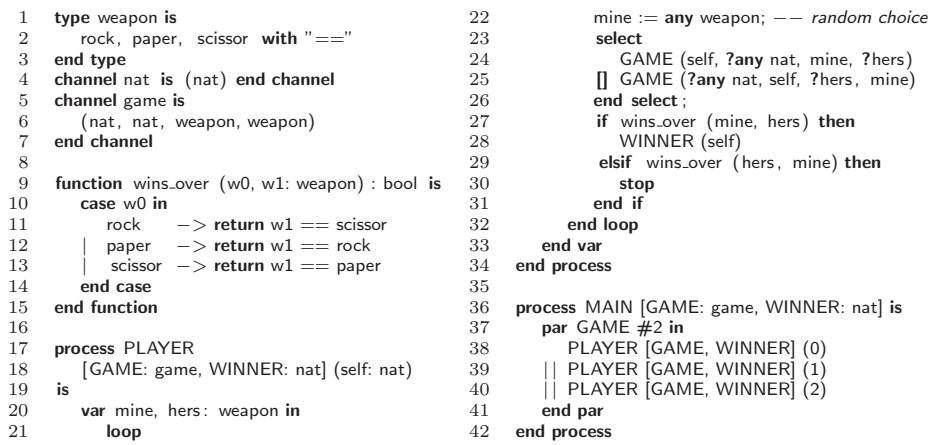
1 Introduction

When designing concurrent systems, the use of formal methods often consists in verifying a *model* of a system, and then writing the actual implementation by hand. The latter is tedious and error-prone, especially in the context of distributed systems, which are notoriously complex. The automatic generation of distributed implementations directly from formal models adresses both difficulties, by speeding-up the production of software, and by letting the programmer operate at the formal model level, with the benefits of formal verification tools. CADP (*Construction and Analysis of Distributed Processes*) [7] is a mature verification toolbox that can analyze concurrent systems modeled in the LNT [3] formal language. In this paper, we present DLC (*Distributed LNT Compiler*, `http://hevrard.org/DLC`), a tool which enables the automatic generation of distributed implementations from LNT models. DLC produces several executables that can be deployed on distinct machines. Moreover, DLC let the end user optionally define interactions between the implementation and its environment.

2 Formal Design with CADP and LNT

The CADP [7] toolbox gathers more than 25 years of research and development in formal methods and offers a comprehensive set of tools including a model checker and a test case generator, among others. The LNT formal language combines a syntax close to mainstream programming languages with powerful concurrency primitives inherited from process algebras. We briefly introduce LNT through a rock-paper-scissors example, illustrated in Fig. 1. For an exhaustive description of LNT including its formal operational semantics, see its manual [3].

The `weapon` type declares the three possible weapons, and requires the equality operator to be defined on its values. Many other types are available in LNT,



```
type weapon is
   rock, paper, scissor with "=="
end type
channel nat is (nat) end channel
channel game is
   (nat, nat, weapon, weapon)
end channel

function wins_over (w0, w1: weapon) : bool is
   case w0 in
      rock     -> return w1 == scissor
    | paper    -> return w1 == rock
    | scissor  -> return w1 == paper
   end case
end function

process PLAYER
   [GAME: game, WINNER: nat] (self: nat)
is
   var mine, hers: weapon in
      loop
         mine := any weapon; -- random choice
         select
            GAME (self, ?any nat, mine, ?hers)
         [] GAME (?any nat, self, ?hers, mine)
         end select;
         if wins_over (mine, hers) then
            WINNER (self)
         elsif wins_over (hers, mine) then
            stop
         end if
      end loop
   end var
end process

process MAIN [GAME: game, WINNER: nat] is
   par GAME #2 in
      PLAYER [GAME, WINNER] (0)
   || PLAYER [GAME, WINNER] (1)
   || PLAYER [GAME, WINNER] (2)
   end par
end process
```

Fig. 1. A rock-paper-scissors game modeled in LNT.

including array and general first-order constructor types which enable the definition of records, lists, etc. The function wins_over uses the **case** pattern matching statement to define the weapons's circular relation. Again, LNT provides many other statements, such as variable assignment, **while** and **for** loops, etc.

The PLAYER process defines a player behavior. Processes are a superset of functions, they additionally enable communication actions, non-determinism and parallel composition. The observable events of a process are *actions* on *gates*. An action contains zero or more *data offers*, whose types form a *profile*. A *channel* lists the profiles supported by a gate. Here, a player, identified by its self argument, performs actions on gates GAME and WINNER, which are restricted by channels game and nat, respectively. A player starts by assigning a random weapon to its mine variable. Then, the select nondeterministic choice statement introduces several possible behaviors, separated by "[]": a player is ready to perform either action on gate GAME—actions differ whether the player's weapon is first or second, identifiers are used for distinction. A player subsequently calls the wins_over function: if it wins, it performs an action on gate WINNER before looping on a new game; if its opponent wins, then the player stops. Otherwise, it is a draw, and both players loop on a new game.

In LNT, processes interact by *multiway rendezvous* with *value matching*, reminiscent of process algebras: one, two or more (multiway) processes synchronize on an action, with the same profile. The value of data offers in received mode (prefixed by "?") of some process is set by other processes. For instance, players can exchange values of type nat and weapon by a rendezvous on gate GAME. The **par** statement in the MAIN process defines which rendezvous are allowed in a parallel composition of three players: an action on gate WINNER can be realized by any player independently, while an action on gate GAME must synchronize any pair among the player processes (m-among-n synchronization [8]).

3 Automatic Distributed Code Generation with DLC

DLC takes as input a parallel composition of LNT processes and generates a corresponding distributed implementation. Each process, also named *task*, is compiled to a distinct executable. Moreover, DLC produces one executable per gate to handle task interactions. Finally, the implementation also contains a *starter* executable that manages the deployment of other executables. For instance, when we apply DLC on our example, we obtain an executable per player, plus two executables for the gates, and the starter executable.

The starter deploys other executables according to a configuration file which associates executables to machine names. By default, DLC produces a configuration file which can be used as a template, where all executables are required to run on the local host. The configuration file adopts a classical UNIX configuration syntax, which makes it easy to be either written by hand or generated by scripts. For instance, here is a configuration file excerpt:

```
edel-12.grid5000.fr                # machine name
   directory = /tmp/task0_PLAYER0  # working directory on the remote node
   files     = dlc_task0_PLAYER0   # name of the executable
edel-36.grid5000.fr
... etc ...
```

3.1 Environment Interaction with Hook Functions

More often than not, the end user wants the generated implementation to interact with other existing systems in its environment, such as a local file system or some web service. DLC enables such interactions through *hook functions*: user-defined C functions that are called upon action events.

We want hook functions to enable not only the monitoring of actions, but also their control. Within the distributed implementation, tasks and gates use a protocol [4] to handle synchronizations while preserving the mutual exclusion of *conflicting* (i.e., targeting the same tasks) rendezvous: when a gate detects a possible action, it starts a negotiation that either succeeds and enables the action realization, or fails. Therefore, we distinguish between *pre-negotiation* hooks that are triggered before a negotiation is started, and *post-negotiation* hooks that are called once the action is achieved. Moreover, each action is both a global event of the system and a local event for each task involved in it. Accordingly, we also distinguish between *global* hooks that are executed by gate processes, and *local* hooks that are executed by task processes. From these categories, DLC provides the three following types of hook functions.

pre-negotiation-global: each gate has a pre-negotiation-global hook that is called before a negotiation starts for an action on that gate. This hook returns a boolean to indicate whether a negotiation must be started for this action.

post-negotiation-global: each gate has a post-negotiation-global hook that is called after a negotiation succeeds for an action on that gate. This hook returns a boolean to indicate whether the action must be realized.

```
/* Function defined in file GAME.gatehook.c */
bool DLC_HOOK_PRE_NEGOTIATION_GLOBAL (DLC_TYPE_ACTION *act) {
    printf ("Allow game between %d and %d ?[y/n] ", act->offers[0], act->offers[1]);
    switch (getchar()) {
        case 'y': return TRUE;
        case 'n': return FALSE;
        default : return FALSE; /* Disallow by default */
    }
}
```

```
/* Function defined in file WINNER.gatehook.c */
bool DLC_HOOK_POST_NEGOTIATION_GLOBAL (DLC_TYPE_ACTION *act) {
    play_sound (act->offers[0]); /* Plays the winner's sound */
    return TRUE;
}
```

Fig. 2. Hook functions enables interactions with the environment

post-negotiation-local: each task has a post-negotiation-local hook that is called when the task realizes an action. This hook returns nothing.

The action under consideration is passed as an argument to all the three types of hooks. Note that the pre-negotiation-global hook can decide whether a negotiation shall be started or not, but a positive response does not guarantee that the subsequent negotiation is successful. When the negotiation does succeed, it is up to the post-negotiation-global hook to eventually decide to realize the action (now that it is certain to be doable) or to abort the negotiation. All hooks can interact with the environment to make choices or perform side effects.

Hook functions are optional, as DLC can produce a stand-alone implementation without them. Hook functions for a gate g (resp. a task t) must be defined in the file named g.gatehook.c (resp. t.taskhook.c). DLC automatically detects such files and embeds the hook functions into the implementation. Besides, DLC has an option to generate hook function templates for a particular gate or task.

Figure 2 illustrates hook functions on the rock-paper-scissors example. The pre-negotiation-global hook of gate GAME let the user decide, at runtime, which games she allows. The post-negotiation-global hook of gate WINNER is used to play some particular sound depending on which player wins a game.

3.2 Overview of Compilation Internals

Figure 3 gives an overview of how DLC proceeds to generate a distributed implementation. DLC relies on the EXEC/CÆSAR [9] tool of CADP to obtain a sequential implementation (in C) of each task process. However, the implementation produced by EXEC/CÆSAR is not complete: it can list the currently possible actions of a process, but does not decide which action shall be realized. This decision is made by the synchronization protocol, and DLC automatically interfaces the code generated by EXEC/CÆSAR with the protocol. Both task and gate protocol logic are implemented once for all in isolated libraries, which nonetheless require information about the specification, such as the interactions

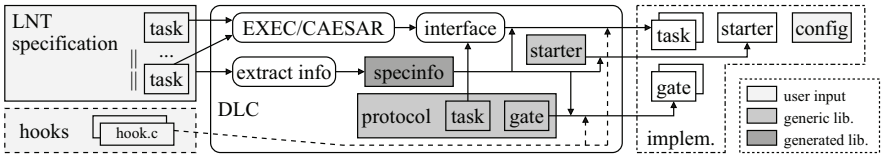


Fig. 3. Overview of DLC architecture.

allowed on each gate with respect to the parallel composition. DLC extracts and gathers this information into the "specinfo" library, which is also used by the starter to know who is who. Moreover, DLC detects and embeds the optional hook functions. Finally, DLC invokes a C compiler to produce the executables.

Implementation Correctness. The sequential implementation of each task is obtained by the existing EXEC/CÆSAR tool of CADP, which has already been employed in a formal context [9]. Interaction between tasks is achieved by a synchronization protocol [6] that we verified [4] using a formal approach that detected possible deadlocks in other protocols of the literature [5]. The actual implementation of the protocol logic is done by hand, but it is isolated in generic modules that can be thoroughly tested. The writing of these modules is a one-time effort, since they are reused in all generated implementations. Therefore, for a given LNT specification, the specific code produced by DLC comes down to the task-protocol interface which is glue code, and the "specinfo" library which only represents information in data structures. Finally, hook functions can avoid some valid actions to happen, but they cannot lead the system into an invalid action. All these considerations let us have a decent confidence in the correctness of implementations generated by DLC.

Current Restrictions. DLC presents two main restrictions. First, values exchanged during an action must fit into a 64bits integer, thus records, lists, and arrays must not appear in action data offers. To be removed, this restriction requires serialization primitives for any LNT types, and we look toward CADP tools to provide them. Second, an action can be *guarded* by a boolean function, i.e., the action is allowed only if its offers let the guard function evaluate to true. Since the code generated by EXEC/CÆSAR does not give access to guard functions, DLC currently ignores the restrictions on data offers possibly induced by them. To be removed, this restriction requires to modify EXEC/CÆSAR such that the generated code gives access to guard functions.

4 Conclusion

In this paper, we presented the DLC tool, which enables the automatic generation of a distributed implementation from the LNT formal model of a concurrent system. From an LNT parallel composition of processes, DLC produces several executables that can be easily deployed on distinct machines. We underline the fact that DLC does not require any special annotations in the LNT source. Process interactions by multiway rendezvous with data exchange are managed by a

formally verified protocol [4]. The end user can also set up interactions with the environment thanks to the hook functions.

We measured the performance of implementations generated by DLC on several examples [6,4]. Our biggest case study so far is the Raft consensus algorithm: from an LNT specification of about 500 lines, DLC produces more than 9000 lines of C code for a Raft server. Across all examples, results illustrate that implementations generated by DLC can achieve more than 1000 rendezvous in sequence per second (and of course much more when rendezvous are realized concurrently on different gates). Hence, we consider implementations generated by DLC to qualify at least for rapid prototyping.

As regards related work, BIP [11] and Chor [1] come with deadlock analysis tools and a distributed compiler. Erlang programs can be verified with McErlang [2], and Dreams [10] generates distributed implementations of Reo models.

Thanks to DLC, a concurrent system can now be modeled in LNT, formally verified with CADP, and automatically compiled to an efficient distributed implementation which is easily deployable and which can interact with its environment. In future work, we plan to get rid of the remaining restrictions of DLC, such that it can handle any LNT specification.

Acknowledgments: the author warmly thanks Frédéric Lang for reviews of this paper, and all other members of the CONVECS team for their support.

References

1. Carbone, M., Montesi, F.: Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming. POPL '13, pp. 263–274, ACM (2013)
2. Castro, D., Gulías, V.M., Earle, C.B., Fredlund, L., Rivas, S.: A Case Study on Verifying a Supervisor Component using McErlang. ENTCS 271 (2011)
3. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., McKinty, C., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LNT to LOTOS Translator (Version 6.1). INRIA/VASY and INRIA/CONVECS (Aug 2014)
4. Evrard, H.: Génération automatique d'implémentation distribuée à partir de modèles formels de processus concurrents asynchrones. Ph.D. thesis, Université de Grenoble (Jul 2015)
5. Evrard, H., Lang, F.: Formal Verification of Distributed Branching Multiway Synchronization Protocols. FORTE/FMOODS'2013, LNCS 7892 (2013)
6. Evrard, H., Lang, F.: Automatic Distributed Code Generation from Formal Models of Asynchronous Concurrent Processes. PDP'2015, IEEE (2015)
7. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. STTT 15(2), Springer (2013)
8. Garavel, H., Sighireanu, M.: A Graphical Parallel Composition Operator for Process Algebras. FORTE/PSTV'99, IFIP (1999)
9. Garavel, H., Viho, C., Zendri, M.: System Design of a CC-NUMA Multiprocessor Architecture using Formal Specification, Model-Checking, Co-Simulation, and Test Generation. STTT 3(3), Springer (2001)
10. Proenca, J., Clarke, D., de Vink, E., Arbab, F.: Dreams: a Framework for Distributed Synchronous Coordination. SAC, ACM (2012)
11. Quilbeuf, J.: Distributed Implementations of Component-based Systems with Prioritized Multiparty Interactions. Ph.D. thesis, Université de Grenoble (2013)