Experiments with distributed Model-Checking of group-based applications

Ludovic Henrio & Éric Madelaine
INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis
2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex - France
Email: First.Last@sophia.inria.fr

I. Behavioural models for Group-based applications

In recent work [3], we have proposed a modelisation of the behaviour of group-based distributed applications, in the form of parameterized networks of synchronised automata (pNets, see [4]). A typical structure in group-based applications is illustrated in Figure 1, where a client sends requests using a synchronous broadcast mechanism (BO) to a number of servers, then collects (CO) the results from these requests in an asynchronous way.

The pNets formalism provides us with a powerful and flexible way to encode labelled transition systems with value-passing, as well as parameterized topologies of processes, and many different communication primitives. But it also has the great advantage that it can be transformed by abstraction into finite pNet models, suitable for finite-state model-checking.

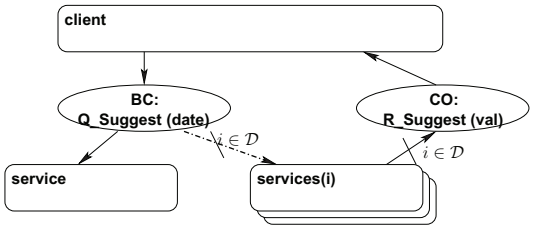

Figure 1. Graphical representation of broadcasting operator

II. Encoding with the Fiacre intermediate language

The middle term goal of these experiments is to integrate automatic model-generation procedures in our VerCors toolset [6]. VerCors includes (graphical) editors for the definition of distributed component-based applications; from such a description, the system generates a pNet behaviour model, that needs to be translated into a language usable as input of a model-checker. We use the CADP verification toolset [8]. Amongst the possible input languages for the CADP engines, we have chosen the recently defined Fiacre format [5], featuring most of the concepts we need for encoding our pNet structures: simple constructive data-types, automata-like processes, parameterized processes, multi-process communication (a la Lotos).

In Figure 2 we show the high-level architecture of our case-study: *Participant[i]* is a group of processes providing services *Suggest* and *Validate*. *Initiator* is a client, that may send requests to the whole group in a broadcast manner. Results from these requests are returned asynchrounously by each group member, collected by a proxy, before being used by the Initiator body. Each Participant has a queue, storing the incoming requests; the participant body is monothreaded, and encodes the service policy.

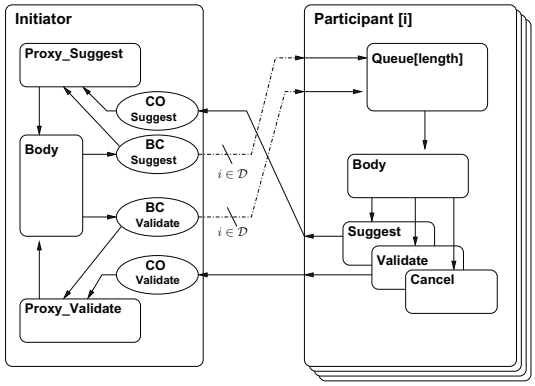

Figure 2. Structure of our case-study

In the current state of our research, we encode manually the pNets into Fiacre code. The most interesting features of the Fiacre language are described here:

Fiacre processes feature standard state-oriented and guarded events concepts, e.g.:

```
process Queue2 [ Q_Suggest: in data,
    Q_Validate: in data2, Q_Cancel:
    none, ...  ]
is
states S_empty, S1, S2, ...
var x:data,  y:data2, ...

from S_empty
  select
    Q_Suggest?x; to S1
  []
    Q_Validate?y; to S2
...
end
```

It also support user-defined data types, and classical programming constructs, as in :

```
const G:nat is 3
type fut_data is union undef | b of
    bool end
type Result_vector is array G of
    fut_data
process Group_proxy [ WaitFor_m: none,
    GetNth_m: out indexG#bool, ... ]
is
states A
var val:bool,
    V : Result_vector
from A
      case V[0] of
  undef -> WaitForNth_m ! 0
      | b(val) -> GetNth_m ! 0,val
      end case;
      to A
```

Fiacre components are used to compose processes hierarchically, with parallel operators inspired from Extended Lotos. They feature explicit declaration of ports, and constructs for specifying multi-way synchronisation of events on these ports:

```
component System [Q_Suggest: data, ...]
is
port  R_Validate0, ...: indexG

par Q_Suggest, ... in
   R_Suggest0, R_Validate0, ... ->
   Initiator [Q_Suggest, R_Suggest0,
       R_Validate0, ...]
 ||
   R_Suggest0, R_Validate0
   -> Participant0 [Q_Suggest,
       R_Suggest0,... ]
 ||
   ...
end
```

In this composition we can observe all synchronisation modes useful for the encoding of our pNets: Events on port `R_Suggest0` are synchronized between components `Initiator` and `Participant0`, events on port `Q_Suggest` are synchronised between *all* participating components (our broadcast communications), events on the local port `R_Validate0` are hidden from outside System, while those on `Q_Suggest` are visible in the global system. Components can have parameters, however they cannot encode directly parameterized topologies, because this would require to specify parameterized synchronisation on their ports. For example here we had to declare one separate port `R_Suggest_i` for each of the possible message from `Participant[i]` to `Initiator`.

III. Using distributor on a cluster infrastructure

In the following tables, we show the figures obtained with the distributed version of the CADP state-generation tools. These figures have been obtained on a cluster, comprising 15 nodes; each node has 8 cores and 32 Go of RAM. The table in Figure 3 measures the overhead due to the deployment of the distributed model-checker. The cost is linear in the number of cores, and mainly due to the copy of the engine executable file on all nodes. There is also a quasi-constant cost, due to preliminary compilation of state-generation code, and to final merging and minimization of the generated state-space.

Subsystem	configuration	Total Time
Initiator :	sequential	11”
	3x4 cores	24”
	3x8 cores	33”
	8x4 cores	38”
	15x4 cores	52”
	15x8 cores	89”

Figure 3. benchs for a small component

Figure 4 shows results obtained for bigger systems. The principal source of state explososion in our example comes from the Queue process as it encodes all possible values of requests with their data arguments, in each position of the queue. We encode a bounded queue structure, with a specific OOB event allowing for checking boundedness properties. Playing with the size of data domains, as well as with the group size, is an easy way to experiment with various strategies and resource configurations.

An important remark is that building systems in a pure compositional way is not always the best strategy: the full state-space of subsystems, when computed out of their context, can be much larger that the part really useful. Here we can observe that the full system size is much smaller than the size of the group of participants, and that trying to compute the group state-space by itself may even fail. This of course is not new, and usual solutions include:

1) generate directly the state space of the server(s) together with their client(s). This is what we have done here in the rows for the “Full system”.
2) generate separately the state-space of the server(s), but providing some constraints on the context behaviour. This would be the idea of a “contract” for using the server(s) in a correct manner. In the CADP toolbox, the projector tool is providing this possibility; the context can be computed from the client code, or can be guessed by the server developper, and checked correct later
3) generate separately the state-space of the server(s), and reduce it by (branching) bisimulation before computing any product.

Solutions 2) and 3) technically involve using Fiacre code for describing the individual subsystems, then using the script language of CADP, SVL, to perform the reduction and parallel product operations. This would have been too complicated for our ongoing experiments, and we have concentrated on the pure distributed features of the tools.

The *distributor* tool of CADP generates state spaces in a distributed way, based on a static hash function ensuring the distribution of states on a number of nodes. The resulting states must then be merged before application of tools that are only available in a sequential implementation, including bisimulation-based minimization, and model-checking. However, some partial-order reduction techniques are available "on-the-fly", during distributed state generation, namely taucompression and tauconfluence [7]. They provide a trade-off be-

Subsystem	generation algorithm	Total Time	States/Transitions	States/Transitions (minimized)
Initiator (sequential)	brute force	12”	3 163 / 152 081	54 / 1 489
Initiator (3x4 cores)	brute force	24”	3 163 / 152 081	54 / 1 489
	taucompression	30”	3 163 / 131 942	54 / 1 489
	tauconfluence	35”	1 219 / 33 815	54 / 1 489
Full system with 3 participants (8x4 cores)	brute force	6’45”	170 349 / 1 646 368	458 / 1 284
	taucompression	11’48”	170 349 / 607 570	458 / 1 284
	tauconfluence	30’	5591 / 14 236	458 / 1 284
Single Participant (sequential)	brute force	9’	9 653 / 31 480	171 / 641
Group of 2 participants (15x8 cores)	brute force	11’32"	13 327 161 / 48 569 764	4 811 / 24 588
	taucompression	30’59"	13 327 161 / 48 569 764	4 811 / 24 588
	tauconfluence	1150’55”	392 961 / 1 354 948	4 811 / 24 588
Group of 3 participants (15x8 cores)	tauconfluence	-	*Out of memory*	-

Figure 4. Benchs for the various on-the-fly reduction strategies

tween space and time consumption, generating less transitions and less states at the price of local "on-the-fly" computations. This trade-off is clearly visible on the full system computation figures, where we generate only 5K states in tauconfluence mode (before merging and minimization). Taucompression is significantly less expensive in time than tauconfluence; here, it appeared that it does not give any benefit for the “Groups of participants” cases. Tauconfluence brings significant reduction in the number of generated states, but it appears here that local computation was too costly in local memory space for the “group of 3” case, preventing us to get a measurable result.

IV. Conclusion

Group-based distributed systems are specific cases of distributed applications with a parameterized topology. They are naturally modelled by systems with a very large state-space. We encode the behavioural semantics of group-based applications using the intermediate format FIACRE. We have experimented with model-checking of such systems, using the CADP verification toolset, and in particular the distributor tool. This allowed us to generate very large but finite state-space on the PacaGrid cloud infrastructure. We have then been able to compare different techniques for generating state-spaces, and experiment with different size of the modelled system and of the experimental platform.

In practice, an efficient solution would rely on a combination of the techniques mentionned in this paper, and in particular on the use of, at the same time, on-the-fly, hierarchical, and contextual techniques. In particular the last technique allows the partial specification of the context in which the system will be used, which will greatly reduce the state space to be generated, and seems a promising method that we want to experiment in future works.

There exists other implementations of distributed model-checking tools, in particular the DiViNe toolset [1], that implements model-checking algorithm for LTL, with specific optimisation for various computing infrastructures, namely clusters, multi-core, and Cuda; and LTSmin [2], that is a MPI-based tool implementing equivalence checking and minimization for various formalisms. Comparisons between these systems is not easy, as it involves encoding the case-studies in quite different formalisms, e.g. the DVE specification language for DiVinE, or μCRL for LTSmin. Furthermore, these toolsets also implement their model-checking algorithms in a distributed way, while the current version of CADP only supports state-generation and on-the-fly reduction in a distributed way, while minimization and model-checking remain sequential.

References

[1] DiVinE, Distributed and Parallel Verification Environment.

[2] LTSmin: Minimization and Instantiation of LAbelled Transition Systems. http://fmt.cs.utwente.nl/tools/ltsmin/.

[3] R. Ameur-Boulifa, L. Henrio, and É. Madelaine. Behavioural models for group communications. In *in proceedings of the International Workshop on Component and Service Interoperability, WICS’10*, Malaga, june 2010. to appear.

[4] T. Barros, R. Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed Fractal components. *Annals of Telecommunications*, 64(1–2), jan 2009. also Research Report INRIA RR-6491.

[5] B. Berthomieu, J. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stoecker, and F. Vernadat. The syntax and semantics of Fiacre. In *Rapport LAAS #07264 Rapport de Contrat Projet ANR05RNTL03101 OpenEmbeDD*, Mai 2007.

[6] A. Cansado and E. Madelaine. Specification and verification for grid component-based applications: from models to tools. In F. S. de Boer, M. M. Bonsangue, and E. Madelaine, editors, *FMCO 2008*, number 5751 in LNCS, pages 180–203, Berlin Heidelberg, 2009. Springer-Verlag.

[7] H. Garavel and G. Serwe. State space reduction for process algebra specifications. *Theorical Computer Science*, 351(2), 2006.

[8] F. Lang, H. Garavel, and R. Mateescu. Cadp 2006: A toolbox for the construction and analysis of distributed processes. In *CAV’07*, 2007.