Validation and evaluation of a software solution for fault tolerant distributed synchronization

P. Ballarini, S. Bernardi and S. Donatelli
Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy
E-mail: {ballarini,bernardi,susi}@di.unito.it

Abstract

This paper presents a case study on the combined use of different tools and techniques for the validation and evaluation, from the early stages of the design, of a fault tolerant software mechanism named distributed synchronization. The mechanism has been specified using UML state charts and sequence diagrams. A number of Stochastic Well-formed Nets (SWN) models have been derived from the specifications: they have been composed using the tool algebra, and the resulting model has been model-checked using the PROD tool for temporal logic properties, thanks to a GreatSPN-to-PROD translator. The quantitative analysis has been performed using the SWN solvers of the GreatSPN tool.

1 Introduction

Modelling and evaluation is traditionally an activity that takes place at the latest stages of the development of a system, sometimes even after the system is already in the operational phase, although in recent years we have seen a number of papers that try to produce performance models from (even partial) specifications of the system, most notably in the field that studies the generation of performance models starting from various UML specification diagrams (see for example [2, 3, 4]). This paper follows this line of research and reports on an effort conducted to support the design of a software synchronization mechanism using a suite of tools based on Petri nets.

The paper addresses the use of Stochastic Well-formed Nets [12, 13] (SWN) and related tools to support the validation and evaluation from the early stages of the design. Peculiar aspects of the early-stage evaluation are:

1. frequent changes in the system being modelled (asking for high flexibility and reuse);
2. delays are usually unknown, or fixed only by an order of magnitude (so that typically sensitivity analysis is performed);
3. need for quick macro models (to quickly evaluate possible macro alternatives)
4. need for detailed models (to validate and evaluate the chosen proposed solution).

The paper addresses points 1, 2 and 4, while an example of analysis for point 3 can be found in [8]: point 3 and 4 are strictly related since it would be interesting to be able to prove some sort of “consistency” between the macro models and the detailed ones. As far as we know there are not many attempts along this line.

We have taken a stochastic approach to evaluation, and we use as our modelling formalism Stochastic Well-formed Nets [12, 13] (SWN) a colored extension of Generalized Stochastic Petri Nets [1] (GSPN). GSPN and SWN allow delays to be exponentially distributed, and many tools allow also non exponential distributions, especially if the solution method is simulation. Indeed in our work we are not much concerned with specific distributions (we are still in the early stages of the design), but we assume that the qualitative and quantitative models have the same behaviour (same reachability graph), and a trivial condition for this to be satisfied is that all distributions have infinite support.

A number of tools allow the definition and solution of GSPN models and of colored GSPN models of some sort, like: APNNtoolbox [16], GreatSPN [11], SMART [14], TimeNET [20], UltraSAN [18]. In this paper we use GreatSPN since we want to take advantage of the efficient solution methods of SWN (both analytical and simulation [5]) and of a number of additional tools/programs that have been recently added to GreatSPN: Multisolve [6], algebra [7] and GreatSPN-to-PROD [9].

Multisolve is a Java interface of GreatSPN to plan and execute solution experiments (using any of the GreatSPN solvers for GSPN and SWN) and to produce gnuplot and postscript files of the results. Similar facilities are present in most performance evaluation tools, and they are of great help while performing the sensitivity analysis demanded by the point 2 above.

algebra is a tool for composing SWN, and it allows the composition over immediate and timed transitions (syn-
chronizing transitions) based on non-injective transition labelling, and the composition over places (communicating places), again based on a non-injective place labelling. algebra also allows us to select whether the variables on input arcs from different models to a synchronized transition should be unified or not. It is an essential tool when frequent changes and reuse are needed, as for point 1 above.

Validation of Stochastic Petri Nets is usually supported by the computation of a number of structural properties, like P- and T-semiflows, conflict sets, structural deadlocks and traps, while a much smaller number of properties are available, and implemented, for SWN. For example, to the best of our knowledge, the only available computation of SWN semiflows is the one in CPN-AMI, but for P-semiflows [15] only and with a number of restrictions on the class of accepted models. Moreover there is in most tools a limited possibility of examining the state space, for GSPN but especially for SWN: GreatSPN provides a textual description of the reachability graph (possibly symbolic for SWN) to-gether with the list of deadlock states, and number of live-locks and of strongly connected components (for GSPN only). These facilities are absolutely inadequate when models become very large and colors come into play.

PROD [19] is a tool for the definition and validation of place/transition nets and of the high level Petri net class known as Predicate/transition (Pr/T) nets. PROD allows the construction of the reachability graph of a net and its in-spection by means of the interactive program probe. It al- lows the verification of the RG by displaying paths between RG nodes and by testing propositional logic formulae and temporal logic formulae expressed in Computational Tree Logic (CTL) [17]. An important characteristic of PROD is that all formulae are expressed in terms of net elements like marking of places and firing of transitions, therefore using a language that is similar (not too different may be) to the one used for the definition of the performance indices.

GreatSPN-to-PROD [9] is a tool that allows the use of the validation facilities of PROD by translating GSPN and SWN nets into algebraic nets in PROD format. The translation also produces a number of ad-hoc macros that allow an easy check of the most common properties of nets, like the computation of the minimum path to a deadlock or the markings enabling a given set of transitions.

The case study of the paper is a software mechanism called distributed synchronization (DS), that has been de-veloped inside the EEC projects TIRAN [10], and its follow-up DepAuDE, as part of a software framework that allows the cheap addition of fault tolerance provision for systems built from standard COTS components. DS allows the synchronization of a number of user tasks over a number of synchronization points, called levels. Both the user tasks and the DS itself can be distributed over different nodes, to tolerate crashes of a node or of a single DS task. In this paper we concentrate on the case of a single DS. The validation and evaluation of the DS plays an important role in the project since it is used by a number of other mechanisms of the framework.

The emphasis of the paper is on the validation and eval-uation process, and on the tools used in the process, since we want to report on the combined use of evaluation and validation from the early stages of the design, more than on the specific performance results.

The paper is organized as follows: Section 2 introduces the distributed synchronization mechanism. Section 3 in- troduces the initial SWN models of the DS components and their composition into a single SWN model. Section 4 in-troduces a variation of the pattern of use of the DS by the users. The “stochastic” evaluation of the system seems satisfactory but a closer look shows that a deeper qualitative analysis is needed to validate the model using GreatSPN-to-PROD and PROD. Section 5 shows the new definition of the DS as suggested by the problems detected in the previous section, and the corresponding SWN model, that is again validated and finally evaluated. Section 6 concludes the paper by taking a critical view point to this modelling activity.

## 2 The DS mechanism

Distributed Synchronization (DS) is a software mecha-nism whose objective is to allow tasks (possibly distributed over a net) to synchronize on the execution of certain activities. Different levels of synchronization are supported, representing different, possibly parallel, independent activities for which synchronization is needed. We consider n-tasks (t_1 \ldots t_n) that require synchronization services on m-levels (l_1 \ldots l_m). Peculiar aspects of the DS behaviour are:

- synchronizing tasks and DS are executing on a distributed architecture, made of a certain number of node;
- participants to synchronization depend on the level: on different levels, participants may be different;
- participating tasks may fail during their execution and this must not block the other tasks that are waiting for a synchronization.

DS has been specified as a UML class with an attribute DStable and two methods, one for checking if a synchroni-zation is possible, according to the table, and one for changing the state of the table according to the stimulus re-ceived.

The DStable is used by the DS to keep track of the evolu-tion of the activities (who’s finished, who’s participating, who’s still executing). The entry DStable[i][j] represents
the state of task $t_i$ with respect to level $l_j$. The possible 
values are: Not Reached (task $t_i$ participates in a synchroniza-
tion at level $l_j$ and has not yet completed), Reached (task $t_i$ 
completed on level $l_j$), Not Available (task $t_i$ fails and it 
should not be taken in consideration for synchronization), 
Restarted (task $t_i$ is alive again and ready to synchronize on 
level $l_j$), and Last Reached (level $l_j$ is the last one on which 
synchronization has been reached).

The change of state of a single entry is described by the 
automaton of Figure 1, while the behaviour of the DS is de-
picted by the automaton of Figure 2. To better describe a 
system a number of sequence diagrams were also provided, 
to show specific paths of execution, together with snapshots 
of the DSTable during the path. As an example, let us con-
side what happens when a ready message from a user task $t$ 
on level $l$ is received by the DS: the state of the automaton of 
Figure 2 changes to checking tasks, and the entry ([i/$l$/]) 
of the DSTable changes from LR to R or from NR to R. If in-
stead a task $t$ fails the automaton moves to the disconnect-
ing task state, and all entries of the table for task $t$ are set to 
Not Available.

A full description of the DS is beyond the scope of this 
paper, but we hope that what has been described is enough 
for the reader to appreciate (or depreciate) the analysis ef-
forts described in the remaining sections.

Figure 1. Automaton of the DS table

3 The Petri net model of the DS

The SWN component models that we consider as basic 
elements for the DS are:

tasks the user tasks requiring synchronization;

backbone the model that accounts for fault and repair (the 
particular name come from the TIRAN project)

ds the distributed synchronization (translation of the 
automaton of Figure 2);

check_synch represents the method for checking if a 
synchronization has been reached.

DStab the DS table (translation of the automaton of 
Figure 1);

All SWN models share the definition of two basic color 
classes T and L, defined as: $T = u T_k$ and $T_k = \{t_1, \ldots, t_n\}$ 
meaning that the class of levels is made of a single static 
class, containing $n$ colors, representing the $n$ tasks of 
the system (the SWN formalism requires the definition of 
the class of tasks in terms of static subclasses, even if there 
is a single one, like in this case) and $L = o L_1 \ldots L_m$, with 
$L_i = \{l_i\}$ meaning that the class of levels is made of $m$ static 
subclasses of one color each; the class is ordered, so that the 
circular successor function can be applied to colors in this 
class and the expression “next level” makes sense.

We now describe each model in isolation: these models 
are later composed to form the SWN model of the whole 
system. The composition operators are superposition over 
place and transitions, based on place and transition labels, 
as defined in [7]. Synchronizing transitions and commu-
nicating places are easily recognizable in the figures, since 
labels are graphically represented by a suffix “[label]name” 
to the place or transition tag. We also remind the reader 
that in SWN syntax $\tilde{S}$ is a constant function that means “all 
colours of a (sub)class”, and that “!” indicates the succes-
sor function (for ordered classes only); in the SWN models 
presented in this paper we have used $x$ and $l$ to indicate a variable of colour T (tasks) and L (levels), respectively.

Figure 3(a) models the user tasks: each task performs 
activities independently from the others and then requires a 
synchronization on a given level by sending a “READY” 
message to the DS through a mailbox (place with label 
"TK_DS_R"): when the synchronization is reached, an ack 
message is sent back to the task through another mailbox 
(place with label “DSTK_R”) that may proceed to the next 
activity towards the next synchronization level (indeed there 
is the successor function “!” on the arc from rcv-global-
synch to place tk1). A task may fail while working and 
later be restarted. Failure and restart events are modeled 
by timed transitions. This model interacts with the back-
bone through transition labels FAULT and RESTART, with 
the DS through place labels TK_DS_R and DS_TK_R. The 
initial marking is non null only in place tk1, where there

Figure 2. Automaton of the DS states
are two tokens, meaning that tasks t1 and t2 are working to reach level l1.

Figure 3. SWN model of tasks (a) and backbone (b)

Figure 3(b) models a super simplified backbone, since the only activities represented are the ones connected to the failure and restart of a task. In case of failure, the backbone sends to the DS mechanism a notification message through a mailbox (place labelled “BK_DS_F”). Please note the use of the # before the name of the variables as a directive to algebra that the variable should be unified upon composition. This model interacts with the user task model through transition labels FAULT and RESTART, with the ds model through place labels “BK_DS_F”. Since all tasks may fail, the net is initialized with as many tokens in place bk1 as there are tasks.

Figure 4 models the distributed synchronization mechanism as specified by the automaton and by the sequence diagrams that, for brevity, we have not reported here. We consider the case of a single instance of the distributed synchronization task. It is easy to see that the structure of the SWN is basically the same as the automaton of Figure 2. The upper part of the net (transition ds5) represents the sending of an acknowledge message to all tasks that are waiting for it: the structure is more complicated that it should due to the fact that marking dependent arcs are not allowed in SWN.

The interface to the check-synch model is the synchronization transition label “check-synch” and the communicating place label “levels”. When the DS receives a READY message for level l from an user task, it puts in the communicating place the value l (through transition ds9 and function (l) on the arc) and it activates the check-synch model; when instead the check is caused by the failure of a user task the DS model passes to check-synch the whole set of levels (function (S) on the arc) since it is not able to determine, a priori, on which level a synchronization will, eventually, take place.

The model is also tightly interfaced with the model of the DSTable, as explained later in this section.

Figure 4. SWN model of the DS

Figure 5 represents the Check-Synch model that checks whether the synchronization conditions are satisfied. It is called by the DS when a READY message from a task is received or when a failed task is detected. Place ck2 levels contains input values passed to the method: for each level contained in this place synchronization conditions are checked until a level is found such that the conditions are satisfied: in this case the level is passed as a return value to the DS mechanism when the transition ck6|reached is fired (function (l) on the input arc to ck6). In case the conditions are not satisfied for any level contained in place ck2 levels no value is passed to the DS mechanism and the transition ck8|not-reached is fired.

Given a level l, the synchronization conditions are:

- **a** for all tasks \( t \in T : \text{DStable}[t][l] \not\in \{LR, NR\} \), and
- **b1** (there \( \exists \) at least a task \( t \in T : \text{DStable}[t][l] = R \) or
- **b2** there \( \exists \) at least a task \( t \in T : \text{DStable}[t][l] = RS \)

If these conditions are satisfied, either one of the two conflicting transitions ck3 (representing condition **a** and **b1**) or ck4 (representing condition **a** and **b2**) fires; otherwise transition ck5 (with lower priority) fires.

The model needs to check the state of the DS table and indeed the four places of labels R, RS, LR, NR are going to be superposed with the places of equal label in the model of the DSTable of Figure 6.

The SWN model of Figure 6 is obtained from the automaton of Figure 1. The DSTable is modified every time
an event on any task occurs, like when a message sent by a
task or by the backbone is received by the DS, and when a
synchronization takes place.

Places all_lev and all_tks, on the right portion of the net,
have been introduced since SWN do not allow marking de-
pendent arcs. All other places are colored with the pair
“T,L”, since there are as many entries in the table as the
cross product of tasks and levels. Initially all entries of the
map are set to the value “not reached”, and therefore the
only non empty place in the net is place NR.

The model is interfaced with the DS model through tran-
sition labels tkfailed, ready and synchro, and through place
label LR, and with the check_synch model though place la-
bes R, RS, LR, and NR.

The program algebra has been used to produce the
complete model; the command:

```
algebra net1 net2 par1 net3
```

composes the SWN models of name net1 and net2 over
places or transitions, or both, depending on the value (p, t,
or b) of par1, and names the resulting net net3. The model
of the whole system has been produced using the following
script:

```
algebra tasks bk t tk-bk
algebra ds check-synch b ds-check
algebra tk-bk ds-check p tk-bk-ds2
algebra DStab tk-bk-ds2 b final
```

State space analysis  The model has 31 tangible states and
731 vanishing for 2 tasks and 2 levels, and it can be very
quickly solved with the SWN solvers of GreatSPN. We have
also produced a version of the model in which tasks are not
restarted upon a failure, that is to say, when a task fails it
does not re-enter play. This was simply done by deleting
the transitions with label RESTART in the task model: we
call this modified model “open”. The run of GreatSPN now
reveals 15 tangible state, 385 vanishing and a single dead-
lock that is easy to interpret as the state in which both tasks
have failed. The number of vanishing would have been bigger
if not for the extensive use of priorities in the model, to
avoid the construction of unnecessary interleavings.

To investigate further we have used GreatSPN-to-
PROD and PROD. The closed and open models have been
translated into PROD using GreatSPN-to-PROD, and this
has required a little help from the user (indeed, as explained
in [9], PROD does not allow inhibitor arcs, so the translator
builds a complementary place for each inhibiting place, but
then user intervention is needed to define the initial mark-
ing of the complementary place and the associated arc func-
tions).

Using the macro of GreatSPN-to-PROD we are able to
easily check a number of properties (see [9] for more
details), and for the open model we have computed the
shortest path from the initial marking to the deadlock state
using the “PathTOdeadlock” macro. The resulting path is
displayed by PROD as follows:

```
0#PathTOdeadlock
Arrow 1: transition tk1|FAULT, x = 3 l = 0
Arrow 0: transition snd|tkfailed, x = 3
Arrow 0: transition failure, x = 3
Arrow 1: transition t2|tkfailed, 1 = 1 x = 3
```
where each line says that a given transition has fired for a given instantiation of the variables. The translator has mapped the colour class L onto PROD colours 0 and 1, and T onto 2 and 3. The initial information on the arrow is not relevant in this context.

The closed model, as expected, has no deadlock, but we would like to check whether there are states in which the system is not performing any “useful activity”, so we have checked if there are states in which the only enabled transition is not performing any “useful activity”, so we have

4 Changing the user task model

The user tasks considered in the previous section reach the synchronization levels in order, that is to say: a new ready message is sent only after a message of “reached synchronization” on the previous level has been received, and all tasks follow the same sequence of synchronization level. This task model is adequate, for example, when the DS is used to synchronize pieces of computations as in a sequence of fork and join statements, but it is not adequate in other cases, for example, the case of the TIRAN mechanism called distributed memory, that manages a replicated and distributed set of variables. The distributed memory tasks, that are running on different nodes, use the DS to synchronize the distributed writes: there is one level per variable, and there is no reason why levels should be ordered, since it is indeed possible that a new request for a write on a variable arrives before the previous request (on another variable) is over.

The only component of the SWN affected by this change is the task model of Fig. 3(a): the initial marking has been changed so that in place tk1 there is now one token per element in the Cartesian product of the color classes T and L, and the successor function has been deleted from the function on the arc from transition rcv-global-synch to place tk1. Also the colors have been changed, since there is no need to keep the color class L ordered and split in static sub-classes, so the new definition is \( L = u L_v \) and \( L_v = \{ 11, 12 \} \).

The model has been produced by running again the same script of algebra commands as in the previous section, simply substituting the old model of the tasks with the new one. The analysis with GreatSPN reveals 97 tangible and 2595 vanishing states for the case of 2 tasks and 2 levels, the steady-state analytical solution produces “reasonable” results (all throughputs are different from zero and there is throughput conservation along paths), but the model contains an error, as we shall see in the following, caused by an incomplete specification of the DS behaviour.

Following the procedure described in the previous section we have run the GreatSPN-to-PROD translator and used the interactive facility of PROD to perform the reachability analysis, on both the open and closed models.

The closed model has the following characteristics: 1 single non trivial terminal strongly connected component (livelock) of 2491 states and 301 strongly connected components of 1 state each. It is not surprising that the steady state results are “reasonable”, since what is happening is that the system evolves through a number of transient states until it reaches the non trivial strongly connected component, where the system stays forever. What is instead unclear is what causes such behaviour. Since the analysis of a livelock of 2491 states is quite time-consuming we have preferred to check the open model first.

The open model presents 4 livelocks of 14 states each, and a single deadlock. The deadlock is the expected one: both tasks have failed and since there is no repair the model is stuck. The livelocks are instead more subtle to understand, but they are of limited size, so we have proceeded as follows: we have used PROD to construct the shortest path from the initial marking to one of the livelocks, and to check the possible execution paths inside the livelock. The first observation that we could draw is that the 4 livelocks present a very symmetric behaviour, that is to say they...
represents the same behaviour but for different pairs (task, level).

By taking a closer look at a single livelock we could observe that the problem found is related to the model component called check_synch. Indeed a possible state of the DS table is that task t1 is in state NR for both levels l1 and l2 and that task t2 is in state R for both levels (implying that two ready messages have been sent by t2, but since t1 has not reached the synchronization point on either of the two levels, no global synchronization has taken place). If at this point task t1 fails, both its entries in the table become NA (not available) and task t2 should be able to pass the synchronization on both levels, but the check_synch model is such that the check ends when the first synchronization is reached.

Is this an error in the model or in the specification? And is our analysis of the causes of the problem right? That is to say: it is true that the DS mechanism does not synchronize tasks as soon as possible? We have decided to express this question as a property to be checked by PROD. Informally the property should state that if the DS table is in a state such that the synchronization over a level can take place, then it should. But what does it mean “it should”? We have formalized our intuition by saying that from any state in which the column for level l of the DS table allows the synchronization over l, then all paths out of that state eventually lead to a state in which the firing of the synchronization transition takes place, passing only through states that do not alter the column l of the table. If α is the condition over the column of the table and β is the condition of reached synchronization, then we need to check that, for all the states of the system:

\[ \text{NOT}(\alpha) \lor \alpha \text{ Until } \beta \]

that is to say, either the table is in a state in which the synchronization is not allowed, or the synchronization takes place in a future state, but passing through states in which α keeps holding. Both α and β have to be expressed as marking condition, that results in α, for the specific level l1, being equal to the following in PROD syntax:

\[
\begin{align*}
&\{ (\text{not } (\text{NR} >= <t1, l1>)) \land \\
&\quad (\text{not } (\text{LR} >= <t1, l1>)) \land \\
&\quad (\text{not } (\text{LR} >= <t2, l1>)) \\
&\land \\
&\quad ((\text{R} >= <t1, l1>) \lor (\text{R} >= <t2, l1>) \lor \\
&\quad (\text{RS} >= <t1, l1>) \lor (\text{RS} >= <t2, l1>))
\}
\]

The condition for β is much simpler, because when there is a synchronization over level l1 a token of color l1 is put into place reached-synch; this translates into:

\[
(\text{reached-synch} >= <l1>)
\]

The test with PROD reveals that the property does not hold on all reachable states neither in the open nor in the closed model, and therefore the DS does not behave as expected.

## 5 Fixing the specification

To fix the problem we first thought that it was enough to change the check-synch method (and the corresponding model), but this is not quite the case. Indeed the check-synch model of Figure 5 can be changed by deleting the transition called “emptying”, so that all levels are checked, but this does not work, as the composed model immediately reveals a deadlock since every time that a synchronization on a level has been reached (firing of transition of label “reached”), this causes a synchronization with the DS model, that synchronizes with the DStable model to change the entries corresponding to the synchronized level, but the DS model was not meant to iterate the interaction of reached synchronization with the check-synch model more than once, and this causes a deadlock. This problem is not a wrong interpretation by the modellers of the specifications, since there is clearly no iteration of the DS automaton over reached synchronizations, but instead a case of underspecification of the system.

The solution adopted was to limit the check-synch method (and therefore model) to check the synchronization over a single specific level that is an input parameter for the method (a single token in place of the place of label “levels”), and to change the behaviour of the DS automaton of Figure 2 (model of Figure 4) so as to call the check-synch a single time in normal situation (a ready message from a user task is received), and as many times as there are levels in the case of a user task failure. The modified check-synch model is shown in Figure 7 and it is clearly simpler than before. The modified DS is shown in Figure 8: the model execution now cycles |L| times through the paths [check-synch, reached, snd-global, ds7, t14] or [check-synch, not-reached, t14], before coming back to the initial state represented by place wait-msg.

The model of the whole system has again been produced with algebra and translated into PROD. The property \(\text{NOT}(\alpha) \lor \alpha \text{ Until } \beta\) is now satisfied on all states, but we still have a livelock in the closed model (of the 2443 states of the model 2110 belong to a livelock). Clearly the model is not able to reproduce the initial state, but why?

We took a closer look to the model and decided to check how many states of the model have the same values of the table entries as in the initial marking, by checking the following query on all nodes:

\[0\# \text{query node } \text{NR} == 1<t1..t2, l1..l2>\]

that builds the set of markings with all table entries equal to NR (place NR contains all pairs of tasks and levels).
The set built by PROD contains 15 markings and it was automatically named %4. If we now build the intersection among %4 and the livelock set (named %0 by a previous PROD query), with:

\[ 0 \# \text{build} \ %0 \& \ %4 \]

we get an empty set: none of the markings of the livelock has the table entries all set to NR. A posteriori this is not surprising, indeed after the first synchronization takes place there is always a level for each task that is equal to LR (to memorize the last level on which the task has synchronized). This behaviour has been approved by the DS designers, and therefore we have not modified the initial marking.

On the final closed model we have performed our experiments, and we report here only an example of study of the size of the reachability graph and of a performance index.

Table 1 reports the number of vanishing and tangible markings for various values of tasks and levels. The first two columns are the ordinary marking, while the last two columns are the symbolic markings, obtained using the symbolic reachability graph solver of GreatSPN. The size of the Markov chain to be solved is given by the number of tangible symbolic markings, and it can be seen that the saving can be quite significant.

As an example of a performance index we have considered the mean waiting time of tasks in place \( \text{tk5} \), where the tasks wait for a message of reached synchronization. The diagram of Figure 9 plots two curves of the mean waiting time in front of a synchronization barrier for a number of tasks of 2 and 3, all with 2 levels, with the rate of the timed transition “working” on the x-axis: as expected the waiting times are larger for the three tasks case, and the faster the task activity the smaller is the waiting time. The two curves have been obtained analytically, while for number of tasks larger than 4 simulation is the only possibility. The diagram of Figure 10 plots instead a similar situation of tasks and levels, but under the hypothesis that no fault can happen. By comparing the two diagrams it appears that the

<table>
<thead>
<tr>
<th>tasks</th>
<th>levels</th>
<th>RG</th>
<th>SRG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TM</td>
<td>VM</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>38</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>98</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>22</td>
<td>339</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>73</td>
<td>2370</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>130</td>
<td>2390</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
<td>202</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1133</td>
<td>45510</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>307</td>
<td>15580</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>15636</td>
<td>898510</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>754</td>
<td>15317</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>388</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>15841</td>
<td>686454</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1451</td>
<td>99390</td>
</tr>
</tbody>
</table>

Table 1. Ordinary and Symbolic Reachability Graphs.
presence of faults decreases the waiting time, which may appear counter-intuitive, but it is perfectly right, since the DS mechanisms does not block, waiting for failed tasks, while trying to reach a global synchronization.

![Mean waiting time to synchronize: fault case](image1)

**Figure 9. Performance measure for the DS**

![Mean waiting time to synchronize: absence of faults](image2)

**Figure 10. Performance measure for the DS**

6 Conclusion

In this paper we have presented the validation and evaluation process for a software solution to distributed synchronization. We have a long list of items that deserve more attention in the future, and of things that we have learned from this case study.

The first observation is that for the analysis from the early stages of the design to be effective it should be fast and not be perceived as a non useful add-on to the project (evaluation from the early stages encounters similar problems as formal specification of systems), and that for a model to be correct it is essential that the specification comes with “properties” to be proved, since the fact that a model has an ergodic behaviour is obviously not enough to tell that its behaviour is correct. If (reasonably formal and complete) specifications are not available, then it may be better to limit the study to the macro behaviour of the system (point 3 of the introduction list).

To be fast enough for the results to be useful, the availability of compositional facilities and of an experiment planner is a necessity.

Symmetries in the model should be exploited also in the analysis: indeed the case of the four livelocks that are instances of the same problem that we have encountered in the case study could have been better tackled if a single parametrized livelock was reported.

Large models can be very tricky, and although the obvious answer is that they can be solved using simulation, still there is a need to show that the model used for the simulation is adherent with reality. Model checkers are well known to validate systems with billions of states using very efficient solution methods, but our experience with PROD showed that no efficient and tricky solution methods could be used due to the presence of priorities over transitions. We have not yet investigated other model checkers, but we shall do it in the near future since, if on one side PROD has the big advantage of allowing properties to be expressed as net elements, on the other side it is not very user-friendly. In particular we found it quite difficult to specify properties, both conceptually (choosing the “good property” is not an easy tasks for performance engineers) and syntactically.

The model that we have built has basically no time associated to the synchronization mechanism. If timed transitions are introduced in the check activity or in sending and receiving of messages through the mailboxes then the number of states increases very significantly: a phenomenon that we still do not master completely.

As a final comment let us draw your attention on the problem of modelling failure: in the model we have assumed that tasks can fail only when they are in the working state, but what happens if instead they can fail in any state, for example while waiting for a synchronization? The model can change significantly and the number of states can increase accordingly.

References


