Int J STTT (2001) 3: 417–430 / Digital Object Identifier (DOI) 10.1007/s100090100065

Implementing compositionality for stochastic Petri nets

S. Bernardi, S. Donatelli, A. Horváth

Dipartimento di Informatica,Università di Torino, Torino, Italy;E-mail: {bernardi,susi,horvath}@di.unito.it

Published online: 24 August 2001 –

Abstract. An implementation of compositionality for stochastic well-formed nets (SWN) and, consequently, for generalized stochastic Petri nets (GSPN) has been recently included in the GreatSPN tool. Given two SWNs and a labelling function for places and transitions, it is possible to produce a third one as a superposition of places and transitions of equal label. Colour domains and arc functions of SWNs have to be treated appropriately. The main motivation for this extension was the need to evaluate a library of fault-tolerant "mechanisms" that have been recently defined, and are now under implementation, in a European project called TIRAN. The goal of the TIRAN project is to devise a portable software solution to the problem of fault tolerance in embedded systems, while the goal of the evaluation is to provide evidence of the efficacy of the proposed solution. Modularity being a natural "must" for the project, we have tried to reflect it in our modelling effort. In this paper, we discuss the implementation of compositionality in the GreatSPN tool, and we show its use for the modelling of one of the TIRAN mechanisms, the so-called *local voter*.

Keywords: Stochastic well-formed nets – Compositionality – Modularity – Fault tolerance – Performance analysis

1 Introduction and motivations

TIRAN (tailorable fault tolerance framework for embedded applications) is a European ESPRIT project, involving six European partners from industries and universities, that is defining and implementing a new approach to fault tolerance in embedded systems. The solution in TIRAN is built around a software solution which provides fault tolerance capabilities to automation systems. TIRAN basically consists of a library of functions that add fault tolerant behaviour to software, a support for their execution, and a language to specify how to react to errors: this is what is called "the TIRAN framework". The framework is meant to allow application programmers to equip their programs with a variety of *software-based* solutions for fault masking, error detection, isolation, and recovery. The framework is currently under development on different hardware platforms, while two pilot applications are being developed to test the framework [5].

Modelling in TIRAN is meant for validation, evaluation, and library documentation. Models are built out of a software specification document. Validation is done with respect to a number of "environment scenarios", that include the application model, fault model, and recovery action specification.

The role of the modelling team in the project is to provide models of the system software and of the individual mechanisms, where each mechanism consists of a set of functions that implement a given fault treatment aspect. Their models have to be flexible to be easily composed with models of the different target applications developed by the partners as test cases for the library. Moreover, each mechanism is usually a set of tasks that interact through the communication primitive offered by the runtime support. Due to the high complexity of the problem, coloured nets [18, 19] have been used for most mechanisms. Important required features of a tool to work with coloured nets include an efficient solution mechanism [7, 8], as well as modularity, reuse, and easy modifications of models.

Figure 1 shows the compositional approach of TIRAN modelling. Starting from the specification of the components (whether they are TIRAN mechanisms or hardware specifications, or user behaviour) a model of each

We acknowledge contribution of the EEC project 28620 TIRAN.

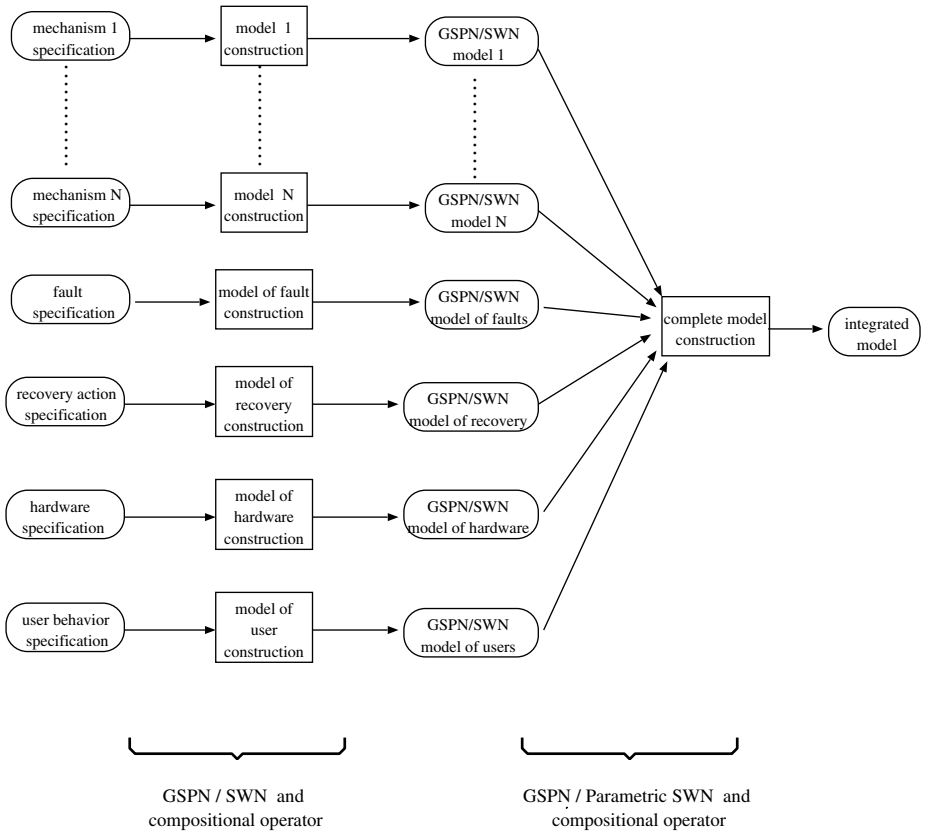

Fig. 1. Compositional modelling in TIRAN

component is built. The different models are then integrated into a single SWN model for evaluation of the whole TIRAN framework. There are a number of boxes all tagged as "model construction", but they may actually require different construction techniques since the input specification can be of different types: the specification of the mechanisms is done with UML diagrams (typically, state diagrams and message charts) plus some textual comments; fault specification is instead taken from the requirements specification document that describes the type of faults and the type of the affected component in a semi-formal language; hardware specification has not been included yet; while the user specification is again based on the requirement specification document.

Most of the mechanisms are built as collections of tasks, and usually there is a state diagram per task, plus a specification of the interactions among them: the corresponding models are built using SWN and the compositional facility of GreatSPN described in this paper.

The construction of the integrated model is, in general, a more complicated task, since this is where model reuse really comes into play. To adequately support this composition, the PSWN class [1] has been recently defined, that allows the definition of parametric colour classes, and for which a compositional operator has been defined that allows importing and exporting values and types. Unfortunately, no implementation is yet available for PSWN.

SWNs were a natural choice in the project for their efficient analysis techniques both for state space generation and for performance evaluation based on aggregated Markov chains. It was natural to choose the tool GreatSPN [9], since it supports state space generation exploiting symmetries, steady state computation exploiting lumpability, and discrete event simulation with confidence interval computation. On the other hand, it also has a number of weak points: a) there is no support for modularity; b) there are very few tools for debugging the model (no invariant computation or check, no reachability analysis is possible apart from checking the properties that are more relevant for performance evaluation, typically the presence of a home state, or to do an inspection of the reachability graph written in ASCII form); and c) there is no concept of a parameterized "library of models".

To overcome these weak points a number of activities are planned and/or are under implementation by the GreatSPN group. In particular:

1. To implement composition over places and transitions for SWN.
2. To implement PSWN [1] to allow reuse of coloured models in different contexts.
3. To export GreatSPN models to PROD nets [23], so as to apply all PROD tools for reachability analysis.

4. To export GreatSPN models to AMI-nets [12], so as to apply the P-invariant computation for coloured nets [11] that is available in CPN-AMI [12].
5. To extend the definition and to implement the $\mathcal{PSR}$ methodology [13] for the modelling of layered hardware and software architecture.

This paper discusses the implementation of the first point, and shows its use in a non-trivial modelling case taken from the TIRAN library: a mechanism to implement application replication and voting.

Of the remaining points, the export to PROD nets is at the testing stage, while the export to AMI-nets has not been started yet. The implementation of PSWN is a non-trivial extension of the composition operator that has been implemented. Indeed, the compositional rule implemented is not very sophisticated from the point of view of the treatment of the colour classes: there is no concept of parametric colour classes, nor of import and export values and types, as is the case in the PSWN class defined in [1].

There are a number of techniques proposed in the literature for the composition or compositional analysis of high-level models: [2, 4, 6, 10, 15, 20], and a very thorough survey of these methods can be found in [20]. In this paper, we concentrate only on tool support for compositional construction, and not for compositional analysis.

To the best of our knowledge there is no other tool offering the possibility of composing stochastic coloured nets based on labelling, although composition based on labelling is a well-established technique for Petri nets [3], and there is an implementation available for a class of high-level nets called M-nets [4], that do not include a notion of time, and that have been used to provide a semantic to the programming language $\mathrm{B(PN)}^2$ [4].

However, there are tools to also assist modular modelling in a stochastic context. CPN-AMI gives the possibility to paste nets next to each other in one model and use the modular services of the graphical interface for the fusion of places or transitions [12]. Design/CPN makes use of hierarchy to assist the user in building complex models: a transition that represents a complex activity may be replaced by a subnet [14]. A similar approach is taken in HiQPN [17], that also offers a larger number of performance evaluation features. UltraSAN models may be combined using the operations *replicate* and *join*; these operations provide common places that can be used for communication between the submodels [22].

The choice of GreatSPN was made because of obvious practical considerations (such as the availability of the source code and of its knowledge), but mainly because of the need to have aggregated symbolic state space generation like the one provided by GreatSPN for SWN (the same analysis is also provided by CPN-AMI, but through an export to GreatSPN).

Section 2 introduces the compositional rule of SWNs and its implementation in GreatSPN. Section 3 describes the local voter mechanism, as given in the TIRAN specification document, while Sect. 4 shows and discusses the SWN models of the local voter. Some results derived from the models are given in Sect. 5. Section 6 concludes the paper.

2 Composition of SWNs in the GreatSPN tool

The proposed composition rule is based on the known concept of "matching labels": transitions and places are labelled and pairs of transitions (or places) with matching labels are superposed. Let L_T (L_P) be the set of labels for the set of transitions T (or places P). The labelling function is denoted by λ, it assigns a subset of the powerset of L_T (L_P) to each transition (place). This implies that more than one label can be associated with a single place or transition (we call this aspect "multilabelling", and it should not be confused with multilabelling in Petri Boxes [3], that refers to a bag of labels). Although the formal definition of superposition is not given here for reasons of space and because it is a simplification of the one presented in [1] for transitions (by not considering parametric colours), we recall the main ideas here through examples. From now on we assume that only the first operand is multilabelled; this restriction is an adequate compromise between the complexity of the implementation and the applicability of the operator. A formal definition of SWNs is given in the Appendix A.

We assume that there are no marking dependent weights and rates and, first, we concentrate on the case in which only transitions are superposed. The two labelled SWNs to be composed are denoted by $\mathcal{N}_i = \langle P_i, T_i, \mathbf{Pre}_i, \mathbf{Post}_i, \mathbf{Inh}_i, \mathbf{pri}_i, \mathcal{C}_i, cd_i, \mathbf{w}_i, \lambda_i \rangle$, $i = 1, 2$. Where P_i is the set of places, T_i is the set of transitions, $\mathbf{Pre}_i$, $\mathbf{Post}_i$, $\mathbf{Inh}_i$ are the functions describing input, output, and inhibitor arcs, $\mathbf{pri}_i$ is the transition priority function, $\mathcal{C}_i$ is the set of basic colour classes, cd_i defines the colour domain of places, and $\mathbf{w}_i$ is the function that associates rates to transitions. The net resulting from the composition is denoted $\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{Post}, \mathbf{Inh}, \mathbf{pri}, \mathcal{C}, cd, \mathbf{w}, \lambda \rangle$. Then, the elements of $\mathcal{N}$ are obtained as follows.

The set of places P is the union of the sets of places, i.e., $P = P_1 \bigcup P_2$. The set of basic colour classes $\mathcal{C}$ and their definitions are assumed to be common for $\mathcal{N}_1$ and $\mathcal{N}_2$. The colour domain function cd is $cd_1(\mathtt{p})$ if $\mathtt{p} \in P_1$, $cd_2(\mathtt{p})$ otherwise.

The unlabelled transitions are considered non-observable with respect to the composition, and those whose labels do not appear in the other operand are not involved in superposition. These transitions are simply copied into T. Remember that $\mathcal{N}_1$ is multilabelled, $\mathcal{N}_2$ is not, and the labelling is non-injective. Let $T_2(l)$ denote the set of transitions `t'` of T_2 with $l \in \lambda(\mathtt{t'})$, where $\lambda(\mathtt{t})$ gives the set of labels of `t`. In $\mathcal{N}$ there will be a copy of $\mathtt{t} \in T_1$ for each element in $\times_{l \in \lambda(\mathtt{t}), T_2(l) \neq \emptyset} T_2(l)$, where $\times$ is the Cartesian product. An example is shown in Fig. 2, for transition `t1`: $\lambda(\mathtt{t1}) = \{l1, l2, l3\}$ and the above defined Cartesian

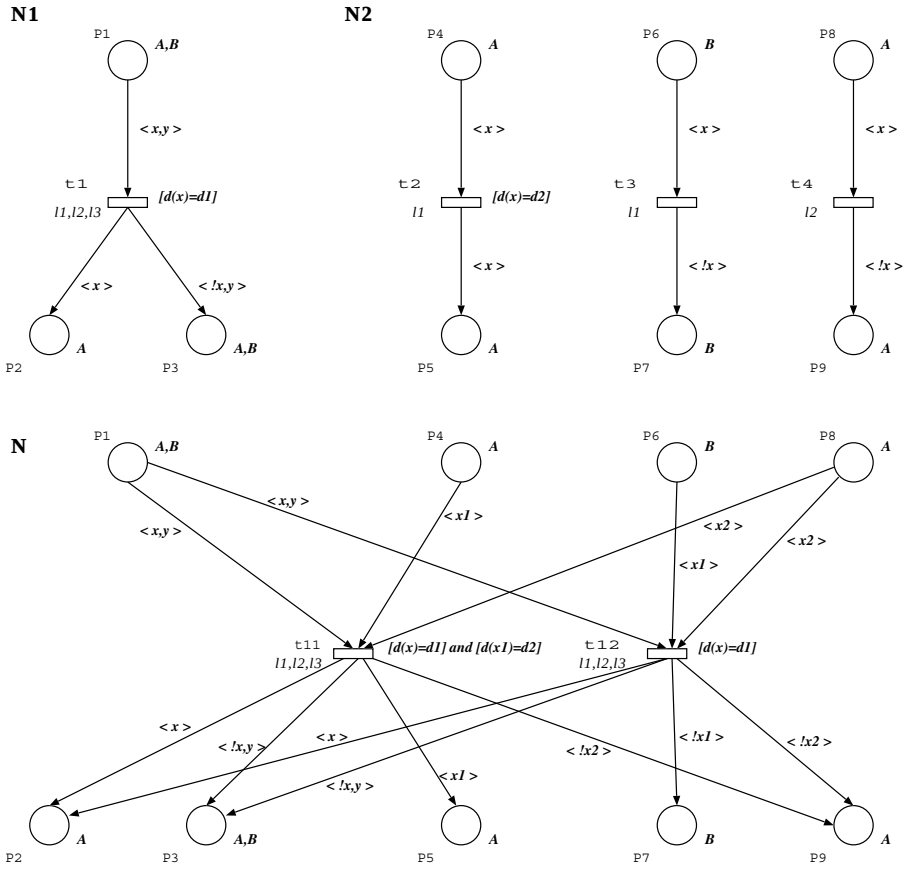

Fig. 2. A multilabelled, non-injective example

product has the elements $\{\texttt{t2}, \texttt{t4}\}$ and $\{\texttt{t3}, \texttt{t4}\}$. In the composed net `t11` is obtained by superposing `t1`,`t2` and `t4`, and `t12` by superposing `t1`,`t3` and `t4`.

If two arcs connected to different transitions involved in the same superposition have identical variable names in their arc expression, then these variables are renamed in the arc expression of all the arcs connected to one of the two transitions. If these variables appear in the guard of the transition whose arcs' expressions are changed, the renaming is performed in the guard as well. As an example, in Fig. 2, during the superposition of `t1`, `t2` and `t4`, the variable x of the arcs and guard function connected to `t2` is renamed to $x1$. (As will be mentioned later, the implemented version of the algorithm allows the user to override the above described renaming rule to "unify" values of the variables.) When two superposed transitions both have a guard function, these guard functions are joined with a logical *and* relation.

The matrices **Pre**, **Post**, **Inh** describing the arc structure of $\mathcal{N}$are built in the following way. The arcs of $\mathcal{N}_1$ and $\mathcal{N}_2$ connected to transitions that are not involved in superposition are simply copied into $\mathcal{N}$. An arc connected to a transition involved in superpositions will have as many instances as many times the transition is superposed. In our example, the arc `P1-t1` has two instances in the composed net: `P1-t11` and `P1-t12`.

The priority function **pri** is left unchanged for the transitions that are not involved in superposition. A transition resulting from superposition inherits the priority value from the involved transition of $\mathcal{N}_1$. The weight function **w** is handled similarly to the priority one.

We basically leave the user the task of redefining **pri** and **w** for the final net, since compositional ways to handle **pri** and **w** are still an open question, although some attempts to address this problem may be found in [16, 21].

The operation to superpose places is the direct counterpart of the operation described above, with the additional and obvious constraint that places of equal label should have the same colour domain. However, the superposition of places is less complex as it does not require renaming of arc or guard expressions.

The simultaneous superposition of places and transitions has two additional features. First, having an arc whose place (transition) is involved in n_p (n_t) superpositions, there will be $n_p \cdot n_t$ instances of the arc in the composed net connecting all the instances of its place with all the instances of its transition. Second, having two arcs whose places and transitions are superposed, the arc expressions of these two arcs are combined. An example for the latter is shown in Fig. 3 where the arc expressions of the arcs $(\texttt{P1}, \texttt{t1})$ and $(\texttt{P3}, \texttt{t2})$ are summed.

2.1 Implementation

The compositional operators described in the previous section are implemented by a program called **algebra**, that uses and produces SWN nets in GreatSPN format. The modeller may build the component nets using the graphical interface of GreatSPN. Since the present interface does not allow one to define labels, they are encoded in the name of the transitions and places. Both transition and place names have the structure `tag|l1|l2...`, where `tag` is the name of the transition or place followed by its labels separated by bars.

The user may define the set of labels which will be taken into account during the composition. This feature may be useful when composing more than two nets and the modeller wishes to avoid that all labels are considered at all stages of the composition.

Right now **algebra** is able to deal with non-injective labelling in both operands, while only one of the two may be multilabelled. This was judged an adequate compromise between the complexity of the implementation and the foreseen use of the operator (this choice is adequate, for example, to support the implementation of the $\mathcal{PSR}$ methodology cited in Sect. 1 as point 5 of the "wish-list" of GreatSPN).

When composing **algebra** attempts to create a well-readable net, using knowledge on the component nets and on the operators. The "shape" of the original components are maintained. The user may define where the individual components are placed in the composed model. If a transition (or place) has multiple instances in the resulting net, the additional instances are placed around the original position of the transition. Renaming of transition and place names may be necessary in order to avoid matching names. When, as a result of the composition, an arc's place and transition are in different subnets, the arc is drawn as "broken arc" in the resulting net. A small example for the output of the program is given in Fig. 4[1]. Figures 4 demonstrates another feature of the program: if a variable name starts with the character #, it is not renamed during the superposition. This allows the modeller to use the same variables in different components, so as to "unify" values.

The tool **algebra** may be called from the command line as

```
>> algebra net1 net2 op labels net
      [placement shiftx shifty]
```

The two operands are `net1` and `net2`, the resulting SWN (GSPN, if the operands are GSPNs) is `net`. The operator is defined by `op` and may be `t` to superpose transitions,

[1] GreatSPN does not display in the graphical interface arc expressions on broken arcs; these have been written on the figure "by hand".

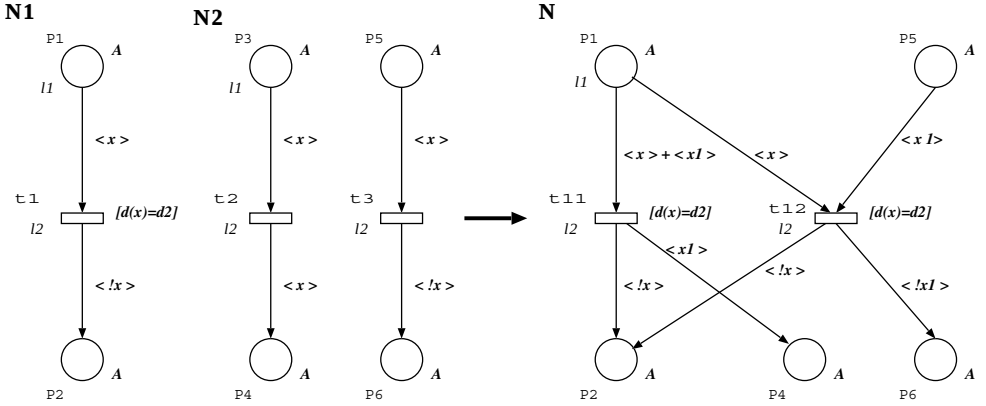

Fig. 3. Superposition of places and transitions

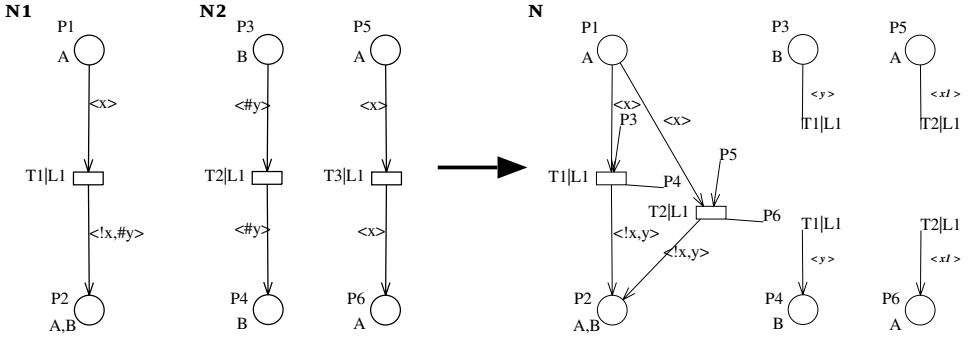

Fig. 4. Superposition using SWN

`p` to superpose places or `b` to superpose both places and transitions. The set of labels over which the superposition will be performed may be given in the file `labels`. This file has the following format:

```
transition={tl1|tl2}
place={pl1|pl2|pl3}
```

The labels that are not given in this file are not considered during the operation. If the file does not exist all labels are considered. The last three arguments may be used to define the placement of the components: if the parameter `placement` is 1 (2) the two nets are placed next to each other horizontally (vertically); if it is 3 the second net is shifted by (`shiftx,shifty`) compared to the first net. For example, to perform the composition depicted in Fig. 4 one calls

```
>> algebra N1 N2 t LABELS N 1
```

where the file `LABELS` either does not exist or contains the list of transition labels `transition={L1}`.

3 The local voter specification

The local voter (LV)mechanism aims at masking occurrences of faults during the execution of the code of an application process. Fault masking is achieved by the adoption of a spatial redundancy of the execution of the piece of code and by the voting on the results coming from the replica.

Depending on the voting technique adopted in the LV and on the spatial redundancy, a limited number of faults may be masked; for instance, by using a majority voting algorithm and by running concurrently K copies, up to $[\frac{K-1}{2}]$ faults can be made transparent for an application process.

Figure 5 shows a graphical representation of LV taken from the specification document of TIRAN; the LV can be used concurrently by several application processes and three replicas are considered per application.

The replicas are executed on separate "planes", that naturally correspond to separate processing nodes. The i^{th} application process that uses the LV mechanism is split in two parts; a part that does not require a replicated execution, and a part that requires it. If there are n applications that can use LV, then each application has its distinct piece of code to be executed.

Each replica of an application i executed on plane j should receive the same input data. Therefore, there is a task IR (input replicator) that performs replication of the input data for the three planes; the data does not go directly to the application replica, but to the input dispatcher of the corresponding plane ID_j, that takes care of passing it on to the appropriate application replica.

A similar approach is used to collect the results: when a replica of the i^{th} application, running on plane j, ends its computation, it sends its output data to the output collector of the corresponding plane OC_j that takes care of forwarding it to the appropriate voting task OV_i; there is one voting task per application.

The components of the local voter interact with the backbone, which is a sort of run-time support for the TIRAN library of mechanisms which handles all exceptions as well as the recovery actions. All interactions among tasks are based on communication through mailboxes.

Table 1 lists the acronyms used in the paper for the different LV tasks, and for each task lists how many copies of that task there are in an LV that serves n applications.

The OV behaviour is described by a state diagram in the specification document, that basically amounts to a 2-

Table 1. Acronyms

Acr.	Description	Do. of copies
APP	application	n
IR	input replicator	1
ID	input dispatcher	K
IST	replicated software to vote upon	$K*n$
OC	output collector	K
OV	output voter	n
BB	backbone	1

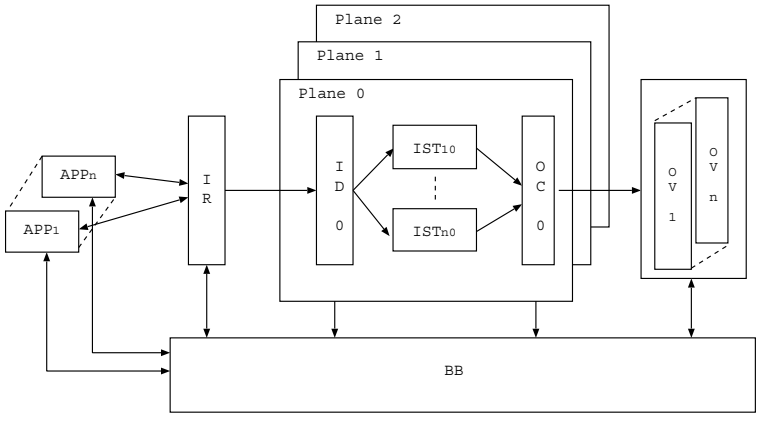

Fig. 5. A description of the local voter

out-of-3 voting (which is equivalent to setting $K = 3$). An additional textual specification also states that, as soon as OV_i for the i^{th} application receives the first output from one of the replicas, it sets a time-out for receiving the other replicas. Each OV sends a message (both in case of success and in case of failure) to the backbone that carries out the proper action.

4 The SWN model of the local voter

The following assumptions were made to model LV: tasks communicate in an asynchronous manner via mailboxes, and there is one mailbox for each ordered pair of tasks; the time required to prepare a message is in general negligible, while the time to actually transmit it from the task output buffer to the recipient mailbox is not. With respect to the graphical representation, we have used cross-lined places to emphasize mailboxes and grey-dot-filled boxes to delimit portions of the nets that correspond to "recovery actions", and that will be explained in the next section. We have also used different fonts in the figures in order to distinguish the different attributes of an SWN model; in particular, Courier font is used for names of transitions/places, italic font for colour classes, arc expressions and guards. Arc expressions are either variables or the diffusion function S. Transition priorities are written as $\pi{:}n$.

Three colour classes have been defined:

AP is the colour class of applications that can request a replicated execution of their code, and it contains a single static unordered subclass App defined as $App = \{ap1, .., apn\}$;

P is the colour of the planes, and it contains a single static unordered subclass Pl with three colors $Pl = \{pl1, pl2, pl3\}$;

Exc is the colour used to distinguish the positive or negative outcome of an LV activity, and it is the union of two static subclasses $Ecx1, Ecx2$, where $Exc1 = \{e1\}$ means that there has been a time-out expiration, while $Exc2 = \{e2\}$ means that there was no time-out expiration.

Figure 6 shows the SWN model of an application, that cyclically executes its own activity (modeled by transition `activity`), sends a message to the task IR of the local voter (`snd_LV`), and waits for a message coming from the backbone (`rcv_reply`). Transition `T1ap` represents the time for a message to reach the mailbox.

Figure 7 shows the SWN model of the input replicator IR: it waits for messages coming from the applications. As soon as a request of replicated execution for the

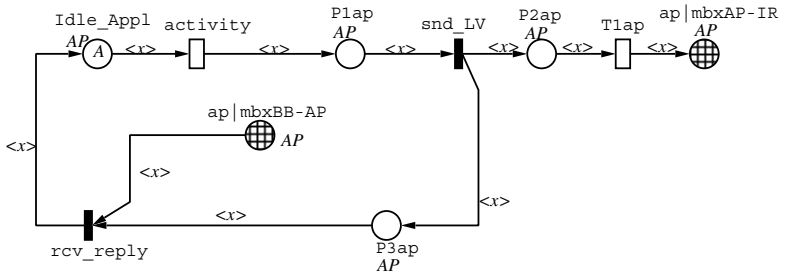

Fig. 6. The application model

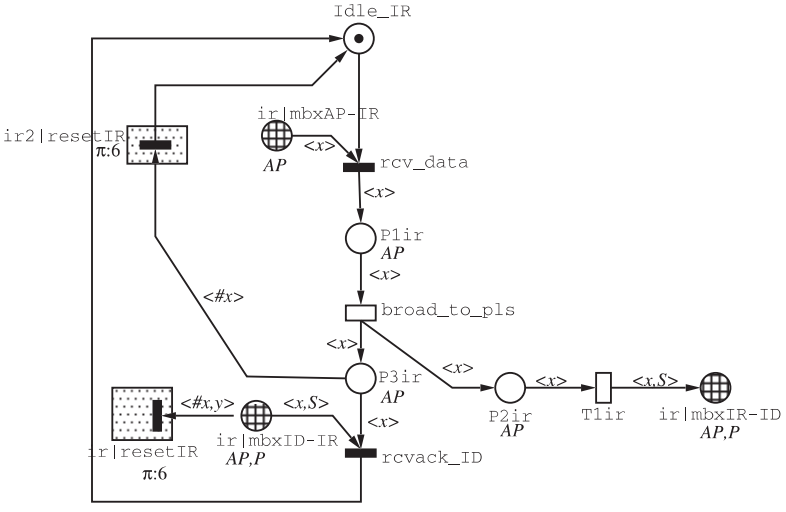

Fig. 7. The input replicator model

i^{th} application is received (`rcv_data`), it broadcasts it to the input dispatcher task of each plane (`broad_to_pls`), and waits for an acknowledgement from ID (`rcvack_ID`). Since there is only one IR task, then no colours are needed.

Figure 8 shows the SWN model of the input dispatcher ID. There is one task ID for each of the three planes: each ID receives from IR the identity of the application to be executed (`rcvdata_IR`), it acknowledges reception to IR (`sndack_IR`), and it activates a task corresponding to the requested replica (called instance) for that plane (`snd(datax,ply)_IST`).

Figure 9 shows the SWN model of the code to be executed on the different planes: since the TIRAN framework allows only static tasks, it is correct to assume that all replica are activated at the beginning and then suspend themselves waiting for a message from the ID tasks. There are $|\mathbf{AP}| \times |\mathbf{P}|$ instances, i.e., one for each application and for each plane. Each instance (x, y) waits for a message (x, y) from ID y (`rcvdata_ID`). When a message (x, y) is received the instance of application x on plane y starts its activity, modelled by timed transition `comp`, and then sends the result of the computation to OC (`sndrepl_OC`).

Figure 10 shows the SWN model of the output collector OC: there is one such process for each plane, and each OC waits for a message coming from the replicas running on its plane (`rcvrepl_IST`), and it forwards it to the output voter (`snd_toOV`). According to the textual portion of the specification, the OC should wait for a "ready message" from OV, but since the conditions under which OV should send this ready message are left unspecified, we assume that OV is always ready to accept messages in its mailbox.

Figure 11 shows the SWN model of the output voter task OV: there is an OV for each application that can use LV. Each OV executes the voting algorithm (majority voting 2 out of 3) on replicas of the same application, independently from the others. OV waits for the replicas outcome (place `idle_OV`) from the three different planes. As soon as the first outcome is received, a time-out for reception of the other two replicas outcome is set (`setTOforx`). Then three situations may occur:

C1 all the three outcomes are received before the time-out expiration, i.e., transition `recv3noTO` fires and voting on the three outcomes takes place;
C2 the time-out has expired and two of the three outcomes have been received (transition `recv2&TO` fires), and a vote on the two replicas takes place;
C3 the time-out has expired and only one of the three outcomes has been received by OV, i.e., firing of transition `recv1&TO`.

Different types of messages are sent to the backbone depending on which of the above conditions is satisfied. Under condition C1 a message of type $e2$ (no time-out has occurred) is sent; while in cases C2 and C3 a message of type $e1$ (a time-out has occurred) is sent. Observe that we are not passing on to the backbone the information on whether the vote was successful or not, although this will be a trivial extension, since the success or failure of the 2-out-of-3 algorithm, according to the state diagram of the

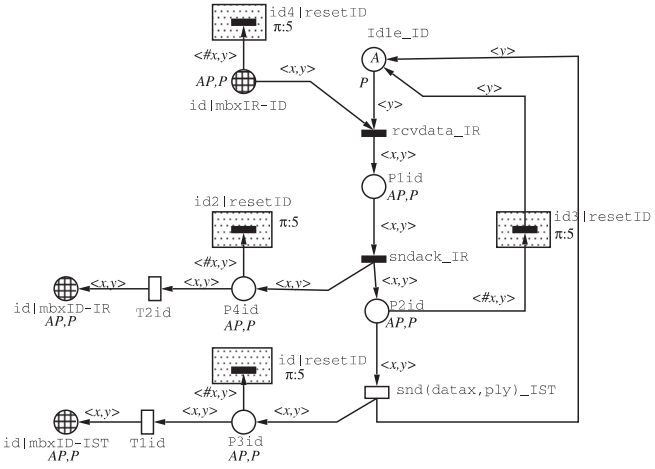

Fig. 8. The input dispatcher model

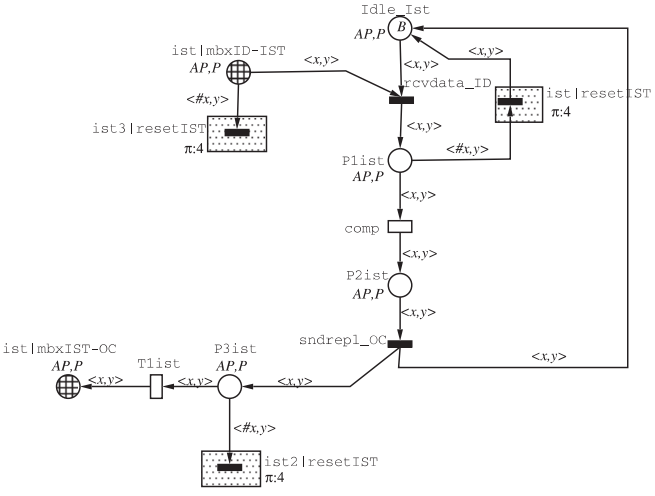

Fig. 9. The model of the replicated code

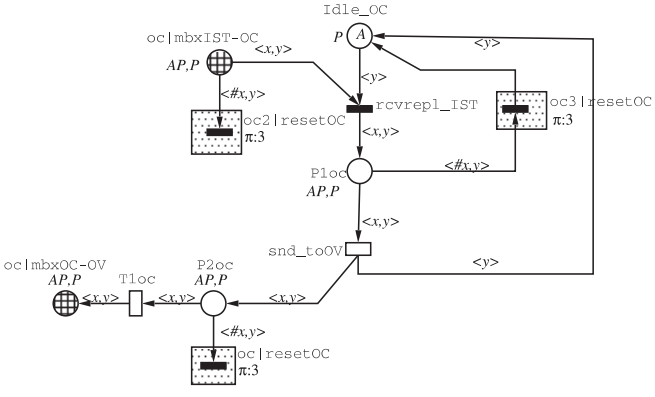

Fig. 10. The model of the output collector

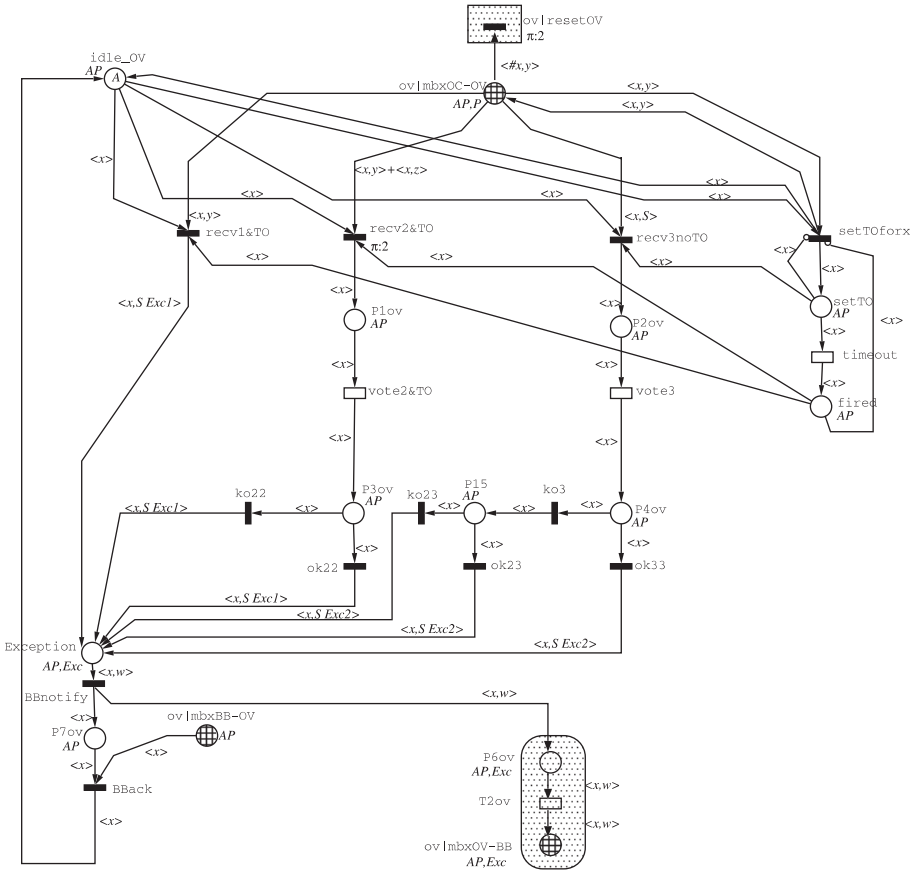

Fig. 11. The model of the output voter

specification document, is modelled in detail in the SWN of Fig. 11.

When a message is sent to the backbone, OV waits for an acknowledge from it to return back into its idle state. Observe that we are assuming that no direct answer goes back directly from OV to the application, not even in the case of a "normal" 3-out-of-3 voting, since we impose that all restarted are caused by the backbone.

Figure 12 shows the SWN model of the backbone task, or, more precisely, of that part of backbone devoted to interactions with LV. The backbone is in an idle state until it receives a message coming from OV. If the message is of type $e2$ (`rcv_OV_noreset`), i.e., no time-out has occurred, then the backbone sends an acknowledge to OV and to the application. If instead the message is of type $e1$ (`rcv_OV_reset`), then a time-out has occurred, and therefore a reset operation is needed, before sending back the messages to OV and to the application.

4.1 Local voter without recovery actions: an open model

A first analysis was performed for the case of a "single run" for each application. In order to obtain the complete model the individual nets have to be composed using the program **algebra** explained in Sect. 2. The nets used are the one without portions included in grey-dot-filled boxes, therefore no message is passed from OV to the backbone, so that each application is executed only once. The composed SWN net has been solved, for the single application case, using the symbolic reachability graph construction of GreatSPN, that produces 859 tangible states corresponding to 4,261 ordinary ones.

The SWN net results in an *open model* and its symbolic reachability graph is not strongly connected. There are three symbolic dead markings, corresponding to seven ordinary dead markings. Each marking is described in terms of dynamic colour subclasses associated with places, and, for each dynamic subclass, its cardinality is given. For each symbolic marking the number of corresponding ordinary markings is given; for instance, the following dead marking:

```
MARKING D856 # ordinary marking: 3 (dead)
Idle_BB(1)
oc|mbxOC-OV(1<App0,Pl1>)
Idle_OC(1<Pl0>1<Pl1>)
Idle_IR(1)
Idle_ID(1<Pl0>1<Pl1>)
Idle_Ist(1<App0,Pl0>1<App0,Pl1>)
P3ap(1<App0>)
Exception(1<App0,Exc10>)
|Exc10|=1 |Exc21|=1 |App0|=1 |Pl0|=1 |Pl1|=2
```

corresponds to a case of time-out expiration: only results from one replica has been received by OV and the time-out has expired while waiting for the remaining replicas.

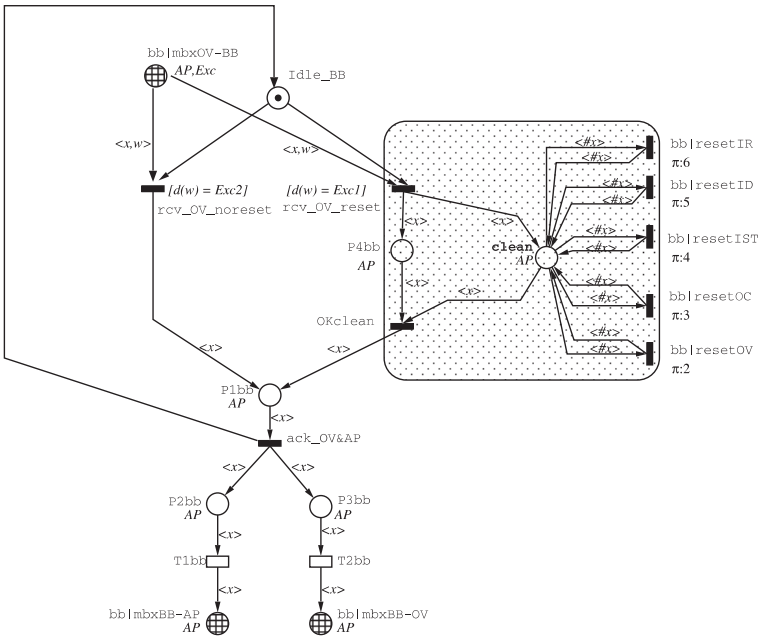

Fig. 12. The model of the backbone

All components, except OV and the application, are in their initial states (idle state), and the application and OV are both waiting for a message from the backbone, that will, of course, never arrive. Observe that a symbolic marking provides a more abstract information than an ordinary one: in this case the abstraction is on the identity of the replica (plane) that has finished first. Other deadlocks represent the case of reception of all the three replicas before the time-out occurs, corresponding to an ordinary deadlock marking, and the case of time-out occuring after the results from two copies have been received, corresponding to three ordinary markings.

4.2 Local voter and recovery actions: an ergodic model

All the deadlocks found describe a "good" (expected) behaviour, so that it makes sense to proceed to also add the reconfiguration activities needed to restart an application. The model obtained by composing all nets, including also the portions of the grey-dot-filled boxes, is ergodic (since there is a single strongly connected component in the reachability graph).
The recovery action taken by the backbone is:

- To remove messages from mailboxes.
- To take the corresponding tasks back to their initial states.

To accomplish this the backbone enables a number of immediate transitions, one per model component, and they are labelled in such a way as to superpose with the resetting transitions in the model components. Observe that these transitions are assigned a different priority, mainly to avoid the generation of useless interleavings, that could significantly slow down the state space generation.

The symbolic reachability graph for the single application case has 1,452 tangible states, while the ordinary one has 7,074. The initial marking is a home state, and therefore the model exhibits cyclic behavior returning infinitely many times to the initial marking. The generation took a few minutes on a 64 MB Pentium 2 machine.

4.3 Local voter without recovery actions and with explicit faults

In the models considered up to now no fault is explicitly included in the model, so that a time-out can expire only due to a delay in the completion of one of the replicas. In order to consider the effect of explicit faults the model of the replicated code has been modified to include a timing transition that models the fault and that takes replicas into an error state place. The resulting model, for the single application case, has 1,185 symbolic tangible markings (corresponding to 6,008 ordinary ones) and there are seven symbolic dead markings, corresponding to twenty ordinary dead markings. Among these, a very interesting one is the marking that represents the state of the model where all the instances are in an error state, and this corresponds to a case in which no replica will ever reach OV, so no time-out will be set. This case was, up to the modelling phase, overlooked by the specification document.

One may observe that to produce an ergodic model it will not be enough to consider the reset activity of the backbone; but an explicit testing for the three replicas all being in an error state should be added to the model. The resulting model, for the single application case, has 2,030 symbolic tangible markings (corresponding to 10,170 ordinary ones).

5 Results

Before proceeding to describe the results we summarize in Table 2 the default timing parameters assigned to transitions. The table contains the mean time needed to perform the operations (the mean is 1 over the rate assigned to the transition). All kinds of transmissions were given the same timing parameter referred to as *tr*. To gain insight on the local voter mechanism irrespective of the applications' characteristics, throughout all the experiments we assumed that the results given by the copies always match (the probability of having matching results is set to 1). However, using the presented models one may study the effect of varying the probability of match as well.

One important issue that may be addressed by the models described in Sect. 4 is the amount of overhead induced by the TIRAN framework with respect to the operation not protected by the local voter. The overhead may be analyzed using the ergodic model of Sect. 4.2 deactivating the time-out. Figure 13 depicts the mean time to execute a computation through the local voter (throughput of transition `activity`) divided by the mean time spent by a single replica to perform the operation (referred to as *comp* in Table 2) as the function of the mean transmission time (given in milliseconds). We used the default values of Table 2 during the computations, varying the mean delays *comp* and *tr*. The calculations were made having only one application which allowed us to compute the results analytically. The local voter mechanism as defined in the TIRAN framework involves the transmission of several messages. As a result, the slower the transmission the higher the overhead, which may lead to rather high overheads if the transmission time is comparable with the time needed to perform the operation.

Another finer point is the length of the time-out. This issue is studied again with the ergodic model of Sect. 4.2. Figure 14 plots the probability of different outcomes of one cycle as a function of the length of the time-out. As before, the parameters that are not given in the figure are set to the default values. These computations as well were performed with one application analytically. Not having explicit faults in the model means the time-out can expire only due to excessive delays of the computations on

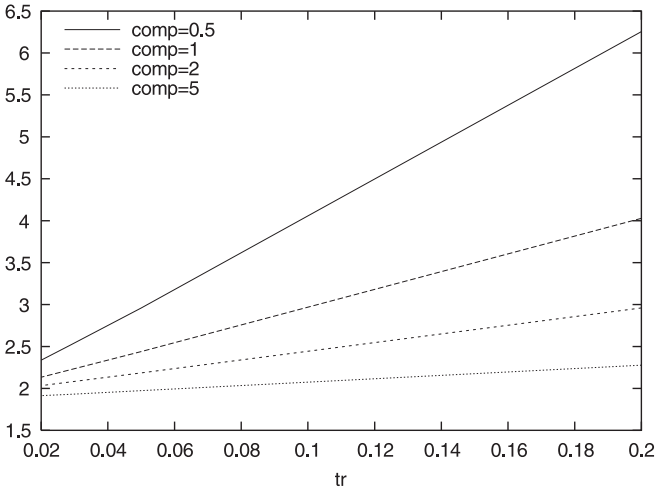

Fig. 13. Overhead induced by the TIRAN framework

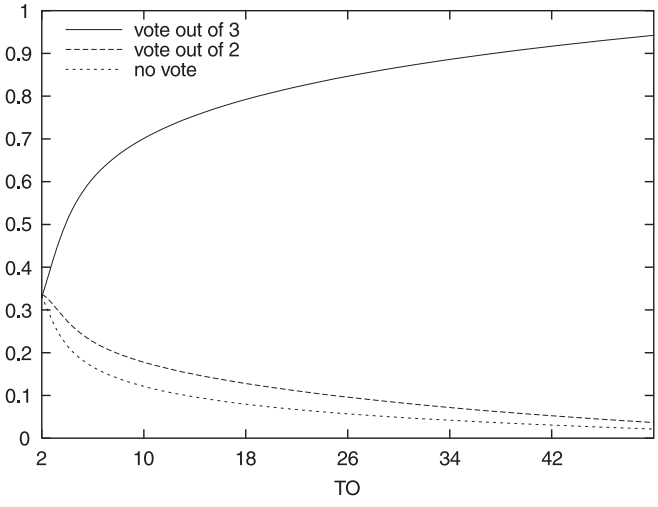

Fig. 14. Probability of the different outcomes of one cycle

the planes which implies that the probability of 3-out-of-3 voting converges to 1 as *TO* go to infinity.

Using the model described in Sect. 4.3 one can study the effect of explicit faults. Figure 15 depicts the probability of performing 3-out-of-3 voting at the end of a cycle as the function of the length of the time-out for different values of the *fault* parameter, the rate of the timed transi-

Table 2. Timing parameters

Param.	Description	Mean value (ms.)
act	time of normal operations of an application (transition `activity`)	1
tr	transmission time through mailboxes	0.1
broad	time for IR to prepare the broadcast messages to IDs (transition `broad_to_pls`)	0.1
snd_toist	time for ID to prepare a message to a same-plane instance (transition `snd_IST`)	0.1
comp	time spent by the instances to perform operations (transition `comp`)	2
snd_toov	time spent by OC to prepare the replica to OV (transition `snd_toOV`)	0.1
TO	time-out (transition `timeout`)	5
vote	time spent by OVs to vote (transitions `vote2&TO`, `vote3`)	0.2

tion that takes IST into an error state. The computations were performed with one application using the discrete event simulation available in GreatSPN with a confidence level of 99 % and accuracy of 10 %. Note that by having explicit faults in the model the probability of 3-out-of-3 voting does not converge to 1.

The ergodic model of Sect. 4.2 was used as well to gain some insight into the cases when the number of applications is higher. Since the runtime of the reachability graph generation grows very significantly into an almost trashing situation even when increasing the number of applications only by 1, we could only use simulation to compute performance indices, varying the number of applications n. Figure 16 gives the mean time needed to execute one application in milliseconds. Two cases are distinguished: a) the amount of resources in the system is assumed to be *infinite*, i.e., the execution of an operation does not make other jobs wait for a resource; and b) the case of limited resources, i.e., only one application may run on a given plane and only one message may be under transmission at a given time. Note that these two cases represent the two extremes, but by varying the service policies any situation between these two may be studied. The simulation was run with a confidence level of 99 % and an accuracy of 10 % with the default parameters of the model.

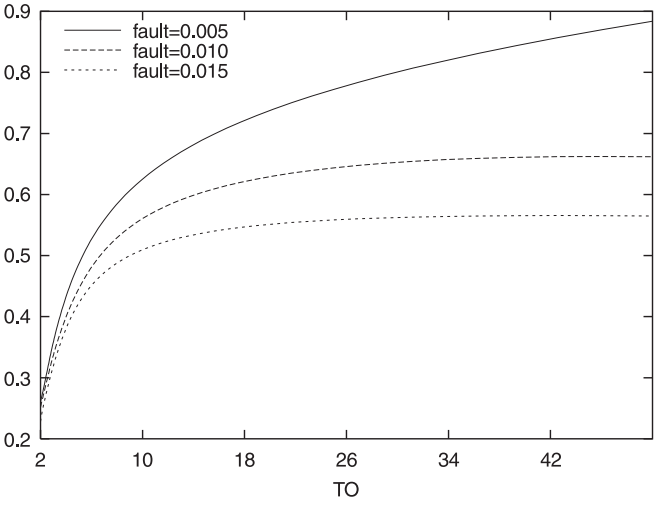

Fig. 15. Probability of 3-out-of-3 voting with explicit faults

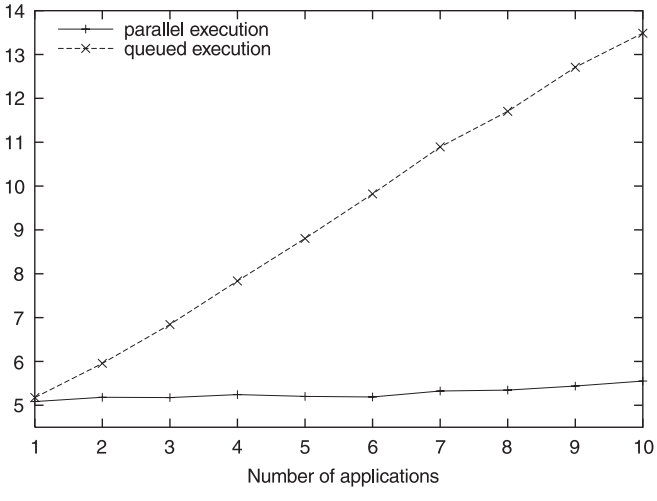

Fig. 16. Mean time to execute an application

6 Conclusions

In this paper we have described the compositional operator that is now implemented in GreatSPN for the superposition over labelled places and transitions for GSPN and SWN nets, and its application to the study of a mechanism for fault tolerance called local voter. Since specifications were changing and the definition of the abstraction level for modelling could not be set in the initial phase of TIRAN, tool support was fundamental in the model developing phase.

To provide more flexibility to these models we should include a modular definition of the communication (for example to investigate rendezvous instead of mailboxes), and different fault models.

Another open point is how to take into account the hardware on which the applications and the framework will run: indeed for this point we are planning to extend the definition of $\mathcal{PSR}$ [13] to SWN models. It seems envisionable that the size of the models produced will only allow simulation since they are quite detailed, being meant for validation, while an interesting open problem is how to derive from these models some more compact ones to be used for analytic performance evaluation purposes.

As we already mentioned in the introduction there is very little support for reachability analysis right now in GreatSPN. As a result, if a net has some unbounded places, the state space generation simply does not stop. To overcome this difficulty, before running the state space generation, we have always run simulation that allows us to check – by checking the accumulated performance indices – whether certain place markings grow in a suspicious manner.

Acknowledgments. We would like to thank the anonymous reviewers for their very careful reading of the paper, and for pointing out various ways to improve it.

References

1. Ballarini, P., Donatelli, S., Franceschinis, G.: Parametric stochastic well-formed nets and compositional modelling. In: Proc. of the 21st International Conference in Application and Theory of Petri Nets, ICATPN 2000, Aarhus, Denmark, June 2000. LNCS 1825. Berlin, Heidelberg, New York: Springer-Verlag, 2000, pp. 43–62
2. Battiston, E., Botti, O., Crivelli, E., De Cindio, F.: An incremental specification of a hydroelectric power plant control system using a class of modular algebraic nets. In: Proc. of the 16th International Conference on Application and Theory of Petri nets, Torino, Italy, 1995. LNCS 935. Berlin, Heidelberg, New York: Springer-Verlag, 1995, pp. 84–102

3. Best, E., Devillers, R., Hall, J.: The Petri box calculus: a new causal algebra with multilabel communication. In: Rozenberg, G. (ed.): Advances in Petri Nets, LNCS 609. Berlin, Heidelberg, New York: Springer-Verlag, 1992, pp. 21–69
4. Best, E., Fleischack, H., Fraczak, W., Hopkins, R., Klaudel, H., Pelz, E.: A class of composable high level Petri nets with an application to the semantics of B(PN)^2. In: Proc. of the 16th International Conference on Application and Theory of Petri nets, Torino, Italy, 1995. LNCS 935. Berlin, Heidelberg, New York: Springer-Verlag, 1995, pp. 103–120
5. Botti, O., De Florio, V., Deconinck, G., Lauwreins, R., Cassinari, F., Bobbio, A., Donatelli, S., Lein, A., Kufner, H., Thurner, E., Verhulst, E.: The TIRAN approach to reusing software implemented fault-tolerance. In: Proc. 8th Euromicro Workshop on Parallel and Distributed Processing (PDP2000), Rhodes, Greece, Jan. 2000
6. Buchholz, P.: A hierarchical view of GCSPN and its impact on qualitative and quantitative analysis. J Parallel Distrib Comput 15: 207–224, 1992
7. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: On well-formed coloured nets and their symbolic reachability graph. In: Proc. 11th Intern. Conference on Application and Theory of Petri Nets, pp. 387–411, Paris, France, June 1990. Reprinted in: Jensen, K., Rozenberg, G. (eds.): High-level Petri nets. Theory and Application. Berlin, Heidelberg, New York: Springer-Verlag, 1991
8. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: A Symbolic Reachability Graph for Coloured Petri Net. Theoret Comput Sci B 176(1/2): 39–65, 1997
9. Chiola, G., Franceschinis, G., Gaeta, R., Ribaudo M.: GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. Performance Eval 24(1/2): 47–68, 1995
10. Christensen, S., Petrucci, L.: Modular state space analysis of coloured Petri nets. In: Proc. of the 16th International Conference on Application and Theory of Petri nets 1995,, Torino, Italy, 1995. LNCS 935. Berlin, Heidelberg, New York: Springer-Verlag, 1995, pp. 201–207
11. Couvreur, J.M., Martinez, J.: Linear invariants in commutative high-level nets. In: Proc. 10th Intern. Conference on Application and Theory of Petri Nets, Bonn, Germany, June 1989
12. CPN-AMI.: `http://www-src.lip6.fr/cpn-ami.html`
13. Donatelli, S., Franceschinis, G.: The PSR methodology: integrating hardware and software models. In: Proc. of the 17th International Conference in Application and Theory of Petri Nets, ICATPN '96, Osaka, Japan, June 1996. LNCS 1091. Berlin, Heidelberg, New York: Springer-Verlag, 1995, pp. 133–152
14. Design/CPN.: `http://www.daimi.au.dk/designCPN`
15. Haddad, S., Moreaux, P.: Evaluation of high level Petri nets by means of aggregation and decomposition. In: Proc. of the 6th International Workshop on Petri Nets and Performance Models, pp. 11–20, Durham, N.C., USA, October 1995
16. Hillston, J.: The nature of synchronization. In: Herzog, U., Rettelbach, M. (eds.): Proc. 2nd Workshop on Process Algebra and Performance Modelling, pp. 51–70, Erlangen, Germany, 1994
17. HiQPN.: `http://www4.cs.uni-dortmund.de/QPN`
18. Jensen, K.: Coloured Petri nets, basic concepts, analysis methods and practical use. vol. 1. Berlin, Heidelberg, New York: Springer-Verlag, 1992
19. Jensen, K.: Coloured Petri nets, basic concepts, analysis methods and practical use. vol. 2. Berlin, Heidelberg, New York: Springer-Verlag, 1992
20. Rojas, I.C.: Compositional construction and analysis of Petri net systems. PhD thesis, University of Edinburgh, 1997
21. Teruel, E., Franceschinis, G., De Pierro M.: Clarifying the priority specification of gspn: detached priorities. In: Proc. 8th Intern. Workshop on Petri Nets and Performance Models, pp. 114–123, Zaragoza, Spain, September 1999
22. UltraSAN.:`http://www.crhc.uiuc.edu/UltraSAN`
23. Varpaaniemi, K., Halme, J., Hiekkanen, K., Pyssysalo, T.: PROD reference manual. Technical Report Series B, number 13, Helsinki University of Technology, August 1995

Appendix : SWN definition

The starting point in the structured definition of the SWN colour syntax is the set of *basic colour classes* $\{C_1, \ldots, C_n\}$. A basic colour class C_i is a nonempty, finite (possibly circularly *ordered*) set of colours; intuitively, a basic colour class can be defined as a set of colours identifying objects of the same nature. A basic colour class is ordered if a *successor function* is defined on its elements, such that it induces a circular ordering on the class elements. Examples are the class of the applications, the class of outcomes, the class of processors, and the class of memories. An example of ordered class is the class of processors connected in a ring topology. Basic colour classes are disjoint (i.e., $\forall i, j : i \neq j, C_i \cap C_j = \emptyset$), moreover, a class may be partitioned into several *static subclasses* ($C_i = C_{i,1} \cup \ldots \cup C_{i,n_i} \forall j, k : j \neq k, C_{i,j} \cap C_{i,k} = \emptyset$): colours belonging to different static subclasses represent objects of the same type but with different behaviour, for example the basic colour class of outcomes could be partitioned into two (disjoint) static subclasses, to distinguish the causes of the outcome.

The place colour domains, are defined by composition through the Cartesian product operator of basic colour classes. The colour domain of a place is similar to a *C*-language structure declaration, i.e., the information associated with tokens comprises one or more *fields*, each field in turn has a type selected from the set of basic colour classes $\{C_1, \ldots, C_n\}$. The identification of the fields is positional (there is no name associated with a field).

The transition colour domains are used to define the *parameters* of transitions and their type; each parameter has a type selected from the basic colour classes, moreover restrictions can be defined on the possible colour instances of a transition (i.e., on the possible values assigned to parameters) by means of a *transition predicate*, or *guard*. Therefore, the definition of a transition colour domain comprises two parts: a list of typed parameters, and the *guard*, defined as a Boolean expression of (a restricted set of) basic predicates on the parameters. Each parameter is associated with a *variable* appearing in the arc functions of the input, output and inhibitor arcs of the transition[2].

We shall denote $var_i(t)$ the subset of transition t parameters of type C_i, and $var(t)$ the whole set of transition t parameters.

Definition 1 (Standard predicates).
A standard predicate (or guard) associated with a transition t is a Boolean expression of *basic predicates*. The allowed basic predicates are: $x = y$, $x =!y$, $d(x) = C_{i,j}$, $d(x) = d(y)$, where $x, y \in var_i(t)$ are parameters of t of the

[2] Observe that the *scope* of a variable appearing on a given arc is the corresponding transition: instances of the same variable appearing in arcs of the same transition actually represent the same parameter, while different instances of the same variable associated with different transitions are independent.

same type, $!y$ denotes the successor of y (assuming that the type of y is an ordered class), and $d(x)$ denotes the static subclass x belongs to.

Arc functions are defined as weighted (and possibly guarded) sums of tuples, the elements composing the tuples are in turn weighted sums of *basic functions*, defined on basic colour classes and returning multisets of colours in the same class. Given this definition, it is more appropriate to refer to the arc inscriptions as *arc expressions* instead of arc functions.

Definition 2 (Arc expressions).
An arc expression associated with an arc connecting place p and transition t has the following form:

$$\sum_k \delta_k.[pred_k]F_k$$

where δ_k is a positive integer, F_k is a function and $[pred_k]$ is a standard predicate. The value of function "$[pred]f$" is given by:

$$[pred]f(c) \;=\; \text{If } pred(c) \text{ then } f(c) \text{ else } 0.$$

Each $F_k : cd(t) \to Bag(cd(p))$ is a function of the form

$$F = \underset{C_i \in \mathcal{C}}{\times} \underset{j=1,\dots,e_i}{\times} f_i^j = \langle f_1^1, \dots, f_1^{e_1}, \dots, f_n^1, \dots, f_n^{e_n} \rangle$$

with e_i representing the number of occurrences of class C_i in colour domain of place p, i.e.,
$cd(p) = \times_{C_i \in \mathcal{C}} \times_{j=1,\dots,e_i} C_i = \times_{C_i \in \mathcal{C}} C_i^{e_i}$.

Each function f_i^j in turn is defined as:

$$f_i = \sum_{q=1}^{n_i} \alpha_{i,q}.S_{C_{i,q}} + \sum_{x \in var_i(t)} (\beta_x.x + \gamma_x.!x)$$

where $S_{C_{i,q}}$, x, and $!x$ are basic functions (defined hereafter), and $\alpha_{i,q}$, β_x, and γ_x are natural numbers.

The multiset returned by a tuple of basic functions is obtained by Cartesian product composition of the multisets returned by the tuple elements. As can be observed in the formal definition of arc expressions, there are three types of basic functions: the *projection* function, the *successor* function, and the *diffusion/synchronization* function. The syntax used for the projection function is x, where x is one of the transition variables (i.e., one of the transition parameters: it is called projection because it selects one element from the tuple of parameter values defining the transition colour instance), the syntax used for the successor function is $!x$ where x is again one of the transition variables, it applies only to ordered classes and returns the successor of the colour assigned to x in the transition colour instance. Finally, the syntax for the diffusion/synchronization function is S_{C_i} (or $S_{C_{i,j}}$): it is a constant function that returns the whole set of colours of class C_i (of static subclass $C_{i,j} \subset C_i$). The class can be omitted when it may be derived from the place domains. This function is called synchronization when used on a transition input arc because it implements a synchronization among a set of coloured tokens contained in a place, while it is called diffusion when used on a transition output arc because it puts several tokens of different colour into a place.

Definition 3 (Stochastic well-formed nets).
A *stochastic well-formed net* is a nine-tuple:

$$\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{Post}, \mathbf{Inh}, \mathbf{pri}, \mathcal{C}, cd, \mathbf{w} \rangle$$

where:

1. P and T are disjoint finite non-empty sets (the places and transitions of $\mathcal{N}$),
2. $\mathcal{C} = \{C_1, \dots, C_n\}$ is the finite set of finite *basic* colour classes, (we use the convention that classes with index up to h are not ordered, while classes with higher index are ordered),
3. cd is a function defining the colour domain of each place and transition; for places it is expressed as the Cartesian product of classes of $\mathcal{C}$ (repetitions of the same class are allowed); for transitions it is expressed as a pair $\langle$ variable types, guard $\rangle$ defining the possible values that can be assigned to transition variables in a transition instance; guards must be expressed in the form of *standard predicates*,
4. $\mathbf{Pre}[p,t], \mathbf{Post}[p,t] : cd(t) \to \mathrm{Bag}(cd(p))$ are the *pre-* and *post- incidence matrices*, expressed in the form of *arc expressions*,
5. $\mathbf{Inh}[p,t] : cd(t) \to \mathrm{Bag}(cd(p))$ is the matrix defining the inhibitor arcs and associated arc expressions,
6. $\mathbf{pri} : T \to \mathbb{N}$ is the priority function,
7. $\mathbf{w}$ is a T indexed vector of functions that assigns rates and weights to transitions:

$$\mathbf{w}[t] : cd(t) \times (\underset{p \in P}{\times} Bag(cd(p))) \to \mathbb{R}^+$$