Memory Fault Tolerance Software Mechanisms: Design and Configuration
Support through SWN Models

P. Ballarini  
Dip. di Informatica  
Università di Torino, Italy  
ballarini@di.unito.it

L. Capra  
Dip. di Scienze dell’Informazione  
Università di Milano, Italy  
capra@dsi.unimi.it

G. Franceschinis, M. De Pierro  
Dip. di Informatica  
Univ. Piemonte Orientale, Italy  
{giuliana,depierro}@mfn.unipmn.it

Abstract

This paper presents a case study of a software fault tolerance mechanisms, the distributed memory, designed and implemented within the European projects TIRAN and DEPAUDE, and currently under study within the Italian project ISIDE. The studied mechanisms are part of a complete framework of general purpose software fault tolerance mechanisms. In this paper we show a method for the compositional construction of models of the DM and of the environment in which it operates, expressed in the Stochastic Well Formed Nets (SWN) formalism. Different versions of submodels, at different detail level are presented and compared using some behaviour inheritance notions taken from the literature.

1. Introduction

The design of safety/mission critical systems is an hard task since it must take into account several different requirements: timing requirements (often hard real time requirements) and dependability requirements. Several projects worldwide have tackled this problem and proposed different methodologies to support the designer in this complex task. The TIRAN (Tallorable fault toleRANce frameworks for embedded applications) and DepAuDE (Dependability of Automation Systems in Dynamic Environments) European projects [5, 1], involving both academic and industrial partners, have proposed a software solution to the fault tolerance (FT) requirements of embedded systems: the so-called TIRAN framework. It is a middleware designed to be easily ported on any operating system (with basic real time functionalities), comprising a library of mechanisms (e.g. watchdog, voter, stable memory, etc.) to be used for error detection, fault containment, fault masking and recovery and a control layer, called backbone (BB), responsible for the coordination of the tasks implementing the mechanisms. The ”Ariel” language can be used to specify how the backbone must react to detected errors.

Some workpackages of both TIRAN and DEPAUDE projects, were devoted to modeling activities, with the aim of supporting the specification and design stages of the mechanisms and of the backbone. To model and analyze the different mechanisms, an high-level Stochastic Petri Net formalism, namely Stochastic Well-Formed Nets (SWN) [6], has been used. It allows to perform both qualitative and quantitative (performability) analysis of the framework components, either in isolation or combined to serve a given application. Besides being useful in the design phase, the models were also intended to be reused in a second stage, when a final user had to configure the mechanisms he/she needed for providing FT to a given application: in this situation the models of the mechanisms could be retrieved from a library of models, composed with a model of the application and analyzed/simulated to evaluate the adequacy of the chosen configuration. The first goal was partially achieved, while the second goal was not achieved in the TIRAN project time frame (but is still being pursued).

This paper describes the steps followed in building and analyzing one of the mechanism of the TIRAN framework, namely the distributed memory (DM). In section 2 we introduce only those aspects of the framework needed to understand the mechanism, however a detailed description of the TIRAN framework is out of scope of this paper; the interested reader can refer to [5, 1] for the details.

The rest of the paper is organized as follows: Sec. 3 introduces the assumptions made on the possible faults that have to be considered in the analysis; Sec. 4 and 5 introduces the incremental modeling approach followed to build a family of models for the DM (at different detail levels), Sec. 6 describes abstract models of the environment components; Sec. 6.3 defines the method used to obtain complete and analyzable models by composing the models described in the previous section. Sec. 7 reports some of qualitative analysis results obtained using the composed model and some example on possible performability indices that
could be computed through the models presented in the paper. Finally in Sec. 8 we draw some conclusions on the lessons learned while developing this case study and give some hints on possible future evolutions of the work.

2. Software fault tolerant memory mechanisms

In TIRAN the achievement of memory fault tolerance involves the interaction between three software mechanisms: the distributed memory (DM), the stable memory (SM) and the distributed synchronization (DS). Each of them was conceived as an independent mechanism providing services to its clients. The SM client is any application using the TIRAN middleware. In turn the SM uses the DM services to protect both its internal variables and the application variables. The SM extends the DM service towards the application with additional dependability features, as explained later on. The DS is located at the lower level in the software structure and its purpose is to provide a distributed synchronization service to both the SM and the DM (as well as to the application, if it needed it). The DS mechanism is extensively treated in [3].

In the rest of the section we describe each mechanism in some detail, and the runtime environment on which the TIRAN framework is intended to execute.

2.1. The TIRAN architecture

Each mechanism in the TIRAN framework is implemented as several replicated tasks cooperating to provide a given service. This avoids a single point of failure typical of any centralized schema. Tasks may be distributed over the nodes of a networked system such as an Ethernet-LAN or may execute on a same host and communicate by message passing (using the two primitives, send and receive). Both primitives have multicast features. The set of tasks implementing a given mechanism identifies a multicast address for the communication primitives, so for example all the DM tasks form a logical group. Each message is sent by a task to a group.

In the sequel we will denote the services offered by the mechanism by combining the mechanism alias (e.g. DM) with the service mnemonic name (e.g. rd), so that for example we will write $DM_{rd}(v)$ meaning the read service of the DM mechanism. Even if the notation resembles a procedure call, actually it hides a send (to group) - receive pair.

1 Actually the final design of the SM and the DM in TIRAN make them tightly related, and designed to work together, although in principle their services are orthogonal.

2.2. The DM mechanism

The DM mechanism aims to protect the physical memory against faults corrupting its contents (e.g. bit flipping) and from faults which might occur during the memory access such as faults which corrupt the data the task is supposed to write or it has just read.

Such a protection is achieved through replication. Any variable $v$ of a DM client is actually replicated by each DM task $i$ in multiple copies $dmv_i, \ldots dmv_{i_n}$. Summarizing, the DM mechanism comprises two dimension of replication: the spatial redundancy and the tasks redundancy. The amount of redundancy is one of the relevant parameters to be set in order to achieve the desired protection; the choice must be done trying to find a good trade-off between adequate reliability and reasonable performance.

Each variable $v$ stored by mean of the DM service is associated with one among the tasks of the DM group which plays a special role throughout any access to that variable, that is the master DM task for $v$.

When the DM client issues a $DM_{rd}(v)$ request, the corresponding service is implemented through two steps of majority voting. In the first step each DM task $i$ performs a majority voting on the $\{dmv_i \}$ replicas it manages. The second step of voting is carried out by the master task for $v$: it gathers the result of the first step from all other DM tasks, performs a majority voting on these results (plus its own local voting result), and sends back a reply to the DM client that issued the read request.

Concerning the $DM_{wr}(v)$ service, the master for $v$ behaves as all other tasks, but it has the additional responsibility of sending back an acknowledge (write completion) message to the client who issued the request as soon as all the DM tasks have synchronized on the completion of the writing of their local replicas. To correctly manage the concurrent access to variables, and in particular to avoid interference between concurrent write/read requests on the same variable, when a write operation starts, all the local replicas are locked, and any attempt to read them fails until the write is completed.

2.3. The SM mechanism

In the TIRAN framework the SM plays the role of a DM client. The SM mechanism adds a further dimension of replication on top of the DM mechanism, namely the temporal redundancy. It is intended to filter the data written by a SM client through its write service so that only a variable value repeatedly written for $k$ times becomes actually visible through a SM read operation. The idea with SM is to provide protection either against errors in the execution of some piece of software that is working on the same stable input data, or against instability of input to a supposedly
error free piece of software (or both).

When the $SM_{w}r(v)$ service is repeatedly invoked on $v$, the value is stored in consecutive elements of a circular array of size $k$, namely the future bank; when all the elements of this array contain the same value then it has stabilized and can become visible through the $SM_{w}rd(v)$ service. The $SM_{w}rd(v)$ service always returns value for $v$ from the current stable version, which is stored in an array of size $k$ called the current bank. When a stabilization occurs in response to a $SM_{w}rd(v)$ request the future and current bank are swapped.

The SM uses the DM to store both arrays as well as the pointers used to manage them, these variables are the current time and current bank indicators pointing respectively to the next position to be written in the future bank array, and to the current bank array.

2.4. The user application

One of the target applications considered in the TIRAN project for demonstration purposes, is a plant control system. According its own current state and the measures coming from the plant the application evolves in a new state producing some output towards the plant. For these reasons we say that it has a cyclic pattern. The application makes use of the SM service to store its own state as well as the corresponding output signals at the end of each cycle. Moreover it reads from the SM both the current state and the output signals to be sent to the actuators in the plant at the start of each cycle.

3. Fault models

So far we described the behavior of the DM mechanism, particularly pointing out how a certain level of protection against memory faults is achieved by combining task/variable replication with a reading approach based on a double level of voting. In this section we provide an analysis which aims to relate the possible situation of faults (number and distribution of faults among the task/variable replicas) with the type of analysis which is performable on the DM models.

The models of the DM we realized are capable to cope with memory errors. In this work we only consider faults affecting the value of some local replica of some DM variable, other kinds of faults like faults affecting the messages circulating within the system or affecting other variables used to implement the DM service are out of the scope of this paper. However it is important to find out all the relevant situation we need to consider in our model of faults and, for each of them, what kind of measures we can expect to obtain by running our DM model coupled with one of them.

We need to distinguish among four relevant fault situations:

1. no faults: in absence of faults we can study the impact of a given DM configuration (number of DM tasks and number of replicas within each DM tasks) on the average time to complete a DM read/write operation. Furthermore, as we will show in the next section, by means of combinatorial arguments (without resorting to a SWN model), it is possible to compute the failure probability of a given configuration. Hence, after having selected a configuration satisfying the reliability requirements, its cost in terms of execution time (conditioned on the absence of faults) can be evaluated by means of SWN models;

2. faults preserving unanimity at second level of voting: this case is given by any configuration which introduces a number/distribution of faults such that none of the local majority of correct data is destroyed (i.e. that will always result in a unanimity at the second level of voting);

3. faults preserving a majority at the second level voting: this case is given by any configuration which introduces a number/distribution of faults such that some local majority may be destroyed while still leaving a majority of uncorrupted values at the second level voting;

4. faults non-preserving a majority at second level of voting: this is the worst possible situation. In this case any number/distribution of faults is allowed, possibly leading to a DM read failure. The aim of the analysis in that case could be that of computing the mean time to reach a read failure actually affecting the application.

As a general criteria we should point out that the greater the number of faults we aim to consider the smaller the probability that they actually take place. Hence, the possibility of verifying how a multiple faults configuration affects the performance indices we are interested in (e.g. mean time to complete a DM read/write for either a preserve unanimity or preserve a majority faults configuration, or mean time to failure for a dont-preser majority situation), should be carefully considered since the presence of very unlikely situations may cause stiffness in the underlying stochastic process, or require sophisticated simulation techniques taking into account rare events. It is more reasonable to limit the modeled faults so that only not very rare situations are actually represented, and then compute separately a worst case measure, showing the degradation in performance in the special (but rare) cases of multiple simultaneous faults.

A separate discussion deserves the analysis of DM task failures, that is the possibility of some DM task to die (or to block and not respond to any external request), but this aspect is out of the scope of this work.
3.1. Selection of a configuration using combinatorial arguments

In this section we discuss a possible method to select a DM configuration by defining a reliability level and using purely combinatorial arguments. In [2] similar arguments have been used to evaluate the efficacy of two alternative bank switching policies in the SM mechanism.

The configuration parameters of the DM mechanism are the number of DM tasks and number of variable replicas for each DM task. The reliability level of the DM can be defined as the probability of getting the correct value when executing a DM read. Of course we need to give a qualitative and quantitative characterization of the possible faults: here we consider faults corrupting the value of single replicas, and we assume that the probability of the occurrence of such fault is , equal for each replica, and that the corruption of a replica does not depend on what happens to the other replicas within the same task, moreover the different tasks are independent from the point of view of possible local failures. We are also assuming totally reliable communications. Under these hypothesis, the probability of not being able to reconstruct the correct value for a given DM variable is equal to the probability of not having a majority of DM tasks with a valid and equal value. Let us consider a single DM task maintaining l replicas of dm: the probability of not being able to produce a correct value is equal to the probability of not having a majority of correct replicas: 

Now if we consider the probability of not being able to produce a correct value when a read is required, it can be computed in a similar way, with the probability replaced by and the number of replicas l replaced by dm (we are assuming that all these DM tasks have the same number l of replicas of dm). Hence, given , we can derive the reliability level of any configuration using the formulas just given. Another aspect to consider when defining the number of DM tasks and their mapping onto the nodes of the architecture is the ability to tolerate node failure: until at least one DM task is alive on any node, the DM services are guaranteed (although the tolerance to faults leading to replicas corruption may decrease). When a set of configurations have been found offering the required reliability, they can be compared from the point of view of the performance overhead they introduce.

4. The modeling approach

In this section the models developed in support of the DM evaluation and configuration are presented. The modeling approach is based on the following guidelines:

1. The overall model is defined as a collection of submodels composed through suitable operators; actually each submodel can in turn be defined as a composition of simpler submodels, so that the overall model will be organized as a multi-level hierarchy of submodels;
2. A submodel represents a system component, or a part of it, however there might be different submodels associated with the same component, corresponding to alternative representations at different abstraction levels; it is desirable to establish a relation between different submodels of the same component; note that when building a complete model by submodel composition, there might be constraints on which submodel versions of each component can be composed;
3. The submodels can be parametric, and parameters in different submodels may be related: when building a complete model from submodels it is necessary to either instantiate each submodel parameters when they are composed, or relate the submodel parameters to higher level parameters in the composed model: when the higher level model parameters will eventually be instantiated, also the related lower level parameters will be automatically instantiated. Of course at some level the model parameters must be related to the real system parameters.

The above guidelines are motivated by the following objectives: make the (sub)models easier to understand, reuse and maintain; make it easy to build models of different scenarios keeping at the same time under control the analysis costs by always using the most appropriate abstraction level for a given analysis objective; make it easy for a non expert user to customize parametrized scenarios, and obtain from the analysis of the resulting model relevant information for decision support (e.g. on the most suitable system configuration).

Of course the possibility of following the above guidelines depends on the availability of appropriate software tools: although this aspect is out of the scope of this paper, some work has already been done in this direction [10], and a few hints on which features should be available in such a tool will be briefly discussed in the conclusions.

5. Modeling the DM read function

In this section we describe the components modeling the DM master task behavior when a read service is requested. The state chart diagram is shown in Fig. 1. For the sake of space we will not show the model of the non master task (that however is a simplified version of the master) and the model portion relative to the write service. The color structure of the models described hereafter is based on the definition of a number of color classes, definition that may change depending on the goal of the analysis and may affect substantially its complexity. Some color classes are defined in parametric form to highlight the fact that some of them
can be seen as parameters, e.g. depending on the specific system configuration considered.

In Table 1 a list of the color class definitions used in the DM models is presented, with a comment explaining the meaning of the elements in the classes, as well as their partitioning into static subclasses. In Table 2 a list of place color domains used in the models is reported, with a comment explaining the meaning of the elements belonging to the color domain. In the rest of the section we give the models for the DM master task only, providing some refinement in case faults are taken into account. The substantial difference when we assume no faults is that there is no possibility of erroneous change in the values of any variable replica. This allows to model both the first level and second level voting operations as simple delays, without need to actually check if there is a majority. Moreover it rules out the need for a correction in case a majority is present but not unanimity.

5.1. The behavior of master tasks

Figure 2 shows the DM master task SWN model. The boxes depicted in the figure mark off the subnets for the first and the second level of voting. Both subnets are models for the simpler version of voting when the “no faults” assumption is taken. Let us first introduce some places of the model that are common to all the subcomponents of the DM. Place MEM represents the contents of each DM task private memory, this place must be shared by all the subcomponents since it is either read or modified by each of them. The label DMMem next to the place name is used by the composition tool to glue together several submodels. The initial marking of this place specifies for all variable replicas in each DM task the initial value and the status, which can be locked or unlocked. Initially, all replicas should be unlocked. Place DM master records the association among the DM master task and the identifier of the DM variable it manages, its initial marking defines which DM task is master for each variable and it remains constant unless a DM reconfigura-

Table 1. Color classes definition.

<table>
<thead>
<tr>
<th>Class and static subclass partition</th>
<th>MEM_FLAG: LOCK, ULOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSG: WR, RD, WCOMP, ST_CHK</td>
<td>message types that can be received by a DM task.</td>
</tr>
<tr>
<td>RDOK, RDKO, RDNA</td>
<td>reply types that can be generated by a DM task.</td>
</tr>
<tr>
<td>RM_RDREQ</td>
<td>message sent from DM master to the DM group.</td>
</tr>
<tr>
<td>SYNCH</td>
<td>message sent from DM tasks to the DS upon ending of a write operation.</td>
</tr>
<tr>
<td>NODE: ND</td>
<td>set of networked nodes.</td>
</tr>
<tr>
<td>SP_R: SPACE_RED</td>
<td>degree of local redundancy within each DM task.</td>
</tr>
<tr>
<td>TASK: SM_TASK_k</td>
<td>( k ) subsets containing one SM task each.</td>
</tr>
<tr>
<td>DM_TASK, DS_TASK, APL_TASK</td>
<td>subsets of DS, DM and APL tasks.</td>
</tr>
<tr>
<td>VAL: APL_VAL</td>
<td>possible values of the application variables.</td>
</tr>
<tr>
<td>T0,T1,T2,B0,B1</td>
<td>possible values of SM variables (time and bank).</td>
</tr>
<tr>
<td>NULLVAL null value identifier.</td>
<td></td>
</tr>
<tr>
<td>VAR: APL_VAR</td>
<td>application variables.</td>
</tr>
<tr>
<td>DM_VAR DM variables.</td>
<td></td>
</tr>
<tr>
<td>BANK, TIME SM variables.</td>
<td></td>
</tr>
<tr>
<td>NULLVAR null variable identifier.</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Color domains definition.

<table>
<thead>
<tr>
<th>Type and Alias</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message format, MBOX</td>
<td>TASK,TASK,VAR,VAL,MSG (dest, sender, variable, value, msg type)</td>
</tr>
<tr>
<td>DM variable rep, MEM_TYPE</td>
<td>VAR,SP,R,TASK,VAL,MEM_FLAG (var_id, replica number, DM task id, value, lock/unlock)</td>
</tr>
<tr>
<td>Pending read request</td>
<td>TASK,VAR (DM task id, DM variable id)</td>
</tr>
<tr>
<td>Task-Node binding</td>
<td>TASK,NODE (task id, node id)</td>
</tr>
</tbody>
</table>
Figure 2. S.W.N modeling the master task behavior after receiving a read request.

tion takes place. Similarly, place Map represents the task to node binding. Place SM_PendRDreq records the read requests being served, namely the pending requests. It is initially empty. Places Waiting and Reading represent the two “macro states” in which a DM task may be: respectively, a DM task is either waiting for messages or working to satisfy a read request (writes are taken into account in a separate submodel not presented in this paper). The place Reading remains marked during the whole cycle, since the reception of a read request message until the waiting state is again reached.

The starting of a reading cycle is controlled by the transition RDMST. When in the DM task mailbox is a read request message and the DM task is not serving a further request on the same variable then the DM task triggers the reading phase which entails the first level voting. Referring to the subnet included in a dashed box and representing such a voting, two alternatives can be observed: the transition sequence RDLock-AnyLock fires if any replica of the requested variable is locked, while the sequence RDLock-Lvotoe fires if no locked replicas exist. In case of a non master DM task the sending of a message to the master with the computed value, modeled by mean of a transition firing, closes the reading cycle and puts the task back to the waiting state.

The behavior so far described is common to both the master and non master roles. Next we give a further description that only a master DM task must accomplish. A DM master after having completed the first level voting phase must perform the second level voting, hence its model comprises a subnet representing such phase (upper submodel in the dashed box). In this subnet it is possible to observe the unanimity case (trans. Unan) and the locked replicas case (trans. any_locked). The subnet formed by place CleanResults and transition End_Clean2 and Clean2 is an auxiliary cleanup mechanism for emptying place Results when some locked replica reply is received (trans. any_Locked removes only one token from place Results, while transition Unan removes all of them at once). The second level voting can take place only when all local voting results have been received: this is accomplished through place CheckResults, which becomes marked after the firing of transition AllComp.

5.1.1. The voting refinements. If faults must be considered then refined models must be built implementing the actual voting mechanism. Figure 3(b) shows the refined version of the first level voting submodel. Several places and transitions are added in between place Serving_RDreq and transitions Ivotoe and AnyLock. If the DM replicas are not locked, these new elements take into account the fact that a majority of equal values may exist or not. A subnet is also added, responsible of realigning the values of all replicas.
within a task if a majority exists but not unanimity (place \texttt{Trig.correct} and transitions \texttt{correct} and \texttt{Correction}).

In absence of faults the refined model is behaviorally equivalent to the simple one: in fact the Tangible Reachability Graphs of both models include exactly the same Tangible Markings and the connections between the states of the TRG are identical in both cases, except that some immediate transition sequences labeling the arcs between states may differ, due to the presence (in the refined model) of some additional step to perform the voting. The rates between corresponding states in the CTMC derived from the TRG are equal for both the models.

We found it appropriate to relate the two versions of the first level voting model by getting inspiration from the life cycle inheritance notion introduced by \cite{wang2010sequential}. It is important to underline that the definitions and theorems in \cite{wang2010sequential} are not directly applicable to our models, since the SWN formalism is a colored and timed formalism (while in \cite{wang2010sequential} P/T nets were considered), which, furthermore, does not implicitly include transition labels (although it is not a problem to add them).

Despite the differences between our component models and the object life cycle models to which the concepts in \cite{wang2010sequential} apply, we believe that the same (or at least very similar) concepts can be transposed to our case; the refined model can be derived from the simpler one by using transformation rules that resemble the PJ and PT rules of \cite{wang2010sequential}; intuitively, the first level voting action is first refined by separating the check for unanimity from the timed activity representing the time actually spent to analyze each replica, this is performed by adding an immediate (silent) transition in between transition \texttt{RDNLock} and place \texttt{Analyze}, then two alternative behaviors in the voting phase are included to account for the possibility of not having unanimity (they are two because there may still be a majority or not): if these two alternative behaviors are blocked, then the refined model implements the same protocol as the simpler one.

It is not difficult to see that in absence of faults the behaviors corresponding to non unanimity are actually blocked (can never occur), moreover the additional steps needed to decide on the presence of unanimity are all (deterministic) immediate transition firings, that can be considered as silent actions from a performance evaluation perspective, hence the resulting stochastic process in the simple and refined models are equivalent.

Of course the behavior is not at all equivalent if faults are present: even a single corrupted replica causes a deadlock in the simple model, while it triggers the alternative behaviors added in the refined model.

The refinement needed for the second level voting is depicted in Fig 2(a). Also in this case there can be either unanimity or a majority or not a majority. In the last case an error message reply must be sent back to the task that issued the read request, moreover the problem should be notified to the backbone. The backbone should also be warned when a majority exists, but not unanimity, so that it may eventually trigger a recovery operation on the tasks containing wrong replicas, these aspects however are not explicitly included in the models we are presenting.

5.1.2. One more abstraction step. Our main objective in building several submodels of the same component at different abstraction levels is to reduce the analysis cost when this is possible. If we choose as a metric for the analysis cost the time and space needed to generate the reachability graph, the refined and simple models just presented do not show very different costs (when both are applicable), in fact the more abstract model has only a few less vanishing markings (some marginal time saving may also arise from having less transitions to check for enabling when developing the RG portions related to the voting phase). Under the hypothesis of no fault and read only workload however, we can do better: in this case the (master + non master) read submodel alone represents completely the relevant DM behavior (the transitions corresponding to the start of a write service, described in next section, are blocked) moreover another abstraction step can
be performed that leads us to more significant savings in the state space size. If only reads can be required to the DM then no replica can ever be locked, hence the paths in the voting subnets representing the presence of a locked replica are blocked: by removing them we realize a first abstraction (related to the previous version via protocol inheritance). Finally a more complicate but effective abstraction step can be performed on the master task submodel: some immediate transitions (namely MST_Check_Comp followed by NotComp&writing, or MST_Comp followed by NotComp&reading) are used to move messages received from non master tasks to place Results for later examination in the second level voting phase, when the master has received all replies and has itself concluded the local voting phase. In case of unanimity (and never locked replicas), the examination simply consists of removing all these tokens at once from Results by transition Unan (hence the cleanup transition Clean2 can never fire, and as a consequence End_Clean2 becomes a redundant immediate step). In conclusion the master model can be simplified by eliminating transitions NotComp&writing, NotComp&reading and AllComp, and modifying the enabling condition of MST_Check_Comp so that all (unanimous) messages of non DM tasks are moved at once from the mailbox place DMinput to Results. Finally immediate transition Unan can be merged with the following timed transition Ivote, leading to the very simple model of Fig. 4.

It can be shown that the TRG of the first simple model presented earlier in this section (under no fault and read only load assumptions) can be transformed into the TRG of the simple model by aggregating equivalent states. (moreover the CTMC of this last model is a lumped version of the CTMC derived from the initial simple model).

Even if it is not so evident, it is still possible to say that the model in Fig. 2 inherits the behavior of the model in Fig. 4: in fact the behavior of the very simple model can be retrieved by blocking the firing sequences MST_Check_Comp followed by NotComp&writing, and MST_Comp followed by NotComp&reading, and considering the sequence of MST_Check_Comp firings (one for each non master task reply message in the mailbox) followed by transition AllComp as a refinement replacing newMST_Check_Comp in the very simple model.

5.1.3. Why bother about behavior inheritance? In the presentation of the DM read submodels at different abstraction levels we have tried to relate them by using a behavior inheritance notion inspired by works [8, 9]: at the moment the ideas presented have not been formalized for the general case, but they worked fine on the non toy example of the DM mechanism, and allowed us to orderly classify the different model versions in a hierarchy and had the very useful side effect to ease the (very frequent) task of updating some tiny portion of model common to all levels in the hierarchy.

6. Modeling the environment

6.1. The SM mechanism

To try different operational conditions of the DM a number of very abstract environments have been developed to be composed with the DM model, very roughly describing the behavior of the application and SM: these models can be interpreted as synthetic workloads used to exercise the DM model. Three environment models have been produced: one performing only read requests, one performing only pure write requests (without first reading the time and bank variables), and finally one performing first the time and bank read and then a write on the appropriate future bank array element. In Figure 5 the model for the read workload is shown. Several SM tasks repeatedly issue read requests to the DM. Actually each SM read results in two read requests for the DM, the first asking for the current bank indicator, the second for the actual variable within the current bank. This means that initially there are several concurrent reads of the same variable (hence causing a temporary overloading of the DM master for the current bank indicator variable), followed by reads of different variables (load distributed among the DM tasks).

6.2. The communication services

In our models the communication layer has been modeled in a very abstract way as shown in Fig. 6. The treatment of any send request is simply modeled by a delay whose duration might depend on the location of the sender and receiver. The two places OutToSM and OutToDM represent the input interfaces of the models while the two places DMinput and SMinput are the output interfaces of the mod-
els. The message queue associated to each mailbox has been represented as a random queue, i.e., the messages are extracted in random order. Since most communications within the framework consist of a broadcast (towards all the tasks in a group) several “send to group” operations were initially modeled by a single timed transition. This simplification has the desirable effect of reducing the model state space but also that of unrealistically making all messages destined to a group to appear at the same time in all mailboxes of all tasks in the group. This approximation had the effect of hiding a problem that will be discussed later. Once the problem has been pointed out, “atomic” broadcast communications were reintroduced for state space reduction reasons in those models where it was appropriate. The boxed subnet of figure 6 represents an extended communication model including an additional input place to be used for atomic broadcast operations. The transition BrCaToDMgroup implements the broadcast to the DM group by using function msgsToDMgroup = <s,S,DM,TASK,i,v,m>, which represents a set of messages with identical sender s and identical content (i,v,m), each destined to a different task in the DM group; its firing time is defined to reflect the fact that it represents n single send messages from a common source to several destinations, some on the same node, some on different nodes.

6.3. Gluing models

All the models are parametric in the number of tasks in the group, the mapping of these tasks on the architecture nodes, and the number of variables (and variable replicas) handled by each task. These parameters correspond to the definition of some color classes (and their static subclasses), of some initial markings, and possibly on some arc function/transition guard. For example in our models color classes TASK, VAR, NODE, SP_R, and the initial markings of places VarMap, Map, MEM, Waiting, etc. must be properly instantiated to reflect a specific configuration. Moreover each model has timing and probability parameters, associated with timed and immediate transitions: the ideal situation is to have the rate and weights of transitions defined as functions of the system parameters (e.g. number of replicas) and of some basic timing parameter (e.g. average time needed for a local/remote communication to complete).

The complete model for a given configuration (and a given workload) is composed by consistently instantiating the parameters of the appropriate submodels, and then by actually composing the instantiated submodels: in our case the composition of the different submodels is performed using place superposition: since the mechanisms interact through message passing using mailboxes, it seemed rather natural to define a set of interface places representing the mailboxes, and perform the composition by place superposition. Assuming that a given submodel represents mechanism X and that this mechanism can send messages to mechanisms Y_i, i = 1, . . . , n, then one input interface place Xin mbox (representing all the mailboxes of the tasks implementing the mechanisms), and n output interface places outToY_i mbox, are included in the submodel (note that if tasks in mechanism X exchange messages among them, there will be also an outToX mbox place).

Place superposition composition has been used also within the submodels representing the different functions of the DM mechanism (common places were highlighted in Sec. 5), so that a complete behavior can be built by gluing

Figure 5. The SM mechanism abstract model

Figure 6. Theme variations of the communication model.
together all function submodels (at the proper abstraction level) that are needed for a given analysis objective.

Since a large number of variables have to be kept under control during the instantiation and composition phase, it is important not only to build a library of well documented submodels, but also automatic tools that helps throughout the construction of the desired scenarios. For such a purpose we have developed an interactive script which prompts the user for the relevant configuration parameters, generates the appropriate net files and eventually composes the instantiated submodels by means of the tool algebra [4].

7. State space analysis and qualitative analysis results

The modeling activity that led to the construction of the models presented in this paper (plus others not presented for the sake of space) had several goals: the first was to enrich the documentation of the mechanisms in the library; the second was to have executable specifications, more detailed than the UML diagrams used by the designers, that could be used to understand the qualitative behaviour of the mechanisms when integrated in different application environments; the third was to provide models for performance and reliability assessment of different configurations.

The first and second goals were achieved by building the refined versions of the model. The modularity of the models not only made them more manageable, but also increased the efficacy from the point of view of the mechanisms documentation. The third goal was achieved by blended use of combinatorial and SWN models. The performance figures were derived by mainly using the simpler models, which allowed to analyze more complex configurations than the refined ones. It was anyway useful to have both the simple and refined models, because they made it possible to evaluate (at least on the smaller configurations) the impact of the abstractions introducing approximations. Having clearly identified the relation between the models at different abstraction levels, it was more clear where and under which assumptions approximations had been introduced. Let us briefly present some concrete results obtained in the model analysis phase.

Analysing the behaviour of the DM models under different application environments allowed us to discover a problem in the DM write function: the current specification does not correctly support concurrent write requests on the same variable. In fact when two clients issue a request to the DM task group, the two messages may arrive in different order to the DM tasks: if this happens all tasks lock their local copies, and then update them to different values; finally they synchronize on the condition “write of variable dmv completed” through the DS. When they receive the second message, the whole protocol repeats, again updating the copies in the different DM tasks inconsistently. It is interesting to observe that the problem was discovered only when using the communication submodel version that moves the messages one by one, because the submodel representing the broadcast as an atomic action always delivered the messages at the same time to all DM tasks.

The performance analysis of the different configurations consisted in generating the reachability graph (up to the current limitations of the GreatSPN tool) and then generating and solving the corresponding CTMC for computing the desired performability indices. When the RG construction failed, we resorted to simulation. Examples of performance figures that can be derived are the average time to complete a read/write under different workload and configuration conditions, with or without faults.

The limiting factor in our models was mainly space for storing the RS due to the very large colored states, combined with the large number of states for relatively large configurations. An ad-hoc optimization of the state space generator was implemented to take into account the presence of places with constant markings (e.g. places expressing mapping of tasks onto nodes, or keeping the identity of master tasks), which allowed us to analyze more complex configurations. In Table 3 some data is reported to give a flavour of the state space growth as a function of some configuration parameters: the table shows both the number of states (tangible and vanishing), and a measure of the space needed to encode the whole reachability set (size of the file containing all markings in the RS). The table allows to appreciate the difference between the simple versus complex models, showing the usefulness of the different representations. The analysis has been performed on a PC with a Pentium III, 866 MHz and 756Mb RAM.

8. Conclusions

This paper presented a study of the distributed memory, a fault tolerance software mechanism designed and implemented within the European projects TIRAN and DEPAUDE. The studied mechanisms are part of a complete framework of general purpose software fault tolerance mechanisms, partly analyzed in previous works. The advantage of using general purpose software fault tolerance mechanisms rather than ad-hoc possibly hardware ones, is their high configurability, their portability to different platforms, and their reusability within different (mission) critical applications, the disadvantage is the time overhead introduced by the mechanisms. In this paper we presented the lessons learned while modeling the above mechanism, which led to a modeling methodology that produced interesting results from several points of view. The modeling activity had several goals: the first was to enrich the documentation of the mechanisms in the library, the second was to
have executable specifications, more detailed than the UML diagrams used by the designers, that could be used to understand the qualitative behavior of the mechanisms when integrated in different application environments, the third was to provide models for performance and reliability assessment of different configurations. The goals were achieved by gradually converging towards a modeling methodology based on the modularization of models, and on the development of several submodels of a given function/component at different abstraction levels. It is important to underline that the submodels representing the same component/function were related by adapting a behavior inheritance notion proposed in [9] (we plan to further investigate if the concepts in [9] can be extended to our formalism in a more formal setting). This allowed us to more clearly point out the assumptions made when moving between different abstraction levels, and state under which conditions the different levels where equivalent (e.g. led to same performance results). By consistently composing submodels at the right abstraction level we could find a good balance between precision in the results, and ability to treat configurations of adequate size. To grow further in the configurations that can be treated by our tools however, we think that the proposed approach must be combined with other techniques, for examples the systematic exploitation of behavioral (partial) symmetries [6] typical of the SWM formalism, or exploiting model (de)composition at the analysis level [7]. Last but not least, proper tool support for handling model libraries and submodels composition is needed: we plan to work also on this line, building on the work presented in [10].

### Acknowledgments

We would like to thank the anonymous reviewers for their suggestions. The work of G. Franceschinis and M. De Pierro has been partially supported by the Italian MIUR under the FIRB project Perf (n. RBNE019N8N).

### References


Table 3. Tangible and vanishing markings for the simple and refined models with 2 SM tasks, a variable number of DM tasks and a read-only workload.

<table>
<thead>
<tr>
<th>DM tasks</th>
<th>Execution Time</th>
<th>RS file size (bytes)</th>
<th>Number of tangible and vanishing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Simple model (with broadcast)</td>
</tr>
<tr>
<td>2</td>
<td>0:39</td>
<td>570.053</td>
<td>2056, 5606</td>
</tr>
<tr>
<td>3</td>
<td>2:31</td>
<td>5.027.022</td>
<td>10204, 40198</td>
</tr>
<tr>
<td>4</td>
<td>15:45</td>
<td>53.207.379</td>
<td>68152, 359978</td>
</tr>
<tr>
<td>5</td>
<td>1:58:09</td>
<td>609.227.364</td>
<td>525772, 3586134</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Refined model (with broadcast)</td>
</tr>
<tr>
<td>2</td>
<td>21:31</td>
<td>1389105</td>
<td>2056, 12470</td>
</tr>
<tr>
<td>3</td>
<td>2:57:56</td>
<td>13664410</td>
<td>11188, 100286</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Simple model (without broadcast)</td>
</tr>
<tr>
<td>1</td>
<td>0:01</td>
<td>12280</td>
<td>93, 159</td>
</tr>
<tr>
<td>2</td>
<td>0:15</td>
<td>1688243</td>
<td>6120, 15991</td>
</tr>
<tr>
<td>3</td>
<td>56:49</td>
<td>335525660</td>
<td>639438, 2550834</td>
</tr>
</tbody>
</table>

| 3. Tangible and vanishing markings for the simple and refined models with 2 SM tasks, a variable number of DM tasks and a read-only workload. |