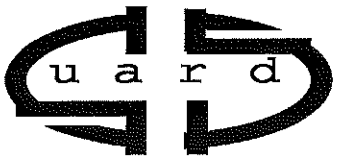

PDCC -Pisa Dependable Computing Centre
Consorzio Pisa Ricerche

Inter-channel State Restoration

A. Bondavalli*, S. Chiaradonna**, F. Di Giandomenico*** and F. Grandoni***

* CNUCE Istituto del CNR, Via S. Maria 36, 56126 Pisa, Italy
a.bondavalli@cnuce.cnr.it

** PDCC - Consorzio Pisa Ricerche, Piazza A. D'Ancona, 1 - 56127 PISA, Italy
S.Chiaradonna@guest.cnuce.cnr.it

*** IEI Istituto del CNR, Via S. Maria 46, 56126 Pisa, Italy
{digiandomenico, grandoni}@iei.pi.cnr.it

Project Reference: GUARDS/II-SA4/AO/6006/b

14/11/1997

Inter-channel State Restoration

Revised version of the "Error recovery at inter channel level" draft.

Main updates: More detailed algorithms regarding multiple-channel instances; More efficient handling of termination condition in Recursive SR; More accurate handling of fault detection along SR.

Very helpful comments have been provided by D. Powell and C. Rabejac.

Rev.	Drafted by	Checked by	Approved by	Date
F				
E				
D				
C				
B	A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, F. Grandoni			13/11/97
A	A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, F. Grandoni			24/09/97

Table of Contents

1. Introduction

In the GUARDS architecture, fault tolerance structures/mechanisms at the inter-channel level have to include system state restoration (SR) provisions. Recall that, in GUARDS [6], inter-channel fault-tolerance is pursued by N-modular redundancy, where critical applications are replicated over N identical channels. The SR mechanism complements the error processing, which is devoted to detect the disagreeing channel(s) and to mask errors occurring in the voted variables at run-time, and fault treatment, which diagnoses (and possibly passivates) faulty channel(s), possibly discriminating the nature (permanent/intermittent against transient) of the faults [3]. System SR mechanisms are employed to restore a correct state in a channel, whether it is brought back into operation after diagnosed being hit by a transient fault, but too heavily corrupted in the internal state, or it is an off-line channel to be re-integrated in the active system.

In GUARDS, the application software is structured as a set of periodic or sporadic 'functions'. Each executes in an *iterative* fashion and is split into a number of sequential tasks [14]: at each iteration the first task gets its input values from sensors and sends them through the ICN (which ensures consistent values across channels). Then the second receives the consolidated values through the ICN and uses them together with the values of some *state variables* (which contain values that are carried over between iterations) to compute the output values. These are again sent through the ICN and the consolidated values are used to feed the third task, which, after voting, exercises the output consolidation circuitry.

State variables are not necessarily consolidated through ICN exchanges (the ICN bandwidth could not possibly support all this traffic and too frequent or heavy consolidations could slow down the control process well below any acceptable limit); on the opposite hand, it could be appropriate in some circumstances (e.g. long computation phases) to consolidate intermediate (possibly not state carrying) variables to decrease error latency.

When a channel is put in service to run replicated applications, its internal state (*channel context*) that is the set of all the state variables of all replicated applications, is to be copied from the active channels, before application programs can be started at the same point in the code. This way, the new channel will run applications consistently with the already active replicas.

The SR is a very sensitive operation. It could happen, for example, that one of the active channels, perhaps just that the state variables are copied from, is affected by a latent fault, which has already corrupted its state but has not yet caused a detectable error. The corrupted state is transferred to the joining channel, thus setting the ground for a catastrophic failure due to a common-mode error, when the joining channel will be put into operation. It is essential, therefore, that an effective detection of errors affecting the state information is

ensured before activating the joining channel. This, in general, can be accomplished whether through consistency checks (if enough redundancy is available) along with the state variables transfer, or by some final check executed on the whole state upon the transfer completion.

2. COTS-OS and synchronisation of the applications

The occurrence of a fault, whichever be its nature (temporary or permanent), may potentially affect a large part of the OS context. In GUARDS, replication is built above COTS OS. This means that the internal state of OS objects is not accessible, let alone controllable. Therefore, application tasks must be restarted in points where the internal state of the restarted OS can be considered equivalent (with respect to the applications) to that of a running channel.

Another issue is where in the code, and how, to re-launch the re-initialised channel: in fact, in a pre-emptive environment, when the ancillary SR tasks do terminate their duty the applicative tasks are suspended, possibly anywhere in their code; their replicas have to be resurrected in the joining channel at the very same points. This would mean, in general, that all the volatile tasks' state, including their stacks, as well as any task descriptor kept by the OS (e.g. open files descriptors), should have been copied, and re-created in the joining channel context. Questions of feasibility in the GUARDS environment may well arise.

The easiest, and most reasonable way to solve the problems above would be to exploit the computation model assumed in GUARDS, by awakening the resurrecting replicas exactly on the next computational iteration beginning. In fact, a synchronous, frame-based computation model is adopted in GUARDS. At the beginning of a frame, all the tasks start with a new iteration, thus the restart time of a re-aligned channel can be chosen as the beginning of the frame next to the last one relative to the SR phase. It has to be assumed that all information carried through successive iterations is stored in the state variables internal to applicative tasks; more explicitly, no residual information is supposed to reside in OS structures at that points. Of course, it has to be ensured that at the beginning of the restart frame the SR action has correctly brought the joining channel's context into agreement with the active channels.

An important issue concerns whether the task(s) implementing the SR procedure has to run in parallel with applicative tasks or not. Obviously, if it were possible to stop the execution of the applicative tasks, the SR would be much simpler. However, it is a crucial point in the design of ultra-reliable fault tolerant system that SR be performed without degrading critical real-time functions. Here it is considered acceptable that the system switches into a special degraded mode, where a reduced number of functionalities are offered by the system, but where a minimum level of service (depending on the specific application) is ensured. Hence both the SR task and (possibly a reduced number of) applicative tasks have to run in parallel. In this scenario, which is the most likely in GUARDS instances, the reintegration of a channel gets more awkward, since during the SR phase, the application tasks may change

values of state variables already restored in the joining channel before the SR task itself has terminated.

Recall that in GUARDS the use of shared variables for inter-task communication is endorsed. This helps in copying and transferring the state information; on the other side of the coin, concurrent access of application tasks to shared state variables along the copy operation requires special care in the design of the restoration procedures.

3. Background

The problem of state restoration has been widely treated in the literature, and several solutions have been devised. However, the SR process is so highly application dependent that a general solution is not attainable, and much effort is still devoted to improving the state of the art.

Restricting to the scenario where SR is necessary to re-align a (failed or erroneous) component belonging to a multiple redundant structure (as an instance of the GUARDS architecture is), the Community Error Recovery (CER) technique is followed. This approach makes use of the assumption that at any given time during the execution there exists a majority of good replicas which can supply information to recover the failed replicas [1], [8], [9], [12], [13]. Erroneous state data can be recovered by periodically replacing the state data maintained by each replica with majority-voted values. Two classes of SR algorithms can be considered, according to whether restoration requires a single step or multiple steps.

One-shot SR. The reintegration of a failed component is accomplished in a single, uninterrupted operation. [1] considers the application of this scheme in highly reliable real-time systems, based on tightly-synchronised replicas and voting, by using a specific technique to detect which memory segments (or in general state variables) have been corrupted, and which then will be restored. This strategy depends on the assumption that the corrupted memory is small enough to be recovered at once without degrading critical real-time functions.

Multi-step SR. In general, restoring the state of a component so that it is congruent with other replicated components is a difficult task. Since the amount of the state data may be large, the SR action may be split into stages. At each stage, only a fraction of the state information is exchanged, and an intermediate vote is taken. Several stages are then needed to refresh the entire state. Provided that enough replicas remain non-faulty at all times, and provided that state data are not contaminated faster than they are repaired, this approach provides self-stabilising SR from transient faults [1], [8].

Intermediate voting needs not to be applied to the whole computation: voting may be limited to certain tasks, or to just some subset of the state variables, or simply to the output value(s) computed by a critical task. Voting large data sets, or voting more often than necessary, can

be very expensive, while restricting the set of voted data or the vote frequency gives rise to lower reliability and longer SR time span.

Multi-step SR schemes can be grouped in two classes, which will be called *Running SR* and *Recursive SR* ; examples can be seen in [2]; Sims, 1996 #21.

In the GUARDS context the one-shot SR does not seem to be applicable, as it is not generally possible to totally suspend vital applications. The Running SR and the Recursive SR schemes appear to be the most suitable in the GUARDS context, and will be described in the following Sections.

A (C, M, I) GUARDS instance, i.e. with C active channels C_1, ..., C_C, will be considered. The state restoration scenario is the following: when one of the channel needs to be re-aligned to the others the system enters a "SR mode". The active channels enter a "put state" mode while the recovering one enters a "get state" mode, where it obtains an up-to-date copy of the state variables of all its replicated applications from the C-1 active pals. Every active channel sends the values of its state variables via the ICN network.

In the frame-based timing of GUARDS, switching from normal computation to the "SR mode" occurs at the beginning of a frame, with the attending change in task scheduling. The SR gets completed on the end of a frame, which is the same for all channels, with normal application scheduling activated on the next frame. This implies that restoration activities started in a frame must be completed in the same frame, both on the active channel(s) and on the joining channel.

Only channel-level interfaces will be presented here. Intra-channel details (e.g. how to organise the inter-channel SR action in a multi-processor channel) are not considered (yet). The following description naturally maps into GUARDS instances with M=1.

4. Running SR logic

Let the channel context be arranged in a single (logical) memory block SM. This can be implemented in several ways: e.g., in Ada by representation clauses, or in C/C++ by using pointers. When SR is started, the active channel(s) enter the "put state" mode. All updates to state variables by the active applications are propagated through the inter-channel network (ICN) and are thus performed also in the joining channel's memory space. At the same time a *Sweeper* task is started on the active channel(s), which conveys the channel context to the joining channel, where it is received and processed by a *Catcher* task. Since a deterministic, finite time is required to copy the memory block, inversely proportional to the bandwidth available for the purpose on the ICN, and any intervening modification to already copied state variables is directly done also in the target memory, the whole state restoration is in turn performed in a deterministic, finite time. Note that in the SR phase the ICN has to support: i) the normal traffic generated by the vital (i.e. non-stoppable) applications, ii) the extra traffic

due to simultaneous state updates (state variables normally do not require agreement protocols to be assigned new values), and iii) the traffic generated by the Sweeper task.

In the frame/cycle/slot timing structure of GUARDS [3]; Coppola, 1997 #11 the Sweeper and the Catcher, which do not necessarily share the iterative nature of the vital application tasks, make use of asynchronous slots in the ICN schedule to exchange context data. The Sweeper runs on CPU time slots left out by vital applications (running in their active SR mode), while the Catcher has most of the CPU time available, as the application tasks on the joining channel are waiting to be restarted after their state has been restored.

4.1 Basic Behaviour

The description of the basic behaviour of the Running SR scheme refers to a simple GUARDS instance: one channel is active, running a Sweeper task and applicative tasks, and a second channel is to be re-aligned, running a Catcher task.

The presentation of the algorithms will focus on the main topics of the state restoration, leaving off much of the details. In particular, checks on communications delays, on missing messages, and in general on exceptional conditions will be omitted. The send() and receive() procedures are intended to be non-blocking. Moreover, to help the reader in understanding the pseudo-code, essential comments on the meaning of variables and functions/procedure are given; however, when the same identifier is used with the same meaning in different algorithms code, the corresponding comment is in general omitted after the first occurrence.

Algorithm of the Sweeper task:

```
/* context_morsel[]:      array  of records containing the portion of context read by the Sweeper introduced  */
/*                        to optimise the use of ICN by sending "packets" fitting into the ICN slot; generic     */
/*                        record structure:  <context variable identifier>,<context variable contents>;          */
/* end_of_context():      Boolean function: returns TRUE when context_morsel[] contains an (end of context)*/
/*                        mark                                                                                   */

{
do
    {
    context_morsel[]=get_next_context_morsel();
    send(context_morsel, Catcher);
    };
while (! end_of_context(context_morsel[]));
}
```

Extensions to the Application Tasks' code

```
/* substitute search assignment instruction directed to a context variable with the following procedure:        */

extended_assignment (context_variable, new_value);
{
context_variable=new_value;
send(context_variable, Catcher);
}
```

Algorithm of the Catcher task:

```
/* context_morsel[]:                    portion of context obtained from the Sweeper for each    */
/*                                      exchange of data  between channels                       */

{
do
    {
    for each taskj in {Active_Tasks} do
        {
        receive(context_variable, taskj );
        update_context(context_variable);
        };
    receive(context_morsel[], Sweeper);
    if(! end_of_context(context_morsel[]))
        update_context(context_morsel[]);
    };
while ( ! end_of_context(context_morsel[]));
}
```

The SR process involves at least three tasks: the active application task(s), the Sweeper and the Catcher, asynchronously running on two separate physical machines communicating through an asynchronous channel. The classic problems of task co-ordination and synchronisation thus arise and must be solved. The following approach is suggested here, as a follow-up to the assumptions made in Section 2:

i) The active applications and the Sweeper task are assigned deadlines in such a way as to ensure that they are scheduled in the same order as the ICN cycles they use to transfer context information.

ii) The Catcher task updates the state variables in the same order that they come from the ICN.

iii) The procedure extended_assignment is executed atomically.

A special case of implementing these assumptions is to reserve the last ICN slots in a frame to the Sweeper, *and* to instruct the scheduler to run the Sweeper *after* all replicated applications. Actually, the assumption (ii) is then not needed anymore, since the values transferred via the ICN are now assured to be congruent, as far as a single frame is considered. It remains to ensure that the Catcher does the variable updates required by applications running at the beginning of next frame *after* it has finished with consuming the previous frame's values; this is easy to do if a timestamping service is provided for interchannel transfers.

The algorithms reported in this Section rely on the just sketched implementation to ensure the validity of the above assumptions.

It has to be noted that, in this simple case, the reliance that can be put on the system just after it has been recovered to the normal configuration is not higher than that of the single working channel, as before the SR procedure. Accurate self-checks on the active channel to be performed before and after the SR operation are thus commendable.

4.2 *Extending the basic scheme to multiple channel instances*

Consider now a more general GUARDS instance, where two or more active channels cooperate to align a joining channel. To efficiently use the ICN bandwidth, the whole block of memory storing the channel context can be split in C-1 (almost equal) sub-blocks $SM_1,...,SM_{C-1}$, where C-1 is the number of active channels. Then, SM_i gets transferred by $Sweeper_i$, i= 1,...,C-1, thus multiplying by (C-1) the amount of context sent through the ICN at each frame. The responsibility for sending the updates performed by application tasks on state variables can also be shared by the active channels in a similar way. Two example policies may be:

i) each channel is responsible for transferring updates to state variables which belong to the sub-block of memory context it is in charge of, or

ii) the set of applications is partitioned into C-1 subsets, and each channel is in charge of transmitting the updates relative to application tasks in an assigned subset. In this case care has to be taken in handling variables shared among applications belonging to different subsets.

The same algorithms detailed for the basic configuration are still applicable in this scenario; there is only the added need of clearly partitioning the workload among the active channels. The end of each memory sub-block has to be marked, so that the function end_of_context() yields TRUE on its occurrence. In the code of $Sweeper_i$, the get_next_context_morsel() function has to start from the beginning of SM_i. The extended_assignment () procedure is to be applied only to a subset of state variable update instructions, complying with the chosen policy (see points i, ii above). Note that a dissimmetry is introduced into the applications, which has to be taken into account in the response timing analysis and in the setup of the scheduler.

In this scheme, the Catcher lacks the redundant information that in the generalised Basic Scheme allowed the adjudication of the correct state values. Therefore other means to ensure that the state has been correctly restored have to be provided for. A couple of solutions are briefly introduced below.

4.2.1 Detection of errors during SR by comparison

Each Sweeper sends the state values not only to the Catcher on the joining channel, but also to the other (C-2) active channels, so that each other active channel will be able to make comparison between the received value with the corresponding local data. In order to cope with Byzantine failures (as required in the GUARDS architecture), a reliable broadcast protocol is used to ensure a consistency in the data sent by each Sweeper to the multiple recipients. The same has to be done by the active applications, while updating their context variables. A Byzantine behaviour exhibited by any of the active channels in sending the information can be so recognised by any of the receivers channels (including the joining one),

while a (solid) value error can be detected only by the active channels, through the comparison with the corresponding own data. Whenever an inconsistency in the state information is detected a signal is raised to the error treatment subsystem, which is in charge of subsequent actions, including suspending or cancelling the ongoing state restoration. Actually, using the reliable broadcast allows some more flexible reactions. In fact, with reference to a 4-channel GUARDS configuration, consider the following scenarios: i) Byzantine behaviour is exposed: SR cannot continue, since a correct value cannot be recovered; ii) only one out the active channels, say C_i, detects a discrepancy between a value sent to the Catcher and its own correspondent value: SR could continue, since the Catcher received a value, which is agreed upon by a majority of the active channels; iii) a value sent, say, by channel C_k, clashes with the corresponding values of the remaining active channels: SR must be suspended, since the Catcher received a wrong value. In the scenarios i) and iii) it is conceivable that a "soft" corrective action could be taken, e.g. by passivating the single failing channel and then continuing the restoration, after properly reconfiguring the involved tasks (i.e. re-partitioning among the remaining sweepers the transfer load).

In the Sweeper task's code the send() operation has to be substituted by a reliable broadcast protocol, where the Sweeper acts as the "general" [4] through a R_broadcast() procedure. The Catcher participates to the protocol by calling a ByzAgreement() procedure, which yields non-redundant data from all the active channels. A new task, name it Checker, has to be activated on each active channel, to compare[1] the data exchanged by ByzAgreement() with the corresponding local value, and to raise calls to the error treatment subsystem whenever needed.

Algorithm of the Checker task:

```
/* N_Sweepers          constant indicating the number of Sweeper tasks, one on each active channel  */
/* Sweeper_stopped     variable to count how many Sweepers have been stopped                        */
/* ByzAgreement(v,S)   Byzantine agreement procedure on the value v received from the sender S      */
/*                     executing a R_broadcast(v)                                                   */
/* compare()           procedure which compares the context values received with the corresponding  */
/*                     local values. Appropriate signalling is raised upon detecting a discrepancy. */

{
Sweeper_Stopped=0;
do
    {
    for each channel_j in {Other Active Channels} do
        {
        for each task_k on channel_j do
            {
            ByzAgreement(context_variable, task_k )
            compare(context_variable, local_context_variable)
            };
        ByzAgreement(context_morsel[], Sweeper); ;
        compare(context_morsel[], local_context_morsel[])
```

---

1 Because of the time dis-alignment between channels, this operation may entail tricky aspects, as, e.g., dealing with multiple updates of a single variable occurring in the time interval between two successive Checker's readings. Careful design of the Sweeper-Chechers interaction is mandatory.

```
        if (end_of_context(context_morsel[]))
            Sweeper_stopped++;
        };
    }
while ( (Sweeper_stopped < N_Sweepers -1));
}
```

This solution, unfortunately, is likely to lose most of the advantage gained in partitioning the memory block to be transferred; on the other hand, the Byzantine agreement promptly points out any error affecting the state.

4.2.2 Detection of errors during SR by signatures

Another procedure for error detection is to have every Sweeper and the Catcher compute a signature [7], [11] of the *whole* local channel state after the state data has been transferred to the joining channel. The signatures are exchanged among all the tasks involved through an interactive consistency protocol (which is available in the GUARDS system libraries). A discrepancy detected in comparing the signatures received by the Sweepers implies that an active channel has failed, while a mismatch between the signature computed by the Catcher and those of the Sweepers points to a failure in the joining channel or to a transient failure in (one or more) active channels during the state transfer (typically, in the transfer medium), but completely recovered by the time the signature has been transmitted. Other cases of mismatch may occur; anyway, if the Catcher's signature agrees with that generated by a majority of active channels, the SR can be considered successfully terminated.

If performance allows, this state signature comparison mechanism could also be used systematically, in every frame, to improve error detection latency.

The implementation of this solution requires additional code in the Sweepers and in the Catcher to compute the signatures, to call the interactive consistency procedure, and to inspect the resulting consistency vector.

This signature-based scheme appears to be more efficient than the previous scheme since there is just a small computational and communication overhead, but the error coverage is strictly related to the adopted signature scheme.

4.3 Mechanisms required to a "GUARDS" implementation

The main difficulty of Running SR is the "simultaneous" update of *all* state variables during the SR phase. In fact, this entails burdening every state variable assignment statement with a call to an ICN routine (via the extended_assignment procedure). To avoid such overhead in the nominal operation, the application software has been supposed to be given a *modal* structure, where entering the SR mode flips a global switch, activating the proper code. The ADE should support/enforce this structure, for example allowing a special type to be defined for state variables, then checking that each update follows the stated rule. Language constructs like Ada95's controlled type allow simple implementation of the extended_assignment

procedure. Since the execution time in the SR mode may substantially differ from that of the nominal mode, both response time analysis and schedulability have to be carried out for both modes. Those analyses have to take into account the heavier traffic in the ICN, which slows further down the operation; it may also happen that the actual ICN throughput is not sufficient to support this mode of operation with respect to the assigned deadlines. In this case, an option could be to design trimmed-down variants of the applications, to be activated in the SR mode; a completely new schedule is also to be provided.

Running SR requires use of ICN bandwidth. Sufficient asynchronous bandwidth is necessary for transferring copied state variables in a reasonable time (possibly a single byte per frame could suffice!). However, extra synchronous bandwidth is necessary for the concurrent propagation of state variable updates.

The Sweeper task can be just another fault-tolerance management-level task. It needs to be given *read* access rights to the state variables of all replicated applications. A corresponding Catcher task runs on the joining channel, receiving the values of the state variables and storing them in their proper places in the application memory space. This means that the Catcher task must be endowed with *write* access rights to the state variables of all applicative replicas in its channel.

5. Recursive SR logic

Here the channel context is supposed to be represented in a more structured way: it can be considered arranged in (or mapped into) an array, where each element holds an elementary state data (say, a simple variable, or a memory word) and a binary tag. A mechanism can be activated in the SR phase, which upon any Write operation on a state data item sets the accompanying tag to TRUE [10]. The applications are not charged of immediately sending the updated variables to the joining channel, as in Running SR. A special tRead operation is assumed to be available, which gets the data item and resets the tag to FALSE.

Similarly to the Running SR mode, the Recursive SR mode leads to the activation of two types of tasks: the Sweeper on the active channel(s) and the Catcher on the recovering channel. Upon activation, the Sweeper starts reading, using tRead, the context data array, and sends the data whose tag is TRUE to the joining channel through the ICN. In the process, the swept tags are cleared to FALSE. When the Sweeper reaches the end of the array, it starts again from the beginning, looking for any tags that could have been set back to TRUE by some application tasks. If this is the case, the corresponding (modified) state variable(s) is (are) re-sent, and so on. A termination condition for the process remains to be defined, because, unlike in the Running SR, it is not possible to determine statically how many times the context array is to be swept. The definition of such condition, however, does not imply that the condition will actually always occur: termination is usually non-deterministic, depending on the detailed timing properties of applications. Of course, it must be ensured that

the occurrence of the termination condition be followed by no further updates on state variables by applicative tasks until the actual end of the SR process, which is supposed to coincide with the end of the frame during which the condition gets satisfied. This would be easily obtained by suspending applications until the end of the frame, which is however unacceptable in GUARDS, where vital activities have to be kept running anyway; hence, a careful scheduling of the applicative tasks and of the Sweeper is necessary, as will be discussed in section 5.1.3.

A preliminary suggestion for a termination condition is the "last shot" technique: when the number of updated variables to be transferred is small "enough" (depending on the maximum time applications are allowed to be suspended), a non-interruptible transfer operation may be arranged. In the following algorithm, this termination condition is adopted.

The impact of the SR process adopting this scheme is less heavy than the Running SR from the applicative tasks point of view (their operation, as far as SR is concerned, remains strictly confined inside the channel); in fact, the overhead necessary to manipulate the variable tags (by a modified Write instruction) is probably small.

5.1 Basic behaviour

The description of the basic behaviour of the Recursive SR scheme refers to a simple GUARDS instance: one channel is active, running a Sweeper task and applicative tasks, and a second channel is to be re-aligned, running a Catcher task.

Algorithm of the Sweeper task:

```
/* context_morsel[]          array  of records containing the portion of context read by the Sweeper  */
/*                            introduced to optimise the use of ICN by sending "packets" fitting into  */
/*                           the ICN slot; generic record structure:                                  */
/*                           <context variable identifier>,<context variable contents>;               */
/*                                                                                                    */
/* check_tag()               Boolean function to check the status of the tag associated to indexed    */
/*                           context variables (TRUE if the tag is on, FALSE otherwise)               */
/*  end_of_transfer          <null context variable identifier><null contents>                        */
/* send_block_dim            dimension of the context_morsel array; related to the ICN slot length   */
/* context_index:            addresses the current item of the context                                */
/*  starting_index           has the purpose to mark the beginning of the scan of context_array ,     */
/*                           to know when a full scan has been completed.                             */
/* termination_condition     Boolean function yielding TRUE when the "last shot" transfer can be      */
/*                           arranged, while blocking the active applications                         */
/* context_array[]           array holding the whole channel context, circularly indexed              */
/*                                                                                                    */
/*  variables & constants declaration & initialisation                                                */
/*                                                                                                    */
{
while (! termination_condition())
  {
     i=0;
     do  {
          if (check_tag(context_array[context_index]))
               {
               context_morsel[i]=tRead(context_array[context_index]);
               i++;
               };
          context_index++;
          };
```

```
    while (i< send_block_dim);
    send (context_morsel[], Catcher);
  };
/*                                                                              */
/* the following actions have to be performed atomically                        */
/*                                                                              */
/*  for the sake of simplicity  it is supposed that the variables remaining to be transferred fit into one   */
/*  single context_morsel[] buffer; the last entry of the buffer is reserved to the end_of_transfer mark     */
/*                                                                              */

starting_index=context_index;
i=0;
do  {
    if (check_tag((context_array[context_index]))
        {
        context_morsel[i]=tRead((context_array[context_index]);
        i++;
        }
    context_index++;
    };
while ((i< send_block_dim  - 1) && (context_index!=starting_index));
context_morsel[i]= end_of_transfer;
send(context_morsel[], Catcher);
/*                                                                              */
/* the application execution is resumed directly by the scheduler in the next frame */
/*                                                                              */
}
```

Algorithm of the Catcher task:

```
/*                                                                              */
/* update_context()        procedure which updates the local state variable identified by the first record */
/*                         field with the value provided by the second field    */
/*                                                                              */
/* receive_block_dim:      dimension of the context_morsel array; same as send_block_dim */
/*                                                                              */

{
do
    {
    receive(context_morsel[], Sweeper);
    i=0;
    while ((i<receive_block_dim) && (context_morsel[i]!=end_of_transfer))
        {
        update_context(context_morsel[i]);
        i++;
        };
    }
while ( (context_morsel[i]!=end_of_transfer));
}
```

5.1.1 Mechanisation of the termination condition

An example mechanisation of the termination condition is the following: let the array SM be associated to a counter variable, which indicates the current number of elements in the array (that is, context variables) whose associated tag is TRUE. At the beginning, when the SR phase starts, this counter variable is set to the total number of state variables. Then, the counter is incremented by only those Write operations which flip the tag from FALSE to TRUE, while it is decremented by every tRead operation. A call to termination_condition() checks the counter, which represents the number of context variables still to be transferred: if the transfer can be performed as a "last shot" the function returns TRUE.

5.1.2 Speeding up termination

To gain in efficiency while performing the "last shot" transfer, which is highly desirable since this transfer has to occur without any interruption from applicative tasks, a list of the addresses of the state variables to be sent in the last shot can be maintained, allowing a fast acquisition of the corresponding values. Let M be the maximum number of context values which can be sent as the "last shot". Then, this list of addresses, named Trailers_List , is set up as a shared variable between the Sweeper and the applicative tasks, which contribute to maintain it continuously updated. The list always holds the addresses of the last M (or less) variables to be transferred by the Sweeper, according to its sequential, circular indexing policy. When the Recursive SR procedure starts, Trailers_List contains the addresses of the last M locations of the whole state memory block: in case no updates are performed by any applicative task by the time the Sweeper gets to the last M locations of the memory block, they are exactly those to be sent as the "last shot". Then, Trailers_List is continuously updated through the Write operation called by applicative tasks on context variables, and through the tRead operation called by the Sweeper. The former acts on Trailers_List as follows: a new assignment to a context variable whose tag was off, makes the address of this variable eligible to be inserted in the Trailers_List. It is actually inserted if: i) Trailers_List currently contains less elements than M, or ii) there exists , in Trailers_List, an address which which will be reached by the Sweeper before the address of the newly assigned variable (i.e. is closer to the current context_index which indicates the most recent memory item examined by the Sweeper). In case ii) the address of the newly assigned variable replaces the one in Trailers_List which is the closest to the current context_index. The tRead operation acts on Trailers_List by simply deleting the addresses of the variables it is working on when the number of updates still to be transferred decreases below M. This situation occurs when the Sweeper starts working in presence of more than M updates to transfer (so, the last shot transfer cannot be arranged), but at the end of its execution less than M updates are left to be sent. A necessary condition for this solution to work, is that the applications and the Sweeper share, besides the Trailers_List structure, also the running pointer used by the Sweeper in accessing the memory block. The pseudo-code representing the actions to manage the Trailers_List, to be included in the code of the Write and tRead operations, is presented in the following.

actions included in the Write operation to manage the Trailers_List

```
/* current_var_address        address of the context variable the Write operation is acting upon   */
/* sweeper_context_index      address of the most recent memory item examined by the Sweeper       */
/* Next_in_List()             function returning the closest address in Trailers_List next to the  */
/*                            sweeper_context_index (according to a circular indexing)             */
/* Next(addr1, addr2)         Boolean function, yielding TRUE if addr1 follows addr2 in the memory */
/*                            block (according to a circular indexing)                             */
.....
if (counter < M)
    insert (Trailers_List, current_var_address);
else {
```

```
    next_to_sweeper_address = Next_in_List(Trailers_List, sweeper_context_index);
    if (Next(current_var_address, next_to_sweeper_address) {
        delete(Trailers_List, next_to_sweeper_address);
        insert(Trailers_List, current_var_address)
    }
}
.....
```

actions included in the tRead operation to manage the Trailers_List

```
.....
if (counter <=M)
    delete(Trailers_List, sweeper_context_index);
.....
```

5.1.3 Termination implementation issues

As introduced in Sect. 5, it is necessary to ensure that no further state updates be performed by applicative tasks after that the termination condition has been met. A simple, albeit unduly restrictive, solution is to mandatorily schedule the Sweeper only at the end of the frame; this is implicitly assumed in the Sweeper version just presented. More efficiency would be gained by allowing the Sweeper to run *also* in CPU time slots internal to the frame: however, the Sweeper must not be allowed to terminate until the frame's end, since otherwise the running applications could modify, unchecked, their context again. The Sweeper should be made aware of when it is the last task running in the current frame, so that only then can it actually terminate its action. This would require a modified implementation of the termination_condition() function, which, in turn, would need to be signalled, from the underlying machine, when it is running the last ride in the frame. A more viable solution is to schedule a special "Send_last_shot" task to run only at the end of a frame, i.e., by giving it a release time of "Frame length - its (fixed) execution time" and a deadline equal to "Frame length", together with a static priority high enough to prevent any pre-emption. This task has to be given the code of the Sweeper as described above, while the mating new Sweeper gets the same code, stripped however of the termination condition part. The Sweeper would keep running, looking for any more updates to do, even though it finds finished its job. It will be killed by the scheduler upon a successful termination signalled by Send_last_shot.

The new Sweeper code is modified as follows:

Algorithm of the Sweeper task

```
/*  variables & constants declaration & initialisation                    */

{
while TRUE
    {
    starting_index=context_index;
    i=0;
    do  {
        if (check_tag(context_array[context_index]))
            {
            context_morsel[i]=tRead(context_array[context_index]);
            i++;
```

```
            };
        context_index++;
        };
    while ((i<=send_block_dim) && (context_index!=starting_index));
     if (i<send_block_dim - 1)
        context_morse![i]= end_of_buffer;
    send(context_morse![], Catcher);
};
```

Observe that the operation of the Sweeper and the Send_last_shot requires some co-operation. In fact, the Sweeper task can be interrupted in the middle of the outer cycle, leaving pending an incomplete buffer. In this case, the number of variables to be transferred is given by the counter reading added to the number of the variables in the buffer. Upon activation, Send_last_shot has to be made aware that there are pending variables to be transferred in addition to those signalled by the counter. A simple solution would be to set up the transfer buffer (and the attending pointers) as a shared variable between the Sweeper and the Send_last_shot, and to let the termination condition take into account the inherited buffer contents.

A slight modification is necessary to the code of the Catcher, to correctly process blocks of context values smaller than the regular buffer size, i.e. containing the *end_of_buffer* mark:

Algorithm of the Catcher task:

```
/*  variables & constants declaration & initialisation                                    */
{
do
    {
    receive(context_morse![], Sweeper);
    i=0;
    while ((i<receive_block_dim) && (context_morse![i]!=end_of_buffer )&&(context_morse![i]!=end_of_transfer))
        {
            update_context(context_morse![i]);
            i++;
        };
    }
while ( (context_morse![i]!=end_of_transfer));
}
```

The Recursive SR does not suffer the problem of inconsistencies between context variables values in the active channel and the corresponding values in the joining channel, provided that tRead and Write operations be performed atomically, since the only agent in charge of sending state values is the Sweeper (or, alternatively, Send_last_shot).

5.2 Extending the basic scheme to multiple channel instances

Consider now a more general GUARDS instance, where two or more active channels co-operate to align a joining channel. Following the same approach as for the Running SR, only organisations based on sharing the SR activities among the active channels will be

considered, thus leaving apart the solution of having all the active channels transferring redundantly their own whole view of the context memory.

A few schemes are examined in the following.

5.2.1 One-sender, Multi-checkers Scheme

According to this solution, only one of the active channels will be given the task of transferring the state. Then, to ensure that the state has been correctly restored, the available redundancy has to be exploited; both techniques presented in Sections 4.2.1 and 4.2.2, i.e. the peer check and the signature-based scheme, can be in principle applied. Now, however, the overhead due to the Byzantine agreement does not affect the applications, as they continue to operate on the original state data structure; its burden is borne by the Sweepers, the Checkers and the Catcher.

The algorithms of the (single) Sweeper and of the Catcher are similar to those already presented in Section 5.1. In the peer check solution the Sweeper sends its values by means of a R_Broadcast() operation. The Catcher gets the state values by means of a ByzAgreement(). Checker tasks are activated on the remaining active channels; they join the ByzAgreement() procedures to acquire the values transferred to the Catcher, and check them against their own corresponding values. Comments given in footnote [1] still apply.

In the signature-based scheme, besides the Sweeper and the Catcher, a task has to be activated on all channels to compute the signature of the whole state, to exchange it through a reliable broadcast, and to make comparisons of the received signatures with its own to detect any discrepancy. Given the asymmetry introduced by having the Sweeper running on only one channel, it is necessary to impose a degree of synchronisation, since the signature has to be computed and processed after the Sweeper and the Catcher have completed, being also sure that the applications have made the same modifications on the channels state, and blocking any further updates by the applications until the end of the SR.

5.2.2 Multi-senders Scheme - 1

As in Sect. 4.2, let the virtual memory block be split into C-1 sub-blocks $SM_1,...,SM_{C-1}$ of similar length; on channel C_i, only SM_i is mapped into a tagged memory structure. Then, each $Sweeper_i$ on the active channels is given the duty of transferring to the joining channel all the updates performed by the applications to the variables in SM_i . The mechanisation of the termination condition is still obtained through a counter variable shared and updated by all applications and the Sweeper. However, this time the counter indicates the number of updated variables in a single memory segment, hence it has in general different values on the different channels. To meet the termination condition at the same time, the counter value has to be exchanged through an interactive consistency check: only if all the Sweepers verify locally the termination condition, the final shot can be arranged by all of them. The same

considerations made in Section 5.1.3, concerning how to guarantee that the termination condition be not followed by further unchecked context variable updates by applications, do apply here, and the same solutions devised there are applicable.

To provide means to overcome errors in the state values of the joining channel, the peer check and the signature-based scheme just recalled in the previous section also apply.

The algorithms detailed in the following make use of R_broadcast() operations and Checkers. A Send_last_shot task, with the same characteristics as that one described in Section 5.1.3, is implemented to ensure a correct termination of the SR procedure.

Algorithm of the Send_last_shot task:

```
/*                                                                              */
/* Interactive_consistency(v,vect[])  input param: v; output param: vect[]      */
/* check_local_t_condition()          is the termination condition locally verified? */
/* context_morsel[]:                  array  of records containing the portion of context read by the Sweeper */
/*                                    introduced to optimise the use of ICN by sending "packets" fit to the */
/*                                    ICN slot; generic record structure:       */
/*                                    <context variable identifier>,<context variable contents>; */
/* end_of_transfer:                   <null context variable identifier><null contents> */
/* end_of_segment()                   Boolean function which checks the end of the block memory assigned */
/*                                    to the Sweeper at hand                    */
/* send_block_dim:                    dimension of the context_morsel array; related to the ICN slot length */
/* context_index:                     addresses the current item of the context */
/* t_condition_for_everyone():        Boolean function yielding TRUE when the "last shot" transfer can be */
/*                                    arranged by all Send_last_shot tasks, while blocking the active */
/*                                    applications                              */
/* R_broadcast()                      Sender's procedure in the Byzantine agreement */
/*end_of_frame()                      boolean_function, yielding TRUE when the end_of_frame signal has */
/*                                    been issued by the underlying run-time support */
/*                                                                              */
/*  variables & constants declaration & initialisation                          */
{
while TRUE
    {
        termination_will=check_local_t_condition()
        Interactive_consistency(termination_will, t_will[])
        if (! t_condition_for_everyone(t_will[]))
    /*
    */
    /*   PART 1   -   recursive transfer                                        */
    /*
    */
    /* The following block has to be executed as an atomic action
    */
            {
            starting_index=context_index;
            do {
                i=0;
                do {
                    if (check_tag((context_array[context_index]))
                    {
                        context_morsel[i]=tRead((context_array[context_index]);
                        i++;
                    }
                    context_index++;
                }
                while ((i< send_block_dim)&& (context_index!=starting_index));
                if(context_index==starting_index)
                {
                    context_morsel[i]= end_of_buffer;
                    R_broadcast(context_morsel[]);
                    wait the end_of_frame signal;
                }
```

```
                else  R_broadcast(context_morsel[]);
            }
            while (! end_of_frame());
        }
        else  {
    /*  PART 2  -  last shot transfer
    */
    /*  for the sake of simplicity  it is supposed that the variables remaining to be transferred fit into one      */
    /*  single context_morsel[] buffer; the last entry of the buffer is reserved to the end_of_transfer mark      */
    /*
    */
    starting_index=context_index;
    i=0;
    do  {
        if (check_tag((context_array[context_index]))
            {
            context_morsel[i]=tRead((context_array[context_index]);
            i++;
            }
        context_index++;
        };
    while ((i< send_block_dim  - 1) && (context_index!=starting_index));
    context_morsel[i]= end_of_transfer;
    R_broadcast(context_morsel[]);
    /*
    */
    /* the application execution is resumed directly by the scheduler in the next frame                          */
    /*
    */
    }
}
```

The code of the Sweeper task, which resembles that of "PART - 1" of the Send_last_shot task, is omitted for the sake of brevity.

Algorithm of the Catcher task:

```
/* update_context()        procedure which updates the state variable identified by the first record      */
/*                         field with the value provided by the second field                              */
/* ByzAgreement(v,S)       Byzantine agreement procedure on the value v received from the sender S        */
/* end_of_transfer():      Boolean function: returns TRUE when context_morsel[] contains the              */
/*                         end_of_transfer mark                                                           */

/*  variables & constants declaration & initialisation                                                    */

{
parallel do
            {
            do  {
                ByzAgreement(context_morsel[], Sweeper1);
                for (i=0; i<=send_block_dim-1; i++)
                    if ((context_morsel[i]!=end_of_buffer))
                        update_context(context_morsel[i]);
                }
            while TRUE;
            };

            {
            do  {
                ByzAgreement(context_morsel[], Send_last_shot1);
                for (i=0; i<=send_block_dim-1; i++)
                    if ((context_morsel[i]!=end_of_transfer)&&(context_morsel[i]!=end_of_buffer))
                        update_context(context_morsel[i]);
            }
            while (! end_of_transfer(context_morsel[]);
            };
```

```
.....
                {
                do {
                    ByzAgreement(context_morsel[], Sweeper_{C-1});
                    for (i=0; i<=send_block_dim-1; i++)
                        if ((context_morsel[i]!=end_of_buffer))
                            update_context(context_morsel[i]);
                    }
                while TRUE;
                };

                {
                do {
                    ByzAgreement(context_morsel[], Send_last_shot_{C-1});
                    for (i=0; i<=send_block_dim-1; i++)
                        if ((context_morsel[i]!=end_of_transfer)&&(context_morsel[i]!=end_of_buffer))
                            update_context(context_morsel[i]);
                }
                while (! end_of_transfer(context_morsel[]);
                };

};
```

Algorithm of the Checker task

```
/* ByzAgreement(v,S)     Byzantine agreement procedure on the value v received from the sender S      */
/* compare(v1, v2)       procedure which compares the context values received, and made consistent    */
/*                       with the Checkers and Catcher running on the other channels through a Byzantine */
/*                       agreement, with the corresponding local values. Appropriate signalling is raised */
/*                       upon detecting a discrepancy.                                               */
/*                                                                                                   */

/*  variables & constants declaration & initialisation                                               */

{
parallel do
                {
                do {
                    ByzAgreement(context_morsel[], Sweeper_1);
                    compare(context_morsel[] , local_context_variable[]);
                    }
                while TRUE;
                };

                {
                do {
                    ByzAgreement(context_morsel[], Send_last_shot_1);
                    compare(context_morsel[] , local_context_variable[]);
                    }
                while (! end_of_transfer(context_morsel[]);
                };

.....

/*  same actions towards messages received from all the other Sweepers, except the                   */
/*  Sweeper  running on the local channel                                                            */
};
```

This solution allows to use CPU time slots available in the middle of a frame, thus substantially increasing the operating speed; the Send_last_shot task has to be scheduled at the end of each frame; the peer checks scheme performed by the Checkers requires special care, as already recalled; the availability of an end_of_frame signal has to be ensured. Note

that a down-to-earth way to implement as an atomic action the main block in the recursive transfer would be the following: given the time span that each Send_last_shot can be endowed, a fixed number of state variable transfers would be programmed in this block. Upon finishing that number of transfers Send_last_shot suspends itself until the end-of-frame signal.

5.2.3 Multi-senders Scheme - 2

This scheme is based on the same approach of apportioning the SR among C-1 Sweepers with the same duties as in the previous scheme; however, in each active channel a snapshot of the whole state memory block is held in a temporary buffer, from which the local Sweeper gets its share of values to be sent to the Catcher. The Sweepers may be allowed to run in any spare CPU time slot, but it must be ensured that the snapshots on the different channels are identical. To this purpose, the iterative nature of applicative tasks is exploited. The snapshots are taken simultaneously at the end of a (common) frame: when the Sweepers finish transmitting the current state buffer, they engage in a rendez-vous, to wait each other. Next, they ask the scheduler to be run as the last task of the frame (whether the current, if it is not too late, or the next one) and suspend themselves. When awoken they lock the state variables and take the new snapshot of those variables whose tag is set. After the snapshot all tags are reset, and the state variables unlocked. At this point there is the opportunity to check for the termination condition; a successful interactive consistency protocol must be now executed on the decision to terminate. If the termination condition has not yet been met (or agreed upon) the Sweepers go on repeating the same steps, otherwise the last shot is sent and the SR ends. The mechanisation of the termination condition is still obtained through a counter variable associated to the tagged memory segment; its value is copied by the Sweepers when taking the snapshot (the counter value is reset thereafter). However, since now the Sweepers work on a frozen copy of the memory, they do not need to decrement the counter value, but they only need to read it once to check if the termination condition is met. Note that the special tRead operation is not required anymore.

The algorithms for implementing this solution are detailed below.

Algorithm of the Sweeper task:

```
/* context_morsel[]:          array  of records containing the portion of context read by the Sweeper  */
/*                            introduced to optimise the use of ICN by sending "packets" fit to the    */
/*                            ICN slot; generic record structure:                                      */
/*                            <context variable identifier>,<context variable contents>;               */
/* end_of_transfer:           <null context variable identifier><null contents>                        */
/* end_of_segment()           Boolean function which checks the end of the segment to be transferred   */
/*                            by  the present Sweeper                                                  */
/* send_block_dim:            dimension of the context_morsel array; related to the ICN slot length    */
/* context_index:             addresses the current item of the context; must be initialised to the     */
/*                            beginning of the segment to be transferred by  the present Sweeper       */
/* t_condition_for_everyone(): Boolean function yielding TRUE when the "last shot" transfer can be     */
/*                            arranged by all Sweepers, while blocking the active applications         */
```

```
/* local_context_array[]:        array holding the whole channel context                          */
/* R_broadcast()                 Sender's procedure in the Byzantine agreement                    */
/* ready_to_get_next_snapshot()  procedure to execute the rendez-vous among all the Sweepers to   */
/*                               synchronise on the next snapshot operation                       */
/* get_next_snapshot()           procedure which takes a new snapshot, copying the whole block of */
/*                               memory and the counter value in local variables                  */
/*                                                                                                */
/*                                                                                                */
/*  variables & constants declaration & initialisation                                            */
{
    do {
        ready_to_get_next_snapshot();
        get_next_snapshot(local_context_array[], local_counter);
        Interactive_consistency(local_counter,counters_array[])
        if (! t_condition_for_everyone(counters_array[]))
            do {
                i=0;
                do {
                    if (check_tag((local_context_array[context_index]))
                        {
                        context_morsel[i]=read((local_context_array[context_index]);
                        i++;
                        }
                    context_index++;
                    };
                while ((i< send_block_dim) && ! (end_of_segment()));
                if (end_of_segment)
                    context_morsel[i]= end_of_buffer;
                R_broadcast(context_morsel[]);
                }
            while(!end_of_segment());
        else  {

/* the following actions have to be performed atomically                                          */
/*                                                                                                */
/*  for the sake of simplicity  it is supposed that the variables remaining to be transferred fit into one */
/*  single context_morsel[] buffer; the last entry of the buffer is reserved to the end_of_transfer mark */
/*                                                                                                */
        i=0;
        do  {
            if (check_tag((local_context_array[context_index]))
                {
                context_morsel[i]=read((local_context_array[context_index]);
                i++;
                }
            context_index++;
            };
        while ((i< send_block_dim - 1) && (context_index!=last_index));
        context_morsel[i]= end_of_transfer;
        R_broadcast(context_morsel[], Catcher);
        exit();
        }
}
```

Algorithm of the Catcher task:

```
/* update_context()        procedure which updates the state variable identified by the first record */
/*                         field with the value provided by the second field                        */
/* ByzAgreement(v,S)       Byzantine agreement procedure on the value v received from the sender S  */
/* end_of_transfer():      Boolean function: returns TRUE when context_morsel[] contains the        */
/*                         end_of_transfer mark                                                     */

/*  variables & constants declaration & initialisation                                            */

{
parallel do
    {
        do {
```

```
            ByzAgreement(context_morsel[], Sweeper1);
            for (i=0; i<send_block_dim; i++)
                if ((context_morsel[i]!=end_of_transfer)&&(context_morsel[i]!=end_of_buffer))
                    update_context(context_morsel[i]);
            }
        while (! end_of_transfer(context_morsel[]);
    };

.....

    {
        do {
            ByzAgreement(context_morsel[], SweeperC-1);
            for (i=0; i<send_block_dim; i++)
                if ((context_morsel[i]!=end_of_transfer)&&(context_morsel[i]!=end_of_buffer))
                    update_context(context_morsel[i]);
            }
        while (! end_of_transfer(context_morsel[]);
    };
};
```

Algorithm of the Checker task

```
/* ByzAgreement(v,S)     Byzantine agreement procedure on the value v received from the sender S      */
/* compare(v1, v2)       procedure which compares the context values received, and made consistent    */
/*                       with the Checkers and Catcher running on the other channels through a byzantine */
/*                       agreement, with the corresponding local values. Appropriate signalling is raised */
/*                       upon detecting a discrepancy.                                               */
/*                                                                                                   */

/*  variables & constants declaration & initialisation                                               */

{
parallel do
    {
        do {
            ByzAgreement(context_morsel[], Sweeper1);
            compare(context_morsel[] , local_context_variable[]);
            }
        while (! end_of_transfer(context_morsel[]);
    };

  .....      /*  same actions towards messages received from all the other Sweepers, except the    */
             /*  Sweeper  running on the local channel                                             */
};
```

5.3 *Mechanisms required to the implementation in GUARDS*

The main mechanism required to implement Recursive SR procedures is the tagged array, and the ancillary access operations.

Tagged memory architectures have been presented in the past [5], [10], mostly with reference to hardware implementations which are not appropriate in a COTS context. In fact, at the very least, the memory word needs to be complemented by the tag bit, resulting into a layout unavailable in off-the-shelf memory boards. Moreover, while the usual Write operation needs "only" a handful of hardware gates to also set the tag, the Read instruction must be accompanied by the novel tRead, in charge of clearing the tag bit. Some device at very low level in software might be possible, such as devoting an unused processor trap (if any) to the

purpose. A hardware solution would then give good performance, but would not be fully transparent to applications.

A more viable solution would be to implement the tagged array concept in software. This approach is definitely not transparent to the user; the design overhead, however, is not larger than that occurred in the Running SR (Section 4) if language features like Ada95' controlled types are exploited. A clean implementation could be attained by means of a global state object, where the three access methods, Read, Write and tRead would be defined over an internal structure. This could be arranged as an array of records, each containing the fields:

<Item No.>, <Variable Name>,<Variable Value>, <Tag>.

The read/write operations could specify both an indexed access (by specifying the <Item No.>) or an associative access (by specifying the <Variable Name>). The methods implementations would of course take care of handling the tag. For example, tRead() could return the Variable Value if the tag is TRUE or a <null> value if the tag is FALSE; the tag would be anyway reset to FALSE on exit.

If the termination condition is based on the counter described in 5.1.1, the counter could be comfortably housed in the context object, and termination_condition() could be implemented as an added method.

A switch to a special *SR mode* is still necessary, to trigger the activation of the Sweeper and of the Catcher tasks. To allot CPU time and ICN bandwidth to the Sweeper, it may still be necessary to run the applicative tasks in a slightly degraded mode. The state object can also easily integrate the mode switch. Two separate versions of the methods, accessed through a single mode-set pointer, promise to be faster than a single program, stuffed with if clauses to steer operations.

The (wildly unconventional) mechanism to signal the Sweeper when it is running the last ride in the frame should be provided by the OS kernel, since it has to interface to the scheduler. If, instead, the simplified solution of reserving the last CPU time slot(s) in a frame to the Sweeper's associate, Send_last_shot, can adopted (as seen in Sect 5.1.3), there is no need for special mechanisms for this purpose. In the former case there is the need to modify the OS kernel, losing its COTS trait; in the latter, there is no explicit visibility of the gimmick at the application code level, which is a non-advisable practice for high-confidence systems.

6. Qualitative comparison between the two schemes

The Running SR requires a finite, deterministic, predictable time to terminate, provided it is allotted sufficient CPU time and ICN bandwidth to transfer at least one byte per frame, AND the applications can run in the SR mode. This last clause actually means that applications, while Running SR code (i.e. including the remote state variable updates), must be given enough CPU time and ICN slots. In fact, the traffic on ICN may substantially increase,

although to a degree that is strongly dependent on the applications. This can be the limiting factor to the viability of this scheme: in a given application environment a preliminary analysis will have to be carried out, to check if the SR mode can be activated without altering the pattern of active applications; in general, save for very over-dimensioned systems, there will be the need to trim the "normal" application accesses to the ICN, whether by putting to sleep some less vital application or by running skimmed down versions thereof. In the general case, a different ICN schedule will be needed for the Running SR mode.

In the Recursive SR, the applications continue to make only local updates to their state variables, with very small overhead: their schedule may well be retained during SR. In fact, the burden of remote updates is handed over to the independently scheduled Sweeper. On the other hand, the termination condition could be never satisfied, and the time required to satisfy it depends on the actual pattern of the context variable updates. On another side, the state information has a more complex structure (because of the tags), and requires one more special manipulation instruction (tRead). A thorough analysis should be done, based on the maximum number of updates along the possible program paths in an iteration; this can yield an estimate, which can be useful to establish the very applicability of the scheme, but gives only little information on the performance in the real life.

The two approaches (together with their variations) must be quantitatively evaluated (possibly on settings taken by real, possibly small scale examples) in order to obtain a sound understanding of their feasibility and to ascertain which one is better where.

7. Conclusions

Two main schemes for inter-channel state restoration have been introduced, to be applied in GUARDS architectures, and some variations of them have been explored. After introducing the need for state restoration and the architectural constraints solutions have to obey, the most known solutions have been briefly recalled. Then the Running SR approach has been described, which basically slows down the running applications and allows state restoration to be achieved in a finite pre-determined time. Some variations of this approach, which allow for different trade-offs between network bandwidth requirements, time to complete the procedure itself and fault tolerance provisions, have been proposed and discussed. Similarly the basic issues of the Recursive SR approach have been introduced, and refinements and enhancements presented. In this note the mechanisms required to implement these solutions in GUARDS systems have been identified, and some qualitative comparison and evaluation carried out. Still, due to the nature of this problem, for which generally applicable solutions are not known, a relevant effort for providing quantitative models is necessary in order to gain the necessary insights on whether (in which conditions) these solutions are really applicable to GUARDS and in which cases one should be preferred to the other.

References

[1] S. J. Adams, "Hardware Assisted Recovery from Transient Errors in Redundant Processing Systems," *19th Int. Symp. on Fault Tolerant Computing (FTCS-19)*, Chicago, IL, USA, 1989, pp. 512-519.

[2] J. Arlat, I. Crouzet and C. Landrault, "Operationally Secure Microcomputers," *FTSD*, 1978, pp.

[3] GUARDS_Consortium, "Specifications and Preliminary Design of Dependability Mechanisms," GUARDS Consortium, Task Output (1st Year Deliverable D1O2) Guards D100 TO 2007 c, march 24 1997.

[4] L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals Problem," *ACM Trans. on Programming Languages & Systems*, vol. 4, pp. 382-401, 1982.

[5] G. J. Myers, *Advanced Computer Architectures*, J. Wiley & Sons, Inc, 1982.

[6] D. Powell, "Preliminary definition of the GUARDS architecture," LAAS-CNRS, Task Output D1O1 D1A1/AO/5000/D, Jan 6 1997.

[7] D. K. Pradhan, S. K. Gupta and M. G. Karpowsky, "Aliasing Probability for Multiple Input Signature Analyzer," *IEEE Trans. Computers*, vol. 39, pp. 586-591, 1990.

[8] J. Rushby, "A Fault-Masking and Transient-Recovery Model for Digital Flight-Control Systems," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, J. Vytopil (Eds.), 1993, pp. 109-136.

[9] J. Rushby, "Reconfiguration and Transient Recovery in State Machine Architectures," *26th Int. Symp. on Fault Tolerant Computing (FTCS-26)*, Sendai, Japan, 1996, pp. 6-15.

[10] T. Sims, "Real Time Recovery of Fault Tolerant Processing Elements," *15th Digital Avionics Systems Conference*, Atlanta, GA, USA, 1996, pp. 485-590.

[11] J. E. Smith, "Measures of the Effectiveness of Fault Signature Analysis," *IEEE Trans. Computers*, vol. 29, pp. 510-514, 1980.

[12] L. Strigini, "Support of Design Fault-Tolerance in Delta-4 Software - Proposals," IEI-CNR, Pisa, Italy, Technical Report N. B4-54 B4-54, December 1988.

[13] K. S. Tso, A. Avizienis and J. P. J. Kelly, "Error Recovery in Multiversion Software," *IFAC SAFECOMP '86*, Sarlat, France, 1986, pp. 35-41.

[14] A. Wellings, "Inter-channel scheduling, offset analysis and the ICN," University of York, Technical Note 7042 a, 17 Sep 1997.