Consistent, global state transfer for message-passing parallel algorithms in Grid environments

PhD thesis

Written by

József Kovács
research fellow
MTA SZTAKI

Advisors:
Dr. Péter Kacsuk (MTA SZTAKI)
Dr. László Kovács (ME)

Leader of Doctoral School:
Dr. Tibor Tóth

Computer and Automation Research Institute,
Hungarian Academy of Sciences
(MTA SZTAKI)

„Hatvany József" Doctoral School of Information Science,
University of Miskolc
(ME)

Budapest, 2008

Abstract

This dissertation introduces a new way to combine and complement existing parallel checkpointing techniques to be applied for software heterogeneous ClusterGrid infrastructure in order to provide transfer (migration) of parallel algorithms (applications) among clusters and nodes. While existing solutions are aiming at providing application transparency by building special middleware, the solution presented in this dissertation provides both application and middleware transparency for the algorithms at the same time.

This dissertation addresses the problem of migrating parallel applications among clusters where source and target clusters may not be operated by identical middleware components. In this case, application may not lay on any special service provided by the middleware running on the cluster. Therefore the solution must somehow be designed in a way that middleware of the cluster is not taking part in the checkpointing procedure and the application programmer must not be forced to touch his/her application source code to reach this functionality.

The solution presented in this dissertation introduces a new checkpointing method by which transparent migration of parallel applications can be solved. The overall checkpointing design space is introduced, requirements are defined and the method is introduced. Two different checkpointing frameworks – targeting different application development methods - have been designed and introduced: Grapnel and TotalCheckpoint. Based on these results process and application migration is introduced providing middleware and application transparency.

Contents

List of Figures

Abbreviations

APP	**APP**lication
ASM	**A**bstract **S**tate **M**achine
CKPT	**C**he**KP**oint**T**
CP	**C**heck**P**oint
CR	**C**heckpoint-**R**estart
CRR	**C**heckpointing and **R**ollback-**R**ecovery
DIWIDE	**DI**stributed **WI**ndows **DE**bugger
FT	**F**ault-**T**olerant
GCA	**G**rid **C**heckpointing **A**rchitecture
GRAPNEL	**GRAP**hical **NE**twork **L**anguage
GRED	**GR**aphical **ED**itor
GRP	**GR**a**P**h (graph representation of GRAPNEL)
GUI	**G**raphical **U**ser **I**nterface
HPC	**H**igh **P**erformance **C**omputing
HTC	**H**igh **T**hroughput **C**omputing
ID	**ID**entifier
IO or I/O	**I**nput-**O**utput
LIB	**LIB**rary
MPE	**M**essage **P**assing **E**nvironment
MPI	**M**essage **P**assing **I**nterface
MW	**M**aster-**W**orker
NFS	**N**etwork **F**ile **S**ystem
OS	**O**perating **S**ystem
P-GRADE	**P**arallel **G**rid **R**un-Time and **A**pplication **D**evelopment **E**nvironment
PVM	**P**arallel **V**irtual **M**achine
PVMD	**PVM D**aemon
PVMLIB	**PVM LIB**rary
PWD	**P**iece**W**ise **D**eterministic
RM	**R**esource **M**anager
SCHED	**SCHED**uler
SGM	Computational **SeGM**ent

SI,SR,SF	**I**nitial **S**tate**, R**unning **S**tate**, F**inishing **S**tate
SRC	**S**ou**RC**e (code)
STID	**S**ystem **T**ask **ID**entifier
TCKPT	**T**otal**C**hec**KP**oin**T**
TID	**T**ask **ID**entifier
TM	**T**uring **M**achine
UTID	**U**ser **T**ask **ID**entifier

1 Introduction

Parallel applications are developed to utilise the computational power of numerous computers to increase performance. These applications consist of processes for calculation and a message-passing framework to exchange sub results among the processes running on different node of a supercomputer, a cluster or a Grid. The executing infrastructure is a collection of machines accumulating mainly processing and storage capacity. Due to the inherently dynamic and error prone behaviour, parallel applications must survive the loss of any resource they are using during execution.

In order to avoid the failure of a parallel application running on multi nodes, special checkpointing techniques are required to save the overall state of the application. Later, the application can be resumed on a different set of nodes based on the saved state. Process migration in distributed systems is an event when a process running on a resource is redeployed to another one in a way that the migration does not influence the result of calculation. To enable this mechanism a very strict and coordinated cooperation of the operating system, scheduler, checkpointer, message-passing framework and the application is required.

The goal of the research is to design and provide a parallel checkpointing technique for parallel applications and execution environments, where the impact generated by the checkpointing facility is minimized for the entities mentioned above. In this work, a highly automated new checkpointing framework is presented. The main design goals are to perform programmer transparency, adaptability, automation and independence from the execution environment.

The dissertation is organised in the following way. Section 1 is to define the scope of the dissertation by introducing Clusters, Grid and ClusterGrid in section 1.1, by over viewing the rollback/recovery and checkpointing techniques in section 1.2, and by summarising existing solutions in section 1.3.

In section 2 the focus is on to develop a general method. First, transparency conditions are identified in section 2.1 and then the method is defined in section 2.2. In both cases a formal description is given based on the model framework introduced in section 2.1.5.2. For the method, the ground model is elaborated in section 2.2.4 by defining the initial states and finally the model is validated in section 2.2.5.

Section 3 aims at giving a concrete checkpointing solution applying the ground model. The P-GRADE checkpointing and migration framework is developed in section 3.1, the corresponding model is introduced in section 3.1.4 and its relation to the ground model is detailed in section 3.1.5. To overcome the limitations of the P-GRADE checkpointer, the TotalCheckpoint framework is designed and introduced in section 3.2. Finally, the corresponding model is developed in section 3.2.6, and the relation to the grapnel model is detailed in section 3.2.7.

Section 4 is detailing the method of migration based on the P-GRADE checkpointer tool. Process migration is defined and introduced in section 4.1, while application migration is elaborated in section 4.2.

Finally, section 5 discusses and summarises the work described in this dissertation and section 6 contains references.

1.1 Cluster, Grid, ClusterGrid

In order to deliver more performance, increasing the number of processors to execute a job is trivial. With the increasing speed of network components and with the decreasing cost of single processor computers building computer clusters in the early 90s became an efficient alternative to the different supercomputers containing hundreds/thousands of processors but with an extremely high-cost.

A computer cluster [26] is a group of tightly coupled computers that work together closely to provide the view of a single computer. The components of a cluster are usually, but not always, connected to each other through fast local area networks. High-performance computing (HPC) clusters are implemented primarily to provide increased performance by splitting a computational task across many different nodes in the cluster, and are most commonly used in scientific computing.

The HPC computer cluster can be defined as follows:

1. Consists of many of the same or similar type of machines (Heterogeneous clusters are a subtype, still mostly experimental)
2. Machines use dedicated network connections
3. All machines share resources such as a common home directory (NFS can be a problem in very large clusters.)
4. They must trust each other so that rsh or ssh does not require a password, otherwise you would need to do a manual start on each machine.
5. Must have communication software such as an MPI or PVM implementation installed to allow programs to be run across nodes

Such clusters commonly run custom programs which have been designed to exploit the parallelism available on HPC clusters. HPCs are optimized for workloads which require jobs or processes running on the separate cluster computer nodes to communicate actively during the computation. These include computations where intermediate results from one node's calculations will affect future calculations on other nodes.

The concept of Grid [28] emerged as a natural extension and generalization of distributed supercomputing or meta-computing [27]. Still one of the important goals of the Grid is to provide a dynamic collection of resources from which applications requiring very large computational power can select and use actually available resources. In that sense Grid is a natural extension of the concepts of supercomputers and clusters towards an even more distributed and dynamic parallel program execution platform and infrastructure. Components of such a Grid infrastructure include supercomputers and clusters and hence parallel application programs and programming tools of supercomputers and clusters are expected to be used in the Grid, too.

These days there are many projects with the idea of extending and generalizing the existing parallel program development and execution environments towards the Grid. The current idea is that a parallel program that was developed for supercomputers and clusters should be used in the Grid, too. Indeed, execution of a parallel program in the Grid does not defer semantically from the execution in a parallel computing system. The main difference comes from the speed, reliability, homogeneity and availability of the exploited resources. In the traditional parallel

systems like supercomputers, the speed of the processors, communication networks, memory access, I/O access are steadily high, the components of the supercomputers are homogeneous and highly reliable and statically available (after allocating them by a resource scheduler). Clusters have more or less the same parameters as the supercomputers though their components might be heterogeneous, they are usually less reliable, a bit slower and the availability of their components is less static than in the supercomputers.

The Grid represents a significant modification in these parameters. The Grid is very heterogeneous, the components (resources) dynamically change both in the respect of their speed and availability and there is no guarantee for their reliability. It means that a parallel program that was developed for a homogeneous, high-speed, static and reliable execution environment should perform satisfactorily well even in a heterogeneous, changing-speed, dynamic and un-reliable execution environment.

In a computational Grid various resources are collected where one or more broker component performs the mapping of applications to resources based on the application requirements and resource capabilities. In this dissertation ClusterGrid is defined as a Grid that can contain clusters represented as one compound and indivisible resource for the broker. Clusters can be maintained by different organizations, so scheduling and execution policy as well as software environment of the clusters can be different. On clusters various schedulers can handle jobs and at the same time the cluster might run different versions of operating systems or message-passing environments. Above all any kind of service can be installed additionally to support the requirements of the organization owning and operating the cluster. These services may differ in every cluster in the ClusterGrid.

Technology of Grid is closely related to cluster computing. The key differences between grids and traditional clusters are that grids connect collections of computers which do not fully trust each other, and hence operate more like a computing utility than like a single computer. In addition, grids typically support more heterogeneous collections than are commonly supported in clusters.

A ClusterGrid in the context of this dissertation is defined as a Grid of clusters, where each cluster is handled as an individual and indivisible resource by the Grid. Each cluster can be owned by different authorisation therefore clusters have their own local policies regarding the handling of jobs to execute.

Since clusters have different policies, their middleware is also defined by the local authority operating the cluster which results in a software-heterogeneous Grid of clusters. Any job to be executed on a cluster might face with different middleware components on the different clusters.

After a short introduction of Clusters, Grids and ClusterGrids, the focus is on the latter one. A more precise definition of ClusterGrid will follow in Section 2.1.

1.2 Overview of rollback-recovery techniques

In the recent decade numerous surveys, classifications and taxonomies have appeared aiming at summarizing existing approaches of the different rollback-recovery [49][50][51] techniques. The main purpose of this section is to give a short overview of the rollback/recovery techniques based on the available literature.

Checkpoint is defined as a designated place in a program at which normal processing is interrupted specifically to preserve the status information necessary to allow resumption of processing at a later time [52]. Checkpointing is the process of saving the status information. By periodically invoking the checkpointing process, one can save the status of a program at regular intervals. If there is a failure one may restart computation from the last checkpoint thereby avoiding repeating computation from the beginning. The process of resuming computation by rolling back to a saved state is called rollback recovery.

A checkpoint can be saved on either stable storage or the volatile storage of another process, depending on the failure scenarios to be tolerated. For long-running scientific applications, checkpointing and rollback-recovery can be used to minimize the total execution times in the presence of failures. For mission-critical service-providing applications, checkpointing and rollback-recovery can be used to improve service availability by providing faster recovery to reduce service down time.

Rollback-recovery in message-passing systems is complicated by the issue of rollback propagation (rolling back to previous state of the application enforced by the inconsistency of message states) due to inter process communications. When the sender of a message m rolls back to a state before sending m, the receiver process must also roll back to a state before m's receipt; otherwise, the states of the two processes would be inconsistent because they would show that message m was not sent but has been received, which is impossible in any correct failure-free execution.

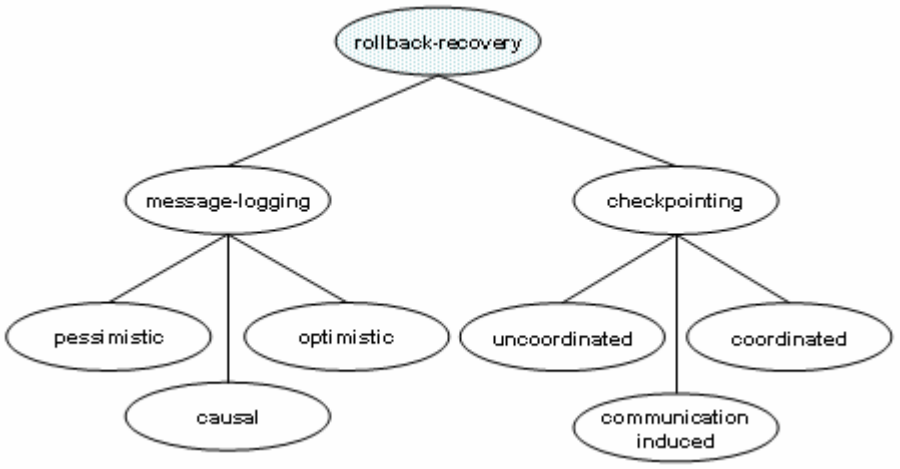

Figure 1 Classification of rollback-recovery algorithms

In some cases, cascading rollback propagation may force the system to restart from the initial state, losing all the work performed before a failure. This unbounded rollback is called the domino effect. The possibility of the domino effect is highly undesirable because all checkpoints taken may turn out to be useless for protecting an application against losing all useful work upon a failure.

In a message-passing system, if each participating process takes its checkpoints independently then the system is susceptible to the domino effect. This approach is called uncoordinated checkpointing. One way to avoid the domino effect is to perform coordinated checkpointing: the processes in a system coordinate their checkpoints to form a system-wide consistent state. A state is consistent if for every process state it is true that 1) every message that has been received has also been sent in the state of the sender and 2) each message that has been sent has also been received in the state of the receiver.

Such a consistent set of checkpoints can then be used to bound the rollback propagation. Alternatively, communication-induced checkpointing forces each process to take checkpoints based on some application messages it receives from other processes. This approach does not require system-wide coordination and therefore may scale better. The checkpoints are taken such that a consistent state always exists, and the domino effect cannot occur.

Systems with more than one processor are known as multiprocessor systems. As the number of processors increase the probability of any one processor failing is high. It has been found in practice that over 80% of the failures in such systems are transient and intermittent [53]. Checkpointing and rollback recovery are particularly useful in such situations. Checkpointing, however, is more difficult in multiprocessors as compared to uniprocessors. This is due to the fact that in multiprocessors there are multiple streams of execution and there is no global clock. The absence of a global clock makes it difficult to initiate checkpoints in all the streams of execution at the same time instance.

The aforementioned solutions rely solely on checkpoints, thus the name checkpoint-based rollback-recovery. In contrast, log-based rollback-recovery uses checkpointing and message logging (see Figure 1). In log-based rollback-recovery a process can deterministically recreate its pre-failure state even if it has not been checkpointed by logging and replaying the nondeterministic events in their exact original order. It is mostly used for applications that frequently interact with the outside world.

1.2.1 Log-based rollback/recovery

Log-based rollback recovery protocols, or message logging protocols, supplement normal checkpointing with a record of messages sent by and received by each process. If the process fails, the log can be used to replay the progress of the process after the most recent checkpoint in order to reconstruct its previous state. This has the advantage that process recovery results in a more recent snapshot of the process state than checkpointing alone can provide. Additionally, log-based approaches avoid the domino effect, since the failed process can be brought forward to the global application state rather than individual processes being forced to roll back for consistency with the failed process.

Log-based recovery protocols rely on the piecewise deterministic assumption (PWD). This assumption dictates that the system has the ability to detect the nondeterministic events that transition to the next state interval. Furthermore, the system must be able to record information about the events such that the important aspects of the event can be recreated in a reconstruction of the process state [57].

An orphan process p is a process that does not fail, but whose state depends on a nondeterministic event that was not recorded to stable storage and the determinant of

which was not recorded on p. Such a process therefore cannot be restored to a consistent state because the information required to replay an event has been lost [49].

There are three main techniques (see Figure 1) used by log-based recovery protocols to guarantee that all processes can be recovered to a consistent state in the event of a failure: Pessimistic, optimistic, and causal. Each of the three approaches has its own tradeoffs for performance, ease of process recovery, and ability to roll back processes that did not fail.

Pessimistic logging techniques, sometimes called synchronous logging, record the determinant of each event to stable storage before the event is allowed to affect the computation. This ensures that the system will easily be able to recover from the failure of any process occurring at any time, because no process can be affected by an event that has not been logged. Pessimistic logging has two key advantages. First, in the event of a failure, processes that did not fail can never become orphans and need not take any special actions. This greatly simplifies the recovery algorithm. Second, garbage collection of message logs and checkpoints is simple - only one checkpoint must be maintained for each process, and message logs older than that checkpoint can be discarded [49].

Optimistic or asynchronous, logging techniques record logs to volatile storage, which is then periodically written to stable storage. This substantially reduces the performance overhead on the application because it does not need to block while waiting for each message to be written to disk. Unfortunately, recovery of the system in the event of a failure is much more complex. Since messages recorded in volatile memory will be lost in the event of a process failure, processes can become orphans. In addition to the recovery of the failed processes, the surviving processes must be rolled back to a state that does not depend on any lost messages [57].

Causal logging protocols maintain the advantages of both optimistic and pessimistic logging, but at the expense have requiring much more complex recovery techniques. The low overhead of optimistic logging is attained by saving logs to volatile storage, similar to optimistic logging [58][59].

1.2.2 Checkpoint-based rollback/recovery

Checkpoint-based rollback/recovery techniques can be classified into three groups based on the abstraction level: kernel-level, library-level and application-level, where the library- and kernel- level checkpointing is also called jointly as system-level checkpointing (see Figure 2).

The *kernel-level* [54] checkpoint support is implemented by a kernel module part of the operating system. Any executed process can be checkpointed in this way, since no changes in the executable are required. The internals of a process are text, data, dynamically allocated data, shared libraries, stack, processor registers, signal handlers and the signal masks, open files and shared resources. A kernel module can easily access the state of theses internals to create a checkpoint. Later the reconstruction of the process is also done by the operating system. This technique produces checkpoint file with binary data (e.g. copy of the memory), so it can be used on computational nodes of a cluster with the same platform.

The *library-level* checkpointing usually requires the application to be relinked with a special library performing the state saving and resumption procedure. In this case special techniques are used to access the process internals. Similarly to kernel-

level the checkpoint information contains bits that constitute the state of the process. Sometimes it is problematic to patch the operating system to support checkpointing, so the main advantage of this approach is to avoid the modification of the OS. Tools implementing this approach are for example Condor [32], Libckpt[31] and Ckpt[29].

The third alternative is *application-level* [55] checkpointing, where applications provide their own checkpointing code. The application is written so that it correctly restarts from various positions in the code by storing certain information to a restart file. The programmer needs to save data to recover the program state. The advantage is that the checkpoint information can be produced in a way that the application is able to checkpoint and restart among nodes with different platforms. The disadvantage of this approach is that it complicates the coding of the application program. Sometimes adding this type of checkpoint support to a parallel application is a comparable task with developing the whole application.

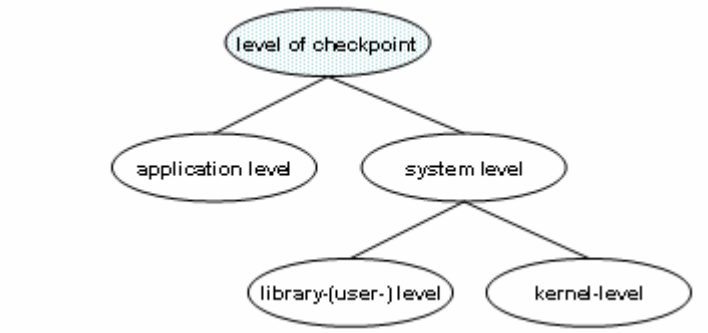

Figure 2 Classification of checkpoint levels

Beyond the previous three directions, two additional aspects (independent from the abstraction levels) are distinguished in parallel checkpointing on how parallel processes are instructed when checkpointing is performed. The three aspects are: coordinated, uncoordinated, communication-induced (see Figure 1).

In *uncoordinated* checkpointing each process of the application can save its state at any time independently from the neighbouring processes. Checkpointing action is not taking the consistency of messages into account, so it can be implemented and executed easily. The problem may occur at restart phase. First, each process reloads its state from the latest checkpoint and then they check whether the application has consistent state [56] regarding messages. Consistency exists if neither duplicated nor lost messages is detected. If the latest checkpoints do not form a consistent state of the application, based on the dependencies of messages rollback propagation is executed. Due to the domino effect it can easily happen that rollback is repeated until the application reaches its initial point i.e. the application is started from the beginning.

Contrary to the previous one, *coordinated* checkpointing creates a consistent state of the application at the time of checkpoint saving instead of trying to create it at resumption phase. Whenever an application must be checkpointed, a global coordination protocol, implemented by exchanging special marker messages, is used to coordinate the state saving of the individual processes by creating a consistent state for the application. A global snapshot is taken from which the computation can be restarted. In this case a distinguished process is responsible to initiate and execute the checkpoint saving protocol. The Chandy-Lamport [56] algorithm is the most well-known protocol for this purpose.

Communication-induced (or quasi-asynchronous) checkpointing provides resilience to the domino effect without requiring global coordination across all checkpoints. Each process takes checkpoints locally, as in uncoordinated checkpointing. However, such protocols allow for processes to be forced to take a checkpoint in order generate a global checkpointed state that will not cause domino effect. Each message passed between processes contains extra protocol information that allows the recipient to determine for it whether or not it should take a forced checkpoint [49].

1.2.3 Checkpoint based migration

Process migration in distributed systems is a special event when a process running on a particular resource is moved to another one in such a way that the migration does not cause any change in the process execution. It means that the process is not restarted instead; its execution is temporarily suspended and then resumed on the new resource. In order to provide this capability migration usually relies on some checkpointing techniques to save the state of the target process and to reconstruct it on the target machine.

Such migration mechanism can be used in several scenarios. First, in supercomputing applications load-balancing is a crucial issue. Migration can solve the problem of unbalanced parallel sites of the Grid. Processes on overloaded machines can migrate to underloaded machines without terminating the entire application. Similarly, load-balancing can be ensured among different sites of the Grid, i.e., when a site becomes overloaded complete applications can migrate to other sites. Third, the migration module can be used for providing fault-tolerance capability for long-running applications if machines are corrupted or need system maintenance during the execution. Fourth, migration can be driven by resource needs, i.e., processes can be moved to a remote site in order to access special unmovable resources. For example, processes may need to use special equipments or huge databases existing on dedicated machines in the Grid.

During migration a tool suspends the execution of the process, collects all the internal status information necessary for resumption and terminates the process. Later it creates a new process and all the collected information is restored for the process to continue its execution from where it was suspended.

1.3 Overview of checkpointing and migration frameworks

In this section an overview is given on various systems in the field of parallel (mostly PVM and MPI) checkpointing. There are numerous systems, but without giving an endless list, the most significant tools on this field are introduced.

1.3.1 CoCheck

CoCheck is a research project which aims at providing Consistent Checkpointing for various parallel programming environments both PVM and MPI. CoCheck implements a coordinated, consistent checkpointing of parallel applications.

In PVM version of CoCheck [64][72] a special process, called Resource Manager (RM) is dedicated to perform decision on task allocation whenever a new task is spawned. When checkpointing, it notifies the tasks by sending signal-message pairs to all the processes to interrupt them as a first step. After successful interruption it synchronizes the messages in order to avoid having in-transit messages in the messaging layer at the time of checkpointing.

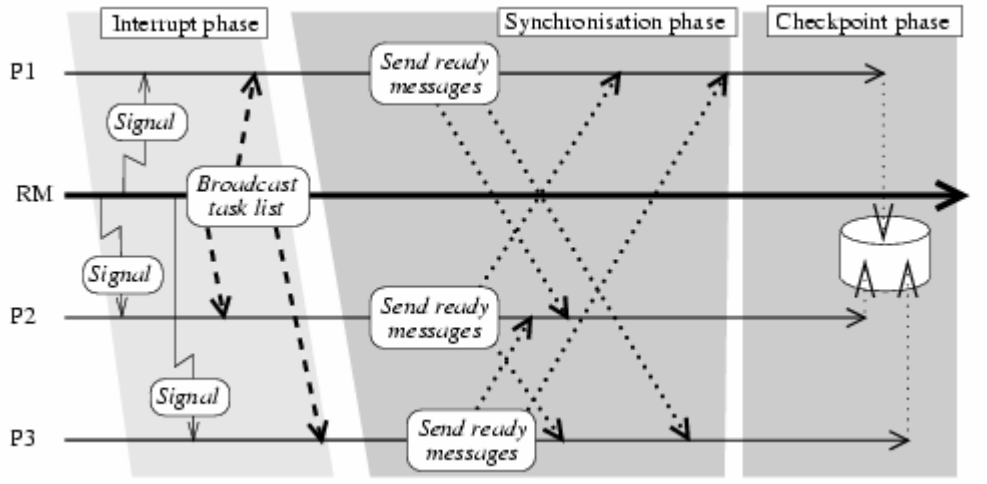

Figure 3 The CoCheck checkpointing protocol

When all the messages have been flushed, the processes detach and finally the process memory is saved (see Figure 3). The restart of a parallel job is steered by a modified startup routine (crt0.o) that belongs to the operating system. This tool was developed in the mid of 90's and it has no more code maintenance, but it became a basic reference work for all parallel checkpointing systems.

To create the MPI version of CoCheck [71] a new MPI implementation has been developed, called tuMPI. To add checkpointing support for tuMPI, the central instance of CoCheck (RM in the PVM version) was integrated as an additional component in the daemon processes of tuMPI. Consistent state of the application processes was ensured by the same algorithm as used in the PVM version. Finally, the tool was optimised for migration of tasks, by transferring checkpoint information to a skeleton process of the migrating one.

One of the main well-known features of CoCheck is its checkpoint protocol, which makes sure in-transit messages are flushed from the message-passing layer. This protocol is derived from Chandy-Lamport algorithm.

1.3.2 Condor

The goal of the Condor Project [33] is to develop, implement, deploy, and evaluate mechanisms and policies that support High Throughput Computing (HTC) on large collections of distributively owned computing resources. Condor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, Condor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.

Condor is able to execute and schedule different types of applications (e.g. sequential, PVM, MPI, Java, etc.) and for PVM they introduced the CoCheck protocol in 1996. Later it became unsupported, so the Condor versions in the last few years changed direction regarding PVM checkpointing.

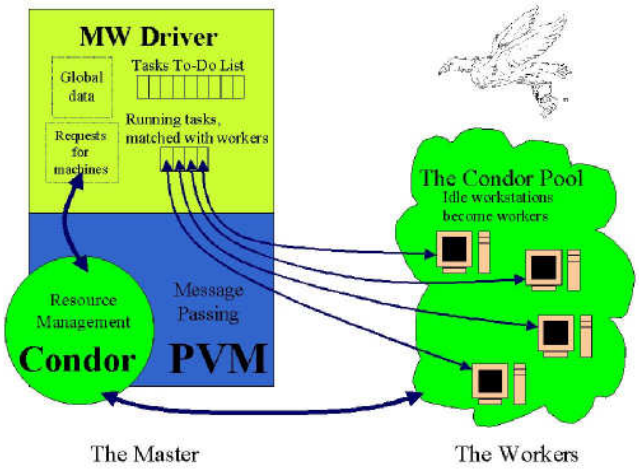

Figure 4 View of the Condor MW paradigm

The new model they introduced is a fault-tolerant execution for Master-Worker (MW) type applications [73]. For Condor, there is a tool for making a master-worker style application that works in the distributed, opportunistic environment of Condor. MW applications use Condor as a resource management tool, and can use either Condor-PVM or MW-File a file-based, remote I/O scheme for message passing. The user must define the code for the Master process, the Worker processes and the system will distribute the works automatically among the workers in a way defined by the programmer. When a worker process aborts or fails, Condor automatically spawns a new worker with the same workpackage, the failed worker owned.

In the Condor job-scheduler tool there is no real checkpointing of parallel application, but fault-tolerant execution of PVM applications are supported in a limited way. However, Condor is able to checkpoint sequential jobs; therefore it is sometimes reused by parallel checkpointing tools, like CoCheck.

1.3.3 Fail-safe PVM

Fail-safe PVM [65] has been designed and implemented by the Carnegie Mellon University. The main purpose of the framework is to provide a fault-tolerant PVM environment regarding single-node failures.

Fail-safe PVM uses checkpoint and rollback to recover from such failures. Both checkpoints and rollbacks are transparent to the application. The system does not rely on shared stable storage and does not require modifications to the operating system. The main goals and advantages of this system are application independence, application transparency, compatibility and minimal overhead.

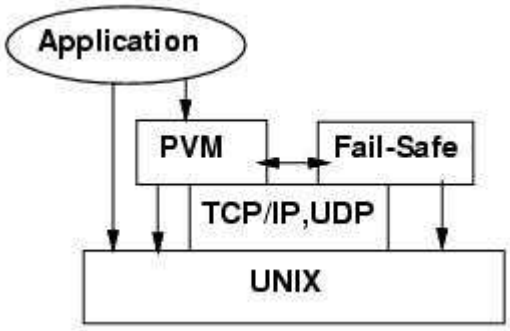

Figure 5 Architecture of Fail-safe PVM

Fail-safe PVM is an enhanced version of PVM, i.e. the design and implementation has been done in a way that PVM internals are modified in order to fulfil the goals.

The main strengths of the system is the capability to detect failed-nodes and to migrate application processes from the failed nodes to the failure-free ones. The PVM daemons are modified in order to synchronize among each other to prepare the application for checkpointing. At recovery the daemons are able to rollback themselves to create a consistent state of the parallel virtual machine. After the daemons are rolled back, they initiate the application to start recovering itself. To do this the application processes are reordered among the error-free PVM daemons to take the work from the failed one. The checkpoint information is distributed among the daemons. After the application has successfully restarted from the last consistent state, it is instructed to continue the execution.

As a summary, FPVM is strongly focusing on migration support of the PVM processes, where the migration is driven in a way to provide fault-recovery of the application.

1.3.4 Dynamite

Dynamite [66] aims to provide a complete integrated solution for dynamic load balancing of parallel jobs on networks of workstations. It contains an integrated load-balancing and checkpointing support for PVM applications.

Dynamite provides dynamic load-balancing for PVM applications running under Linux and Solaris. It supports migration of individual tasks between nodes in a manner transparent both to the application programmer and to the user, implemented entirely in user space. Dynamically linked executables are supported, as are tasks with open files and with direct PVM connections. In Dynamite, a monitor process is started on every node of the PVM virtual machine. This monitor communicates with the local PVM daemon and collects information on the resource usage and availability of the nodes.

The architecture of Dynamite is modular. It is possible to use just the dynamic loader of Dynamite and get checkpoint/restart facilities for sequential jobs that do not use PVM. Even when using PVM, it is not required to use the Dynamite monitor/scheduler: the user can migrate tasks manually from the PVM console (using the new move command) or from custom programs (using the new pvm_move function call). This gives Dynamite extra flexibility, and makes its components reusable for different projects.

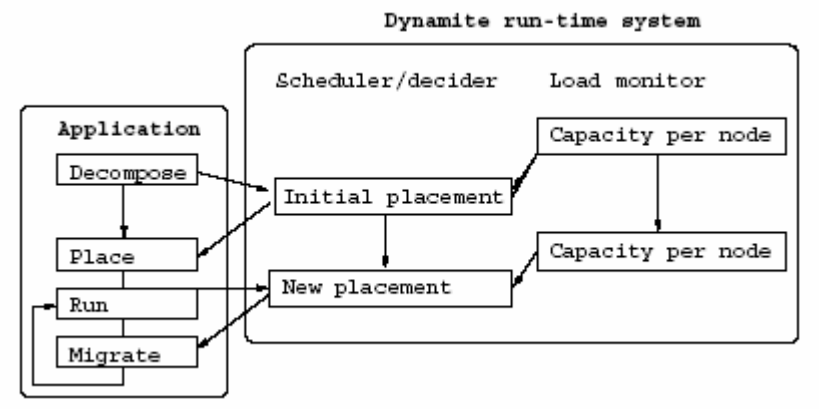

Figure 6 Architecture of the Dynamite PVM system

The system focuses on migrating individual PVM processes. To provide this capability, Dynamite replaces the dynamic loader and the whole PVM implementation for the daemons.

1.3.5 MPVM (MIST)

In MPVM [67], the interface between the PVM daemons (pvmd) and the Resource Manager has been extended to accommodate task migration, allowing the MPVM Resource Manager to use dynamic scheduling policies.

In order to support task migration, both the pvmd and PVM library (pvmlib) is modified. The modifications made were also driven by the goals of source code compatibility, portability, and migration transparency. To ensure source code compatibility, the modifications maintain the same function calls, parameters and semantics, as provided by PVM. To maximize portability, the migration mechanism is implemented at user-level, using facilities available through standard Unix library routines and system calls. Migration transparency is addressed by modifying the pvmd and pvmlib such that the migration could occur without notifying the application code and by providing ”wrapper” functions to certain system calls.

To avoid explicit modification of the source code of the PVM application for installing the signal handler, the (m)pvmlib defines its own main() function which executes the necessary initialization and then calls a function called Main(). A Migration Transparent Version of PVM linked with the pvmlib, the resulting executable will have the pvmlib's main() as the entry point, allowing execution of the migration initialization code prior to the execution of the application's code.

An important component of the migration protocol is what is collectively called Control Messages. These control messages or CMs are special system messages added to the pvmds and the pvmlib for the primary purpose of managing task

migration. Just like other system messages, these control messages are invisible to the application code.

The medium for the state transfer is TCP connection between the source pvm task and the destination migrated task. MPVM uses the approach of transferring the entire virtual address space of a process at migration time. That is mainly the reason why MPVM does not support fault-tolerance.

The protocol implemented in MPVM is designed in way that after the migration the task has the same task identifier, so translation tables are not needed. However, there is a need for routing tables in the PVM daemons.

1.3.6 tmPVM

The tmPVM [68] system aims to provide efficient migration of PVM tasks. It does not create application wide checkpointing, i.e. it does not support fault-tolerant, the application does not have a stored consistent global state.

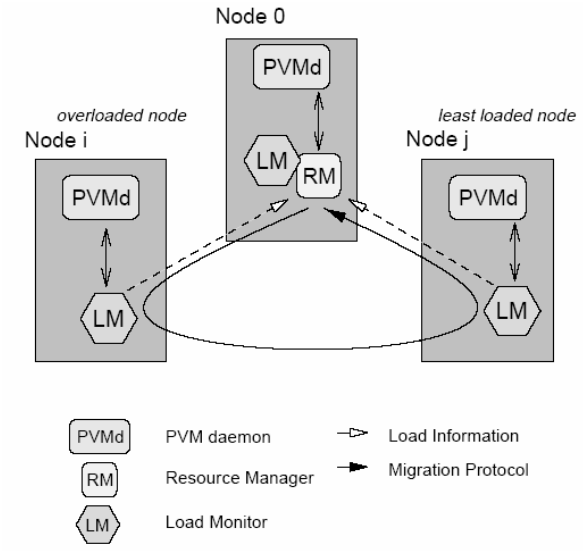

Figure 7 The tmPVM virtual machine

A task to be migrated creates a checkpoint file by spawning an external extractor to scan the /proc file system. This task waits until the task is restarted on the destination node, and in the mean time forwards any messages that it receives to the destination (new) instance. The destination instance has a new PVM task identifier after the migration, so a special TID alias directory (mapping table) is maintained in the PVM daemons to translate between the original and current task identifier.

A load monitor module is spawned on each node to collect local load information. In tmpPVM, a centralized resource manager gathers and maintains local information for each participating node. All load monitor modules are assumed to be active till the shutdown of the tmPVM virtual machine. Users are required to include an initialisation call (named TCP_TM_init()) before any statement. The routine will install the necessary mechanism, data structures and notification to the load monitor modules.

1.3.7 DAMPVM

DAMPVM [69] (Dynamic Allocation and Migration Parallel Virtual Machine) is an extension to the PVM environment. DAMPVM traces the changes of some

parameters on the machines in the virtual machine like nodes loads, other users activities, measures the speeds of the nodes to get to know how much processing power is available on the nodes.

The programs can be written analogously to PVM ones but the programmer must use the library supported by DAMPVM instead of PVM. This library offers communication and creation functions analogous to the PVM ones plus some extra connected with process instrumentation. DAMPVM kernels (one on each machine) start processes on the appropriate machines. DAMPVM is a regular PVM application so there are processes - kernels (one on each machine) and additionally the DAMPVM library.

The goal of DAMPVM is to minimize the total execution time of the whole application (all the processes started by the user of DAMPVM) - DAMPVM performs dynamic allocation of processes. Heterogeneous migration is possible because it is performed on the code level. A programmer must support two functions for packing and unpacking the state of a process to enable DAMPVM kernels to migrate a process. Dynamic allocation and migration is performed automatically by DAMPVM.

1.3.8 Charm

Charm [70] is a checkpoint/restart system, where PVM calls are wrapped in order to modify the behaviour of the calls and to support checkpointing. To preserve communication it uses a similar version independent protocol to the one defined in CoCheck, but it is optimised for faster migration, since message flushing is only performed for migrating tasks. The messages sent to the migrating processes are stored in a delaying buffer and when the migration has finished the content of the delaying buffer is forwarded to the new destination. The key advantage of this system is that no modification of PVM is performed. The system has a C-manager that is responsible for performing the proper protocol among the processes of the application.

C-manager is implemented as a PVM task. Users start the charm system (cmanager), then cmanager spawns the user’s PVM task. It also reads in the checkpointing-related .rc files. By this technique the application will know, the Task ID of the C-manager and will be able to cooperate with it in order to perform checkpointing.

With the slight modification of the start-up mechanism of the PVM application, Charm is able to migrate PVM processes. At the same time this system does not provide application wide checkpoint, i.e. it is not possible to checkpoint the whole application, shutdown all the running component and to restart the checkpointed application on a different cluster.

1.3.9 CLIP

CLIP [74] (Checkpoint Libraries for Intel Paragon) is a semi-transparent checkpointer for the Intel Paragon. CLIP can checkpoint programs written in either NX or MPI. The conceptual structure of CLIP is depicted in Figure 8. In checkpointing mode, CLIP code acts as middleware between the user's application and the Paragon libraries.

To use CLIP, a user must link the CLIP library with the application program. Additionally, the user must place one or more subroutine calls in the code specifying when checkpoints can occur. When the code is executed, CLIP takes periodic checkpoints of the program's state to disk. If for some reason (hardware or software

failure) the program is terminated prematurely, it may be restarted from the checkpoint file.

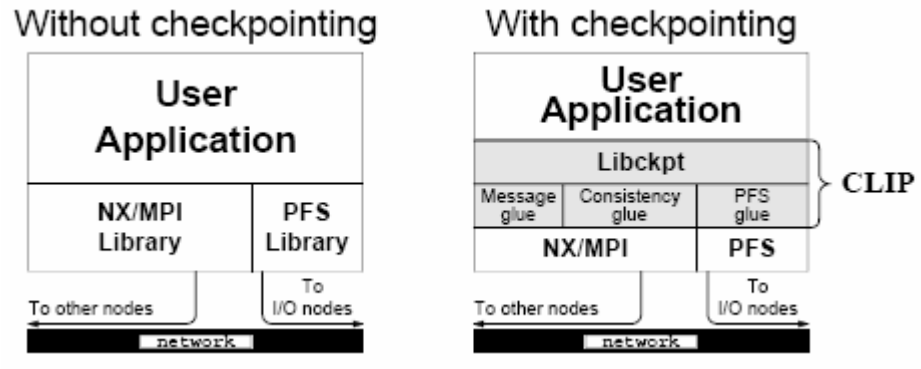

Figure 8 Structure of CLIP

1.3.10 ZapC

ZapC [75] is a novel system for transparent coordinated checkpoint-restart of distributed network applications on commodity clusters. ZapC provides a thin virtualization layer on top of the operating system that decouples a distributed application from dependencies on the cluster nodes on which it is executing.

This decoupling is done by introducing a PrOcess Domain (POD) abstraction. POD enables ZapC to checkpoint an entire distributed application across all nodes in a coordinated manner such that it can be restarted from the checkpoint on a different set of cluster nodes at a later time.

ZapC is designed to support migration of unmodified legacy applications while minimizing changes to existing operating systems. This is done by leveraging loadable kernel module functionality in commodity operating systems that allows ZapC to intercept system calls as needed for virtualization.

1.3.11 LAM/MPI

LAM/MPI [76] is a high performance implementation of the Message Passing Interface (MPI) standard. LAM/MPI provides a System Services Interface (SSI), a modular component system that allows easy extensibility to new environments. One of these services is designed to provide a transparent checkpointing and rollback recovery (CRR) with high portability.

The CRR framework implements most management of coordinated checkpointing and rollback recovery for MPI parallel applications. MPI applications running under LAM/MPI can be checkpointed to disk and restarted at a later time. LAM requires a 3rd party single-process checkpoint/restart toolkit for actually checkpointing and restarting a single MPI process - LAM takes care of the parallel coordination.

Currently, the Berkeley Labs Checkpoint/Restart package (BLCR) [77] is supported which is a kernel-level single-process-CRR tool for Linux. It works as a dynamically loadable kernel module, so it is not a universal tool for different Linux versions and platforms.

LAM/MPI allows new back-end checkpointing systems to be "plugged-in" simply by providing a new CR SSI module. An example to replace kernel-level

checkpointing of LAM/MPI with user-level checkpoint and recovery is introduced by [78] where BLCR is replaced with libcsm. New CR modules are designed and the tool is modified to provide the needed APIs. Because LAM/MPI supposes checkpointing is thread-based and implements the framework on this assumption, its workflow had to be modified to integrate the library.

1.3.12 Charm4MPI

ChaRM4MPI [84] is a Checkpoint-based Rollback Recovery (CRR) and Migration System for Message Passing Interface, specially designed and implemented for Linux Clusters. It is based on coordinated checkpointing protocol, synchronized rollback recovery to provide process migration.

In ChaRM4MPI users can migrate MPI processes manually from one node to another for load balance or system maintenance. This tool is implemented in a user-transparent way and uses the techniques employed by libckpt [31] and CoCheck [72]. The whole implementation is based on MPICHP4 [79], which is an MPI implementation developed at Mississippi State University and Argonne National Laboratories that employs P4 [79] library as the device layer.

Regarding the architecture, one management process, called manager, is implemented as the coordinator of parallel CRR and process migration. It can operate users' checkpoint or migration commands received through a GUI. The source code of P4 listener is altered to perform signal interruption of computing processes in case commands received from the coordinator. The start-up procedure of MPICHP4 is modified, so all MPI processes will register themselves to the coordinator. In this way, the latter can maintain a global process-info-table to manage the whole system.

1.3.13 FT-MPI

The aim of FT-MPI [80] is to build a fault tolerant MPI implementation that can survive failures. The system offers a wide range of recovery options for the application developer other than just returning to some previous checkpoints. FT-MPI is built on the HARNESS [81] meta-computing system.

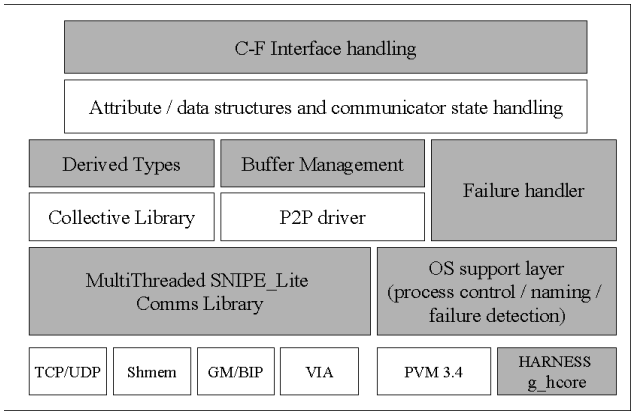

Figure 9 Structure of the FT-MPI implementation

FT-MPI extends the MPI communicator states with additional states (e.g. ok, detected, recover, recovered, failed). A communicator changes its state when either an MPI process changes its state, or a communication within that communicator fails for some reason. By allowing the communicator to be in an intermediate state the application has the ability to decide how to alter the communicator and its behaviour.

The current implementation is built as a number of layers and integrates a wide variety of underlying services to provide fault-tolerance (see Figure 9). Different services provide FT-MPI with notification of failures from communications libraries as well as the OS support layer.

Since FT-MPI provides fault-tolerance by reporting failures through e.g. modification of communicator states, the responsibility to handle the undesirable situations is still remains a duty of the application programmer. The typical usage of FT-MPI would be in the form of an error check and then some corrective action such as rebuilding a communicator and so on.

1.3.14 Starfish MPI

Starfish [82] MPI is a daemon based implementation approach for MPI. Each host of a cluster must have a Starfish daemon (see Figure 10) forming a parallel computer over the cluster. Application processes can only be migrated within the so-called parallel computer i.e. where a starfish daemon is running.

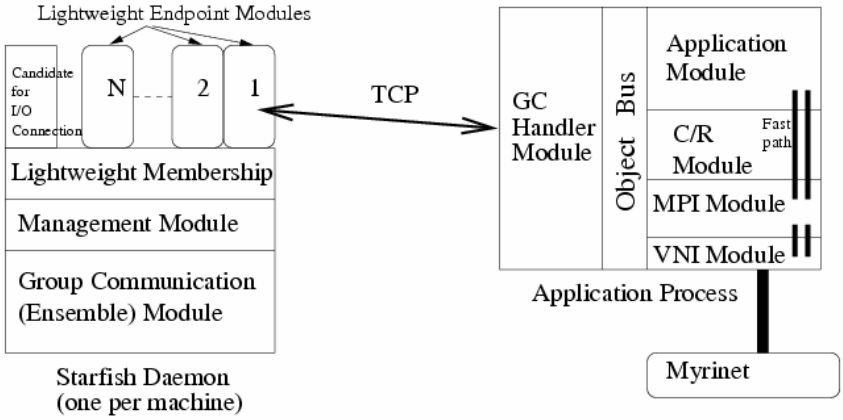

Figure 10 Architecture of the Starfish MPI

Starfish uses its own distributed system to provide built in checkpointing. Starfish MPI handles communication and state changes which are built upon the Ensemble system [83] and managed by strict atomic group communication protocols among the daemons.

2 New checkpointing method on ClusterGrid

2.1 Transparency conditions in ClusterGrid environments

2.1.1 Overview

Checkpointing of message-passing based parallel algorithms or applications can be realised using various techniques and methods. A concrete solution must always face the requirements imposed by the Grid middleware. The goal of the first group of thesis is to identify the requirements and conditions towards the checkpoint and migration techniques imposed by the ClusterGrid environment and to elaborate a new (abstract) technique which fulfils the already identified requirements.

As a preparation of thesis 1.1, I have defined the main characteristics of the ClusterGrid environments and several desirable use cases for state migration mechanism. As a next step, I have identified the cluster components which may influence the internal operation of a checkpointing system and then I determined the requirements in 4+1 points. By using the ASM formal framework for modelling I have developed the model of the cluster components and their most relevant characteristics, and then based on this model I have elaborated a formal description of the requirements and conditions for checkpointing techniques. Finally, based on these criteria I have analysed, evaluated and classified the existing solutions and stated thesis 1.1.

> **Thesis 1.1:** *In a cluster environment a formal framework of requirements can be defined for message-passing parallel algorithms, which enables transparent checkpointing of the algorithms for the scenarios defined in the dissertation. In addition, currently there are no checkpointing and migration facilities fulfilling the defined requirements in a cluster environment for the defined scenarios.*

Related publications are [3][4][19].

2.1.2 Identification of the components

Based on the abstraction levels and coordination types (both defined in Section 1.2), checkpointing of a parallel application can be implemented in several ways. In this section the most relevant components are identified which contribute to the whole checkpoint and restart procedure. The following (logical) components are identified as possible contributors to the checkpoint or restart procedure (see Figure 11) in a cluster environment:

1. Operating system (“OS”)
2. Scheduler (“sched”)
3. Source code of the application (“src”)
4. Linked libraries of the application (“lib”)
5. Message-passing layer (“message-passing system”)
6. External coordination process (“Coord.”)

An application is shown (see Figure 11) with three processes ("ProcA", "ProcB", "ProcC") running on three nodes of a cluster. Processes - from logical point of view - contain user written code ("src") and libraries linked ("lib") to the code of the processes. The distinction is introduced to distinguish between library or application level checkpointing.

In case of application level checkpointing the *source code* (depicted as "src" in Figure 11) must be prepared by the programmer as it has been detailed in section 1.2.

If automatic library (system) level checkpointing is supported, a special *library* (depicted as "lib" in Figure 11) is linked to the process. The library is usually activated during execution; it scans through all the memory space of the process and saves the information into a temporary storage for later usage. Saving results into a binary file includes all memory segments, registers, signals and so on.

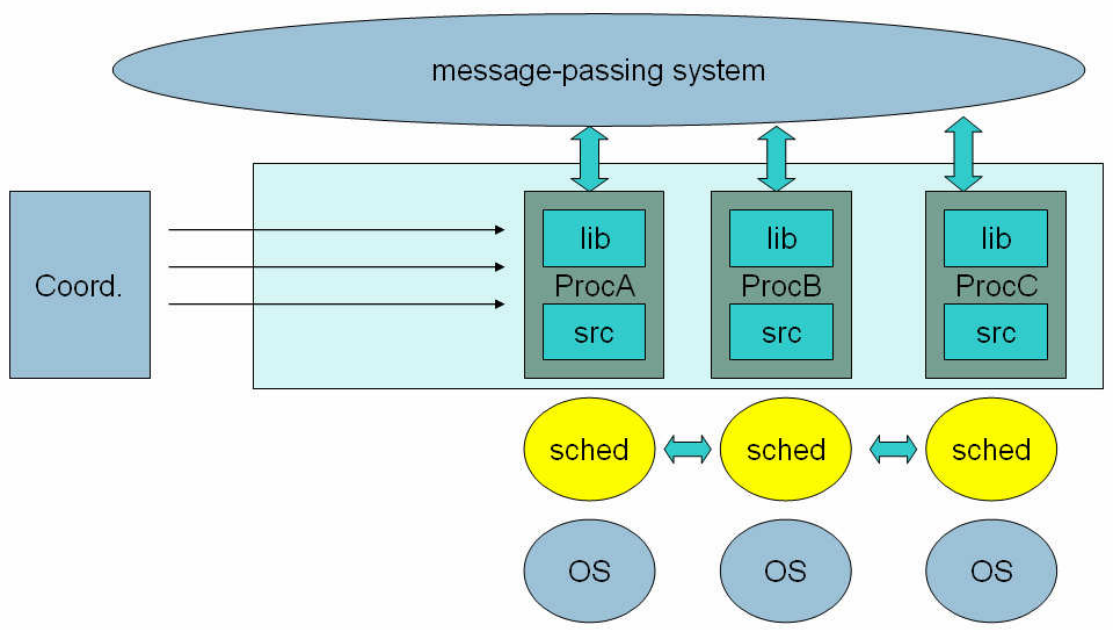

Figure 11 Components of a checkpointing environment

The *message-passing system* (MP) (see Figure 11) is running in the background to transport messages among the processes. In certain solutions (e.g. Fail-safe PVM [65], Dynamite [66], MPVM [67], etc.) this layer is modified to support checkpointing of the messages and of the links among processes. This support usually means cooperation with the application to be prepared for checkpointing in a way that consistency of in-transit messages and connections are preserved.

In order to facilitate the *coordination* of checkpoint among the processes of the application an external checkpoint coordination process (depicted as "Coord." in Figure 11) can be connected to each process of the application. This auxiliary process usually performs coordination of checkpoint saving, exchange of checkpoint related information or identifiers among the application processes. Furthermore, it can manage open files and play a central role in the resumption procedure, too.

Checkpoint support in schedulers can be an integrated part. Most of the schedulers for clusters (depicted as "sched" in Figure 11) perform checkpointing up to some extent (e.g. Dynamite [66]). Checkpointing of an individual process is usually fully supported by these schedulers, which means checkpoint/restart mechanism is integrated. The Scheduler decides (usually based on load of the nodes) when checkpoint must be carried out and shutdown a process.

Last, but not least another component where checkpoint support might be implemented is the *operating system* (depicted as "OS" in Figure 11) in case of kernel-level checkpointing as it has been described in section 1.2.

To summarize, the list of components/modules/parts defined above represents potential/possible modification points that existing checkpoint frameworks usually exploit. Based on the type of checkpointing, different combination of the components are added, replaced or modified. As a consequence, depending on the components modified by the checkpointing system, different levels of compatibility can be reached among the clusters in case of a ClusterGrid.

2.1.3 Use cases

This section is aimed at defining some of the use cases that are typical in a ClusterGrid system concerning a job life-cycle including submission, fault-related events, checkpointing and migration. Before defining use cases it is necessary to determine some basic key features of the Grid system and the critical characteristics of the jobs to be submitted in order to understand the potential behaviour of the entire system.

In the current context the execution of the submitted application is realised in a Clustergrid containing numerous clusters independent of each other. The main expectations towards these clusters are defined as follows:

Assumptions for the proposed ClusterGrid (*AsmpCls*):

AsmpCls 1. Each node of every cluster in the Grid is compatible at OS level to each other. It is necessary that the binary of the submitted application can be executed on any node of any cluster. In a real environment it is not always true. In that case we consider the biggest subset of the Grid consisting OS compatible nodes as the ClusterGrid environment.

AsmpCls 2. Each node of every cluster must have the same type of message-passing system installed, i.e. each must have MPI or PVM or both.

AsmpCls 3. The scheduler is able to allocate a process of the application to any node of the supervised cluster.

AsmpCls 4. Every node (except the front-end node) in a cluster are identical regarding software environment.

Assumptions for the application (*AsmpApp*):

AsmpApp 1. The application uses PVM or MPI as message-passing layer.

AsmpApp 2. The physical requirements (memory size, disk capacity, etc.) of the application fit to the capabilities of the clusters and nodes.

AsmpApp 3. The application is authorised to be executed on any of the clusters in the Grid. In a real environment it is not always true. In that case we consider the biggest subset of the ClusterGrid the application is authorised nodes as the ClusterGrid environment.

Points (degree) of freedom for the proposed ClusterGrid (*FreeClu*):

FreeClu 1. Clusters of the Grid are not expected to have a specific checkpointing service installed on their nodes. Decision on installed services is made by the local policy of the cluster.

FreeClu 2. Clusters of the Grid are not expected to have installed the same version of the message-passing software, but still must accomplish the assumptions defined in *AsmpCls2* and *AsmpCls3*.

FreeClu 3. Clusters of the Grid are not expected to operate a predefined scheduler. Decision on scheduler to be used on a cluster is made by the local policy.

FreeClu 4. Clusters of the Grid are not expected to apply a predefined OS version installed on their nodes, but still must accomplish the assumptions defined in *AsmpCls1*.

FreeClu 5. Clusters of the Grid are not expected to perform OS or kernel related updates to support checkpointing.

FreeClu 6. Clusters of the Grid are not expected to assign all of their nodes to one application. A cluster might serve for execution of multiple (parallel) applications at the same time. Decision on scheduler rules to be applied on a cluster is made by the local policy.

The proposed assumptions (AsmpClu, AsmpApp) and the points of freedom (FreeClu) in the middleware of the clusters determine the potential behaviour and interaction among the clusters in the entire ClusterGrid.

Several scenarios are now defined to be supported during the execution of application performed by the middleware of cluster.

Scenarios for process and application migration:

Scenario 1. Immediate process migration to free resources

This scenario defines a simple case when one of the processes of the application (job) migrates from its source node to another (target) one. The reason is that the source node is about to be pre-empted by the local scheduler.

1.1. User submits job containing k processes
1.2. Job is scheduled to Cluster A having n nodes where n>k
1.3. Job is executed on Cluster A on nodes 1..k and nodes k+1..n are free
1.4. Scheduler initiates termination of process running on node i, where $1 \leq i \leq k$
1.5. Scheduler assigns new node j for the job, where $k < j \leq n$
1.6. Process from node i is migrated to node j
1.7. Job continues execution
1.8. Job finishes
1.9. User gets results

Scenario 2. Delayed process migration to free resources

This scenario defines a simple case when one of the processes of the application is removed from its node, but the migration cannot be realised due to unavailable free resource.

2.1. User submits job containing k processes
2.2. Job is scheduled to Cluster A having n nodes where n>k
2.3. Job is executed on Cluster A at nodes 1..k and nodes k+1..n are reserved
2.4. Scheduler initiates termination of process running on node i, where $1 \leq i \leq k$
2.5. No resource is available, process is waiting for free node
2.6. Node j becomes free, where $k < j \leq n$
2.7. Scheduler assigns node j for the job
2.8. Process from node i is recovered on node j
2.9. Job continues execution

2.10. Job finishes
2.11. User gets results

Scenario 3. Application migration among clusters

This scenario defines a case when the whole application is pre-empted by the scheduler to vacate all the nodes of the cluster due to the request originated from the Central Broker of the ClusterGrid. Central Broker is the component that manages a queue of jobs submitted to the Grid and assigns the jobs to the clusters for execution.

3.1. User submits job containing k processes
3.2. Job is scheduled to Cluster A having n nodes where $n \geq k$
3.3. Job is executed on Cluster A on nodes 1..k
3.4. Central broker initiates pre-emption of the entire application through the local scheduler
3.5. Scheduler initiates vacation of all the nodes the application is using
3.6. Application is removed
3.7. Central broker reschedules the job to Cluster B having at least k available free nodes
3.8. Application processes are recovered on Cluster B
3.9. Application continues execution
3.10. Job finishes
3.11. User gets results

Scenario 4. Process and application migration among clusters

This scenario defines a simple case when some of the processes of the application are removed from their nodes, but the migration cannot be realised due to unavailable free resources. Job removal is initiated by the local scheduler.

4.1. User submits job containing k processes
4.2. Job is scheduled to Cluster A
4.3. Job is executed on Cluster A
4.4. Scheduler initiates vacation of processes on some of the nodes of the cluster
4.5. Processes are removed and application is waiting for free resources
4.6. All the remaining nodes in Cluster A are reserved, therefore after a timeout the scheduler initiates removal of the job from the cluster
4.7. Central Broker reschedules the job to Cluster B
4.8. Application is recovered on Cluster B
4.9. Job continues execution
4.10. Job finishes
4.11. User gets results

Scenario 5. Application initiated application migration among clusters

This scenario defines a simple case when some of the processes of the application are removed from their nodes, but the migration cannot be realised due to unavailable free resources. Job removal is initiated by the application itself.

5.1. User submits job
5.2. Job is scheduled to Cluster A
5.3. Job is executed on Cluster A
5.4. Scheduler initiates removal of some of the processes
5.5. Processes are removed and application is waiting for free resources

5.6. After a timeout the application initiates self termination (this point is the main difference comparing to Scenario 4)
5.7. Central broker realises the preemption of job and reschedules it to Cluster B
5.8. Application processes are recovered on Cluster B
5.9. Job continues execution
5.10. Job finishes
5.11. User gets results

As the various scenarios show, basically two different types of migration are required in order for a job to perform its calculation in a Grid system. Without checkpointing and/or migration the whole application should be re-executed from its initial point.

In the following sections the focus is on the middleware components. Detailed analysis reveals the requirements of the middleware.

2.1.4 Requirements in ClusterGrid

In case of ClusterGrid infrastructure where clusters can have different software components installed, the relevant design goals or requirements of a parallel checkpoint tool are *compatibility* (with the surrounding software components) and *integrity* (of the checkpoint information of the application). While the first goal ensures the seamless operation of checkpointer on clusters with various middleware components, the second one is a basis for application migration among clusters.

In order to fulfil the compatibility requirement the following conditions must be taken into consideration:

Condition 1. Operating system does not provide checkpointing facility

Condition 2. Solution does not rely on checkpoint support of the job manager

Condition 3. Solution relies on the native version of message-passing system

Condition 4. Dependence from external auxiliary process does not exist

The four conditions correspond to compatible operation of checkpointing frameworks. Following these conditions an application can be checkpointed in software heterogeneous ClusterGrid environment, i.e. under the control of any cluster environment (consist of the components defined in Section 2.1.2).

Checkpointing can support fault-tolerance (restoration from a previous checkpoint is done due to middleware error causing application abort) and migration (restoration is done on another cluster) of a parallel application. Migration and fault-tolerance requires different support from the tools, since migration itself does not require checkpointing the state of the entire application (e.g. process migration) while supporting fault-tolerance requires.

Migration facilities in some of the cases temporarily store checkpoint information (e.g. in memory) about the checkpointed/migrating processes only. Since in a ClusterGrid infrastructure application may not be allowed to reach nodes that are part of another cluster, this type of migration technique would fail.

Therefore only those checkpointing and migration techniques are usable that creates a correct set of checkpoints of the entire application. It is called the integrity of checkpoints.

Based on the previous explanation a new requirement must be defined as a complementary one to the previous four. It is called integrity requirement. In order to fulfil it the following condition must be satisfied:

Condition 5. Application-wide (included all processes) checkpoint saving is performed

The five conditions together form a framework in which a checkpoint tool must fit in order to provide parallel application checkpoint/restart support on a general ClusterGrid infrastructure. By accomplishing these conditions applications can be migrated among the different nodes of a cluster and among the clusters.

2.1.5 Formal definition of the requirements

2.1.5.1 Abstract State Machines

Abstract State Machines represent a mathematically well founded framework for system design and analysis [36] [38] introduced by Gurevich as evolving algebras [39]. The motivation for defining such a method is quite similar to that of Turing machines (TM). However, while TMs are aimed at formalizing the notion of computable functions, ASMs are for the notion of (sequential) algorithms [40]. Furthermore, TMs can be considered as a fixed, extremely low level of abstraction essentially working on bits, whereas ASMs exhibit a great flexibility in supporting any degree of abstraction.

In every state based systems the computational procedure is realized by transitions among states. In contrast with other systems, an ASM state is not a single entity or a set of values but ASMs states are represented as (modified) logician's structures, i.e. basic sets (universes) with functions and relations interpreted on them. Experience showed that any kind of static mathematic reality can be represented as a first-order structure [40]. These structures are modified in ASM so that dynamics is added to them in a sense that they can be transformed.

Applying a step of ASM **M** to state (structure) *A* will produce another state *A'* on the same set of function names. If the function names and arities are fixed, the only way of transforming a structure is changing the value of some functions for some arguments. The transformation can depend on some condition. Therefore, the most general structure transformation (ASM rule) is a guarded destructive assignment to functions at given arguments [36].

ASMs are especially good at three levels of system design [36]. First, they help elaborating a ground model at an arbitrary level of abstraction that sufficiently rigorous yet easy to understand, defines the system features semantically and independent of further design or implementation decisions. Then the ground model can be refined towards implementation, possibly through several intermediate models in a controlled way. Third, they help to separate system components. ASM is not a paper theory but it has been applied in various industrial and scientific projects like verification of Prolog [41] and Occam [42] compilers, Java virtual machine [43], PVM specification [44], ISO Prolog standardization, validating various security and authentication protocols, VLSI circuits, and many more. The definition of ASMs is written in [39] and [45] and a tutorial can be found in [46]. A short overview is here as follows.

A vocabulary (or signature) is a finite set of function names, each of fixed arity furthermore, the symbols *true, false, undef*, =, the usual Boolean operators and the unary function Bool. A state *A* of vocabulary Ψ is a nonempty set *X* together with interpretations of function names in Ψ on *X*. *X* is called the super-universe of *A*. An r-ary function name is interpreted as a function from *X^r* to *X*, a basic function of *A*. A 0-ary function name is interpreted as an element of *X*.

In some situations the state can be viewed as a kind of memory. Some applications may require additional space during their run therefore, the *reserve* of a state is the (infinite) source where new elements can be imported inside the state.

A location of *A* (can be seen like the address of a memory cell) is a pair $l=(f,\mathbf{a})$, where *f* is a function name of arity *r* in vocabulary Ψ and **a** is an r-tuple of elements of *X*. The element $f(\mathbf{a})$ is the content of location *l*.

An update is a pair $a=(l,b)$, where *l* is a location and *b* is an element of *X*. Firing *a* at state *A* means putting *b* into the location *l* while other locations remain intact. The resulting state is the sequel of *A*. It means that the interpretation of a function *f* at argument *a* has been modified resulting in a new state. This is how transition among states can be realized. An update set is simply a set of consistent updates that can be executed simultaneously.

2.1.5.2 Basic universes, functions and relations

An application (element of universe APPLICATION) is comprised of several processes (elements of universe PROCESS) that cooperate in some way. Their relationship is represented by the function *app: PROCESS → APPLICATION* that identifies the application the process belongs to. Processes are running on nodes (elements of universe NODE) which belong to a cluster (universe CLUSTER). Their relationship is represented by the function *cluster: NODE → CLUSTER*. However, processes might exist that do not belong to any application on the cluster (*app* is evaluated to *undefined*).

A process is logically divided into user defined code (universe CODE) and libraries linked to the code (universe LIBS). Each PROCESS element has a corresponding element from CODE and LIBS universes. The relationships are defined by functions *code: PROCESS → CODE* and *libs: PROCESS → LIBS*.

A node can have: an operating system (universe OS), a scheduler logical component (universe SCHED) and a service for handling message-passing activities of any process running on the node (universe MPE). The relationships are defined by the functions *os: NODE → OS, sched: NODE → SCHED* and *mpe: NODE → MPE*. The assignment of any components and nodes are defined by the function *mapped: {PROCESS, OS, SCHED, MPE} → NODE*.

In the checkpointing and resumption activities basically two different features are distinguished. A component (e.g. OS, SCHED, MPE, CODE, LIBS) may perform the checkpoint saving and resumption activity or may perform checkpoint coordination. Relationship is represented by the function *ckpt: {CODE, LIBS, SCHED, MPE, OS} → {true, false}*. The existence of the coordination activity is expressed by the relation *coord: {CODE, LIBS, SCHED, MPE, OS} → {true, false}*.

Every application has its momentary internal state (universe STATE) which represents a running application. Result of a checkpoint activity is one or more image (universe IMAGE) belonging to a PROCESS. This relationship is represented by the

function *image: PROCESS → IMAGE*, and multiple images are represented by the universe $IMAGES \in IMAGE^N$.

Checkpointing of an application is basically depending on the checkpointing technique the actual cluster is applying. So, the checkpointing activity maps the state of an application on a specific cluster to a certain image set. Resume is doing the same to the opposite direction. Accordingly, there is the function called *checkpoint: {CLUSTER x STATE} → IMAGES* and the other called *resume: {CLUSTER x IMAGES} → STATE*.

2.1.5.3 Formal definition of the requirements

Based on the representation of the logical components taking part in a checkpointing procedure, this section is aimed at giving a formal description of the requirements defined in section 2.1.4. The following description uses the symbols defined in section 2.1.5.

Condition 1. Operating system does not provide checkpointing facility

$$\forall C \in CLUSTER, \forall n \in NODE, cluster(n) = C : ckpt(os(n)) = false$$

The expression denotes that no checkpoint support exists in the operating system of each node n belonging to a given cluster C.

Condition 2. Solution does not rely on checkpoint support of the job manager

$$\forall C \in CLUSTER, \forall n \in NODE, cluster(n) = C : ckpt(sched(n)) = false$$

The expression denotes that no checkpoint support exists in the scheduler (or job manager) running on each node n belonging to a given cluster C.

Condition 3. Solution relies on the native version of message-passing system

Native version means a version with no checkpoint support added.

$$\forall C \in CLUSTER, \forall n \in NODE, cluster(n) = C : ckpt(mpe(n)) = false$$

The expression denotes that no checkpoint support exists in the message passing system (or layer) running on each node n belonging to a given cluster C.

Condition 4. No dependence exists from external auxiliary processes

Assume we have an application (A) running on a cluster (C). We must ensure that any process (P) which does not belong to any application, but running on the cluster does not contain any checkpoint related functionality.

$$\forall C \in CLUSTER, \forall A \in APPLICATION,$$

$$\forall p \in PROCESS, app(p) \neq A, cluster(mapped(p)) = C :$$

$$ckpt(p) = false$$

Condition 5. Application-wide (including all processes) checkpoint saving is performed

The fifth condition ensures that all information for resuming all processes of the application is available. So let us define *m* as an image set belonging to an application on a given cluster (C) which contains images for each process of the application.

$$\forall C \in CLUSTER, \forall A \in APPLICATION, \forall p \in PROCESS, app(p) = A:$$
$$\exists m \in IMAGES, checkpoint(C, state(A)) = m, resume(C, m) = state(A) \Rightarrow$$
$$image(p) \subset m$$

2.1.6 Analysis of the requirements

2.1.6.1 Existing checkpointing techniques

Based on the five conditions defined in section 2.1.4 an analysis is carried out on the existing PVM checkpointing and migration systems. The goal of the analysis is to summarize the solutions used by the various tools and to check against the conditions defined in section 2.1.4 to examine the conformity of the tools.

In the list below the most commonly used programming techniques and methods are summarized used by existing PVM [1] checkpointing and migration systems. The identification of these techniques is necessary since they have a great impact whether they can be used in ClusterGrid or not for transparent. The programming techniques and methods listed below are the most common ones and at the same time unfortunately they cannot accomplish the conditions defined in section 2.1.4. They are as follows:

A. Replaced PVM Resource Manager

In the PVM environment the scheduling functionality belongs to a special process named Resource Manager (RM) [25]. This functionality can be dynamically reassigned to another PVM process, which then becomes the new RM coordinating the assignment of the newly spawned processes on the machines. Checkpoint solutions replacing the PVM RM may fail to coordinate more than one application at a time since only one RM is allowed to be assigned on a host under PVM. Another problem may occur when the scheduler of the cluster already resides in PVM as RM. In this case registering the checkpointer as RM leads to a conflict between the scheduler and the checkpointer components.

This checkpointing technique violates Condition 3 defined in section 2.1.4.

B. Modified PVM daemon

In the PVM environment daemons running on each node are forming the virtual machine. To provide checkpointing functionality for PVM application daemons are often patched to handle checkpointing of communication. Since these daemons (obviously) cannot be replaced dynamically, during the migration the application may be unable to resume in case the target cluster environment does not contain the daemon with the same patch or another version is installed. This checkpoint technique is not compatible with the different software environments in case special modification is required in one of the installed software components part of the middleware.

This checkpointing technique violates Condition 3 defined in section 2.1.4.

C. OS level modification

In order to provide checkpoint/restart functionality some tools offer a solution kernel level checkpointing. Since, the homogeneity of software environment of the

various clusters in a ClusterGrid cannot be defined as a requirement; this approach cannot fulfil the compatibility requirement.

This checkpointing technique violates Condition 1 defined in section 2.1.4.

D. Auxiliary process

There are solutions using external coordination process or daemon running in the background continuously on (usually the master node of) the clusters. These auxiliary processes may vary from cluster to cluster in a ClusterGrid infrastructure. The internal protocol and behaviour of the coordination processes are not compatible among the different checkpoint/restart tools, so the migrated application looses its checkpoint support in case the target cluster does not have the very same coordination process. The checkpointer tool must be prepared for different software environment in order to be compatible with the existing tools running on a cluster.

This checkpointing technique violates Condition 4 defined in section 2.1.4.

E. Partial checkpoint of the application

Checkpoint tools usually provide an optimized version of migrating a process of the application from one node to another within a cluster (e.g. tmPVM [68]) by storing the checkpoint information in memory. It is fast and uses only a low amount of resources on the nodes. When the migration is happening internal data are copied directly from the memory of the source process to the memory of the target one. This solution cannot be used in case the application must migrate from one cluster to another since connection does not exist among the worker nodes of different clusters. A parallel checkpoint/restart solution must take this fact into account.

This checkpointing technique violates Condition 5 defined in section 2.1.4.

2.1.6.2 Classification of related works

After introducing the most commonly used checkpointing techniques for PVM checkpoint/restart tools, Table 1 summarizes the aforementioned solutions used by the various existing checkpointing tools. Each technique represents a category signed by letters (A-E) used in section 2.1.6.1 and the techniques of the analysed tools are classified into these categories.

In Table 1 those techniques and methods are listed only those are relevant in the context of ClusterGrid. The listed techniques A-E (vertical columns of Table 1) make any checkpointing tool unable to fulfil the compatibility (column A-D) and integrity (column E) requirements for a ClusterGrid infrastructure. In the next paragraphs a summary of the tools are given using the forbidden techniques in their design or implementation.

CoCheck [64] is a research project that aims at providing Consistent Checkpointing for various parallel programming environments like PVM and MPI based on Chandy-Lamport [56] algorithm. The checkpoint/restart capability of this tool is lying on the replacement of the default PVM resource manager which belongs to category A and it is using a single checkpointing tool requiring the update of the process startup mechanism at OS level belonging to category C.

The goal of the Condor [32] Project is to develop, implement, deploy, and evaluate mechanisms and policies that support High Throughput Computing (HTC) on large collections of distributively owned computing resources. The model they follow is a fault-tolerant execution of Master-Worker (MW) [73] type applications therefore

Condor does not provide application-wide checkpointing which belongs to category E. Since Condor dynamically deploys PVM daemons through the Resource Manager functionality of PVM, the solution covers category A. In addition, fault-tolerant execution is limited to a programming framework and a fixed topology.

The main purpose of Fail-safe PVM [65] is detecting failed nodes and migrating application processes from the failed nodes to the error-free ones. To do this PVM daemons are modified which belongs to category B.

	Approaches to implement checkpointing/migration for PVM applications				
Tools / **Techniques**	**A. Replaced PVM Resource Manager**	**B. Modified PVM daemon**	**C. OS level modification**	**D. Auxiliary process**	**E. Partial checkpoint of the application**
	Violates condition 1-4.				***Violates cond. 5.***
CoCheck	✸		✸		
Condor	✸				✸
Fail-safe PVM		✸			
Dynamite		✸	✸	✸	✸
MPVM (MIST)		✸			
tmPVM		✸		✸	✸
DamPVM		✸		✸	✸
CHARM				✸	✸

Table 1 – Classification of existing PVM checkpointing tools

Dynamite [66] aims to provide a complete integrated solution for dynamic load balancing of parallel jobs on networks of workstations. It contains an integrated load-balancing and checkpointing support for PVM applications. The system focuses on migrating individual PVM processes. It does not provide application wide consistent checkpointing and it also uses a wide variety of techniques. Dynamite replaces the

dynamic loader of the kernel and the whole PVM implementation for the daemons. Background processes like monitor helps the system in its operation. This design makes the tool belong to category B, C, D and E as well.

In MPVM [67], the interface between the pvmds and the Resource Manager has been extended to accommodate task migration by modification of both PVM daemon and PVM library. This point in the design belongs to category B.

The tmPVM [68] system aims to provide efficient migration of PVM tasks therefore it does not perform application wide checkpointing. After analysing its architecture, it turns out that special background monitoring components and resource managers must be deployed in order to support tmPVM. As a summary, this tool belongs to category B, D and E.

DAMPVM [69] (Dynamic Allocation and Migration Parallel Virtual Machine) is an extension to the PVM environment. PVM library and the daemon are patched to provide migration of PVM processes (only). Programmer must update its source code to fit to DAMPVM requirements. Altogether, the tool with its architecture fits into category B, D and E.

CHARM [70] is a checkpoint/restart system where a special external process called C-manager is responsible for performing the proper protocol among the processes of the application to realise process migration. The technique used in this tool belonging to category D while supporting only process migration covers category E, too. Besides, the startup mechanism of the PVM application is also modified by using the C-manager.

Based on the analysis it can be seen that the introduced and examined PVM checkpointing tools are not ClusterGrid compliant in the way it is defined in section 2.1.4, i.e. each tool breaks at least one of the requirements, therefore they are not able to realise application and middleware transparent checkpointing with migration among the clusters.

Continuing the analysis of the parallel checkpointing tools of the MPI, different techniques can be identified causing dependencies on the Grid middleware components running on a cluster. Analogously to the PVM checkpointing, the design and implementation techniques violating some of the compatibility or integrity conditions defined in Section 2.1.4 can be found in the tools.

CLIP [74] (Checkpoint Libraries for Intel Paragon) is a semi-transparent checkpointer for the Intel Paragon. CLIP can checkpoint programs written in either NX or MPI. To use CLIP, the user must place one or more subroutine calls in the code specifying when checkpoints can occur. Inspecting its architecture and examining its internal behaviour shows that the existence of the native Intel/Paragon NX communication support is a requirement for checkpointing MPI application. The solution applied in this tool depends on a specific version of the MPI, nonetheless architecture of CLIP itself showing a layering structure.

The design and implementation of CLIP violates Condition 3 defined in Section 2.1.4 requiring the usage of the native i.e. checkpoint-free version of the message-passing communication service.

ZapC [75] is an application transparent coordinated checkpoint-restart of distributed network applications on commodity clusters. It provides a thin virtualization layer on top of the operating system that decouples a distributed

application from dependencies on the cluster nodes on which it is executing. This requires the installation of a loadable kernel of into the OS of the node. Modification of the kernel in this way unfortunately violates Condition 1 defined in Section 2.1.4.

LAM/MPI [76] is a high performance implementation of the Message Passing Interface (MPI) standard. It implements most management of coordinated checkpointing and rollback recovery for MPI parallel applications and it integrates the BLCR [77] kernel-level single-process checkpointer tool. Since LAMMPI integrates the coordination of checkpointing and applies a kernel-level checkpointer, it violates Condition 3 and 1 defined in Section 2.1.4.

ChaRM4MPI [84] is a Checkpoint-based Rollback Recovery (CRR) and Migration System for Message Passing Interface, specially designed and implemented for Linux Clusters. It is based on coordinated checkpointing protocol, synchronized rollback recovery to provide process migration. In this solution the startup mechanism is modified that makes checkpointing support dependent on this actual version of MPI implementation and an external so-called coordinator is required during execution to which the processes are connected. The implementation of this tool violates Condition 3 and 4 defined in Section 2.1.4.

FT-MPI [80] aims to build a fault tolerant MPI implementation that can survive failures and is built on the HARNESS [81] meta-computing system. In FT-MPI the MPI communicator states are updated i.e. additional return values are defined those are not even part of the MPI standard. With this update modification of the message-passing layer is realised, therefore Condition 3 defined in Section 2.1.4 is violated.

Starfish [82] MPI is a daemon based implementation approach for MPI. Application processes can only be migrated within the so-called parallel computer i.e. where a starfish daemon is running. Starfish uses its own distributed system to provide built in checkpointing. Based on these facts, Starfish MPI has its checkpointing system integrated, therefore no native version of the MPI is used i.e. Condition 3. defined in Section 2.1.4 is violated.

With this overview altogether 14 related works has been examined. The result of the examination is that currently no tool providing parallel checkpointing is able to satisfy the conditions defined in Section 2.1.4 which means transparent behaviour of the examined tools is not possible. However, simple use cases defined in Section 2.1.3 show the real need of parallel checkpointing tools focusing on middleware transparency.

2.2 The ClusterGrid checkpointing method

2.2.1 Overview

The list of requirements introduced in thesis 1.1 defines several criteria for transparent checkpointing. The various solutions proposed by the different checkpointing techniques form a design space. In order to designate my proposed solution I have defined the operational and architectural details of the desired transparent solution. As a result, the definition of the proposed solution is summarised by a seven point list and the definition has been suited into the ASM formal model (CP_{ground}) introduced previously in thesis 1.1. In addition, I have elaborated the necessary ASM rules to describe the internal operation mechanism. Finally, I have proved thesis 1.2 through introduction of abstract events.

> **Thesis 1.2:** *The newly elaborated checkpointing method defined by the ASM model called CP_{ground} implements transparent checkpointing both for the programmer and for the middleware at the same time.*
>
> Related publications are [3][4][5][19].

The solution defined by the CP_{ground} abstract model enables the saving and restoration of the consistent, global state of a message-passing based parallel application or algorithm in a transparent way.

2.2.2 Introduction of the ClusterGrid method

In this section the main cornerstones of a new parallel checkpointing method are introduced. The following seven key definitions form a method which meets the requirements defined in section 2.1.4. The following method creates a more tight solution range than it is allowed by the compatibility and integrity requirements.

The conditions defined in section 2.1.4 restrict the various checkpointing approaches to those implementing the whole checkpoint restart functionality inside the application. There are two different checkpointing levels which can be applied:

In *application-level* checkpointing applications can perform checkpointing by providing their own checkpointing code. Applying this level of checkpointing a solution can be designed which fulfils the requirements defined in section 2.1.4 since no dependence on auxiliary components exists. The disadvantage of this solution is that it requires a significant programming effort to be implemented while library level checkpointing is transparent for the programmer.

Library level checkpointing requires a special library linked to the application that performs the checkpoint and restart procedure. Using library level technique can also result in a solution providing compatible method for checkpointing.

Apart from the two abstraction levels one must distinguish two further aspects on how parallel processes are coordinated in case checkpointing performed: *coordinated* and *uncoordinated* (see section 1.2.2). Both directions can satisfy the conditions defined in section 2.1.4.

In the coordinated version (see section 1.2.2) a designated process controls the checkpoint saving procedure to ensure the consistency of messages among the processes in the application to avoid message loss or duplication. In uncoordinated version (see section 1.2.2) consistency is ensured at restart time. During execution checkpoints must be stored from time to time for each process without removing the

ones created previously. At restart checkpoints for each process are searched through and attempted to make a selection in a way that they form a consistent state for the application and that they represent the latest valid state. While the coordinated version forces the processes to synchronize, uncoordinated checkpointing gives freedom for the processes to create checkpoint at any time. In coordinated checkpointing one checkpoint per process is enough to perform a successful resumption of the application, while in uncoordinated version the likelihood of successful resumption can only be increased with raising the number of checkpoints per process. In extreme cases when consistency is not established the application must be started from the beginning.

After giving a short overview of the potential techniques, here is the proposed transparent ClusterGrid method summarized by seven key definitions:

Definition 1. Library level checkpointing technique is used

The proposed checkpointing method performs library level checkpointing. A library performing checkpoint and restart functionalities are linked to the application. Therefore, application carries its own checkpoint facilities in its executable code. Behaviour of the application is also modified to enable the linked checkpoint facility to catch and wrap system and message-passing calls in the application. Library level checkpointing technique is the only alternative to provide a transparent solution for the programmer and for the surrounding ClusterGrid environment at the same time.

Definition 2. Application wide checkpointing is performed

Each time a checkpoint is taken; all processes must take part in the checkpoint creation process. The generated checkpoint information must contain state information of every process of the application. It is required since the checkpoint itself must store enough information for the entire application to be resumed.

Definition 3. Checkpoint information is stored in files

Migration of an application between two clusters is converted to checkpoint, terminate, movement of working files, resubmit and resume steps. Checkpoint information must be stored into files in the working directory of the application in order to transfer the generated checkpoint information transparently to the target cluster in case of migration. There are checkpointing tools using storage techniques like temporary memory, or sending the checkpoint information into a socket (to a process being resumed) which are not possible among independent clusters. Transferring checkpoint information through a file server can also be an alternative, but it is not scalable, fault-tolerant and transparent for the middleware.

Definition 4. Parallel checkpointing technique is coordinated

The most problematic feature of uncoordinated checkpointing is the possible inconsistency of individual checkpoints. In case only the last set of checkpoints is stored, the likelihood of storing an inconsistent set is fairly big. In most cases resumption results in rolling the application state back to the initial point if one checkpoint per process is stored at a time. To store several checkpoints for each

process raises a significant overhead for the storage system and at the same time it still does not guarantee that consistency exists among one of the combinations of the checkpoints. To overcome this problem coordinated checkpointing techniques are used.

In message passing systems, earlier performance studies by Bhargava et al (1990) [60] showed that coordinated checkpointing algorithms are costlier because they incur extra communication. Later simulation studies by Elnozahy et all (1992) [61], however, revealed that coordinated algorithms are better than independent algorithms. The cost of coordination is much lower when compared with the cost of maintaining multiple checkpoints and/or logging messages.

Definition 5. Coordination process is part of the application

In order to provide a coordinated checkpoint of the application a coordination process is required. Originally, it can be an external one or the representative functionality can be part of the application. Since Condition 4 in section 2.1.4 prohibits applying external (auxiliary) processes, coordination functionality must be built in the application, which includes alternatives using separate and non-separate processes inside the applications.

Definition 6. Processes migrate within a cluster without terminating the application

In case a process needs to be migrated from one node to another within a cluster there are two alternatives. First one is that only the migrating process terminates and restarts while the rest of the application is suspended. Second one is that the whole application terminates after checkpointing which is then resubmitted to the cluster with different process-host mapping. We can see that both results in process migration. This definition states that the checkpointing solution must include the possibility of providing migration of processes the former way.

Definition 7. Application source code is unmodified

All the checkpoint techniques listed above must be provided in a transparent way for the programmer. The original application that has been created by the programmer must be made checkpointable in a way that the source code does not need to be changed.

Based on the proposed method described above the following scenario can be outlined. The programmer has an application developed in a certain programming environment. Without source code modification he/she adds checkpoint support to its application through recompilation or relinking of the application. The resulted executable with the required input files are submitted to a ClusterGrid as job. The application is assigned to a cluster by a broker, it is submitted and started. During execution local scheduler decides to deallocate some computational nodes because of some reasons (overload, maintenance, etc). Before any node is lost by the cluster, all jobs running on them are terminated gracefully. Checkpoint facility built in the application detects the start of the termination of some of its processes and initiates a checkpoint saving. It creates an application-wide consistent checkpoint represented as

working files in the application working directory. At this point two different continuations can happen.

(1) If some of the processes are terminated, the application tries to reallocate the terminated processes on different nodes. When nodes are available new processes are spawned that reload the checkpoints produced by the terminated processes. At the end of reloading a checkpoint, migration is successfully performed for a process. When all the processes are ready to run and all the necessary migration has finished the application continues its execution.

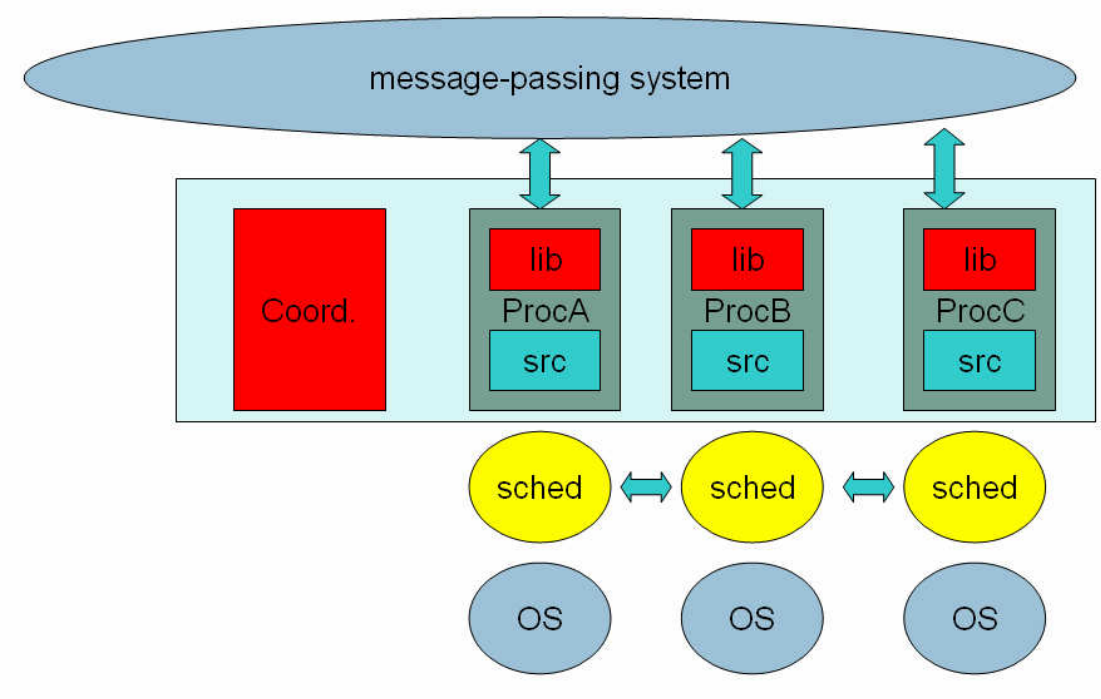

Figure 12 Structure of the proposed checkpointing technique

(2) In case all the processes are terminated the scheduler on the cluster detects a successful finish of the. At this point the central broker of ClusterGrid reassigns the job to another resource (cluster), transfers the executable and all working files (including checkpoints) to the target cluster and resubmits the job. When the application is being started the built-in checkpoint/restart facility detects the existence of checkpoints and performs resumption. After successful resumption of every process of the application execution continues. At successful finish of the application existing checkpoints are removed. Checkpoint saving procedure can also be initiated periodically.

As a summary, an architectural overview is shown in Figure 12. It can be seen that the application must be extended with a task performing checkpoint/restart functionality, but in a way that the application programmer is not forced to make modification in the application i.e. in a transparent way.

2.2.3 Formal definition of the ClusterGrid checkpointing method

In this section a formal ASM definition of the ClusterGrid checkpointing method (detailed in section 2.2.2) is introduced in order to give a precise description. The formal definitions below are based on the universes and functions, relations introduced already in section 2.1.5.

Definition 1. Library level checkpointing technique is used

Library level checkpointing is defined by the "ckpt" function, which returns true in case checkpoint related functionality belongs to the component passed as an input parameter.

$$\forall C \in CLUSTER, \forall A \in APPLICATION, \forall p \in PROCESS, app(p)=A:$$

$$ckpt(code(p)) = false \ \& \ ckpt(libs(p)) = true$$

$$\&$$

$$\forall n \in NODE, cluster(n)=C, \forall comp \in \{SCHED, MPE, OS\}, mapped(comp)=n:$$

$$ckpt(comp) = false \ \& \ coord(comp) = false$$

The goal of the formula is to prohibit checkpoint functionality in the user code, to ensure linked libraries to support it and finally to prohibit checkpoint functionality in any other component of the cluster.

Definition 2. Application wide checkpointing is performed

Let us assume that we have an image set, which stores the state of each process of the application.

$$\forall C \in CLUSTER, \forall A \in APPLICATION:$$

$$\forall p \in PROCESS, app(p)=A:$$

$$\exists m \in IMAGES:$$

$$image(p) \subseteq m \ \&$$

$$checkpoint(C, state(A))=m \ \&$$

$$resume(C, m)=state(A)$$

The latter two additional restrictions for the image ensure that this image is the result of the checkpoint and the application can be resumed from it.

Definition 3. Checkpoint information is stored in files

To represent the image(s) of the application in files, we state that working file(s) of the application exist which are assigned to the image(s).

$$\forall C \in CLUSTER, \forall A \in APPLICATION, \exists m \in IMAGES:$$

$$checkpoint(C, state(A))=m \ \&$$

$$resume(C,m)=state(A) \ \&$$

$$\forall i \in IMAGES: imagefileofapp(i)=A$$

The function '*imagefileofapp*' returns the application if input parameter has a working file representation owned by an application otherwise it returns the value of undefined. The file is usually stored in the working directory. In other cases the only requirement that the application owns it i.e. can access and read it and has file representation which assumes that can be accessed in the same way as a file.

Definition 4. Parallel checkpointing technique is coordinated

The assumption follows the idea that in coordinated checkpointing the cluster must have a component (part or not part of the application) somewhere on its nodes performing checkpoint coordination.

$$\forall C \in CLUSTER, \exists comp \in \{SCHED, MPE, OS\}:$$

$$cluster(mapped(comp))=C \ \& \ coord(comp) = true$$

$$OR$$

$$\forall C \in CLUSTER , \exists p \in PROCESS:$$

$$cluster(mapped(p))=C \ AND \ (coord(libs(p)) = true \mid coord(code(p)) = true))$$

Definition 5. Coordination process is part of the application

The first part of definition ensures that exactly one process exists among the application processes to which coordination activity belongs to.

$$\forall C \in CLUSTER, \forall A \in APPLICATION:$$

$$\exists p \in PROCESS: cluster(mapped(p))=C \ \& \ app(p)=A \ \&$$

$$(coord(libs(p)) = true \mid coord(code(p)) = true)$$

$$AND$$

$$\forall comp \in \{SCHED, MPE, OS\}: cluster(mapped(comp))=C \ \&$$

$$coord(comp) = false$$

The second part clarifies that no coordination activity exists in a component which is not part of the application.

Definition 6. Processes migrate within a cluster without terminating the application

There are two possibilities to alter the mapping of (i.e. to migrate) the processes of an application. It is possible to checkpoint the whole application, shutdown all the processes and restart the whole application with a different mapping of processes. The second possibility is to checkpoint all or some of the processes, but shutdown those processes only that need to be remapped to another node. The method follows the second alternative.

Let assume that formally a run of an ASM M is defined as an ordered set (M,<) of moves *m*, of its agent satisfying the following conditions (this method is introduced in [47]):

- Each move has its finite predecessors, i.e. for each $m \in M$ the set $\{m' \mid m' < m\}$ is finite.
- The set of moves $\{m \mid m \in M\}$ is linearly ordered by $<$
- Each finite initial segment X of (M, <) has an associated state $\sigma(X)$ which is the result of all moves in X with m executed before m' if $m<m'$. For every maximal element $m \in X$ is the result of applying move m in state $\sigma(X - \{m\})$.

Let us define the cluster and the parallel application as follows:

$$C \in CLUSTER, A \in APPLICATION:$$
$$\exists n_1, n_2, ..., n_c \in NODE, \forall n_j : 1 \le j \le c : cluster(n_j) = C;$$
$$\exists p_1, p_2, ..., p_k \in PROCESS, \forall p_i : 1 \le i \le k : app(p_i) = A;$$

where *c* is the number of nodes, *k* is the number of processes in the cluster.

Let assume we have two mappings (G_1, G_2) between the processes and nodes which are defined as follows:

$$G_1 \in \{n_1, n_2, ..., n_c\}^k, \forall G_{1,i} : 1 \le i \le k : G_{1,i} = mapped(p_i)$$
$$G_2 \in \{n_1, n_2, ..., n_c\}^k, \forall G_{2,i} : 1 \le i \le k : G_{2,i} = mapped(p_i)$$

.Let us define two (sub)set of moves $M_1 \in M$ and $M_2 \in M$ where applying every move in M_1 results in state $\sigma(M_1)$ which is the state where the mapping of application is exactly equal to G_1 and applying every move in M_2 results in state $\sigma(M_2)$ where G_2 exists. We assume that $M_1 \subset M_2$ to define the ordering of corresponding states. Based on the ordering of moves we can define the ordering of states, where $\sigma(M_1)$ happens before $\sigma(M_2)$ if and only if $M_1 < M_2$.

Let us define the states between M_1 and M_2 representing the start of the migration and the end of migration of the application, where *d* is the number of mappings that leads to M_2 from M_1:

$$d = | M_2 \setminus M_1 |$$
$$S_1 ... S_d \in S : S_i = \sigma(M_1 \cup \bigcup_{j=1}^{i} \{m_j\} : m_j \in (M_2 \setminus M_1), 1 \le j \le d)$$

Now, we can define the state of ASM at the point where the two consecutive mapping have happened.

$$\sigma(M_1) : G_1 = (n_1, n_2, ..., n_k), \forall p \in PROCESS : phase(p) = RUNNING \ \& \ app(p) = A$$
$$\sigma(M_2) : G_2 = (n'_1, n'_2, ..., n'_k), \forall p \in PROCESS : phase(p) = RUNNING \ \& \ app(p) = A$$
$$\exists i, 1 < i < k : n_i \neq n'_i$$

And finally, we can make definitions regarding the process phases between the two states. Definition 6 can be fulfilled, if and only if there is least one process that does not terminate between the two states while at least one is migrating. If a process is terminated and restarted it must go through the resumption phase, but at the same time does not necessary holds RUNNING phase continuously. Therefore the expression is as follows:

$$\exists p \in PROCESS, \forall S_i, 1 \le i < d : phase(p) \neq RESUMING$$

This expression says that at least one process exists during the execution of the migration protocol (from S_1 to S_d) that is not restarted.

Definition 7. Application source code is unmodified

Using the ‘ckpt’ and ‘coord’ functions it is possible to describe that checkpointing activities cannot exist in the user code of any process of the application.

$$\forall A \in APPLICATION, \forall p \in PROCESS, app(p)=A:$$

$$ckpt(code(p)) = false \ \& \ coord(code(p)) = false$$

As a summary, the definitions above clearly define each definition of the proposed checkpointing method for Clustergrid environments.

2.2.4 Definition of the CP_{ground} ASM model

In this section a formal model - called CP_{ground} - is defined in the ASM framework. This model forms the basis for a distributed checkpointing tool that provides all the features introduced by the Clustergrid method in section 2.2.3. This model is a ground model since in section 3, two additional refinements of this model are going to be introduced.

2.2.4.1 Universes and signatures

The model presented here is a multi-agent ASM, where agents are processes i.e. elements from the *PROCESS* universe. The nullary *Self* function represented here as *p* allows an agent to identify itself, so it is interpreted differently by the different agents [47].

Basic universes and signatures used in the CP_{ground} ASM model are introduced in section 2.1.5.2, only extension of universes and signatures are described in the following paragraphs.

To give a detailed internal behaviour of a parallel multi-process application the communication activity must be modelled. To realise communication, processes are sending and receiving messages (universe MESSAGE), where every message has a sender (*from: MESSAGE→PROCESS*) and a receiver (*to: MESSAGE→PROCESS*). The actual content of the message is irrelevant in this model.

During execution of the application, its processes interact with each other and the surrounding cluster middleware components. Every interaction has an initiator and a target. These interactions are modelled as “events”. There are numerous events occurring among the participants, so modelling each of them is a difficult task and irrelevant from checkpointing point of view. Therefore only the checkpoint related events are modelled by the universe EVENT. The following events are represented: *spawn* to create a new process, *send* to send a message, *receive* to receive a message, *terminate* to finish execution of a certain process, *checkpoint* to perform checkpointing of a process, *resume* performing resumption of a process, *exit* to notify a process to finish execution. The occurrence of an event is represented by the external function *event: PROCESS → EVENT*. Events are generated by an external function called *egen*. This function describes events generated by certain instructions which are located at certain points in the code. Each universe (representing the components of a cluster middleware) contains elements representing instructions to be executed. The function *egen* maps the instructions and an instruction pointer to an event: *egen: ({CODE, LIBS, SCHED, MPE, OS}, INSTRUCTION_POINTER) → EVENT*. (Note: the referred universes are defined in section 2.1.5.2)

A program represented by an instruction set (INSTRUCTION_SET) that can be divided according to the components that execute them. Elements of the INSTRUCTION_SET are fragments of program codes, like instructions, procedures, and so on. Therefore we define universes CODE for user defined code, LIBS for linked libraries, SCHED for scheduler, OS for the operating system and MPE for

message-passing subsystem. Accordingly, we can separate functional components such as user program, library, operating system routine, message-passing procedures.

In the following sections only those rules are introduced and explained, where functions related to the INSTRUCTION_SET universe are always related to LIBS. In other words we model the library component. The rules that operate on CODE, SCHED, OS and MPE universes, i.e. user program, scheduler, message-passing components are omitted in the CP_{ground} model since well-known functionalities are assumed such as creating a process, managing memory and handling I/O operations, etc.

Processes are going through different phases during their execution represented by the function *phase: PROCESS → {init, waiting, receive_waiting, running, checkpointing, resuming, terminating}*. Any process of the application may be marked to be checkpointed (*process_to_checkpoint: PROCESS → {true, false}*), marked to be terminated (*process_to_terminate: PROCESS → {true, false}*), marked to be resumed (*process_to_resume: PROCESS → {true, false}*) or marked to be stored (*process_to_store: PROCESS → {true, false}*).

Two different roles are defined for the processes that are expressed by the function *role: PROCESS → {coordinator, userdefined}*. The function *role* returns *userdefined* for a process if it is programmed by the user. Every process has a function for startup (*startupmode: PROCESS → {normal, resume}*) marking the process to be resumed and a relation (*master: PROCESS → {true, false}*) that is evaluated to true for exactly one process that is spawned first in the application.

2.2.4.2 Initial state

In this model, an application starts one process (master) initially which then performs spawning the required additional processes forming the application. Spawning the first process is the responsibility of the middleware components of the cluster, while spawning the rest of the processes is down by any of the application processes e.g. by the master. Therefore the initial state is exactly one process where the functions are interpreted as follows:

```
∃p∈PROCESS: app(p)≠undef,
            phase(p)=init,
            role(p)=undef,
            master(p)=undef,
            startupmode(p)=undef
```

The initial state claims that the master process requires an application (identifier) to belong to (“app(p)≠undef”) and the execution phase to be set to *init* (“phase(p)=init”). The other three functions must be set to *undef*, since they are updated internally.

2.2.4.3 Rules

1. Rules for initialisation

At startup, each process decides its role in the application. Two different roles exist: *coordinator* that manages the checkpoint related activities and *user-defined* that is programmed by the user. Coordinator has zero instruction set belonging to *CODE* universe, while user-defined processes have instruction sets both from *CODE* and from *LIBS* universe. The first initialisation rule fires once for user-defined processes

and twice for coordinator process. This rule selects only the first (master) process to be the coordinator. All the processes are updated to *waiting* phase by this rule.

CPground-R1a

```
if phase(p)=init then
      if role(p)=undef then
            role(p)=coordinator
      else
            phase(p):=WAITING
      endif
      if master(p)=undef then
            master(p)=true
      endif
endif
```

When a coordinator is selected it decides at start-up, whether to execute the application from the beginning (*normal*), or resume it from a previous checkpoint (*resume*). This decision is based on the existence of the checkpoint information.

CPground-R1b

```
if role(p)=coordinator & phase(p)=waiting & startupmode(p)=undef
then
      if ∃i∈IMAGES: imagefileofapp(i)=app(p) then
            startupmode(p):=resume
      else
            startupmode(p):=normal
      endif
endif
```

In case of normal execution one user-defined process is created that is going to build the whole application, otherwise *resuming* phase is started.

CPground-R1c

```
if role(p)=coordinator & phase(p)=waiting & startupmode(p)≠undef then
      if startupmode(p)=normal then
            phase(p):=running
            process_to_store:={}
            process_to_checkpoint:={}
            process_to_terminate:={}
            process_to_resume:={}
            extend PROCESS by child with
                  app(child):=app(p)
                  phase(child):=init
                  role(child):=userdefined
                  startupmode(child):=normal
                  master(child):=false
            endextend
      else
            phase(p):=resuming
            event(p):=resume
      endif
endif
```

2. Rules for process creation

Whenever a process executes an instruction (programmed by the user) that generates *spawn* event, a new process is created with the appropriate parameter set. The newly created process (*child*) belongs to the same application. Obviously, the new process will fire the rules of initialisation, after created. The way of creation and allocation of process on a node is irrelevant in this model.

CPground-R2a

```
if role(p)=userdefined & phase(p)=running & event(p)=spawn then
      extend PROCESS by child with
            app(child):=app(p)
            phase(child):=init
            role(child):=userdefined
            startupmode(child):=normal
            master(child):=false
      endextend
endif
```

This rule ensures that the newly created process starts running after it fired the initialisation rules.

CPground-R2b

```
if role(p)=userdefined & phase(p)=waiting &
   startupmode(p)=normal & process_to_checkpoint(p)=false
   then
      phase(p):=running
endif
```

3. Rule for sending a message

Processes in a distributed system interact with each other via message passing. Although, in modern systems there are higher level constructs (e.g. RPC, RMI, etc. [86]) with a rich set of sophisticated message-passing services, in the lowest level they all based on simple send-receive communication primitives.

A running process programmed by the user may execute an instruction (from the CODE universe) that generates a *send* event to send a message to another process. This event is handled by this rule that implements a non-blocking version of sending operation. The content of the message is irrelevant, the source (*from*) and target (*to*) process is modelled.

CPground-R3a

```
if role(p)=userdefined & phase(p)=running &
   event(p)=send(dest)
   then
      extend MESSAGE by msg with
            from(msg):=p
            to(msg):=dest
      endextend
endif
```

4. Rules for receiving a message

A running user process may execute an instruction (from the CODE universe) that generates a *receive* event to send a message to another process. This event is handled by this rule that implements a blocking version of receiving operation. In case the message does not exist yet, the process waits for it by changing its phase. If the message exists, it is removed from the universe MESSAGE, i.e. it is received by the process. The source process of the required message may (*source*) or may not (*any*) be specified to support nondeterministic behaviour.

CPground-R4a

```
if role(p)=userdefined & phase(p)=running &
   event(p)=receive({source|any})
   then
      if (∃msg∈MESSAGE):to(msg)=p & {from(msg)=source| }
   then
            MESSAGE(msg):=false
      else
            expecting(p):={source|any}
            phase(p):=receive_waiting
      endif
endif
```

If a message exists from the appropriate (*expecting*) process, the message is removed and process continues its execution by changing its phase.

CPground-R4b

```
if role(p)=userdefined & phase(p)=receive_waiting &
   ((∃msg∈MESSAGE):to(msg)=p & {from(msg)=expecting(p)| })
   then
      MESSAGE(msg):=false
      phase(p):=running
      expecting(p):=undef
endif
```

5. Rules for initiating checkpoint procedure

Checkpoint procedure is initiated based on the occurrence of the appropriate event during the execution of the application. In this model the *exit* event initiates the checkpoint. *Exit* event can be generated by any middleware component, but usually it is done by the scheduler component when a process is intended to be removed from its node, i.e. process is marked to be pre-empted. This model handles this case by performing an application-wide checkpointing.

There are two situations: In case the master (coordinator in this model) process gets the exit event every process (including master) must be checkpointed and terminated. Otherwise, after performing an application-wide checkpointing, the processes that got this event must be terminated and restarted. These two situations are handled by the following three rules.

Whenever a *userdefined* process receives an *exit* event, it is marked to perform checkpoint, terminate and resume operations sequentially:

CPground-R5a

```
if role(p)=userdefined &
   phase(p)={running|receive_waiting|checkpointing} & event(p)=exit
   then
      process_to_checkpoint(p):=true
      process_to_terminate(p):=true
      process_to_resume(p):=true
endif
```

In case the coordinator gets an exit event, the whole application (including the coordinator itself) is going to be checkpointed and terminated, since without the coordinator the application cannot continue its execution. Therefore, every process is marked for checkpointing and termination and phase is changed accordingly.

CPground-R5b

```
let allproc:=((∀cp∈PROCESS):app(cp)=app(p) & cp≠p)
if role(p)=coordinator & phase(p)=running & event(p)=exit
   then
      process_to_checkpoint:=allproc
      process_to_terminate:=allproc
      process_to_resume:={}
      process_to_store:=allproc
      phase(p):=checkpointing
      event(p):=checkpoint
endif
```

Whenever the coordinator detects the need of checkpointing it initiates checkpointing of all the processes by changing their phase to *checkpointing* and by notifying them with a *checkpoint* event. Every process must take part since application-wide checkpointing is required by the method defined in section 2.2.3.

CPground-R5c

```
let allproc:=((∀cp∈PROCESS):app(cp)=app(p) & cp≠p)
if role(p)=coordinator &
   (∃p1∈PROCESS:app(p1)=app(p) & p1≠p &
   process_to_checkpoint(p1)=true &
   phase(p1)={running|receive_waiting})
   then
      do forall p2 :p2∈allproc & app(p2)=app(p) & p2≠p &
                    phase(p2)={running|receive_waiting}
            process_to_checkpoint(p2):=true
            phase(p2):=checkpointing
            event(p2):=checkpoint
      enddo
endif
```

6. Rule to create checkpoint

If a user process receives a *checkpoint* event, it creates an image file with its own state and then changes phase to *terminate* if needs to be terminated, otherwise it waits for the coordinator. Checkpoint creation of a single process is modelled by a high-level macro called *SINGLE_PROCESS_STATE_CHECKPOINT*. The way – how the relevant information for saving and restoring a single process is collected – is irrelevant in this model. This macro takes the process and image file as parameter. Refinement of this macro is out of scope of this model.

CPground-R6a

```
if role(p)=userdefined & phase(p)=checkpointing & event(p)=checkpoint
   then
      extend IMAGE by imagefile with
            imagefileofapp(imagefile):=app(p)
            SINGLE_PROCESS_STATE_CHECKPOINT(p,imagefile)
      endextend
      if process_to_terminate(p)=true then
            phase(p):=terminating
      else
            phase(p):=waiting
      endif
endif
```

7. Rules to terminate processes

The following rule handles the case when the user defined process reaches the end of its execution and intends to terminate normally. This notification is realised by the *terminate* event generated by the appropriate instruction belonging to the *CODE*

universe. The detection of the event causes the process to change its phase to *terminating* which is handled in the rules detailed later.

CPground-R7a

```
if role(p)=userdefined & phase(p)=running & event(p)=terminate then
      phase(p)=terminating
endif
```

The following rule handles two cases. In the first case when all user defined processes reached the end of their execution and are being terminated normally, coordinator removes existing checkpoint information (assumption: checkpoint information became useless, since application finished successfully), instructs the processes to finish their execution and terminates. In the second case, an application-wide checkpoint is just about to finish (finished executing its user code), therefore coordinator performs self-checkpoint and application terminates.

CPground-R7b

```
if role(p)=coordinator & process_to_resume={} &
   (∀p1∈PROCESS: p1≠p & app(p1)=app(p) & phase(p1)=terminating )
   then
      if phase(p)=running then
            do forall imagefile : imagefile∈IMAGE &
                                  imagefileofapp(imagefile)=app(p)
                  IMAGE(imagefile):=false
            enddo
            do forall p2∈PROCESS: app(p2)=app(p) & p2≠p
                  event(p2):=terminate
            enddo
            PROCESS(p):=false
      endif
      if phase(p)=checkpointing then
            extend IMAGE by imagefile with
                  imagefileofapp(imagefile):=app(p)
                  SINGLE_PROCESS_STATE_CHECKPOINT(p,imagefile)
            endextend
            do forall p2∈PROCESS: app(p2)=app(p) & p2≠p
                  event(p2):=terminate
            enddo
            PROCESS(p):=false
      endif
endif
```

The following rules ensures that the process terminates i.e. the process is removed from the *PROCESS* universe.

CPground-R7c

```
if role(p)=userdefined & phase(p)=terminating & event(p)=terminate
then
      PROCESS(p):=false
endif
```

8. Rules to resume processes

Whenever an application is started and checkpoint information exists, application continues its execution from checkpoint. This procedure starts with resuming the state of the coordinator. The macro called *SINGLE_PROCESS_STATE_RESUME* expresses a service providing the state recovery of the calling process. Similarly to the macro called *SINGLE_PROCESS_STATE_CHECKPOINT* the refinement is omitted, library-level resumption is assumed. As a next step user defined processes are spawned and their resumption is initiated.

CPground-R8a

```
if role(p)=coordinator & phase(p)=resuming & event(p)=resume
   then
      imagefile ∈ IMAGE: imagefileofapp(imagefile)=app(p)
      SINGLE_PROCESS_STATE_RESUME(p,imagefile)
      do forall child : process_to_store(child)=true
            extend PROCESS by child with
                  app(child):=app(p)
                  phase(child):=init
                  role(child):=userdefined
                  startupmode(child):=resume
                  master(child):=false
            endextend
      enddo
      process_to_checkpoint:={}
      process_to_terminate:={}
      process_to_resume:=process_to_store
      phase(p):=running
endif
```

This rule ensures the restart of processes that have finished checkpointing and are about to be resumed. These are the processes that got *exit* event previously, performed checkpoint and must be restarted on another node of the cluster. The method of resource allocation is irrelevant in this model; it is performed by the scheduler.

CPground-R8b

```
if role(p)=coordinator & phase(p)=running &
   ∃p1∈PROCESS: phase(p1)=terminating & process_to_resume(p1)=true
   then
      event(p1):=terminate
      extend PROCESS by child with
            app(child):=app(p)
            phase(child):=init
            role(child):=userdefined
            startupmode(child):=resume
            master(child):=false
      endextend
endif
```

A user defined process may get to *waiting* phase in two different ways: the process finished checkpointing (and is waiting to continue execution) or the process has just been started (and is waiting to resume and/or continue execution). To handle this case, the coordinator changes the phase of every process accordingly.

CPground-R8c

```
if role(p)=coordinator & phase(p)=running &
   (∀proc∈PROCESS:app(proc)=app(p) & proc≠p & phase(proc)=waiting)
   then
      do forall pp : pp∈PROCESS & app(pp)=app(p) & pp≠p
            if process_to_resume(pp)=true then
                  phase(pp):=resuming
                  event(pp):=resume
            else
                  phase(pp):=running
            endif
      enddo
endif
```

Resumption of the state of a user defined process is performed when it gets a *resume* event. It is done by the SINGLE_PROCESS_STATE_RESUME macro that has been detailed at rule CPground-R8a. After resumption, execution is continued depending whether the process was communicating or not at the time of the last checkpoint.

CPground-R8d

```
if role(p)=userdefined & phase(p)=resuming & event(p)=resume
   then
      imagefile ∈ IMAGE: imagefileofapp(imagefile)=app(p)
      SINGLE_PROCESS_STATE_RESUME(p,imagefile)
      if expecting(p)=undef then
            phase(p):=running
      else
            phase(p):=receive_waiting
      endif
endif
```

2.2.5 Validation of the CP_{ground} model

In the CP_{ground} model an application interacts with the middleware components through events. When an application requires service from one of the components, it generates an event which is caught and served by a handler of that component. To express which event is generated by a component ($Event_{generator}$) and which one is handled ($Event_{handler}$) by a component, the following functions are introduced:

$$Event_{generator}: EVENT \times \{CODE, LIBS, SCHED, MPE, OS\} \rightarrow \{true, false\}$$

$$Event_{handler}: EVENT \times \{CODE, LIBS, SCHED, MPE, OS\} \rightarrow \{true, false\}$$

Let us define the following assumptions:

Assumption 1. During the execution of a distributed application, the application relies on the following four basic functionalities or services that need external support: process creation (*spawn* event) served by the message-passing layer (it interacts with the operating system and scheduler), termination (*terminate* event) served by the scheduler, message sending (*send* event) and receiving (*receive* event) both served by the message-passing layer.

$$\forall e \in \{spawn, send, receive, terminate\}, \exists c \in CODE: Event_{generator}(e, c)=true$$

$$\forall e \in \{spawn, send, receive\}, \exists mp \in MPE: Event_{handler}(e, mp)=true$$

$$\forall e \in \{terminate\}, \exists s \in SCHED: Event_{handler}(e, s)=true$$

Assumption 2. During the execution, termination (exit event) of a process can be initiated by an instruction belonging to the scheduler (universe SCHED), and the event can be handled by instructions executed by the user code (universe CODE) or libraries (universe LIBS). The events *terminate* and *exit* both causes the process to finish its execution, but are distinguished based on the initiator.

$$\exists s \in SCHED: Event_{generator}(exit, s)=true$$

$$\forall p \in PROCESS: Event_{handler}(exit, code(p))=true \mid Event_{handler}(exit, libs(p))=true$$

All events other than the ones defined in the previous two assumptions (*checkpoint* and *resume* in this model) belong to the checkpointing functionality. The next step is to define which interaction set/universe the *checkpoint* and *resumption* event generation and handling belongs to.

The definition of the CP_{ground} model (see section 2.2.4) details that the rules defined for the CP_{ground} model are updating the locations related to the LIBS universe.

Therefore, any event generation or event handling activity realised by the rules named CPground-* updates locations related to the LIBS universe.

After analysing the rules of CP_{ground} model, checkpoint event generation occurs in rules CPground-R5b and CPground-R5c, while resume event generation occurs in rules CPground-R1c, CPground-R8c.

Definition A:

$$\forall e \in \{checkpoint, resume\}, \exists l \in LIBS:$$
$$Event_{generator}(e, l)=true \text{ and}$$

$$\forall e \in \{checkpoint, resume\}, \forall i \in \{CODE, SCHED, MPE, OS\}:$$
$$Event_{generator}(e, i)=false$$

Checkpoint event is handled in rule CPground-R6a, while *Resume* event is handled by the rules CPground-R8a and CPground-R8d, respectively.

Definition B:

$$\forall e \in \{checkpoint, resume\}, \exists l \in LIBS:$$

$$Event_{handler}(e, l)=true \text{ and}$$

$$\forall e \in \{checkpoint, resume\}, \forall i \in \{CODE, SCHED, MPE, OS\}:$$

$$Event_{handler}(e, i)=false$$

Before checking the ASM model against the seven definitions (defined in section 2.2.3), *ckpt* and *coord* rules (see section 2.1.5.2) must also be expressed using the functions $Event_{generator}$ and $Event_{handler}$. We can say that a checkpoint activity exists in a component, if checkpoint related event generation or event handling is performed by that component. For instruction set that handles checkpoint related events the function *ckpt* evaluates to true and for the instruction set that generates these events the function *coord* evaluates to true. Therefore

Definition C:

$$\text{if } \exists e \in \{checkpoint, resume\},$$

$$instruction \in \{CODE, LIBS, SCHED, MPE, OS\}:$$

$$Event_{handler}(e, instruction) = true$$

$$\text{then } ckpt(instruction) = true,$$

$$\text{otherwise } ckpt(instruction) = false$$

Similarly *coord* function can be expressed in the following way:

Definition D:

$$\text{if } \exists e \in \{checkpoint, resume\},$$

$$instruction \in \{CODE, LIBS, SCHED, MPE, OS\}:$$

$$Event_{generator}(e, instruction) = true$$

$$\text{then } coord(instruction) = true$$

$$\text{otherwise } coord(instruction) = false$$

Based on the definitions A, B, C and D, now it is possible to evaluate the seven definitions of the ClusterGrid checkpointing method defined in section 2.2.3.

Definition 1. Library level checkpointing technique is used

$$\forall C \in CLUSTER, \forall A \in APPLICATION, \forall p \in PROCESS, app(p)=A:$$

$$ckpt(code(p)) = false \textbf{ (expression 1.1)}$$

$$\& \; ckpt(libs(p)) = true \textbf{ (expression 1.2)}$$

$$\& \; \forall n \in NODE, cluster(n) = C,$$

$$\forall comp \in \{SCHED, MPE, OS\}, mapped(comp) = n:$$

$$ckpt(comp) = false \textbf{ (expression 1.3)}$$

$$\& \; coord(comp) = false \textbf{ (expression 1.4)}$$

Evaluation of logic expression in definition 1 for the CP_{ground} model is divided into four parts. The evaluation of the four parts is carried out in the following way:

(expression 1.1) ckpt(code(p)) =

$$Event_{handler}(checkpoint, code(p)) \mid Event_{handler}(resume, code(p)) =$$

$$(false \mid false) = false;$$

(expression 1.2) ckpt(libs(p)) =

$$Event_{handler}(checkpoint, libs(p)) \mid Event_{handler}(resume, libs(p)) =$$

$$(true \mid true) = true;$$

Since no checkpoint or resume events are generated or handled by any instruction belonging to SCHED, MPE and OS universes, the last part is also evaluated to *false* for each of the universes as described below:

(expression 1.3) $\forall comp \in \{SCHED, MPE, OS\}$:

$$(Event_{handler}(checkpoint, comp) \mid Event_{handler}(resume, comp)) =$$

$$(false \mid false) = false$$

(expression 1.4) $\forall comp \in \{SCHED, MPE, OS\}$:

$$Event_{generator}(checkpoint, comp) \mid Event_{generator}(resume, comp) =$$

$$(false \mid false) = false$$

Altogether, the expression for Definition 1 is satisfied, since all the four parts satisfies the equivalence, separately.

Definition 2. Application wide checkpointing is performed

$$\forall C \in CLUSTER, \forall A \in APPLICATION:$$

$$\forall p \in PROCESS, app(p)=A:$$

$$\exists m \in IMAGES:$$

$$\text{image}(p) \subseteq m \;\&$$

$$\text{checkpoint}(C, \text{state}(A)) = m \;\&$$

$$\text{resume}(C, m) = \text{state}(A)$$

This definition says that for each process a checkpoint image is created at checkpoint time. To decide whether the image is created, the appropriate rules of the *coordinator* and the *userdefined* processes must be analysed. The rules *CPground-R5b, CPground-R5c* on the coordination side ensures that in case of termination all the processes gets the checkpoint event and the rule *CPground-R6a* ensures that an image is created by the process that gets the event.

Definition 3. Checkpoint information is stored in files

$$\forall C \in \text{CLUSTER}, \forall A \in \text{APPLICATION}, \exists m \in \text{IMAGES}:$$

$$\text{checkpoint}(C, \text{state}(A)) = m \;\&$$

$$\text{resume}(C,m) = \text{state}(A) \;\&$$

$$\forall i \in \text{IMAGES}: \text{imagefileofapp}(i) = A$$

This definition expresses a further restriction the need of mapping the image into a file. Image creation is performed by the rules *CPground-R6a* and *CPground-R7b*, respectively. In both rules, the function *imagefileofapp* ensures that the image is mapped on to a file belonging to the application.

Definition 4. Parallel checkpointing technique is coordinated

$$\forall C \in \text{CLUSTER}, \exists \text{comp} \in \{\text{SCHED, MPE, OS}\}:$$

$$\text{cluster}(\text{mapped}(\text{comp})) = C \;\&\; \text{coord}(\text{comp}) = \text{true} \quad \textbf{(expression 4.1)}$$

OR

$$\forall C \in \text{CLUSTER}, \exists p \in \text{PROCESS}:$$

$$\text{cluster}(\text{mapped}(p)) = C \;\&\; (\text{coord}(\text{libs}(p)) = \text{true} \mid \text{coord}(\text{code}(p) = \text{true}) \quad \textbf{(expression 4.2)}$$

The checkpoint method (modelled by CP_{ground}) conforms definition 4 if at least one component performs coordination activity (i.e. *coord* function is evaluated to *true* for one of the INSTRUCTION_SET subuniverses).

Expression 4.1 is evaluated to *false*, since based on definition A there are no checkpoint related events generated by any of the SCHED, MPE, OS instruction sets.

At the same time, the expression 4.2 is evaluated to *true* according to definition A and D since coordination activity exists among the processes of the application. The rule CPground-R1a ensures that at least one process sets the expression *role(self)* to *coordinator* and the rules CPground-R5b, CPground-R5c perform checkpoint event generation in this process.

Definition 5. Coordination process is part of the application

$$\forall C \in \text{CLUSTER}, \forall A \in \text{APPLICATION}:$$

$$\exists p \in PROCESS: cluster(mapped(p))=C \ \& \ app(p)=A \ \&$$

$$(coord(libs(p))=true \mid coord(code(p))=true)$$

$$AND$$

$$\forall comp \in \{SCHED, MPE, OS\}: cluster(mapped(comp)=C \ \&$$

$$coord(comp)=false$$

This definition is a more restrictive version of the previous (definition 4) one. While the previous one requires a coordination component on the cluster, this definition requires it to be integrated into the application. The existence of the coordinator activity has already been showed in definition 4.

Definition 6. Processes migrate within a cluster without terminating the application

To see, whether the CP_{ground} model satisfies this point, we have to examine the state transition of a process during migration. After analysing the rules a phase transition diagram can be created as seen in Figure 13.

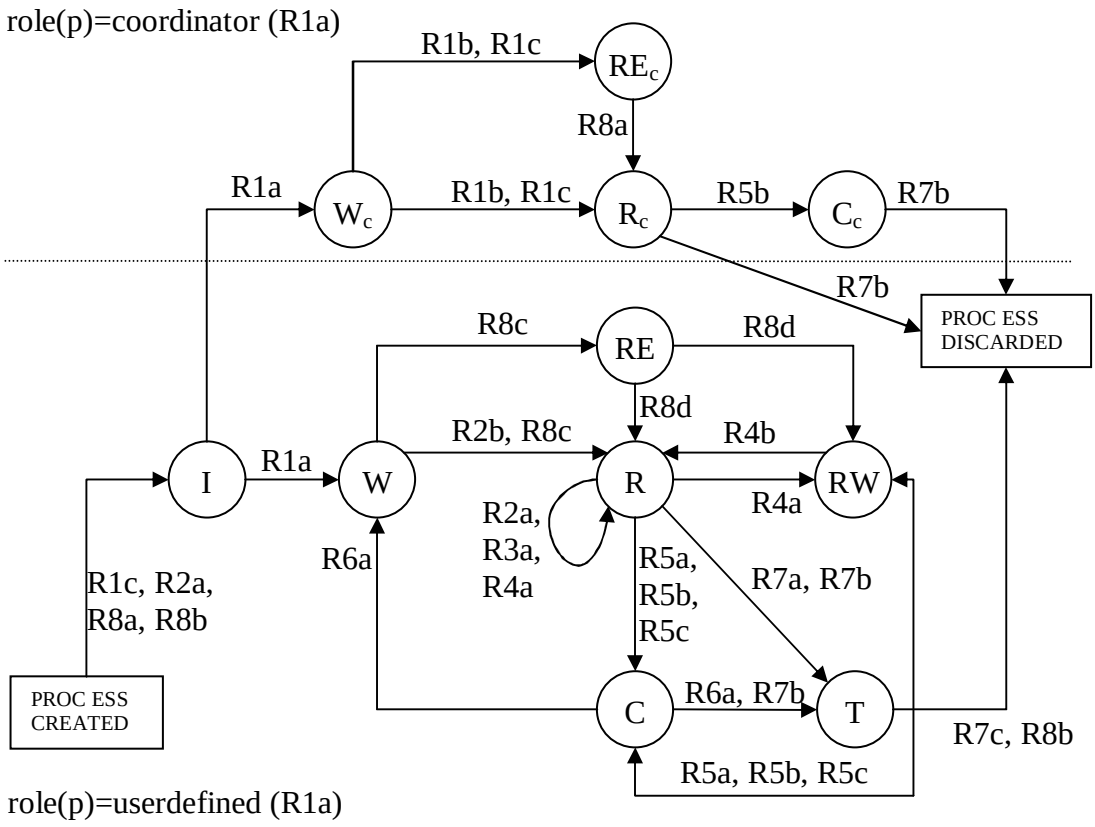

Figure 13 Phase transition diagram of the CP_{ground} model

Nodes represent the possible phases of the processes and the directed arcs represent the firing of rules that change the phase of a process accordingly. Abbreviations for the name of the phases are as follows:

I: init, W: waiting, R: running, RW: receive_waiting,

T: terminating, C: checkpointing, RE: resuming

Based on the phase transition diagram, the following four statements can be derived:

1. All user defined (i.e. *role(p)=userdefined*) processes at normal execution are going through the following phases:

   a. At startup: *init* $\Rightarrow$ *waiting* $\Rightarrow$ *running*

   b. At receiving a message running $\Rightarrow$ *receive_waiting* $\Rightarrow$ *running*

2. Migrating user defined processes are going through the following phases:

   a. *running* $\Rightarrow$ *checkpointing* $\Rightarrow$ *terminating* $\Rightarrow$ *init* $\Rightarrow$ *waiting* $\Rightarrow$ *resuming* $\Rightarrow$ *running*

3. Non-migrating user defined processes are going through the following phases if at least one migrating process exists in the application:

   a. *running* $\Rightarrow$ *checkpointing* $\Rightarrow$ *waiting* $\Rightarrow$ *running*

4. Coordinator process does not change phase during the migration of the user defined process(es).

Based on the four statements derived from the phase transition diagram of CP_{ground} model, the original (Definition 6) expression can be evaluated to true, since the non-migrating processes and the coordinator process never terminate during the migration of migrating processes of the application. The coordinator terminates only in two cases: if every user defined process has finished execution or if coordinator gets an exit event from the scheduler. The first case means the normal termination of the application, while the second one happens when the whole application is shut down by the scheduler before it could finish the execution.

Definition 7. Application source code is unmodified

$$\forall A \in \text{APPLICATION}, \forall p \in \text{PROCESS}, \text{app}(p)=A:$$

$$\text{ckpt}(\text{code}(p))=\text{false} \ \& \ \text{coord}(\text{code}(p))=\text{false}$$

ckpt(code(p)) which is evaluated to *false* according to Definition 1 for the CP_{ground} model. The second part can be evaluated taking definitions A and D. Definition D that state the *coord* function is evaluated to true for the elements of the universe if it contains at least one instruction that generates checkpoint or resume events.

Since definition A says that

$$\forall e \in \{\text{checkpoint, resume}\}, \forall \text{instruction} \in \{\text{CODE, SCHED, MPE, OS}\}:$$

$$\text{Event}_{\text{generator}}(e, l)=\text{false}$$

the expression *coord(code(p))* is also evaluated to false, accordingly.

3 Checkpointing PVM applications

3.1 The GRAPNEL checkpointing framework

3.1.1 Overview

The theoretical background introduced in the first group of theses – which resulted in a checkpointing solution defined by an abstract model – forms an appropriate basis for developing a concrete tool. To utilise the theoretical results I have chosen the P-GRADE graphical parallel program development environment developed by MTA SZTAKI. The goal of the second group of theses is to apply the abstract model for message-passing PVM applications or algorithms created by the P-GRADE environment.

As a preparation of thesis 2.1 I have designed and elaborated a checkpointing technique for PVM applications created by the P-GRADE development environment based on the abstract method defined in thesis 1.2. I have studied the architectural design of the GRAPNEL application. I have defined the required modifications on the architecture and redesigned the communication primitives in a way that the checkpointing operation can be activated at any time during the execution. In addition I have elaborated an abstract model ($CP_{grapnel}$) that fits to the solution introduced in P-GRADE. Finally, I have proven the correctness of refinement between the CP_{ground} and $CP_{grapnel}$ models. Based on the results thesis 2.1 is stated.

> **Thesis 2.1:** *The checkpointing technique integrated in GRAPNEL applications – following a static process model and developed by P-GRADE – realises transparent checkpointing and its corresponding $CP_{grapnel}$ ASM model is a correct refinement of the original model called CP_{ground}.*
>
> Related publications are [1][3][13][16][17][18].

The elaborated solution enables transparent checkpointing operation – both for the programmer and for the middleware – for parallel applications which follow a static process model and developed by the P-GRADE environment.

3.1.2 P-GRADE environment and GRAPNEL language

In order to cope with the extra complexity of parallel and distributed programs due to inter-process communication and synchronization, a graphical programming environment called P-GRADE [1][48] have been designed. Its major goal is to provide an easy-to-use, integrated set of programming tools for development of general message-passing applications to be run on both homogeneous and heterogeneous distributed computing systems like supercomputers, clusters and Grid systems.

The central idea of P-GRADE is to support each stage of the parallel program development life-cycle (e.g. designing, executing, debugging [2], monitoring) by an integrated graphical environment (see Figure 14) where all the graphical views applied at the various levels are associated with application designed and edited by the user.

The parallel program design is supported by the GRAPNEL [35] (GRAphical Process NEt Language) language and the GRED (see Figure 14) graphical editor. In GRAPNEL, all process management and inter-process communication activities are defined graphically in the user's application. Low-level details of the underlying message-passing system are hidden. P-GRADE generates automatically all message-passing library calls (either PVM or MPI) on the basis of the graphical notation of GRAPNEL. Since graphics hides all the low level details of message-passing, P-

GRADE is an ideal programming environment for application programmers who are not experienced in parallel programming (e.g., for chemists, biologists, etc.). GRAPNEL is a hybrid language: while graphics is introduced to define parallel activities, textual language parts (C/C++ or FORTRAN) are used to describe sequential activities.

GRAPNEL is based on a hierarchical design concept supporting both the bottom-up and top-down design methods. A GRAPNEL program has three hierarchical layers which are as follows from top to bottom:

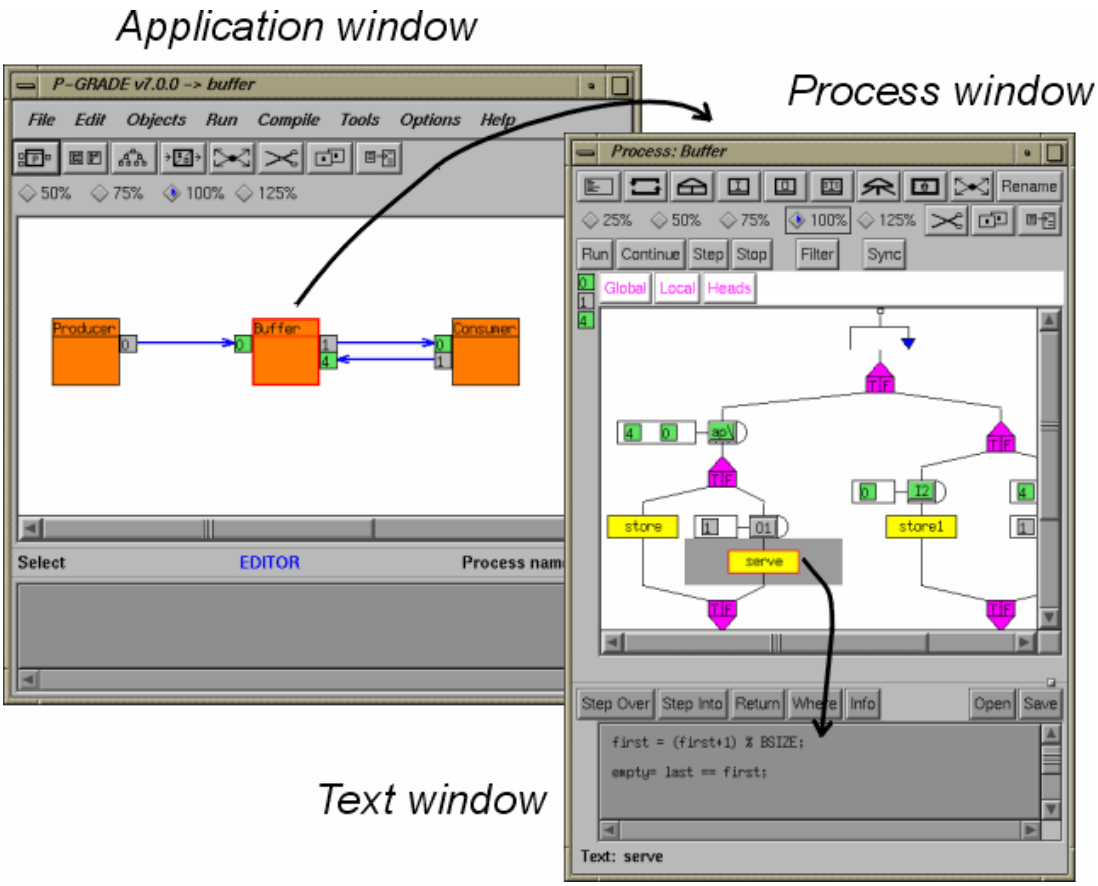

Figure 14 Hierarchical design in the P-GRADE environment

- Application layer is a graphical layer which is used to define the component processes, their communication ports as well as their connecting communication channels (directed arrows between processes on the application window on Figure 14). Shortly, the Application layer serves for describing the interconnection topology of the component processes.
- Process layer is also a graphical layer to define the internal structure of the component processes by a flow-chart like graph (see Figure 14). The basic goal is to provide graphical representation for the message passing function calls. As a consequence, every structure that contains message passing calls should be graphically represented. The following types of graphical blocks are applied: loop construct, conditional construct, sequential block, message passing activity block and graph block. Sequential blocks must not contain any message passing calls.
- Text layer is used to define those parts of the program that are inherently sequential and hence a textual language like C/C++ or FORTRAN can be

applied at this level. These textual codes are defined inside the sequential blocks of the Process layer (see Figure 14).

The top-down design method can be used to describe parallel activities of the application program. At the top level the topology and protocols of the interprocess communication can be defined and then in the next layer the internal structure of individual processes can be specified. At this level and in the Text layer the bottom-up and top-down design methods can be used in a mixed way. In the case of the top-down design method, the user can define the graphical structure of the process and then uses the Text layer to define the C/C++ or FORTRAN code for the sequential blocks. In the bottom-up design approach, the user can inherit code from existing C/C++ or FORTRAN libraries and then can build up the internal process structure based on these inherited functions. Moreover, GRAPNEL provides predefined scalable process communication templates that allow the user to generate large process farm, pipeline and mesh applications fast and safely.

The GRED editor helps the user to construct the graphical parts of GRAPNEL programs in an efficient and fast way. GRAPNEL programs edited by GRED are saved into an internal file called GRP file that contains both the graphical and textual information of GRAPNEL programs. The main concepts of GRAPNEL and GRED are described in detail in [1].

After the edition of the application has finished the pre-compilation and compilation of GRAPNEL programs takes place. The goal of pre-compilation is to translate the graphical language information of the GRP file into PVM or MPI function calls and to generate the C or FORTRAN source code of the GRAPNEL program. For the sake of flexibility, PVM and MPI function calls are not called directly in the resulting C code; they are hidden in an internal library, called the GRAPNEL Library which has two versions. In the first version (GRP-PVM Library) the GRAPNEL Library functions are realised by PVM calls, and in the second version (GRP-MPI Library) they are realised by MPI function calls.

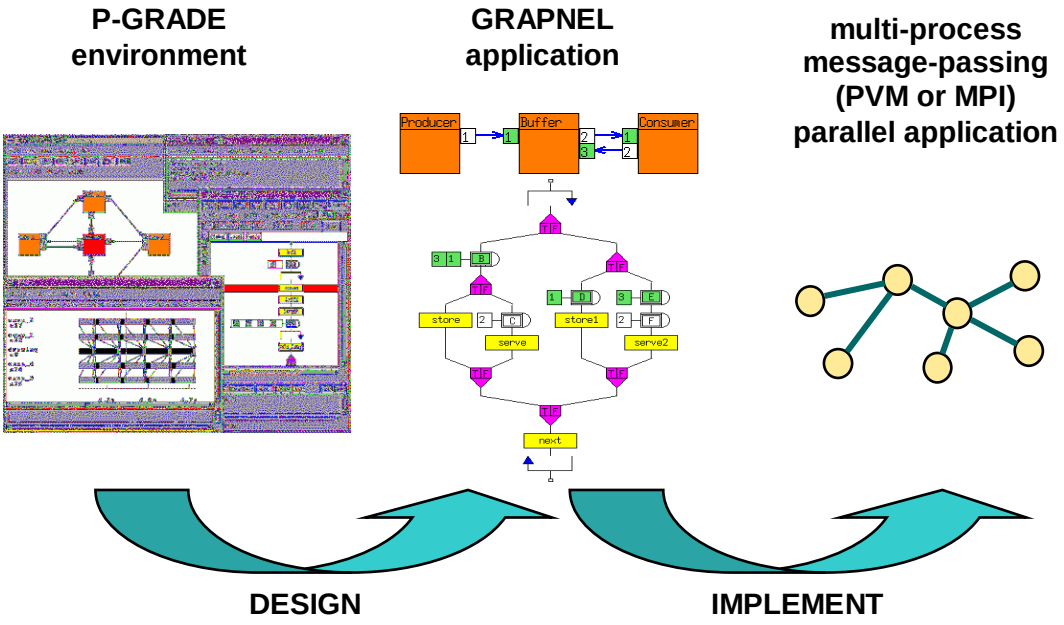

Figure 15 Relation of P-GRADE and GRAPNEL application

As a result of the compilation, one executable is created containing the code for all communicating processes. The execution of each process is defined by the following triplet: user code, GRAPNEL library and the concrete message-passing library (PVM or MPI).

3.1.3 The GRAPNEL checkpointing solution

In the P-GRADE parallel application development environment the user designs and develops an application by using native source code and graphical annotations. Based on this information the executable (binary) is implemented i.e. generated and compiled (see Figure 15). Details about how the generation and compilation is performed can be found in [35].

The proposed checkpointing method defined in section 2.2 is adopted in order to implement library(user-) level, fully automatic self-checkpointing for Grapnel applications. Details are introduced in this section.

3.1.3.1 Overview of the solution

The solution of design is driven by the following key features of the grapnel application:

- A built-in additional coordinator process to maintain the process creation and building up the connection among the processes i.e. creating the layout of the application
- An existing additional software layer in the application (explained in the next few paragraphs) between the user source code and the underlying message passing layer servicing the seamless checkpoint integration and information layer about the application structure

Based on the key features detailed above the design of the grapnel application checkpoint can be summarised by the following key points, which is in alignment with the ClusterGrid checkpointing method defined in Section 2.2:

1. Library-level checkpointing of the individual processes is realised by an existing single-process checkpointing library called Ckpt[29], in consistency with Definition 1 of the ClusterGrid method.
2. Consistent – application wide - checkpoint is performed by using the well-known Chandy-Lamport [56] algorithm that provides a solution to avoid in-transit messages before checkpointing of the individual processes takes place. Every process of the application executes the algorithm and creates its checkpoint. This approach is consistent with Definition 2 of the ClusterGrid method.
3. The created checkpoint information is saved into files in the working directory of the application by the user processes or optionally (in case a shared working directory does not exist) by a checkpoint server that can be integrated (if needed) into the built-in extra server process of the grapnel application. This consideration is consistent with Definition 3 of the ClusterGrid method.
4. Coordinated checkpoint is performed by the built-in server process which is called grapnel server. This consideration is consistent with Definition 4 of the ClusterGrid method.

5. By default the grapnel application has a built-in manager process which is going to be extended to perform the checkpoint coordination required by the application. This consideration is consistent with Definition 5 of the ClusterGrid method.
6. Coordination process (grapnel server) schedules the suspending of processes, synchronisation of messages, checkpointing, termination and re-spawn of migrating processes to enable migration of processes without terminating the entire application. This consideration is consistent with Definition 6 of the ClusterGrid method.
7. All the checkpointing support is integrated in a seamless way into the grapnel library of the application, which resides between the application source code and the message passing e.g. PVM library. The designed grapnel application in the P-GRADE environment is considered to be the original source code of the application. Adding the checkpoint support to it does not require even a bit of change in the application, only re-linking is performed. This consideration is consistent with Definition 7 of the ClusterGrid method.
8. Checkpoint related information and checkpoint control messages are distributed by the grapnel server through the process creation and through the hidden communication channels between the user processes and the grapnel server. It is needed to perform consistent checkpointing.
9. Existing extra software layer (GRAPNEL) between the user code and the underlying message passing layer enables the altering of the communication primitives, where required.
10. Code generation and compilation is done by the P-GRADE environment, extra libraries can be linked without user interaction. Using this feature the checkpoint libraries can be integrated into the application.

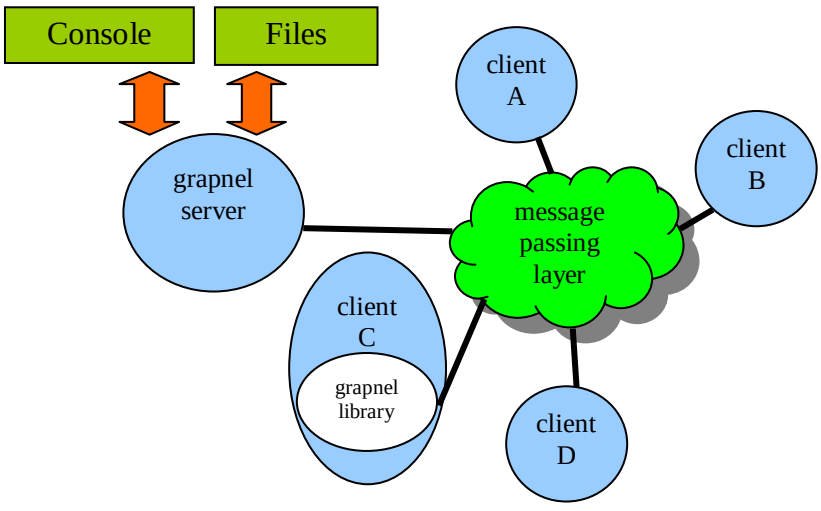

Figure 16 Structure of a GRAPNEL application generated by P-GRADE

The following sections detail the design and implementation of the Grapnel parallel checkpointing tool for the P-GRADE generated Grapnel applications.

3.1.3.2 Structure of the GRAPNEL checkpointing framework

In order to understand the operation of the grapnel checkpointing framework, a short overview is presented in the next paragraphs about the checkpoint-free operation of the grapnel application.

The P-GRADE compiler generates [35] the grapnel executables which contain the code of the client processes (signed by “client A, B, C, D” in Figure 16) defined by the user and an extra process (signed by “grapnel server” in Figure 16), called grapnel server which coordinates the run-time set-up of the application. The client processes at run-time logically contain the user code, the message passing primitives and the grapnel library (signed by “grapnel library” in Figure 16) that manages logical connections among them. To set-up the application, first the grapnel server starts and then it creates the client processes containing the user computation. As a result of the co-operation between the grapnel server and grapnel library the message passing communication topology is built up. To access all necessary input-output files and the console (see Figure 16), the server acts on behalf of the client processes. The client processes send requests to the server for reading and writing files and console and the necessary data are transferred when the action is finished by the server on its master host.

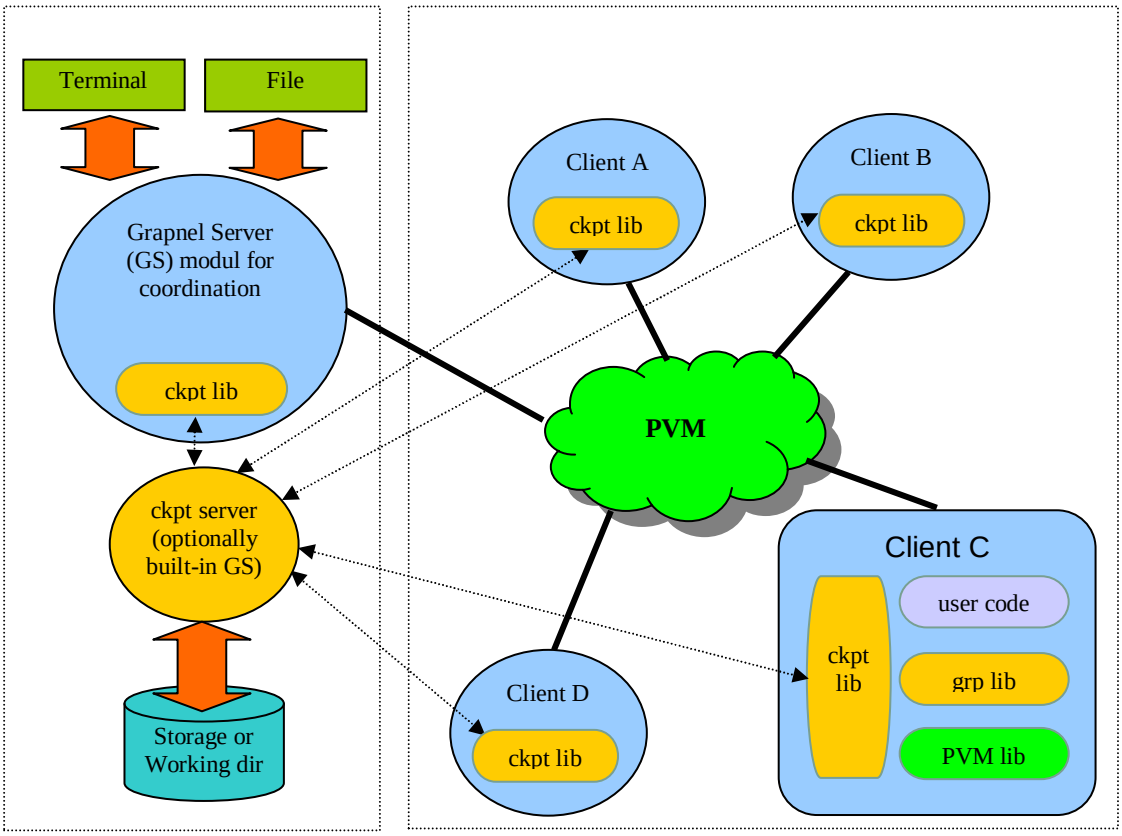

Figure 17 Structure of the Grapnel application in checkpoint mode

Based on the previously introduced application structure, the checkpoint version (see Figure 17) of the grapnel application contains the following elements:

- Grapnel Server (GS) (signed by “Grapnel server module for coordination” in Figure 17): an extra co-ordination process that is part of the application and generated by P-GRADE. It sets up the application by spawning the

processes and defining the logical communication topology for them. Checkpoint extension of GS coordinates the checkpointing activities of the client processes e.g. executes the checkpoint protocol at initialisation, saving and resumption phases.

- Grapnel Library (GL) (signed by “grp lib” in Figure 17): a layer between the message passing library and the user code, automatically compiled with the application, co-operates with the server, performs preparation for the client process environment and provides a bridge between the server process and the user code. Checkpoint extension of this library in client processes prepares for checkpoint, performs synchronisation of messages and re-establishes connection to the application after a process is rebuilt from checkpoint.
- Checkpoint Server (CS) (signed by “Ckpt Server” in Figure 17): a component that receives data via socket and puts it into checkpoint file of the storage and vice versa. This component is optional; functionality can be built in GS.
- Dynamic checkpoint library (CL) (signed by “ckpt lib” in Figure 17): loaded at process start-up and activated by receiving a checkpoint event, reads the process memory image, creates the checkpoint image and passes this image to the CS. CL can be transformed into static library as well in order to be part of the application or can be transported as a working file of the application.
- Storage is a virtual component and can be established locally or remotely. It is a repository for checkpoint files which can be of any type. The simplest solution is using the working directory of the application if it is shared among the nodes.
- PVM is the message passing layer of the application that performs process management on PVM level and transfers messages among the GL and GS components dealing with user data, grapnel-level process management and checkpoint coordination.
- PVM lib is the client side part of the PVM message-passing layer i.e. provides access for PVM services through function calls.

The resulted executable itself is a PVM application. The grapnel server process with clients A, B, C, D (see Figure 17) form the application. Processes are connected through the PVM system and each contains an individual process checkpointer library linked to them. The necessary checkpoint protocol is executed among the GS and the GL components of the client processes. Grapnel protocol messages are transported by PVM, but are invisible for the user code.

3.1.3.3 The GRAPNEL checkpointing protocol

Before starting the execution of the application, an instance of the Checkpoint Server (CS) is running in order to transfer checkpoint files to/from the dynamic checkpoint libraries (CL) linked to the application. CS is optional component since in systems where common working directory exists among the hosts checkpoint file saving and loading is performed locally. Otherwise GS plays this role. In the rest of this section, CS and GS will be referenced individually to split functionalities clearly.

The overall execution of application with checkpointing support can be divided into 5 main phases, which are detailed in this section.

1. Initialisation
2. Interruption
3. Synchronisation
4. Saving
5. Resumption

Initialisation phase

When the application is launched, the first process that comes to live is the GS performing the coordination of the Client processes. After initialisation it starts spawning the Client processes. Whenever a process comes to live, it first gets connected to GS in order to download parameters, settings, etc. When each process has performed the initialisation, GS instructs them to start execution or start the resumption depending on the existence of the checkpoint information.

Each (user-defined) process of the application at start-up loads automatically the CL that is going to perform the single process checkpointing or resumption later. The processes connect to GS and wait for decision, whether to perform normal start-up or initiate resumption. If GS decides to initiate resumption, user-defined process continues with restoring its memory, otherwise it starts the execution from the beginning.

Interruption phase

While the application is running and the processes are performing the user defined computation the checkpoint mechanism is inactive. When a checkpoint is required, GS instructs the client processes by sending a message/signal pair to them. This step of the protocol is more detailed in section 3.1.3.4. When a process is successfully notified about the start of checkpointing, the execution of the user code is suspended and the grapnel library is started instead.

The GL first notifies GS about the successful interruption and waits for further instruction. The interruption phase finishes when all the client processes of the application are interrupted successfully.

Synchronisation phase

The Grapnel server initiates the synchronisation phase by sending a message to all the processes of the application. This message notifies the processes to start the synchronisation and contains a list of process identifiers. Having received the identifiers each process executes the well-known Chandy-Lamport [56] algorithm.

The algorithm executes the following protocol. Each process first sends an end-of-channel message to the processes that were included in the synchronisation notification message before. After that, the sender starts listening on the message channels one after the other. When receiving a message two different actions might be performed. First, in case of receiving a user generated message i.e. the message is part of the application protocol designed by the user, the message is stored in the memory and the channel is repeatedly checked by the process. Second, in case of receiving an end-of-channel message, the channel is clean i.e. no message is currently transported

on this channel. Therefore, the next unclear channel is started to be checked by the process.

When end-of-channel messages arrived through all the channels to a process and each processes of the application got this from all partners, the underlying communication system demonstrably does not hold any message sent by/to be delivered to any of the processes of the application. In that case the communication system reached a consistent state regarding message delivery. At this point the synchronisation phase is finished, which is signed by a ready-to-save message sent by each process to the GS.

Saving phase

The Grapnel server initiates the low-level checkpoint saving phase by sending a do-save instruction message to all Grapnel client processes. In this phase the main step is to store the memory image map. This operation is done by CL. Before this operation the last step is to leave the message-passing (i.e. PVM) subsystem. When the process successfully exits from PVM, the control is taken by CL which collects all memory segments and necessary information and stores them into a file or optionally sends it to the checkpoint server. After the checkpoint information is successfully stored, the processes might continue their execution or terminate immediately. If the application is started to resume its state, all processes turns to resumption phase.

Resumption phase

In the resumption phase after the process started (spawned by GS or continued after checkpointing), the first step is to establish connection with the message-passing (i.e. PVM) system. At this point the process gets a new identifier. After successful connection, the process notifies GS about its new identifier and waits for permission to continue the execution. When all the processes are ready to run, GS notifies them to continue their execution.

There are two different ways to continue the execution, depending on the point where the process was interrupted. If the process got the checkpoint signal while it was executing computation, the resumption simply means to continue the execution from the point where it was interrupted by the signal. If the process got the checkpoint signal while it was executing receive operation, the communication must be repeated since the receive operation was aborted. (The situation how invalid user messages are generated explained in details in Section 3.1.3.4.) Send operation cannot be interrupted, since it is made atomic by disabling interruption.

After each successful resumption, whenever a process executes the receive operation, existing messages (stored in the memory at a previous checkpoint) must be scanned through. In case the message (the process is waiting for) is found, it is removed from the list and passed to the user code, so the receive action is simulated. In case appropriate message is not found in the list, a real communication is executed.

3.1.3.4 Checkpoint aware communication primitives

In order to prevent communication from malfunctioning (e.g. losing messages or duplicating message), it is necessary to prepare the communication primitives for proper handling of the underlying message-passing subsystem.

In order to checkpoint the application, the execution of all user processes must be interrupted. The interruption might happen at the time of executing communication

and non-communicating operation. In the latter case the interruption does not cause any problem regarding the message-passing layer.

The former case might cause faulty operation. In the PVM message-passing system the target process is identified by a task id (TID). Whenever a process connects, a new identifier is associated even if the process was already member of the communicating processes previously i.e. disconnection and reconnection steps are performed. This fact might cause an initiated receive operation to be unfinished forever or finished with invalid message in case the target process performs a disconnection and reconnection. Reconnection is inevitable when checkpointing or resuming a process of a parallel application.

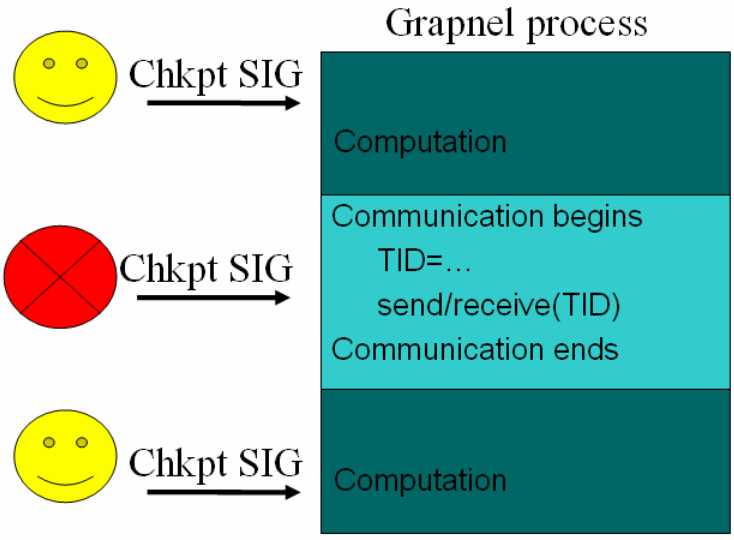

Figure 18 Guarded communication primitives to prevent interruption

The receive operation stores the TID of the target process and it is not possible to modify the TID that has been passed for the communication call. In order to prevent the application to get to this situation a process that is still executing a communication operation (see Figure 18) must not be interrupted. The communication primitives are therefore defined as atomic operations.

Examine the following situation: an application consists of only two processes M and S. Process M is the master process and S is the slave process. Process S is performing a calculation and at the same time Process M is waiting for the result i.e. executing a receive operation. At this point a checkpoint is required which start with the ‘interruption’ phase. Process S is successfully interrupted, but Process M is impossible to be interrupted, since communication is atomic. Process M is waiting for the result from Process S in an endless loop. This need to be resolved somehow, because checkpointing may only work properly if each user process is successfully interrupted.

Solution is based on the fact that the receive operation must be forced to be finished. A possible way to do this is to make the receive operation accept messages from a source different from the specified. To implement this mechanism three requirements are needed to be fulfilled:

1. support for wildcard receive operation by the MP layer
2. existence of method for communication primitive redefinition

3. existence of a different source process

For PVM based GRAPNEL applications the requirements defined above can be fulfilled by the support of the GRAPNEL layer. The P-GRADE code generator redefines every receive operation to act as a wildcard receive operation and inserts the Grapnel Server's identifier as an extra possible source of the message (see Figure 19). In this case if a process is stuck into a receive operation and no messages are in the queue the Grapnel Server is able to force the process to continue its execution by sending a checkpoint message to the process. Obviously, in this case, the receive operation should be repeated after the resumption of the checkpointed process (see Figure 19), since the process has got an invalid message from its point of view just before the checkpoint procedure.

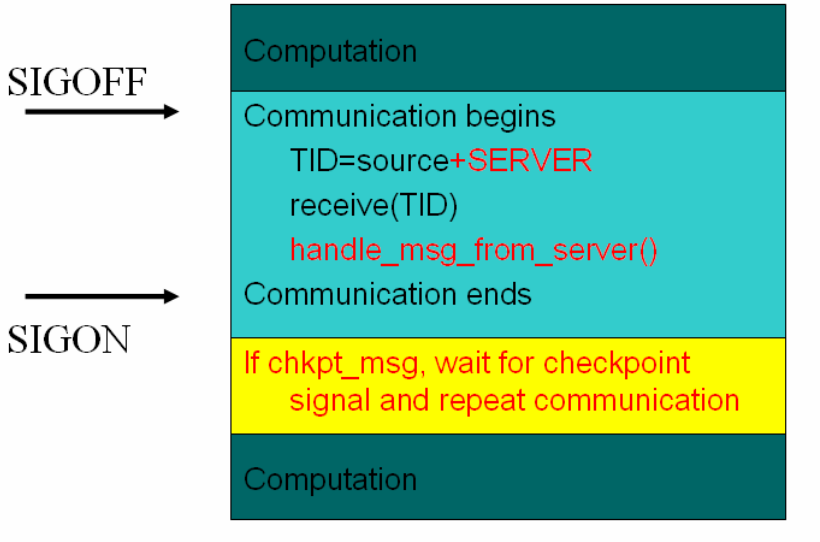

Figure 19 Modification of receive communication primitives to multi-receive

As a summary, there are two situations for interruption. In the first case when the user process is in computing phase (i.e. executing non-communicating code) a unix signal works perfectly for interruption. In the second case the user process is executing communication method and the checkpoint message forces the process to finish the operation. In both cases the server performs interruption and messaging in a combined way.

The latter technique is required only when the process is receiving a message. In case of send operation there is no need for the second technique since the send operation must finish in a limited/short period because send operation is non-blocking in this environment. Blocking and synchronized send operation is realised by a non-blocking send and blocking receive communication pair where the receive operation is waiting for an acknowledgement. The communication primitive that performs the receiving of an acknowledgement is also implemented like it is explained above.

3.1.3.5 Interruption and consistent cut

To checkpoint a message-passing parallel application, the state of the individual processes and the state of the communication channel must be collected. The states of the components forms the so called ‘system state’. Whenever a system is checkpointed, a snapshot of each component are created and stored.

Components of a message-passing application in a parallel environment are executed in a concurrent way. Each process and the message-passing subsystem are using the same resources to perform its job. In a concurrent system where snapshot of communicating processes is taken, key factor is the time elapsed between the snapshots of the individual components.

The ideal approach is that the snapshot of the individual components to form the 'system state' lasts exactly zero time. In practice this is not possible. Each snapshot takes time greater than zero. To create a snapshot of the 'system state' i.e. global snapshot, individual snapshots must be created for the components one after the other. Since time – greater than zero - elapses between the snapshot of the individual components, a 'system state' might be invalid, because components are changing their state by the time the snapshot of the other components are taken. This state is invalid because inconsistency occurs among the state of the components. The global snapshot forms a consistent system state or consistent cut if the following conditions are satisfied:

C1. Every message that has been received has also been sent in the state of the sender.

C2. Each message that has been sent has also been received in the state of the receiver.

For example, the cut T1 (see Figure 20) is consistent, but T2 and T3 are inconsistent. Condition C2 is not satisfied for message D in case of cut T2 and T3. At the same time message E is also against condition C1 in case of cut T3, since the state of process P3 shows that message E is received, but in the state of process P1 message E is not yet sent, which is called 'orphan' message.

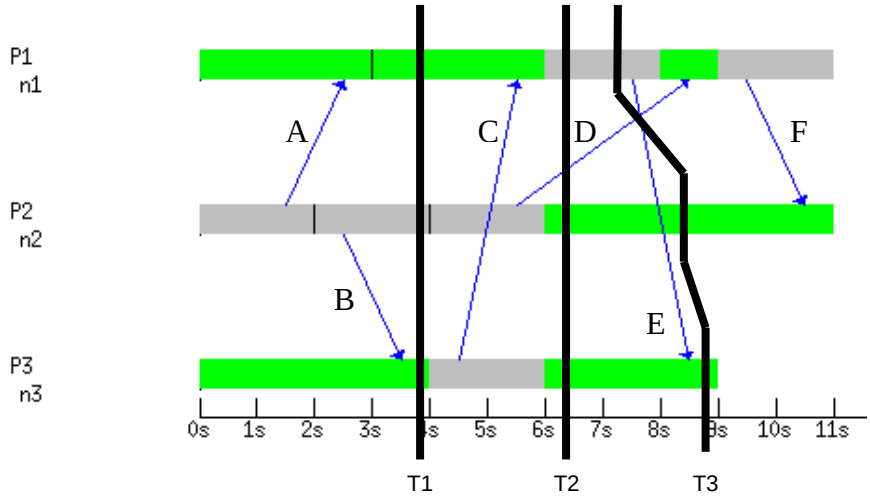

Figure 20 Consistent and inconsistent cuts in a system state

Orphan messages [49] cannot be part of a consistent system state. Messages that are sent but not yet received are called in-transit messages which can be removed from the system state in order to make T2 a consistent cut.

The consistent cut in the Grapnel checkpointing framework are created in a way that both conditions defined above are satisfied. Condition C1 is satisfied by

using the so-called blocking coordinated checkpointing. Consistent cut is formed by the points in the time when the notification from the GS reaches the user processes. Whenever a process is notified, it interrupts the execution and does not continue until all checkpoint related activities finish.

To satisfy condition C2, in-transit messages must be removed. This is realised by an extra end-of-channel notification sent by every process to all its neighbours. In case there is a reliable communication layer where delivering messages is done in FIFO order, receiving the end-of-channel message means no more messages are on its way to the receiver from the source process.

3.1.3.6 Redesigned communication algorithms

One of the key solutions in parallel checkpointing of the grapnel applications is to update the receiver communication primitives of the GL in the user processes in a way which provides interruptible receiver primitives on grapnel level and at the same time it is atomic i.e. non-interruptible on PVM level. The problem and the proposed solution is already introduced in section 3.1.3.4. The following few paragraph details the technique how this feature is implemented in the grapnel layer.

The main communication primitives altered in the GL are:

1. asynchronous send operation
2. synchronous send operation
3. blocked receive operation
4. blocked wildcard (alternative) receive operation

The proposed solution detailed in section 3.1.3.4 redefines the primitives to wildcard receiving operation that waits for any message. From the list above there are three primitives requires redefinition: blocked alternative receive operation, blocked receive operation and synchronous send operation. The former two are straightforward, the last one requires redefinition because a synchronous send is implemented by an asynchronous send-receive operation, where the receive operation is to wait for an acknowledgement.

As a reference, the pseudo-like code original (checkpoint-free) algorithm performing the receiver operation in the GRAPNEL processes is introduced below:

```
1 GRP_RECEIVE_FUNC_BEGIN( sendertype, source)
2       grp_set_comm_parameters() //setting communication parameters
3       grp_set_remote_tid(source) //gets tid of source process
4       exec_receive(source) //performing receive operation from the source process
5       extract_msg_from_buffer() //unpacks content of message
6       IF(sendertype==blocked)
7               send_acknowledgement() //sending acknowledgement
8 GRP_RECEIVE_FUNC_END
```

To enable the receiver operations of the processes to be interruptible, the following algorithm has been designed:

```
1 GRP_RECEIVE_FUNC_BEGIN( sendertype, source)
2    IF(chkpt_mode)
3         checkpoint_interrupt_disable()
4    grp_set_comm_parameters() //setting communication parameters
5    DO
6         chkpt_msg=NULL
7         from_queue=FALSE
8         grp_set_remote_tid(source) //gets tid of source process
9         IF(!chkpt_mode)
10                exec_receive(source) //performing receive operation from the source process
11        ELSE
12                IF (grp_chkpt_restore_stored_msg(source) < 0) //checking stored messages
13                        set_receive_target(server+source) // sets  matching function
14                        exec_multiple_receive() // performs matching and set chkpt_msg flag
15                        reset_receive_target() //resets matching function
16                ELSE
17                        from_queue=TRUE;
18                IF(chkpt_msg)
19                        chkpt_get_proc_list()  // receiving chkpt msg, unpack and store proc list
20        IF(!chkpt_msg)
21                IF(from_queue)
22                        extract_msg_from_queue() //unpacks content of message
23                ELSE
24                        extract_msg_from_buffer() //unpacks content of message
25                IF(sendertype==blocked)
26                        send_acknowledgement() //sending acknowledgement
27        IF(chkpt_mode&&chkpt_msg)
28                checkpoint_interrupt_enable()
29                        wait_for_checkpoint_interrupt()    -->checkpoint interrupt activates here
30                                                           -->saving and restoration returns here
31                checkpoint_interrupt_disable()
32    WHILE(chkpt_mode&&chkpt_msg)
33    IF(chkpt_mode)
34         checkpoint_interrupt_enable()
35 GRP_RECEIVE_FUNC_END
```

The routine „grp_receive" is modified to be interruptible. The main modifications are as follows:

- disable and enable checkpoint interruption at the enter and exit point of the function (lines 2-3 and 33-34)
- searching for appropriate message in the message queue storing in-transit message caught at the previous checkpoint(s); (line 12)
- performing wildcard receive operation with the server process identifier added into the list of acceptable input sources; (line 13-14)
- receives and stores process list (part of checkpoint procedure) stored in the checkpoint message; (line 18-19)
- in case the message is taken from the memory i.e. from the message queue, different way of message unpack is required; (line 21-22)
- in case the message is a checkpoint message the overall receive operation must be interrupted and checkpoint should be performed by simply letting the checkpoint interruption to realise; (line 28-31)

In order to make the Grapnel application be interruptible at any time, the pattern of program code – based on the solution proposed in section 3.1.3.4 and introduced above - is built into all operations performing any receive action.

3.1.4 Definition of the $CP_{grapnel}$ ASM model

In this section a new ASM model called $CP_{grapnel}$ is introduced, which is a refinement [37] of the CP_{ground} ASM model. Refinement for $CP_{grapnel}$ means to elaborate a model on the basis of CP_{ground} model that is less abstract i.e. more concrete in point of its execution. The aim of the refinement is to show that the Grapnel checkpointing framework implements the same features and behaviour as CP_{ground} model defines.

In $CP_{grapnel}$ the following points are elaborated in a more detailed way:

- message types are introduced, messages are distinguished depending on serve as application messages or checkpoint control messages
- interrupt of processes are elaborated based on the checkpoint control messages
- synchronisation of message channels are modelled
- storing application messages are modelled
- static process model of the GRAPNEL application is implemented in the model

3.1.4.1 Universes and signatures

Universes and signatures for $CP_{grapnel}$ are the same as it is defined for (inherited from) CP_{ground}. In this section the differences are detailed.

In $CP_{grapnel}$ events have been redefined. First, the EVENT called *interruption* has been introduced to model the interruption phase of parallel checkpointing based on control messages. Furthermore, the event called *checkpoint* has been redefined in order to distinguish message synchronisation and checkpoint saving activities. Therefore the universe EVENT is defined as follows: *EVENTS={spawn, send, receive, terminate, interrupt, checkpoint(synch, save), resume, exit}*.

In order to implement message synchronisation a function has been introduced to mark the actual state of the synchronisation process. The function is defined in the following way: *eoch: (PROCESS × PROCESS) → {true, false}*. (note: "*eoch*" stands for "end of channel"). This function returns *true* if synchronisation among two processes has been successfully finished, *false* if synchronisation is being done and *undef* if it has not been started yet.

To model message handling, message types are introduced to distinguish application and control messages. Application messages are always sent among user defined processes and are necessary for proper operation of the processes. Control messages are sent by the checkpointing tool and are hidden from the programmer. Application message type is called *userdefined*, while control messages are defined to mark the beginning or the end of an activity to be performed. Therefore the type is defined as follows: *type: MESSAGE → {userdefined, interrupted, endofchannel, synchronised, saved, resumed, exiting, exited }*.

In this model the universe MESSAGE contains the messages sent among the processes. It can be considered as a message queue handled by the underlying communication layer. Supplementary to this, a new universe (*universe MESSAGE_STORE*) is introduced which stores messages, but the represented queue is implemented in the memory of the process. Therefore, functions defined on the

MESSAGE universe are also defined for the MESSAGE_STORE universe in CP_{ground}: (*from: MESSAGE_STORE→PROCESS*) to represent the source, (*to: MESSAGE_STORE→PROCESS*) to represent the target and (*type: MESSAGE_STORE→{userdefined, interrupted, endofchannel, synchronised, saved, resumed, exiting, exited }* to represent the type of the message.

Note, that this model follows the static process model of GRAPNEL, since spawning new processes at runtime is not allowed in P-GRADE [1][48]. The model uses a predefined set of processes (*universe GRP_DEF_PROC_LIST*) that is assumed to be created by the P-GRADE environment at the time of compilation.

All the universes and functions that are not mentioned in the previous paragraphs – but used in the model – are defined in sections 2.1.5.2 and 2.2.4.1, respectively.

3.1.4.2 Initial state

Initial state of the $CP_{grapnel}$ model is equivalent with the initial state of CP_{ground} which is as follows:

```
∃p∈PROCESS: app(p)≠undef,
            phase(p)=init,
            role(p)=undef,
            master(p)=undef,
            startupmode(p)=undef
```

Detailed explanation of parameters can be found in section 2.2.4.2.

3.1.4.3 Rules

1. Rules for initialisation

Two of the three initialisation rules for $CP_{grapnel}$ are equal to the ones defined for CP_{ground}: **CPgrapnel-R1a≡CPground-R1a, CPgrapnel-R1b≡CPground-R1b**

Since static process model is followed by GRAPNEL, all processes must be spawned at initialisation. These processes have been defined by the programmer (GRP_PROC_DEF_LIST) in P-GRADE. The rule R1c is modified accordingly.

CPgrapnel-R1c (modified version of CPground-R1c)

```
if role(p)=coordinator & phase(p)=waiting & startupmode(p)≠undef then
      if startupmode(p)=normal then
            phase(p):=running
            process_to_store:={}
            process_to_checkpoint:={}
            process_to_terminate:={}
            process_to_resume:={}
            do forall x : x ∈ GRP_DEF_PROC_LIST
                  extend PROCESS by child with
                        app(child):=app(p)
                        phase(child):=init
                        role(child):=userdefined
                        startupmode(child):=undef
                        master(child):=false
                  endextend
            enddo
      else
            phase(p):=resuming
            event(p):=resume
      endif
endif
```

2. Rules for process spawning

Since, there is no process creation during execution – initiated by the programmer - the corresponding rules are disabled by performing a *skip* operation.

CPgrapnel-R2a (modified version of CPground-R2a)

```
if role(p)=userdefined & phase(p)=running & event(p)=spawn then
      skip
endif
```

CPgrapnel-R2b (modified version of CPground-R2b)

```
if role(p)=userdefined & phase(p)=waiting &
   startupmode(p)=normal & process_to_checkpoint(p)=false
   then
      skip
endif
```

3. Rule for sending a message

Refinement in this rule introduces the type of messages.

CPgrapnel-R3a (modified version of CPground-R3a)

```
if role(p)=userdefined & phase(p)=running &
   event(p)=send(dest)
   then
      extend MESSAGE by msg with
            from(msg):=p
            to(msg):=dest
            type(msg):=userdefined
      endextend
endif
```

4. Rules for receiving a message

Receiving a message in $CP_{grapnel}$ model is realised by a two-phase message checking. First, messages stored in the memory (see definition of MESSAGE_STORE in section 3.1.4.1) are checked, and then real communication is performed. In-transit messages just before checkpointing are elements of this universe and after resumption of a process these elements are removed whenever a receive operation is called by the user code. Rules CPgrapnel-R4a and R4b are modified in this way.

CPgrapnel-R4a (modified version of CPground-R4a)

```
if role(p)=userdefined & phase(p)=running &
   event(p)=receive({source|any})
   then
      if (∃msg∈MESSAGE_STORE):to(msg)=p & {from(msg)=source| }
         then
            MESSAGE_STORE(msg):=false
      else if (∃msg∈MESSAGE):to(msg)=p & {from(msg)=source| }
         then
            MESSAGE(msg):=false
      else
            expecting(p):={source|any}
            phase(p):=receive_waiting
      endif
endif
```

CPgrapnel-R4b (modified version of CPgroundR4b)

```
if role(p)=userdefined & phase(p)=receive_waiting
   then
      if (∃msg∈MESSAGE_STORE):to(msg)=p & {from(msg)=expecting(p)| }
         then
            MESSAGE_STORE(msg):=false
            phase(p):=running
            expecting(p):=undef
      else if (∃msg∈MESSAGE):to(msg)=p & {from(msg)=expecting(p)| }
         then
            MESSAGE(msg):=false
            phase(p):=running
            expecting(p):=undef
      endif
endif
```

5. Rules for interrupting the execution

Whenever a process gets an *exit* event, application checkpointing must be performed. Only coordinator can initiate checkpointing, therefore user defined processes must notify it in case they get the event. Notification is sending an *exiting* message to the coordinator. Rule CPgrapnel-R5a realises this operation.

CPgrapnel-R5a (newly introduced)

```
let coord=c∈PROCESS:role(c)=coordinator & app(c)=app(p)
if role(p)=userdefined & phase(p)=running & event(p)=exit
   then
      if master(p)=true then
            event(coord):=exit
      else
            extend MESSAGE by m with
                  from(m):=p
                  to(m):=coord
                  type(m):=exiting
            endextend
      endif
endif
```

Whenever the coordinator gets an *exiting* message first time, it initiates the interrupt of all the processes of the application by sending them an *interrupt* event.

CPgrapnel-R5b (newly introduced)

```
let allproc:={∀cp∈PROCESS:app(cp)=app(p) & cp≠p}
if role(p)=coordinator &
      ((∃msg∈MESSAGE):to(msg)=p & type(msg)=exiting)
   then
      process_to_terminate(from(msg)):=true
      process_to_resume(from(msg)):=true
      MESSAGE(msg):=false
      if process_to_checkpoint={} then
            process_to_checkpoint:=allproc
            if (∀p'∈PROCESS): p'≠p & app(p')=app(p) &
                  process_to_checkpoint(p')=true &
                  phase(p')={running|receive_waiting}
               then
                  do forall pp∈PROCESS:process_to_checkpoint(pp)=true
                        event(pp):=interrupt
                  enddo
            endif
      endif
endif
```

The next rule handles the case when the coordinator gets an *exit* event. It means application-wide checkpoint and shutdown must be performed including the

coordinator. Therefore all processes marked to be checkpointed and terminated and they are all notified. Changing the phase to checkpointing indicates the need for checkpointing the coordinator.

CPgrapnel-R5c (newly introduced)

```
let allproc:={(∀cp∈PROCESS):app(cp)=app(p) & cp≠p}
if role(p)=coordinator & phase(p)=running & event(p)=exit
   then
      process_to_checkpoint:=allproc
      process_to_terminate:=allproc
      process_to_resume:={}
      process_to_store:=allproc
      if (∀p'∈PROCESS): process_to_checkpoint(p')=true &
                        phase(p')={running|receive_waiting}
         then
            do forall pp∈PROCESS: process_to_checkpoint(pp)=true
                  event(pp):=interrupt
            enddo
      endif
      phase(p):=checkpointing
endif
```

Interruption is modelled in $CP_{grapnel}$ by changing its phase to checkpointing when *interrupt* event is detected and coordinator is notified by an *interrupted* message. The execution of the user code becomes suspended.

CPgrapnel-R5d (newly introduced)

```
let coord=c∈PROCESS:role(c)=coordinator & app(c)=app(p)
if role(p)=userdefined & phase(p)={running|receive_waiting} &
   event(p)=interrupt
   then
      phase(p):=checkpointing
      extend MESSAGE by m with
            from(m):=p
            to(m):=coord
            type(m):=interrupted
      endextend
endif
```

6. Rules for message synchronisation among the processes

Message synchronisation begins when all user defined processes have been successfully terminated. If the coordinator detects this case, it notifies all the processes by sending a *checkpoint(sync)* event with the process list as parameters. The function called *eoch* is also initialised to mark the beginning of the synchronisation activity.

CPgrapnel-R6a (newly introduced)

```
let N:=sizeof({(∀cp∈PROCESS): app(cp)=app(p) & cp≠p})
if role(p)=coordinator &
   (∃msg1,…,msgN∈MESSAGE):(∀i∈[1..N]:to(msgi)=p &
   type(msgi)=interrupted & process_to_checkpoint(from(msgi))=true)
   then
      do forall proc : proc∈PROCESS &
                        process_to_checkpoint(proc)=true
            MESSAGE((m∈MESSAGE):from(m)=proc &
                        type(m)=interrupted):=false
            event(proc):=checkpoint(synch, process_to_checkpoint)
            do forall p' : process_to_checkpoint(p')=true
                  eoch(proc,p'):=undef
            enddo
      enddo
endif
```

This rule ensures that each user defined process starts the message synchronisation by sending an end-of-channel marker message to all other process.

CPgrapnel-R6b (newly introduced)

```
if role(p)=userdefined & phase(p)=checkpointing &
   event(p)=checkpoint(synch,proclist) &
   (∀proc∈PROCESS, proclist(proc)=true:eoch(p,proc)=undef)
   then
      do forall neighbour : neighbour ∈ {proclist\p}
            extend MESSAGE by m with
                  from(m):=p
                  to(m):=neighbour
                  type(m):=endofchannel
            endextend
            eoch(p,neighbour):=false
      enddo
      eoch(p,p):=true
      event(p):=checkpoint(synch,proclist)
endif
```

During message synchronisation every process read the messages from all the other ones, until the end-of-channel marker message is detected. Userdefined messages are stored in the memory to be part of the checkpoint image of the process.

CPgrapnel-R6c (newly introduced)

```
if role(p)=userdefined & phase(p)=checkpointing &
   event(p)=checkpoint(synch,proclist) &
   ((∃proc∈proclist):eoch(p,proc)≠undef) &
   ((∃msg∈MESSAGE):proclist(from(msg))=true)
   then
      if type(msg)=userdefined then
            extend MESSAGE_STORE by mb with
                  from(mb):=from(msg)
                  to(mb):=to(msg)
                  type(mb):=userdefined
            endextend
            MESSAGE(msg):=false
      else if type(msg)=endofchannel then
            eoch(p,from(msg)):=true
            MESSAGE(msg):=false
      endif
      event(p):=checkpoint(synch,proclist)
endif
```

A process finishes message synchronisation when it got all end-of-channel marker messages from all the other processes. In this case, the coordinator is notified by a *synchronised* message.

CPgrapnel-R6d (newly introduced)

```
let coord=c∈PROCESS:role(c)=coordinator & app(c)=app(p)
if role(p)=userdefined & phase(p)=checkpointing &
   event(p)=checkpoint(synch,proclist) &
   (∀proc∈proclist:eoch(p,proc)=true)
   then
      extend MESSAGE by m with
            from(m):=p
            to(m):=coord
            type(m):=synchronised
      endextend
endif
```

7. Rules for checkpoint saving of processes

After message synchronisation, the next step is to save the checkpoint image. Coordinator instructs every *userdefined* process one by one to save its state in case synchronisation finished.

CPgrapnel-R7a (modified version of CPground-R5c)

```
if role(p)=coordinator &
   (∃msg∈MESSAGE: to(msg)=p & type(msg)=synchronised &
   process_to_checkpoint(from(msg))=true))
   then
      MESSAGE(msg):=false
      event(from(msg)):=checkpoint(save)
endif
```

Checkpoint image creation is performed by a *userdefined* process when it got a *checkpoint(save)* event. The SINGLE_PROCESS_STATE_CHECKPOINT macro performs the state saving operation. The internal details of this macro are irrelevant, since this model focuses on the distributed nature of checkpointing.

CPgrapnel-R7b (modified version of CPground-R6a)

```
let coord=c∈PROCESS:role(c)=coordinator & app(c)=app(p)
if role(p)=userdefined & phase(p)=checkpointing &
   event(p)=checkpoint(save)
   then
      extend IMAGE by imagefile with
            imagefileofapp(imagefile):=app(p)
            SINGLE_PROCESS_STATE_CHECKPOINT(p,imagefile)
      endextend
      extend MESSAGE by m with
            from(m):=p
            to(m):=coord
            type(m):=saved
      endextend
endif
```

8. Rules for terminating the processes

The next rule is fired when a process successfully performed saving the checkpoint image. In this case coordinator decides whether the process should be terminated or must wait for further instruction.

CPgrapnel-R8a (modified version of CPground-R6a)

```
if role(p)=coordinator &
   ∃msg∈MESSAGE:to(msg)=p & type(msg)=saved
   then
      if process_to_terminate(from(msg))=true)) then
            event(from(msg)):=terminate
      else
            phase(from(msg)):=waiting
      endif
      MESSAGE(msg):=false
endif
```

Rule R8b ensures that a user defined process starts the *termination* phase whenever a *terminate* event is delivered. In this case the coordinator is notified.

CPgrapnel-R8b (modified version of CPground-R7c)

```
let coord=c∈PROCESS:role(c)=coordinator & app(c)=app(p)
if role(p)=userdefined & phase(p)={running|checkpointing} &
   event(p)=terminate then
      extend MESSAGE by m with
            from(m):=p
            to(m):=coord
            type(m):=exited
      endextend
      phase(p):=terminating
endif
```

Next rule is fired at normal termination. The condition is evaluated to true, when every process has exited, but no resumption is needed. In this case all checkpoint information is removed (assuming, it is not needed if execution completed).

CPgrapnel-R8c (modified version of CPground-R7b)

```
if role(p)=coordinator & phase(p)=running &
   (∃msg1,…,msgN∈MESSAGE:(∀i∈[1..N]:to(msgi)=p &
   type(msgi)=exited & process_to_resume={}
   then
      do forall imagefile : imagefile∈IMAGE &
                              imagefileofapp(imagefile)=app(p)
            IMAGE(imagefile):=false
      enddo
      PROCESS(p):=false
endif
```

Next rule fires at the end of the application-wide checkpoint and shutdown operation. When every process has exited, coordinator creates its checkpoint and exits.

CPgrapnel-R8d (modified version of CPground-R7b)

```
if role(p)=coordinator & phase(p)=checkpointing &
   (∃msg1,…,msgN∈MESSAGE:(∀i∈[1..N]:to(msgi)=p &
   type(msgi)=exited & process_to_resume={}))
   then
      extend IMAGE by imagefile with
            imagefileofapp(imagefile):=app(p)
            SINGLE_PROCESS_STATE_CHECKPOINT(p,imagefile)
      endextend
      PROCESS(p):=false
endif
```

This rule ensures that processes in termination phase are removed from the *PROCESS* universe.

CPgrapnel-R8d (newly introduced)

```
if role(p)=userdefined & phase(p)=terminating then
      PROCESS(p):=false
endif
```

9. Rules for resuming the processes

The rule that ensures the resumption of the coordinator and the spawning of the userdefined processes in $CP_{grapnel}$ is equivalent to the one defined in CP_{ground}. Therefore

CPgrapnel-R9a (≡CPground-R8a)

After a process exited and it must be resumed, a new process is created by the coordinator. This activity is realised by the rule CPgrapnel-R9b.

CPgrapnel-R9b (modified version of CPground-R8b)

```
if role(p)=coordinator & phase(p)=running &
   (∃msg∈MESSAGE: to(msg)=p & type(msg)=exited &
   process_to_resume(from(msg))=true)
   then
      MESSAGE(msg):=false
      extend PROCESS by child with
            app(child):=app(p)
            phase(child):=init
            role(child):=userdefined
            startupmode(child):=resume
            master(child):=false
      endextend
endif
```

When all the processes are in *waiting* phase, there are two alternatives. The first one is that a normal startup of the application has been performed. In this case the phases of processes must be changed to *running* to start the execution of the application. The second alternative is that there are processes that have been created and need to be resumed. In this case their phases are changed to *resuming* and an additional *resume* event is sent to notify them.

CPgrapnel-R9c (modified version of CPground-R8c)

```
if role(p)=coordinator & phase(p)=running &
   (∀proc∈PROCESS:app(proc)=app(p) & proc≠p & phase(proc)=waiting)
   then
      if process_to_resume={} then
            do forall pp : pp∈PROCESS & app(pp)=app(p) & pp≠p
                  phase(pp):=running
            enddo
      else
            do forall pp : pp∈PROCESS & process_to_resume(pp)=true
                  phase(pp):=resuming
                  event(pp):=resume
            enddo
      endif
endif
```

In *resuming* phase and with a *resume* event a process recovers its state from a checkpoint file. When done, the coordinator is notified.

CPgrapnel-R9d (modified version of CPground-R8d)

```
let coord=c∈PROCESS:role(c)=coordinator & app(c)=app(p)
if role(p)=userdefined & phase(p)=resuming & event(p)=resume
   then
      imagefile:=x∈IMAGE:imagefileofapp(x)=app(p)
      SINGLE_PROCESS_STATE_RESUME(p,imagefile)
      extend MESSAGE by m with
            from(m):=p
            to(m):=coord
            type(m):=resumed
      endextend
endif
```

The next rule ensures that an application continues its execution after some of its processes have been terminated, created and resumed i.e. migrated.

CPgrapnel-R9e (newly introduced)

```
if role(p)=coordinator & (∃pa∈PROCESS:app(pa)=app(p) & pa≠p &
   phase(pa)=resuming)
   then
      if (∀pb∈PROCES): app(pb)=app(p) & pb≠p &
         (phase(pb)=waiting |
         (phase(pb)=resuming & ((∃msg∈MESSAGE):from(msg)=pb &
         type(msg)=resumed))))
         then
            do forall pc : pc∈PROCESS & app(pc)=app(p) & pc≠p
                  if expecting(pc)=undef then
                        phase(pc):=running
                  else
                        phase(pc):=receive_waiting
                  endif
                  if (∃m∈MESSAGE):from(m)=pc & type(m)=resumed
                     then
                        MESSAGE(m):=false
                  endif
                  process_to_checkpoint(pc):=false
                  process_to_terminate(pc):=false
                  process_to_resume(pc):=false
            enddo
      endif
endif
```

3.1.5 Correspondence of CP_{ground} and $CP_{grapnel}$ ASM models

In this section, the models CP_{ground} (defined in section 2.2.4) and $CP_{grapnel}$ (defined in section 3.1.4) are analysed and the correspondence of the two models are justified.

3.1.5.1 Notion of equivalence

Based on the level of abstraction, many definitions of equivalence can be created. ASMs offer a possibility to precisely define what equivalence means at a certain case. There are two possibilities: comparison involves only the results (i.e. final states) of the two systems or the comparison is expanded to (some of) the internal states. In the first case equivalence can be defined based on relations among the input and output of the analysed system. Otherwise, the definition of equivalence must rely on comparing the series of consecutive states produced by the corresponding runs.

In CP_{ground} and $CP_{grapnel}$ model, final states of a message-passing application do not necessarily characterise all relevant attributes. Therefore, the second way of equivalence definition is followed, which is as follows:

Definition CR1 Correctness of refinement [37]. An ASM M* is called a correct refinement of an ASM M if and only if for each M*-run S_0*, S_1*, … there is a corresponding M-run S_0, S_1, … and sequences $i_0<i_1<\ldots$, $j_0 < j_1 < \ldots$ such that $i_0 = j_0 =0$ and $S_{ik} \equiv S_{jk}$* for each k and either

- Both runs terminate and their final states are the last pair of equivalent states, or
- Both runs and both sequences $i_0<i_1<\ldots$, $j_0 < j_1 < \ldots$ are infinite.

In order to perform the comparison of two ASM models, the following items must be defined:

- states

- (representative) states of interests
- computational segments where each single M-step leads from one corresponding state of interest to (usually the next) corresponding states of interest.

Based on the definition of the previous items, a comparison can be performed and the equivalence can be proven by applying definition CR1.

3.1.5.2 Proof of equivalence

In CP_{ground} and $CP_{grapnel}$ ASM models an agent has several functions (*phase*, *startupmode*, *expecting*, etc…) that may (or may not) be updated during the execution. After an analysis, it becomes obvious that the most characterising function of the different agents is: *phase*.

The operation of every process in the application basically depends on its actual *phase*. Any activities performed by processes can be assigned to a certain phase.

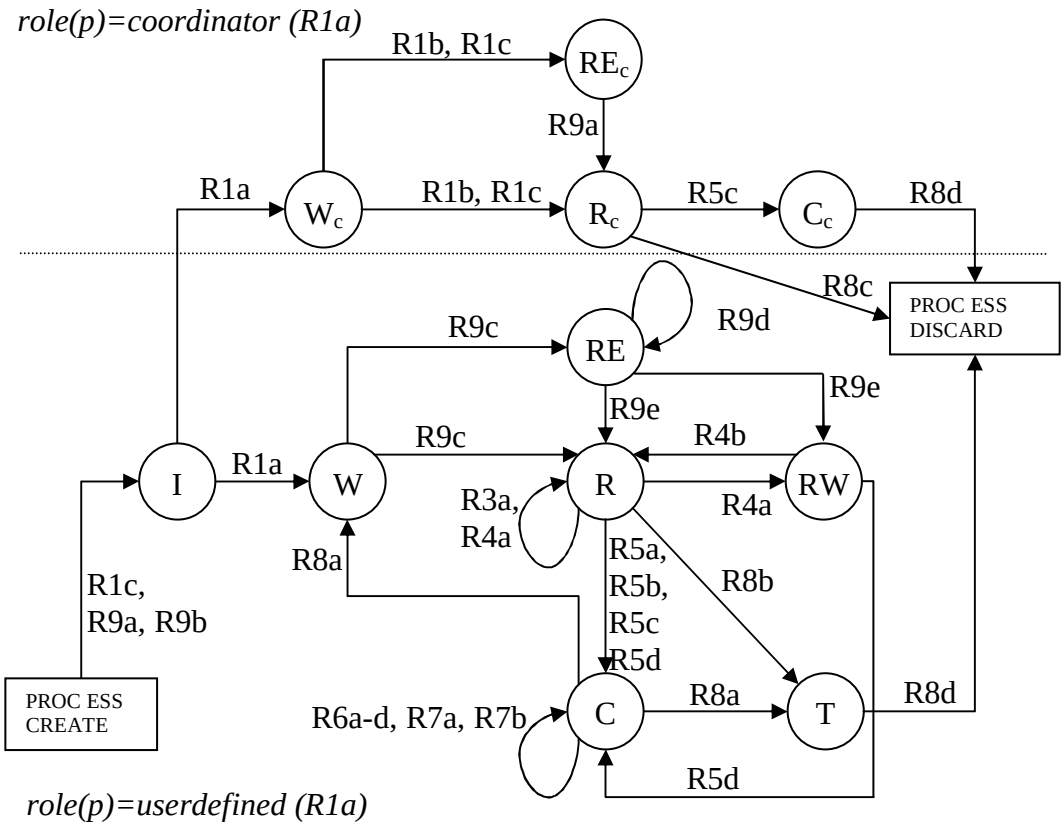

Figure 21 Phase-transition diagram of processes in $CP_{grapnel}$ ASM model

In both (CP_{ground} and $CP_{grapnel}$) models, there are two types of processes executed in the application that have different activities (*role*) to perform: *coordinator*, *userdefined*. Since activities are strongly depending on the role of a process, it must be part of the representative state of a process. In an application there are always exactly one *coordinator* and at least one *userdefined* process during execution.

Definition D1: the state of a process agent is expressed by the value of *phase* and *role* functions: S(p)={*phase*(p), *role*(p)}

The phase of a userdefined process can be one of the followings: init (I), waiting (W), receive_waiting (RW), running (R), checkpointing (C), resuming (RE), terminating (T). To analyse, how a process can change its phase during execution a phase-transition diagram is created. For CP_{ground} model the diagram has already been elaborated, that can be seen on Figure 13 on page 60. The same kind of diagram for $CP_{grapnel}$ can be seen on Figure 21.

On Figure 21 a directed graph can be seen, where nodes represent the different phases of a process and arcs represent transitions between phases. Each arc has one (or some) rule(s) assigned to it that may transfer a process from one phase to another. Based on the value of the *role* function a process may traverse one (upper half) or the other (lower half) branch of the graph. The lifetime of processes starts with creation (box called PROCESS CREATED) and ends up with removing it (box called PROCESS DISCARD).

The **states of interest** are defined to be the following ones:

- *SI* : the initial state of the system (One process of the application is created [see 'Initial state' definition of the models])
- *SR* : the state of the system at point of repose (Every process is in running phase)
- *SF* : the state of the system at the finishing point (No process of the application is running i.e. every process has been terminated and discarded)

Definition D2: $S_{INTEREST}$:={SI, SR, SF}

The node SI represents (see Figure 22) the beginning and SF represents the end of an execution, while the node SR represents an intermediate state of the execution when all the processes are in running phase. It follows that SI and SF occur once in the lifetime of an application, while SR can occur any number of times.

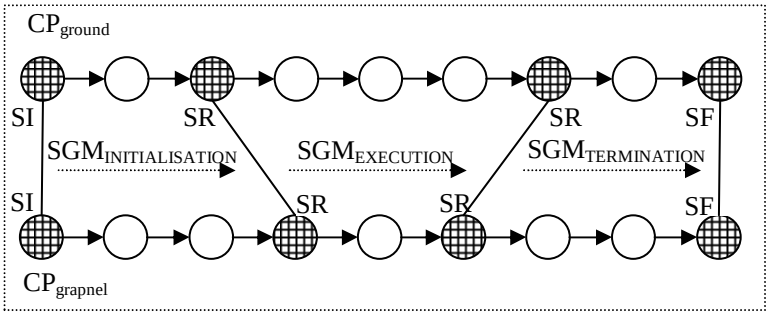

Figure 22 Computational segments in CP_{ground} and $CP_{grapnel}$ models

When a state (e.g. SI, SR, SF) of a model is in relation to the state of another model, they are called corresponding states. Computation segments represent a sequence of steps that leads from a corresponding state of interest (SI, SR, SF are represented by shaded nodes in Figure 22) to (usually the next) corresponding state of interest. Computational segments therefore defined as $SGM_{INITIALISATION}$, $SGM_{EXECUTION}$ and $SGM_{TERMINATION}$.

Definition D3: $SGM_{INITIALISATION}$ contains steps that lead from the related states SI to the related states SR, $SGM_{EXECUTION}$ leads from SR to SR and $SGM_{TERMINATION}$ leads from SR to SF.

In order to prove the correctness of the refinement, it must be shown that for every run of CP_{ground} model there exists a corresponding run of $CP_{grapnel}$ where the states of interest are in relation. Therefore, the computation segments must be defined to show that the corresponding segments of the two models always lead between the corresponding states of interest as it is defined by definition D3.

Computational segments are defined by a sequence of states of the system. System in this context means the application with all of its processes. The state of an application in these models depends on the number of processes, their phases and roles. Two states are different if any of these properties are different and two sequences are different if at least one state is different. Practically, the number of different sequences can be infinite since the number of processes is unlimited. Therefore the number of processes is considered as a fix number that is low enough that the overall state can be handled and high enough to represent all different runs.

Every application contains exactly one *coordinator* process and at least one *userdefined* one. However, at least two of the userdefined processes are required in a parallel application. At the same time, behaviour of processes during the checkpointing can be divided into two representative groups: checkpoint with termination and checkpoint without termination. Based on these considerations, state sequences are represented by three processes: 1 *coordinator* and 2 *userdefined* processes. A state with any number of *userdefined* processes can be derived from this two processes by scaling (multiplying) them up to the required number.

Based on the phase-transition diagrams (shown in Figure 22) and the analysis of the rules of the two ASM models, exactly two different scenarios can be defined for each (SI, SR, SF) of the three different computational segments. Altogether, six different computation segments can be defined where each one represents an atomic operation in the model. Atomic since the whenever the computation segments leaves its initial state, there is no event that could inhibit the model to reach the next state in the sequence.

For segment $SGM_{INITIALISATION}$ the two scenarios are

1. Normal startup of the application and
2. Resumption of the application from checkpoint.

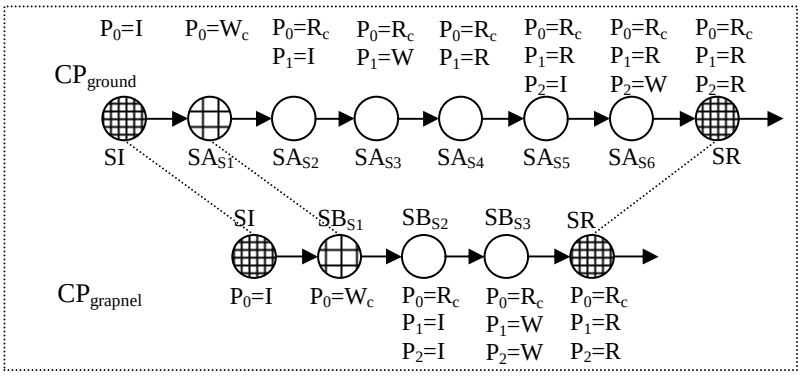

Figure 23 Startup phase of $SGM_{INITIALISATION}$ in CP_{ground} and $CP_{grapnel}$ models

The states for normal startup of CP_{ground} and $CP_{grapnel}$ models are shown on Figure 23. Nodes represent the overall state of the system and directed arcs show the next state of the system. As it can be seen both sequences start with the initial state (SI) and end up with the running state (SR). Intermediate steps are represented by

SA_{S1}–SA_{S6} for CP_{ground} and SB_{S1}–SB_{S3} for $CP_{grapnel}$. (Note: steps are named in a form like "**S***X*$_{SY}$" where X can be "A" for CP_{ground}, "B" for $CP_{grapnel}$ and SY means the Yth state in the sequence)

For each state, phases of every process (P_0, P_1, P_2) are shown above or below the nodes in Figure 23. P_0 represents the *coordinator* process, while P_1 and P_2 are the *userdefined* processes. The corresponding states are shown as shaded nodes for SI and SR and with thin grid pattern for intermediate states. Relation of a pair of states is shown by a dotted line.

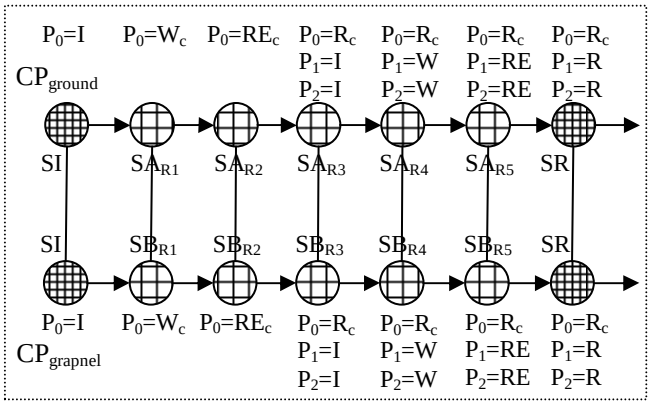

Figure 24 Resumption phase of $SGM_{INITIALISATION}$ in CP_{ground} and $CP_{grapnel}$ models

Normal startup of the CP_{ground} assumes that the first *userdefined* process is created by the *coordinator* the second one is spawned by the *userdefined* process, since the ground model applies dynamic process creation. Contrary to that, in the grapnel model processes cannot be spawned at run-time, therefore *coordinator* spawns all *userdefined* processes in one step. These two scenarios are shown in Figure 23, where the sequences of both system starts and ends up in related states (in this computation segment), while one pair of intermediate related states (SA_{S1}–SB_{S1}) still exists. Analysing the rules, one can come to a conclusion that every state can lead only to the next state defined in the aforementioned sequences (in Figure 23) for both models. The rules ensure that during the startup phase state of the agents are not depending on any external event. The exception is only the *spawn* event sent by the programmer, which is predefined (assumed that CP_{ground} model gets a spawn event after the first process started) in this situation to perform comparison of the two runs.

The comparison of the resumption computation segments of CP_{ground} and $CP_{grapnel}$ (in Figure 24) shows correspondence between all the states. Both model implements resumption of an application in the same way, i.e. by going through the same states.

For segment $SGM_{EXECUTION}$ there can be two scenarios, namely

1. Communication among two processes
2. Process checkpoint and restart (i.e. migration)

Computation segment in the run of both models happens to change state to *receive_waiting* only in case a process is blocked on receiving a message. Any other communication does not result in phase change. Whenever a blocked process gets the message it returns to *running* phase again. These two segments are therefore considered to be in relation (see Figure 25).

The second computation segment in $SGM_{EXECUTION}$ describes a scenario where one of the *userdefined* processes initiates an application-wide checkpoint due to an *exit* event (assumed to be sent by the scheduler).

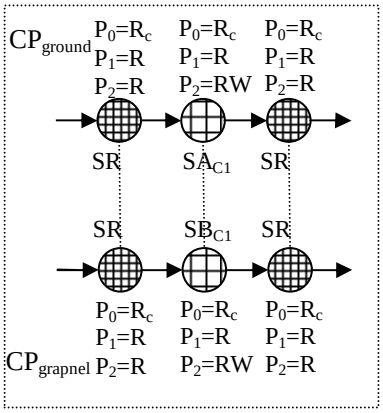

Figure 25 Communication phase of $SGM_{EXECUTION}$ in CP_{ground} and $CP_{grapnel}$ models

After checkpointing the notified process terminates, restarts and resumes, while the other is suspended. When resumption finished, application continues execution. The computation segment for this scenario is depicted in Figure 26.

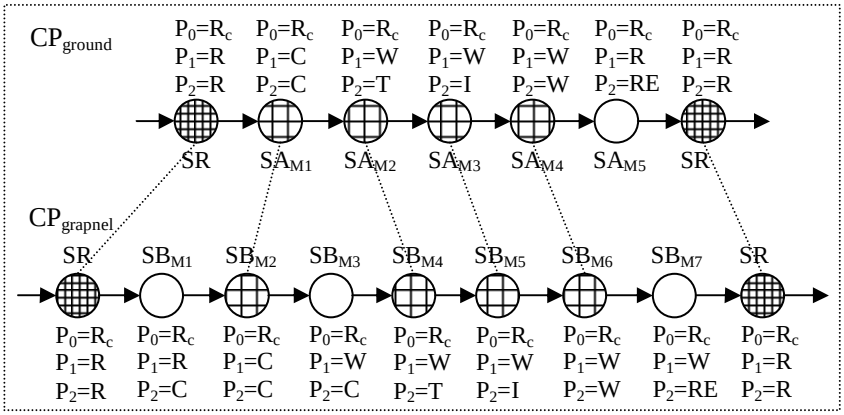

Figure 26 Checkpoint/Restart phase of $SGM_{EXECUTION}$ in CP_{ground} and $CP_{grapnel}$ models

The third segment $SGM_{TERMINATION}$ leads the system state from SR to SF. There can be two scenarios implementing it. These are

1. Normal termination
2. Shutdown (termination with checkpoint)

In case of normal termination every *userdefined* process finishes its work and exits. After it, *coordinator* terminates, too. The sequence of states realizing this scenario is depicted in Figure 27 for both models.

After examining the termination segment of both models, all intermediate states are considered to be in relation.

The final segment called shutdown can be seen in Figure 28. In this scenario, the *coordinator* process gets an *exit* event (assumed to be sent by the scheduler) which initiates an application-wide checkpointing. When checkpointing is performed, every *userdefined* and finally the *coordinator* processes exit, i.e. the whole application finishes execution.

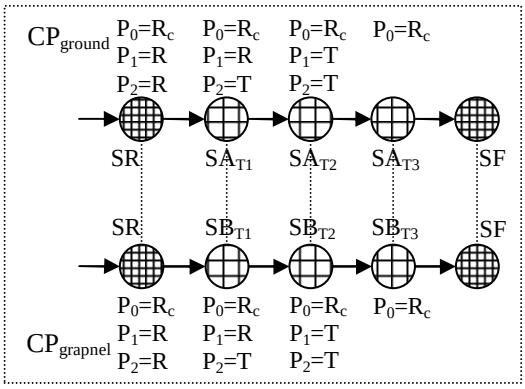

Figure 27 Termination phase of $SGM_{TERMINATION}$ in CP_{ground} and $CP_{grapnel}$ models

Analysis of the state sequences shows that the main difference comes from the fact that while in CP_{ground} model the *coordinator* instructs all *userdefined* processes to terminate in one step (i.e. within one rule), the other model instructs them to do that one-by-one whenever checkpointing is finished for a process.

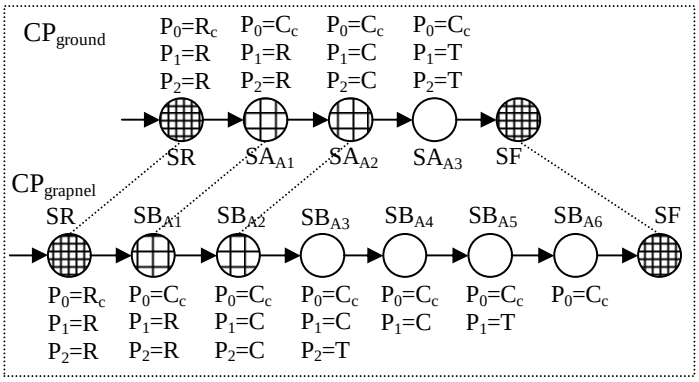

Figure 28 Shutdown phase of $SGM_{TERMINATION}$ in CP_{ground} and $CP_{grapnel}$ models

As a summary, the model $CP_{grapnel}$ is considered to be a correct refinement of CP_{ground}, since for every run of CP_{ground} model there is a corresponding run of $CP_{grapnel}$. Every possible run of the models has been defined by

- showing that the model may only has three different types of computation segments: $SGM_{INITIALISATION}$, $SGM_{EXECUTION}$, $SGM_{TERMINATION}$
- introducing the only two possible runs for each of the three segments

3.2 The TotalCheckpoint framework

3.2.1 Overview

As a preparation of thesis 2.2 I have designed and elaborated a checkpointing technique for native PVM applications based on the abstract checkpointing method defined in thesis 1.2 and as a generalisation of the method developed in thesis 2.1. I have studied PVM applications and services. Afterwards, I have defined the structure of the PVM application and solved the problem of interruption for communication primitives. I have elaborated an abstract model (CP_{tckpt}) that fits to the introduced solution. Finally, I have elaborated a model refinement procedure to prove that the model CP_{tckpt} – implemented by the TotalCheckpoint (TCKPT) tool – is a correct refinement of the $CP_{grapnel}$ model. Based on the results thesis 2.2 has been appointed.

> **Thesis 2.2:** *The checkpointing technique for native PVM applications – following a dynamic process model and realised by the TotalCheckpoint tool – performs transparent checkpointing and its corresponding CP_{tckpt} model is a correct refinement of the $CP_{grapnel}$ model and of the CP_{ground} model.*
>
> Related publications are [4][5][8][9].

The solution introduced in thesis 2.2 gives transparent checkpointing operation – both for the programmer and for the middleware – for native PVM applications which follow static or dynamic process model.

3.2.2 Structure and principles

A native PVM application cannot migrate among nodes during execution, because some parts of the application state are communication-related. The communication-related part of the state is hidden in the messaging layer i.e. in the PVM environment.

Since the modification of the communication environment is undesired (defined by Condition 3. in Section 2.1.4) the application must be modified. At the same time only those techniques can be applied that do not require any changes in the application source code (defined by Definition 7. in Section 2.2.2).

The basic design principles of the TotalCheckpoint framework can be summarised by the following theses:

- PVM daemons of the virtual machines are not checkpointed.
- Migrating PVM environment is converted to shutdown and restart steps, i.e. inner state of the PVM is lost after migration.
- Before checkpointing the processes leave PVM and during resumption they reconnect to PVM. Processes leave PVM in order not to save such internal state for the individual, disconnected processes showing connected status.
- After the process memory is rebuilt from checkpoint, the process is reconnected to PVM.
- Every reconnection causes the PVM process identifier to be changed.

Based on the Conditions (defined in Section 2.1.4) and Definitions (defined in Section 2.2.2) special techniques must be introduced that hide communication-related changes from the user and give transparency for the user at the same time. To

understand the overall solution, the proposed architecture is introduced first (see Figure 29).

In PVM, one process daemon per node is running in the background in order to provide a contact point for the PVM processes for communication. PVM daemons are connected to each other forming a virtual environment where the connected user processes can send messages to the others through the daemons [25].

In Figure 29 PVM daemons (represented by a PVM cloud) form the parallel virtual machines and A, B, C, D are the user processes of the application. They are connected to the daemons and exchange messages through them. Checkpoint server [CS] stores and retrieves the checkpoint information. This information is gathered inside the processes by a single process checkpointer library [CL] (denoted as 'ckpt lib' in Figure 29) linked to the user processes.

TotalCheckpoint library [TL] (denoted as 'tckpt lib' in Figure 29) is linked to each processes of the application. They are connected to the Totalcheckpoint coordinator [TC] and execute the checkpoint related commands issued by TC.

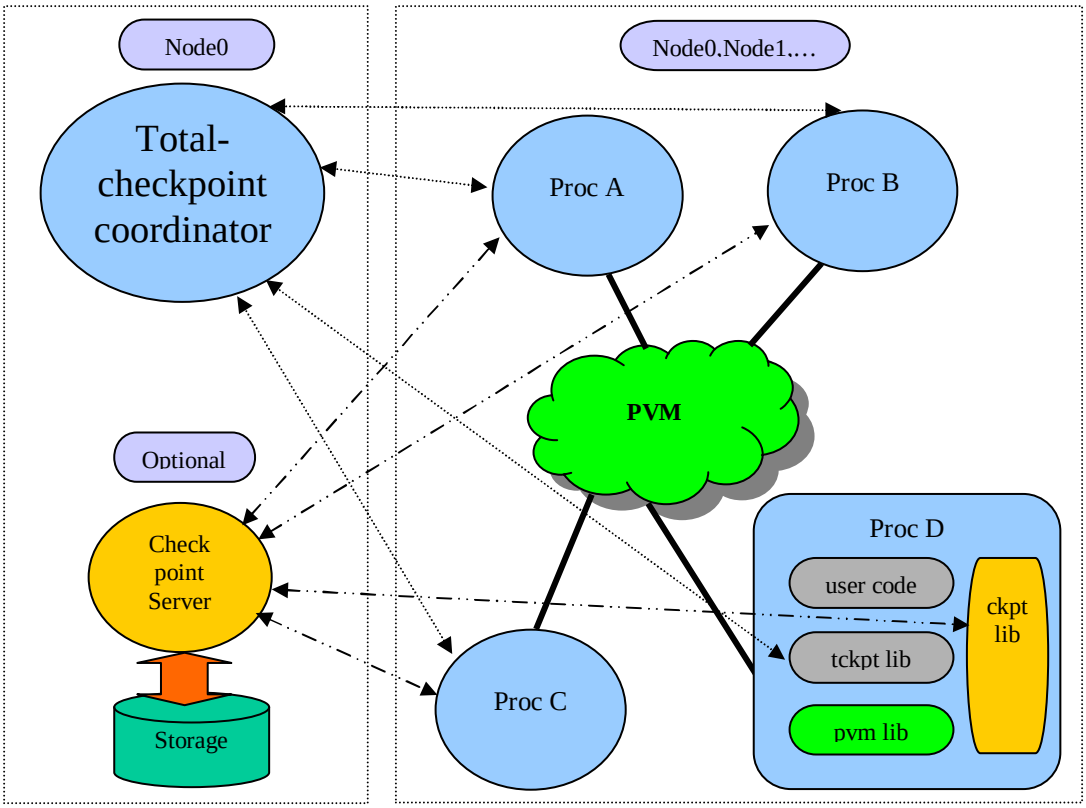

Figure 29 Architecture of the Totalcheckpoint framework

In order to prepare the PVM application to be checkpointable, modification of the internal behaviour of the application is needed. To avoid changes in the user code special techniques are used, changes are transparent for the user. The main cornerstones of the solution designed for TCKPT are as follow:

- To perform checkpoint of a single process with its internals, a checkpoint library (CL) is dynamically loaded at startup.

- At process creation time an initialisation function is automatically invoked to initialize the underlying checkpointing system i.e. to initialize TL.
- Using signals lets TL interrupt the user code and execute its own algorithms.
- Pre and post functions of TL are registered in CL to be called when a checkpoint or resumption phase starts or finishes.
- Linker wrapping technique enables the modification of the behaviour of the PVM operations. When linking the application the original PVM routines are renamed and they are replaced by the functions of TL with the same name. Therefore, application user code invokes the function calls defined in TL instead of the real PVM calls. TotalCheckpoint library can invoke the real PVM calls if necessary.
- One instance of TC is running in the background (usually on the frontend node of the cluster) to help PVM processes in synchronising their checkpoint activities. PVM processes get contacted to TC at startup.

3.2.3 Design issues and solutions

3.2.3.1 Identification of processes

The user code of a PVM application usually stores process (task) identifiers (TID) and refers to them when sending or receiving messages. In case a process leaves PVM the task identifier is discarded and when re-entering a new identifier is generated for the process [25]. Since processes exit and enter PVM when they are checkpointed and resumed, process identifiers change (if checkpointed at least once) during the life of the application.

During the execution, the checkpointer continuously keeps track of process identifiers. For transparency, real TIDS are replaced by virtual ones as returned by PVM. Virtual process (or task) identifiers (USER TASK ID, UTID) are kept the same during the whole lifetime of the application, while the continuously changing real ones (SYSTEM TASK ID, STID) are mapped to them. This mapping is performed by the checkpointer library and the checkpoint coordinator.

If an application spawns and terminates its processes frequently, it might happen that the STID returned by the PVM system is already assigned for a process as UTID. Since, the virtualisation algorithm prefers to use the same value for UTID as STID, duplication can occur. Therefore, when spawning a new user process coordinator always have to check for duplication and choose an unreserved value for UTID if duplication is found.

In case a process is resumed and got a different process identifier it cannot be addressed by its neighbours. Therefore the process after migration gets connected to the coordinator and reports its new identifier which is then distributed among all the processes of the application before coordinator let them continue their execution. In other words the coordinator is used for a rendezvous point for identification of processes.

3.2.3.2 Dynamic process creation

At the time of checkpoint, the Chandy-Lamport protocol is applied for saving the in-transit messages among the processes. It requires the coordinator to have the

exact list of processes. According to the list, the processes can cross-post synchronisation messages to their neighbours.

Hence, dynamic process creation and termination requires keeping track of a process list. As a first step all processes must be registered in the coordinator, directly by the newly spawned process and indirectly by its parent when querying the TID of its child process. During the process creation a checkpoint interruption is undesired, so this operation must be defined atomic and uninterruptible.

In case of process termination there is an obvious solution. There are several system or PVM calls that may result in process termination. Using wrapping techniques enable the tool to redefine any call, e.g. insert notification of the coordinator. However, there are numerous such calls and processes might terminate even without any system call. Above all there can be situations when a process aborts due to any unexpected event. Applications are usually prepared for child process termination; the checkpointer tool must also handle this case. Therefore, it is not failsafe to rely on the notifications of the modified system calls, since it has no effect when process aborts.

To continuously track the number of processes, the coordinator needs a mechanism to automatically detect the termination of a process. The solution used in the TotalCheckpoint tool is based on the detection of loss of connection towards the user process. Since at startup the user process builds up a connection to the coordinator, process termination can easily realised by the connection loss detection mechanism. When a process terminates the coordinator detects that the connection has been lost and registers the assigned process as terminated and updates the actual number of processes spawned by the application.

As a summary, in a Grapnel application the number of processes is constant during the execution while in the native PVM version new process can be created at any time during execution. Therefore in the latter case, registering the actual number of processes in a continuously varying application requires careful design.

3.2.3.3 Starting the execution

In this section the normal startup (not resumed from checkpoint) of the application is introduced. During the startup mechanism a predefined protocol between user processes and the coordinator is executed.

Based on the situation there are 2 different ways of startup:

1. Normal startup of a single process
2. Normal startup of a child process

The following protocol (depicted in Figure 30) is executed for normal startup of a single process:

1. The user process gets connected to the coordinator using the value stored in the **CHKPT_COORD_ADDR** environment variable.
2. The user process decides on the mode of startup based on the value of the **CHKPT_RESUME** environment variable that can be **RUN**, for normal startup or **RESUME** for starting from checkpoint.
3. The user process identifies itself with the value stored in **CHKPT_APPID** environment variable by sending a **CHKPT_NEW_PROC** message to the

coordinator. The startup mode **CHKPT_WM_RUN** is also attached to let the coordinator override the mode if necessary.

4. Coordinator informs the user process with a message **CHKPT_CKPT_INFO** storing the information about the location (**CKPT_SERVER**) and about the name (**CKPT_ID**) of the checkpoint data.
5. Coordinator checks the conditions for performing a **RUN** startup mode and acknowledges it with the **CHKPT_RUN** message.

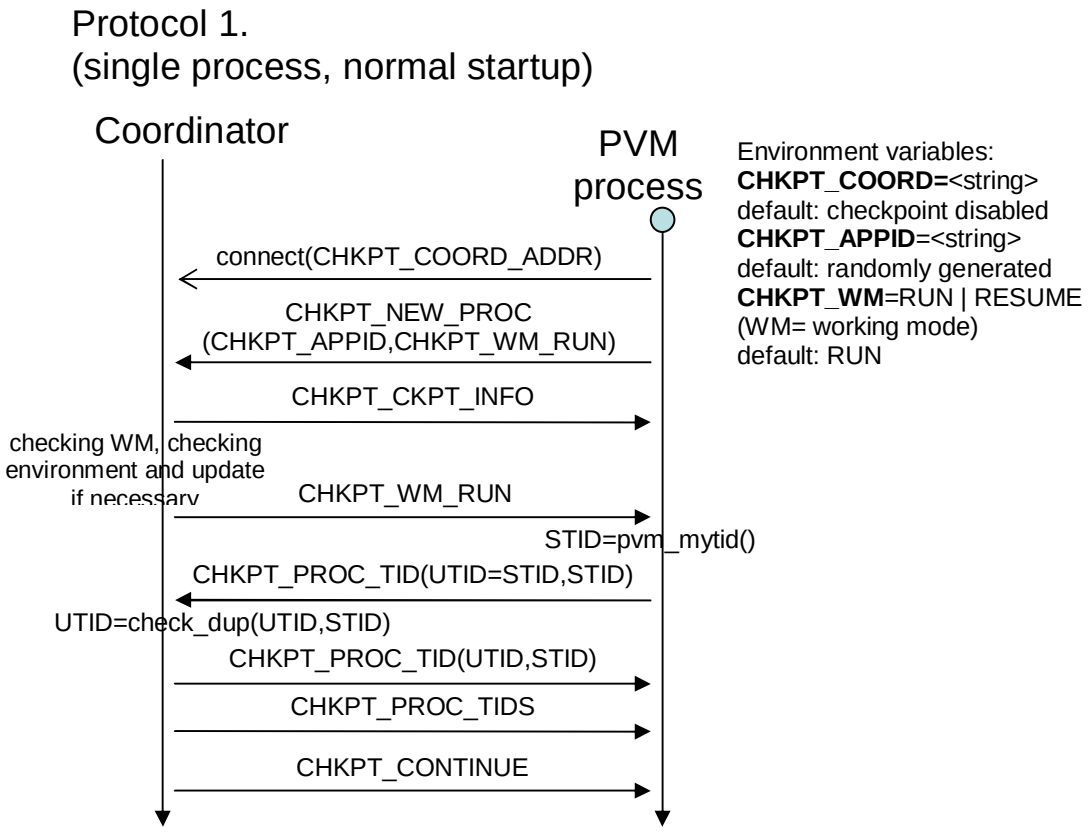

Figure 30 Protocol of normal startup for a single process

6. The user process sets the checkpoint related parameters and initiates the PVM subsystem.
7. The coordinator checks the TID values against duplication sent by the user process (**CHKPT_PROC_TID**) and updates (**CHKPT_PROC_TID**) if necessary.

The last two steps of the protocol are executed when all process initialisation of user processes have reach this point. The last two steps therefore are executed simultaneously among the user process when the coordinator decides to continue the execution of the entire application.

8. Coordinator updates the process table of the user process by the message called **CHKPT_PROC_TIDS**. This message contains **USER_TID**, **SYSTEM_TID** pairs of every process of the application.
9. Coordinator lets the user process start execution by a **CHKPT_CONTINUE** message.

User process starts execution and coordinator switches to standby mode i.e. waits for checkpoint related events to happen.

The following protocol (depicted in Figure 31) is executed for normal startup of a child process:

1. Parent process spawns the child and sends a request (**CHKPT_CHILDTID**) to the coordinator to get the UTID, STID pairs for the child process.
2. Coordinator adds the request to a waiting queue until the TID values are decided.
3. Overlapped with the first two steps, the child process executes protocol 1.
4. After the startup protocol of the child process finished, coordinator informs (**CHKPT_CHILDTID**) the parent process about the TID values of the child process.

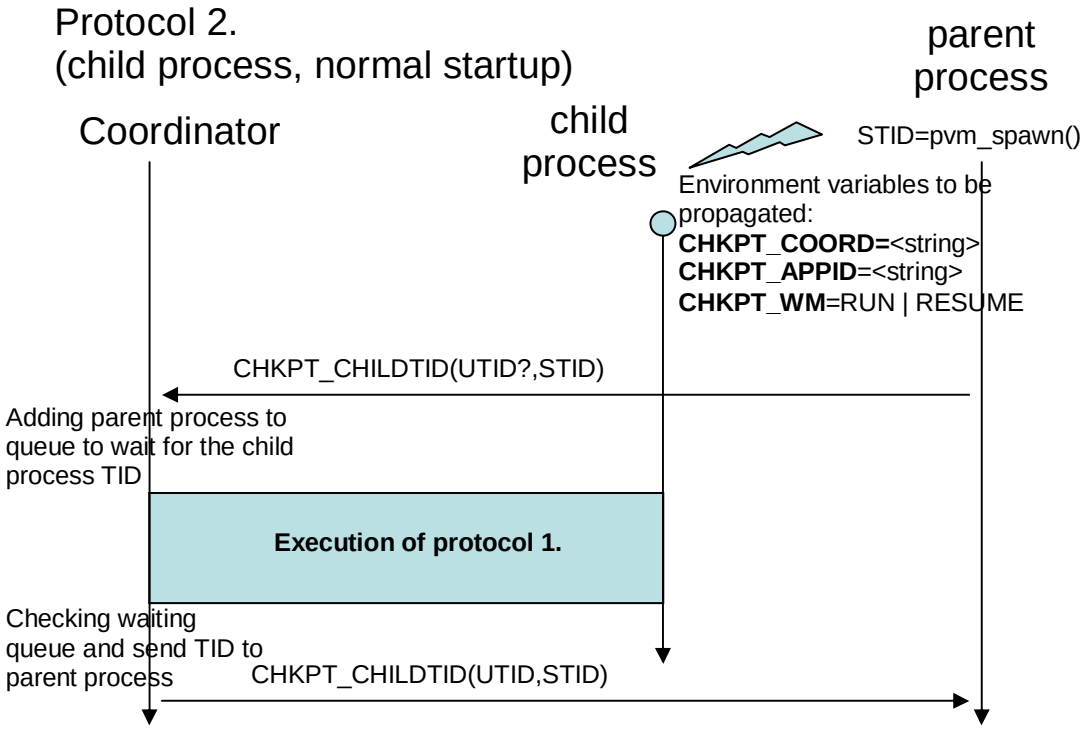

Figure 31 Protocol of normal startup for the child process

Based on the previously defined two protocols, the application can be rebuild with checkpoint support in a seamless way as these protocols are hidden from the user. To rebuild the application two additional protocols are defined and introduced in the next section.

3.2.3.4 Recovering the execution

In case the application execution is resumed from checkpoints, the recovering protocol needs to be changed accordingly.

Based on the resumption alternatives, there are two different startup mechanisms distinguished. These are the followings:

1. Resumption of a single process
2. Resumption of a child process or the entire application

The protocol of resumed startup for a single process (depicted in Figure 32) is defined in the following way:

1. The user process gets connected to the coordinator using the value stored in the **CHKPT_COORD_ADDR** environment variable.
2. The user process decides on the mode of startup based on the value of the **CHKPT_RESUME** environment variable that can be **RUN**, for normal startup or **RESUME** for starting from checkpoint.

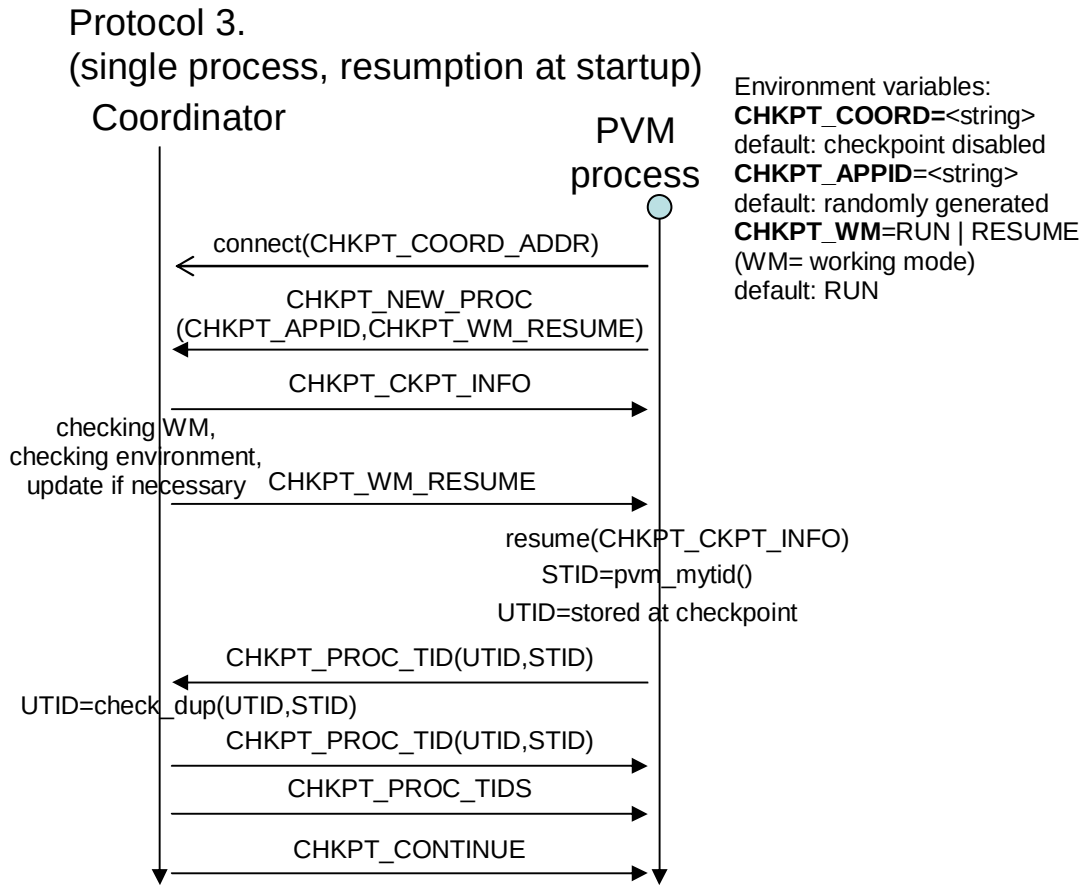

Figure 32 Protocol of resumption at startup for a single process

3. The user process identifies itself with the value stored in **CHKPT_APPID** environment variable by sending a **CHKPT_NEW_PROC** message to the coordinator. The startup mode **CHKPT_WM_RESUME** is also attached to let the coordinator override the mode if necessary.
4. Coordinator informs the user process with a message **CHKPT_CKPT_INFO** storing the information about the location (**CKPT_SERVER**) and about the name (**CKPT_ID**) of the checkpoint data to be used for resumption.
5. Coordinator checks the conditions for performing a **RESUME** startup mode and acknowledges it with the **CHKPT_RESUME** message.
6. The user process sets the checkpoint related parameters and initiates the resumption of the internal state based on the checkpoint information addressed by the coordinator. Now, the process memory is overwritten, protocol is continued with the initialisation of the PVM system.
7. The coordinator checks the TID values against duplication sent by the user process (**CHKPT_PROC_TID**) and updates (**CHKPT_PROC_TID**) if necessary.

The last two steps of the protocol is executed only if all process initialisation of user processes have reach this point. The last two steps therefore are executed simultaneously when the coordinator decides to continue the execution of the entire application.

8. Coordinator updates the process table of the user process by the message called **CHKPT_PROC_TIDS**. This message contains **USER_TID**, **SYSTEM_TID** pairs of every process of the application.
9. Coordinator lets the user process start execution by a **CHKPT_CONTINUE** message.

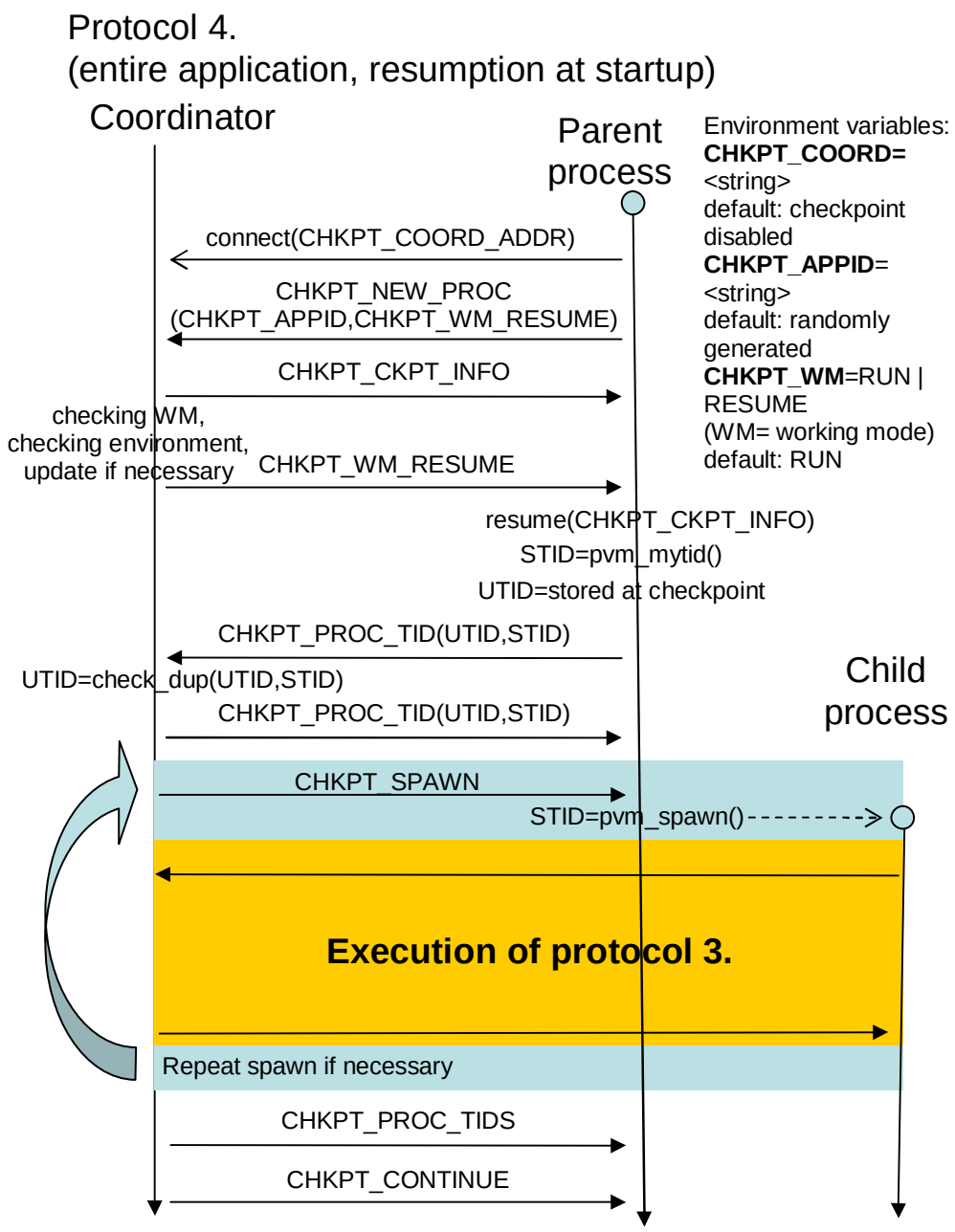

Figure 33 Protocol of resumption at startup for the entire application

The next protocol (Protocol 4) is used for starting the whole application from checkpoints. By forcing the parent processes to spawn their children, the relationship

of the processes in the application is preserved. The spawning operation is repeated until the whole application is resumed. The protocol (depicted in Figure 33) is as follows:

1. The first block (until the message **CHKPT_SPAWN**) of steps defined for the parent process in this protocol is equal with the ones defined by Protocol 3 without the last two steps. It means that the protocol of resuming a new child process is executed after the parent process has successfully finished its protocol and is waiting for the coordinator to continue the execution.
2. In case a new child process is about to be spawned, based on the relationship of the processes at checkpoint time, coordinator selects the parent process and instructs it to spawn a child by sending a **CHKPT_SPAWN** message to it.
3. When a child process resumes it executes Protocol 3.
4. Coordinator repeats this step until the entire application is built. Spawning operations are executed overlapped in time.
5. When the last process has successfully resumed, coordinator updates the TID table of the processes by sending a **CHKPT_PROC_TIDS** message to all of them.
6. After all the processes are aware of the identifiers, coordinator instructs them to continue the execution with a **CHKPT_CONTINUE** message.

Based on the protocols defined in this section, the processes of the application can be started in 4 different ways.

3.2.3.5 Checkpointing the application

In this section the overall application state saving mechanism i.e. checkpointing is introduced. The solution is derived from the previously mentioned Chandy-Lamport algorithm. The main steps of the flow of checkpoint (depicted in Figure 34) are the followings:

1. Checkpointing of the application is started by an interruption of the processes. Each process reports this fact to the coordinator with a **CHKPT_SYNC_START** message.
2. When the coordinator gets this notification from all the processes of the application, synchronisation is started with a **CHKPT_SYNC_TIDS** message containing the TIDs of the processes performing the checkpoint.
3. In the next step processes are performing a cross-messaging and turn into a message receiving mode i.e. saves all incoming messages appear in the channel until a **CHKPT_SYNC_MSG** is taken.
4. Saving the in-transit messages is signalled to the coordinator by a **CKPT_SYNC_FINISHED** to which responding with a **CHKPT_SAVE_START** initiates the checkpoint of the individual processes.
5. Checkpointing of the individual process means to leave message-passing layer i.e. PVM and perform the real internal state saving (i.e. copying the whole memory) of the process internals. The operation is finished with a **CHKPT_SAVE_FINSIHED** message to the coordinator and after enrolling again to PVM the new identifier is reported by a **CHKPT_PROC_TID** message. Leaving and re-entering the PVM is required to reset the state of the PVM library contained by the process itself, otherwise the internals of the PVM library of a resumed process represents connected state which is not true, since reconnection must be perform after resumption.

6. When all the processes have finished the previous steps coordinator distributes the new identifiers (message **CHKPT_PROC_TIDS**) and let the processes continue (**CHKPT_CONTINUE**) their execution.

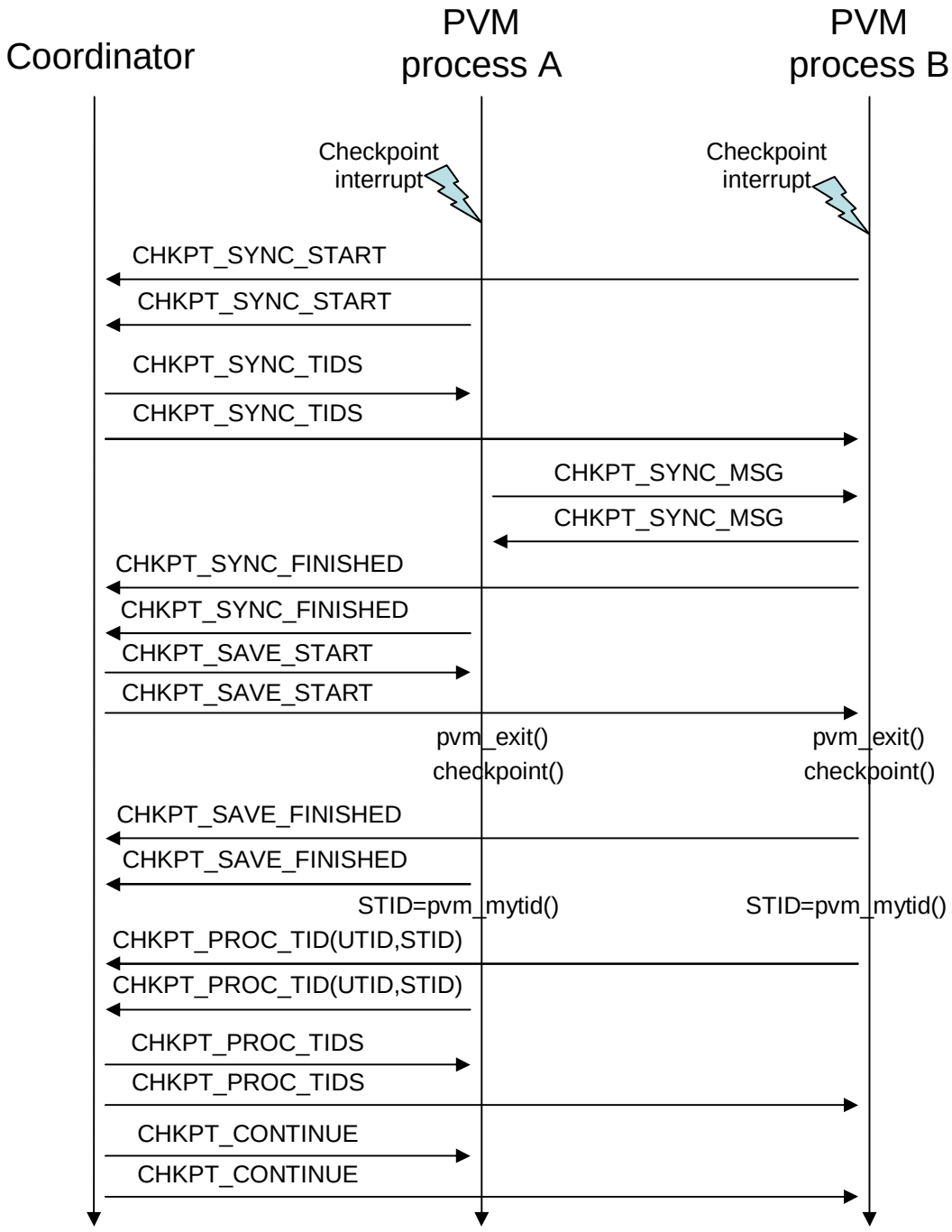

Figure 34 Protocol of checkpoint saving in TCKPT

The introduced protocol is responsible to perform a checkpoint of the whole application i.e. all processes must take part. However, it can happen that only some of them interrupted, since it is decided by the scheduler. The processes that have not been interrupted by the scheduler are interrupted by any other suspended process.

3.2.3.6 Restoring message buffers with dynamic message format

In case the processes execute the synchronisation of the in-transit messages in the checkpoint saving protocol the following requirement must be fulfilled:

- PVM offers filtering of messages by type (called tag) and by partner. During receiving incoming messages, filtering of messages by type must not be applied; any type of incoming messages must be received, read and saved. Otherwise filtered messages are lost.
- Each process must possess the identifier of the other processes that are included in the procedure of message synchronisation.
- The receiver process must be able to save any incoming message stored in a message buffer. Saving a PVM message buffer requires the knowledge of the message format which is not part of the message or the buffer.

To fulfil the third requirement the checkpoint library must recognize the protocol i.e. the format of the message. Since it is implicitly coded in the user algorithm, the only possible way is to add extra information to the message through redefinition of the message creation/packaging calls. Redefinition is demonstrated in the next block of pseudo code.

Messaging in the application with TCKPT:

```
elementA        :element with user type A
TA              :element type A
elementT        :element for storing a type
TT              :element type for describing a type

Sender side:

Function packing(TA,elementA,messagebufferX)
Begin
        packing(TT, TA, messagebufferX)
        packing(TA, elementA, messagebufferX)
End function

Receiver side:

Function unpacking(TA, elementA, messagebufferY)
Begin
        unpacking(TT, elementT, messagebufferY) ;ommiting type information
        unpacking(TA, elementA, messagebufferY)
End function
```

Whenever the user adds a new element to a buffer through a packaging method, the method inserts the format (type and length) into the buffer before the real user data. This extra information has a fixed message format and is inserted before every user message element. With this extension the checkpoint facility is able to unpack the appropriate format of data elements from the buffers. On the receiver side, format information is automatically omitted by the corresponding message handling calls if the message is read by the user code.

Saving the content of a message buffer is also required for created-but-not-sent buffers at the time of checkpointing. Since any number of message buffers can be created in PVM by the programmer, saving and restoring these buffers are also

inevitable tasks for the checkpointing facility. To save the content of these types of buffers the previously introduced method can also be applicable.

Last, but not least virtualisation of the PVM buffer identifiers similarly to the PVM task identifiers (TID) must be applied. PVM buffers have discrete identifiers and the buffers with their identifiers together are also removed in case of detaching from the PVM environment. When re-entering PVM the resumption mechanism must take care of the initialisation of all the buffers existed at the time of checkpoint. New buffers are created, their identifiers are assigned to the original ones and their content is also added. When the user code refers to a message buffer after resumption a mapping of the original and new identifier is performed by the checkpointing layer.

3.2.4 Comparison of GRAPNEL and TCKPT checkpointing

In the following few paragraphs an overview of the GRAPNEL and TCKPT checkpointing tool is given in order to reveal the differences between these two solutions and in order to show the motivation for the final design step in TCKPT.

The basic difference is that GRAPNEL version is aimed at supporting checkpointing of applications developed in a high level graphical environment, TCKPT is a design for native PVM applications. However, TCKPT could be utilised for checkpointing GRAPNEL applications, several advantages and extra assistance offered by the GRAPNEL layer would not be exploited by TCKPT.

In GRAPNEL application the GRAPNEL layer itself introduces several restrictions regarding the topology. A significant restriction is that the topology of a GRAPNEL application is fixed, the layout is static. No creation of additional child processes is possible in the PGRADE environment. This feature enables the GRAPNEL checkpointing support to simplify its internal process management mechanism.

The essential support provided by the GRAPNEL layer can be summarised by the following points:

- Process topology is known by the GRAPNEL server process that gives significant help in the process management mechanism of the checkpointer
- Virtualisation of the process identifiers is already implemented with GRP identifiers, so this functionality is not required from the checkpointer
- Message format is static in the sense the protocols of the channels connecting two processes must be defined in advance and no changing is possible. Therefore the information of the message format is available for the checkpointer to unpack and save messages in the memory while executing the synchronisation of messages

The previously defined restrictions and supports given by the GRAPNEL layer confirms the need for both GRAPNEL and TCKPT checkpointer since for GRAPNEL application a general native PVM checkpointer like TCKPT is not efficient enough while GRAPNEL checkpointer is not able to work on non-PGRADE applications.

During the design of TCKPT several services provided by the GRAPNEL layer had to be replaced and the layering is also modified (depicted in Figure 35) according to that.

In the software layers (see Figure 35) designed for TCKPT a virtual PVM API is inserted (replacing the functionality of the GRAPNEL layer) in order to enable the modification of the behaviour of the PVM calls serving transparent checkpointing for the programmer. This insertion is automatically done at the time of linking the application and no modification of the source code is required.

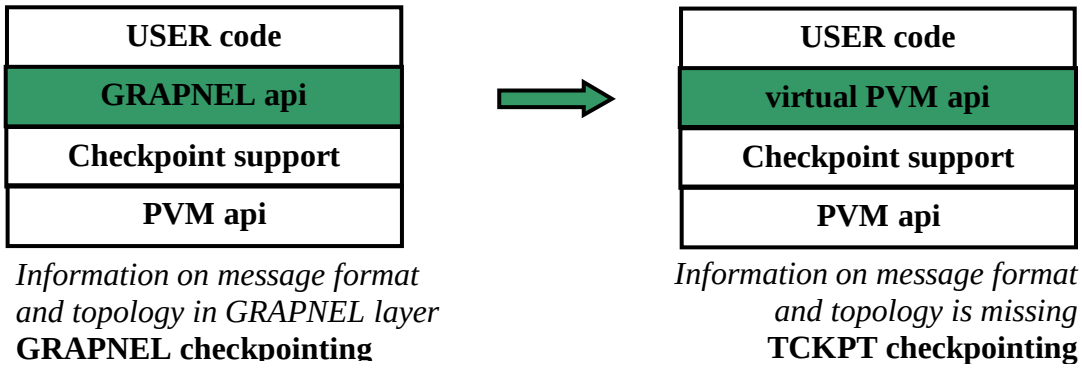

Figure 35 Software layers in GRAPNEL and TCKPT checkpointing

However, most of the required services of the GRAPNEL checkpointing are successfully substituted in the TotalCheckpoint tool, one critical difference in the current design is that the coordinator is a standalone, independent, background process usually running on the front-end node of the cluster while coordination in the GRAPNEL version is part of the application as shown in Figure 36.

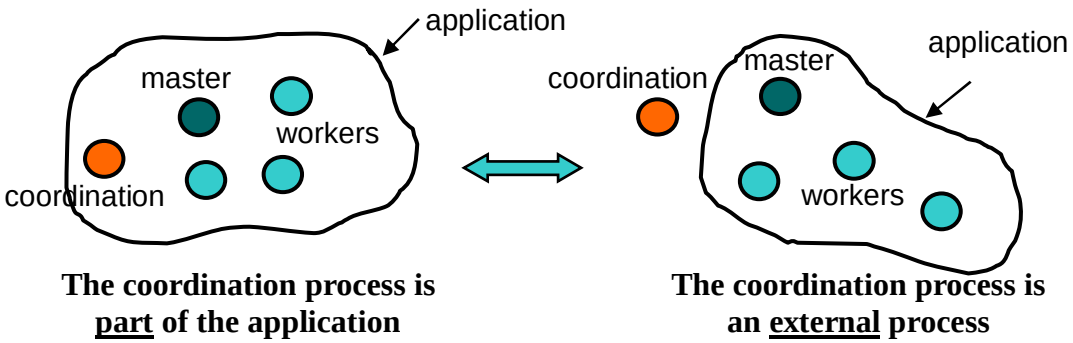

Figure 36 Comparison of structures of GRAPNEL and TCKPT checkpointing

The original version of TCKPT has been designed for Hungarian ClusterGrid [85] where each cluster has the coordinator as a deployed service. Since the current work aims at designing and developing a checkpointer tool with application and middleware transparency, coordinator must be somehow integrated into the application. Otherwise it causes the application to depend on a surrounding grid middleware environment which results in failing to fulfil the requirement defined by Condition 4. defined in Section 2.1.4.

3.2.5 Overview of the Enhanced TCKPT

In order to reach the desired architecture of the TCKPT and make it an application and middleware transparent and ClusterGrid compliant checkpointer tool the coordinator must be integrated with the application. The main advantages of the

introduced tools and the resulted enhancement is summarised and depicted on Figure 37.

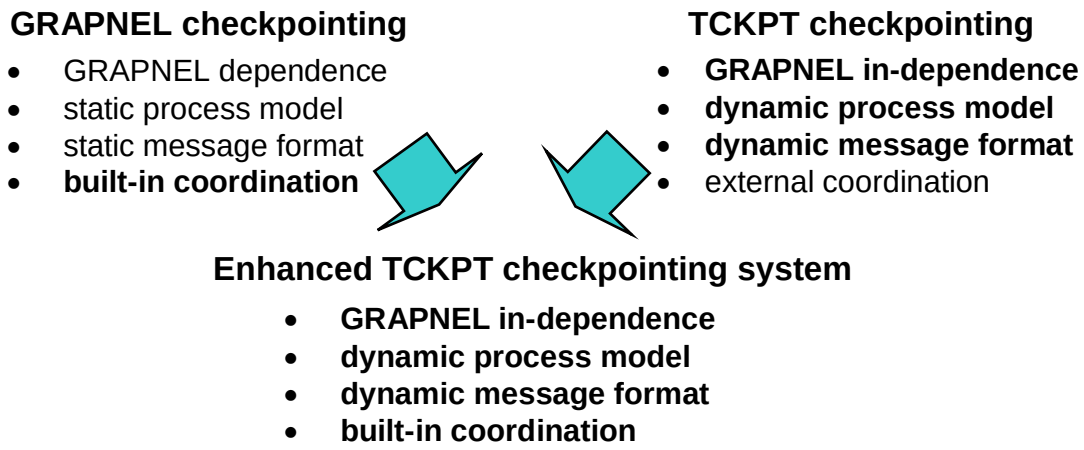

Figure 37 Evolution of the Enhanced Totalcheckpoint framework

In Figure 37 a classification of the two introduced tools is depicted. There are four aspects, namely: GRAPNEL relation, process model, and message format and coordination type. Features are bold representing the desired ones for the enhanced version.

The weaknesses of the GRAPNEL solution are the dependence on the graphical language, the static process model that the designed application follows and the static message format which must be defined in advance without the possibility to modify it during the execution. In the TotalCheckpoint tool these points have been fixed in order to make the tool independent of the graphical language, to handle dynamic process creation in the application and to support message formats compiled on the fly. However, the built-in coordination is still missing from the final (enhanced) version that is a strong point of the GRAPNEL version.

In order to integrate the coordination into the application a special technique is used. The coordinator functionality is transformed into a procedure and compiled into a static library. This library then linked to the application at compile time as part of the checkpointing library. Creation of the coordinator is the responsibility of the first instance of processes (master) when starting up the application as depicted in Figure 38.

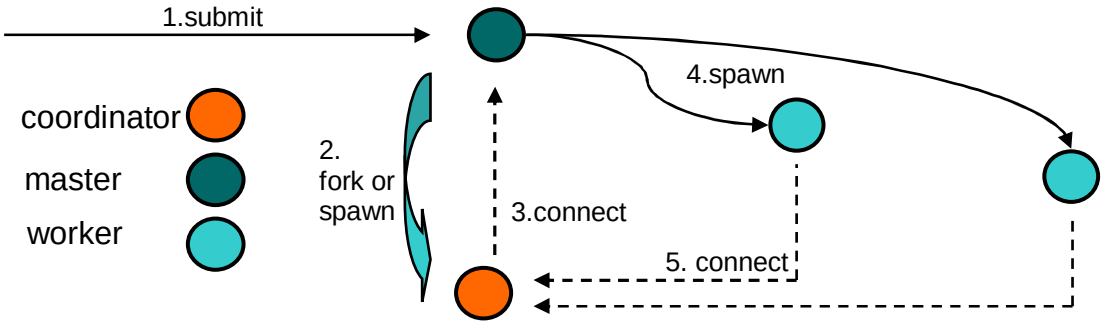

Figure 38 Coordinator initialisation in the Enhanced TCKPT framework

When the first process (master in Figure 38) is created at startup, it forks or spawns the coordinator process before any user code is executed. When the

coordinator is started, master gets connected and all the start-up protocol defined in section 3.2.3.3 and 3.2.3.4. is executed

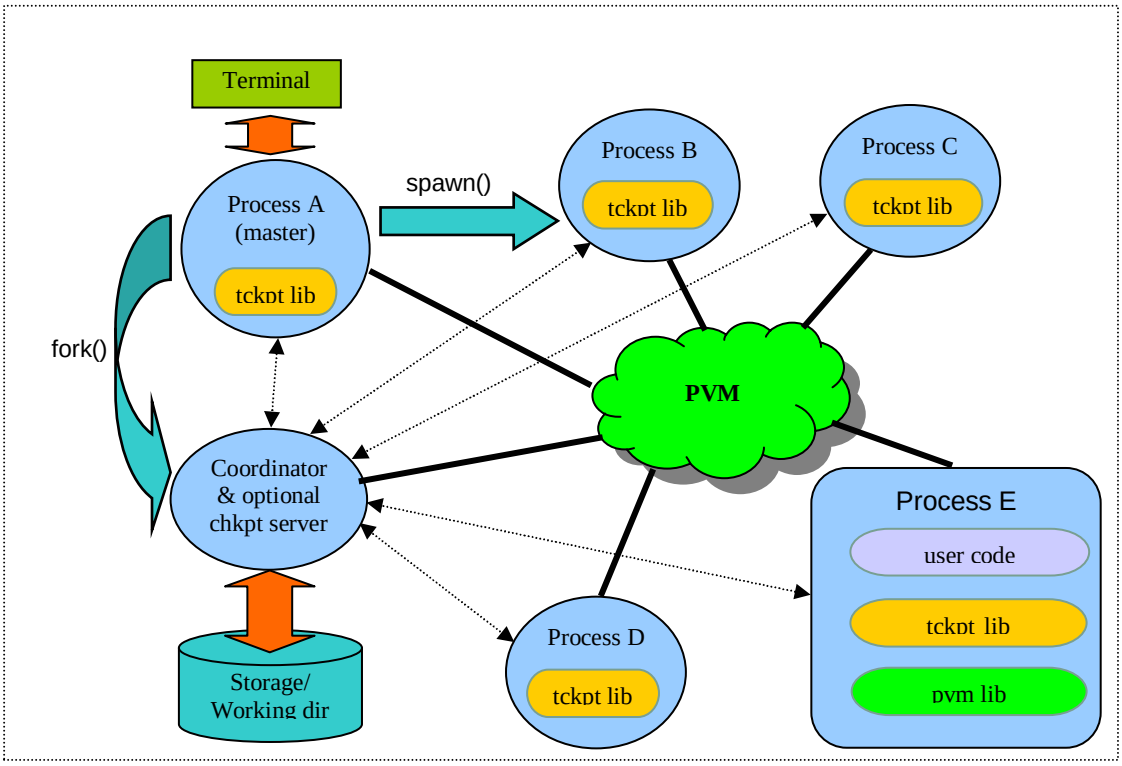

Figure 39 Structure of the Enhanced TCKPT framework

After applying the mechanism introduced in the previous paragraph the layout or structure of the application changes in the way depicted in Figure 39.

In the Enhanced TCKPT framework the coordination became a child of the Master process of the application. The lifetime of the coordination starts when the Master creates it and ends up with the event when all its connections to the user processes (A, B, C, D, E in Figure 39) break. Coordinator in this context also serves as a checkpoint server storing and retrieving the checkpoint information of the user processes.

3.2.6 Definition of CP_{tckpt} ASM model

In order to verify the proper behaviour of the TotalCheckpoint framework, a CP_{tckpt} model is elaborated and its relationship to the CP_{ground} model is analysed. The following section introduces the CP_{tckpt} model.

3.2.6.1 Universes, Signatures and Initial state

The universes and signatures used by CP_{tckpt} are inherited from $CP_{grapnel}$ with only one modification and the initial state is also the same as in $CP_{grapnel}$ model. The exact definition can be found in section 3.1.4.1 and 3.1.4.2, respectively.

The one and only modification is related to the message type. Since in this model processes can only be spawned by user defined processes, the coordinator must be able to control their spawning. This ability is required at resumption of the application. New message type called “spawn” is introduced, so the definition is as

follows: *type: MESSAGE → {userdefined, interrupted, endofchannel, synchronised, saved, resumed, exiting, exited, spawn}.*

3.2.6.2 Rules

1. **Rules for initialisation**

The very first process (that is the master process of the application) has the task of creating the *coordinator* as it is introduced in section 3.2.5. Other details are the same as it was defined in $CP_{grapnel}$.

CPgrapnel-R1a (modified version of CPgrapnel-R1a)

```
if phase(p)=init then
      if role(p)=undef then
            role(p)=userdefined
            extend PROCESS by c with
                  app(c):=app(p)
                  phase(c):=init
                  role(c):=coordinator
                  startupmode(c):=undef
                  master(c):=false
            endextend
      else
            phase(p):=WAITING
      endif
      if master(p)=undef then
            master(p)=true
      endif
endif
```

CPtckpt-R1b (≡CPgrapnel-R1b)

Comparing to CP_{ground} or $CP_{grapnel}$ models, in rule CPtckpt-R1c no process spawning needs to be performed since by the time *coordinator* starts one *userdefined* (master) process is already created.

CPtckpt-R1c (modified version of CPgrapnel-R1c)

```
if role(p)=coordinator & phase(p)=waiting & startupmode(p)≠undef then
      if startupmode(p)=normal then
            phase(p):=running
            process_to_store:={}
            process_to_checkpoint:={}
            process_to_terminate:={}
            process_to_resume:={}
      else
            phase(p):=resuming
            event(p):=resume
      endif
endif
```

2. **Rules for process spawning**

CPtckpt-R2a

```
if role(p)=userdefined & phase(p)=running & event(p)=spawn then
      extend PROCESS by child with
            app(child):=app(p)
            phase(child):=init
            role(child):=userdefined
            startupmode(child):=normal
            master(child):=false
      endextend
endif
```

Dynamic process creation during execution is supported similarly to the CP_{ground} model. The rule CPtckpt-R2a ensures the process to be created, while the rule CPtckpt-R2b ensures the process to start its execution.

CPtckpt-R2b

```
if role(p)=userdefined & phase(p)=waiting &
   startupmode(p)=normal & process_to_checkpoint(p)=false
   then
      phase(p):=running
endif
```

3. **Rule for sending a message**

CPtckpt-R3a (≡CPgrapnel-R3a)

4. **Rules for receiving a message**

CPtckpt-R4a (≡CPgrapnel-R4a)

CPtckpt-R4b (≡CPgrapnel-R4b)

5. **Rules for interrupting the execution**

CPtckpt-R5a (≡CPgrapnel-R5a)

CPtckpt-R5b (≡CPgrapnel-R5b)

CPtckpt-R5c (≡CPgrapnel-R5c)

CPtckpt-R5d (≡CPgrapnel-R5d)

6. **Rules for message synchronisation among the processes**

CPtckpt-R6a (≡CPgrapnel-R6a)

CPtckpt-R6b (≡CPgrapnel-R6b)

CPtckpt-R6c (≡CPgrapnel-R6c)

CPtckpt-R6d (≡CPgrapnel-R6d)

7. **Rules for checkpoint saving of processes**

CPtckpt-R7a (≡CPgrapnel-R7a)

CPtckpt-R7b (≡CPgrapnel-R7b)

8. **Rules for terminating the processes**

CPtckpt-R8a (≡CPgrapnel-R8a)

CPtckpt-R8b (≡CPgrapnel-R8b)

CPtckpt-R8c (≡CPgrapnel-R8c)

CPtckpt-R8d (≡CPgrapnel-R8d)

CPtckpt-R8e (≡CPgrapnel-R8e)

9. **Rules for resuming the processes**

CPtckpt-R9a (modified version of CPgrapnel-R8a)

```
let allproc=((∀rp∈PROCESS):app(rp)=app(p) & rp≠p)
let master=((m∈PROCESS):app(m)=app(p) & master(m)=true
if role(p)=coordinator & phase(p)=resuming & event(p)=resume
   then
      imagefile ∈ IMAGE: imagefileofapp(imagefile)=app(p)
      SINGLE_PROCESS_STATE_RESUME(p,imagefile)
      do forall child : process_to_store(child)=true &
                              master(child)=false
            extend MESSAGE by msg with
                  from(msg):=p
                  to(msg):=master
                  type(msg):=spawn
            endextend
      enddo
      process_to_checkpoint:={}
      process_to_terminate:={}
      process_to_resume:=process_to_store
      phase(p):=running
endif
```

At resumption every *userdefined* process must be created – except the master one since it is already created – to resume them. It is done by notifying the master to make it on behalf of the *coordinator* which is realised by the rule CPtckpt-R9a.

CPtckpt-R9b (≡CPgrapnel-R9b)
CPtckpt-R9c (≡CPgrapnel-R9c)
CPtckpt-R9d (≡CPgrapnel-R9d)
CPtckpt-R9e (≡CPgrapnel-R9e)

The rule CPtckpt-R9f makes spawning possible by the *userdefined* processes on behalf of the *coordinator* process. When receiving a "spawn" message, process creation is performed.

CPtckpt-R9f

```
let coord=c∈PROCESS:role(c)=coordinator & app(c)=app(p)
if role(p)=userdefined &
   (∃msg∈MESSAGE: from(msg)=c & to(msg)=p & type(msg)=spawn)
   then
      extend PROCESS by child with
            app(child):=app(p)
            phase(child):=init
            role(child):=userdefined
            startupmode(child):=undef
            master(child):=false
      endextend
      MESSAGE(msg):=false
endif
```

3.2.7 Relation of $CP_{grapnel}$ and CP_{tckpt} ASM models

In this section, CP_{tckpt} is examined in order to show the relation to CP_{ground} model. Based on the definition of the rules, it is obvious that CP_{tckpt} is more similar to $CP_{grapnel}$ model than to CP_{ground}. Since $CP_{grapnel}$ is a correct refinement, the models $CP_{grapnel}$ (defined in section 3.1.4) and CP_{tckpt} (defined in section 3.2.6) are analysed to see whether CP_{tckpt} is a correct refinement of $CP_{grapnel}$ and of CP_{ground} at the same time.

3.2.7.1 Correctness of refinement

In order to examine the equivalence, first the differences between CP_{tckpt} and $CP_{grapnel}$ are identified based on the definition of the rules:

1. Initialisation rules are different, since the startup mechanism of the coordinator is changed.
2. Resumption rules are different, due to the difference in the startup mechanism.
3. New message type is introduced, but is has no effect on the states of the models
4. Dynamic process creation is introduced compared to $CP_{grapnel}$

Proving the correspondence (i.e. one is a correct refinement of the other) of CP_{tckpt} and $CP_{grapnel}$ can be carried out in the same way as in section 3.1.5.2. The definition of the state, states of interest and the three computational segments ($SGM_{INITIALISATION}$, $SGM_{EXECUTION}$, $SGM_{TERMINATION}$) are exactly the same in this case at it was introduced in section 3.1.5.2. Based on the differences listed above, only $S_{INITIALISATION}$ computational segment is required to be justified in CP_{tckpt} by showing the sequences of states for normal startup and for resumption. For normal startup, a comparison of the state sequences can be seen in Figure 40.

Any run of a normal startup of CP_{tckpt} always leads from SI to SR which can be derived from the logic of the rules. A startup sequence is atomic, since the logic of the rules only depends on internal parameters set by the rules themselves i.e. cannot be interrupted by any event.

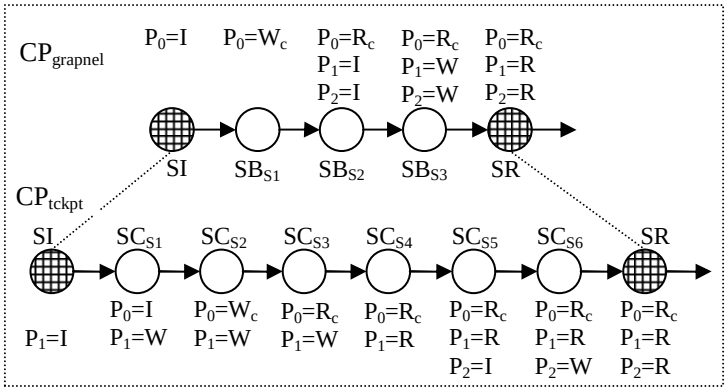

Figure 40 Startup phase of $SGM_{INITIALISATION}$ in $CP_{grapnel}$ and CP_{tckpt} models

An alternative computational segment for initialisation is the resumption (i.e. startup based on checkpoints), which can be seen on Figure 41.

Resumption of an application is always a sequence from SI to SR. Looking at the sequences there is a slight difference at state SI in CP_{tckpt}. The state of the process is marked with "P_1=I", while for $CP_{grapnel}$ SI is represented by "P_0=I". The difference is only syntactical, since both models have the same initial state (defined in section 3.1.4.2) which says that there is exactly one process with INIT state, by the *role* is set to *undefined*. Therefore, the correct expression for SI would be "P=I" in both models. Indices are introduced in only to express which role the created process will get in the forthcoming steps of sequences.

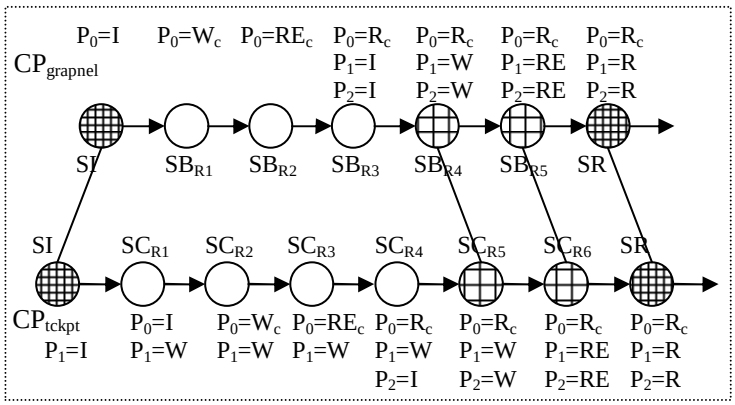

Figure 41 Resume phase of $SGM_{INITIALISATION}$ in $CP_{grapnel}$ and CP_{tckpt} models

As a summary, CP_{tckpt} is considered to be a correct refinement of CP_{ground} and $CP_{grapnel}$ since all computation segments have been defined and it has been shown that every segment leads to a corresponding state of interest between the models. In this case only $SGM_{INITIALISATION}$ segments were shown because the other two segments are equivalent in every state, since the related rules were inherited from $CP_{grapnel}$.

4 Migration of PVM applications

Process migration in distributed systems is a special event when a process or application running on one or several resource(s) is redeployed to other one(s) in a way that the migration is transparent for the process execution. The migration mechanism can be advantageously used in several cases like load-balancing, preemption, fault-tolerance or providing special resource needs.

Depending on the complexity and level two approaches are defined from migration point of view in this work:

1. **Process migration**: process migration in a parallel application refers to the event when the execution of the application is paused, some processes are terminated and restarted (i.e. migrated) on a different machine and finally the application continues the execution. The number of migrating processes (m) in an application (consisting n processes) must be $0 < m < n$. The process that never migrates is the built-in checkpoint coordination process (e.g. GS in a Grapnel application).
2. **Application migration**: application migration in a parallel application in the current context refers to the event when the execution of the application is paused, an application-wide checkpoint is performed and all processes - including the checkpoint coordination process - are terminated. The broker or the user then resubmits the application to a different cluster after all the files (executable, input, output and checkpoint) have been moved to the target cluster. After resubmission the application reloads the previous state from the checkpoint files and continues the execution.

Process migration is typically used inside a cluster among nodes where

- not all the resources (nodes executing the processes) of the application are revoked by the schedule
- the master process of the application is not terminated but continuously running
- revoked resources are replaced within a relatively short period of time
- the processes of the application are notified about the revoking of the resource
- the application is not removed from the queue of the scheduler
- the master process is not checkpointed

Application migration is typically used inside a Grid among Clusters where

- all the resources of the application are removed
- the master process of the application must be terminated, it cannot continue execution
- the revoked resources are not replaced within a short period of time
- the processes of the application are not notified individually about the removal of the resource, the migration is initiated through/by the master process

- the application is removed from the queue of the scheduler in order to be restarted on a different cluster
- master/coordination process performs its own checkpoint

4.1 Process migration on Clusters

4.1.1 Overview

After the theoretical background elaborated in the first group of theses and the concrete checkpointing solution developed in the second group of theses, the goal of the third group of theses is to justify that transparent migration – based on the previous solutions – can be realised in ClusterGrid environment. There are two options: migration of certain processes of the application among the resources of the hosting cluster and migration of the entire application among clusters. To demonstrate the elaborated techniques the well-known job scheduler called Condor has been selected.

Thesis 3.1 focuses on transparent process migration by analysing the interaction between the GRAPNEL application – integrating the transparent checkpointing facility developed in thesis 2.1 – and Condor maintaining the resources of a cluster. In this thesis after short overview of Condor, I define the basic conditions of operation, the components and I determine the steps of migration mechanism. The theory of this mechanism is justified by the analysis of state-transition diagram derived from the $CP_{grapnel}$ model and by mapping the flow of migration procedure into the state-transition diagram. Based on the introduced results I have stated thesis 3.1.

> **Thesis 3.1:** *The migration of processes of the GRAPNEL applications – developed in P-GRADE – is realised in a transparent way for the schedulers and the elaborated solution adapts itself to the internal rules of the $CP_{grapnel}$ ASM model.*

Related publications are [10][11][12][14][20]

In the introduced solution transparency is ensured from several aspects, since the migration does not require any modification in Condor, nor in PVM, nor in any components of the operating system and nor in the source code of the application.

4.1.2 Condor

Condor is a specialized workload management system for compute-intensive jobs [33][34]. Condor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.

The aim of Condor is to exploit the unused computing cycles of the nodes under its control. Condor continuously monitors the usage of the machines and jobs are started on idle nodes. When the machine becomes non-idle (some processes are executed by the owner of the machine), the execution of condor job is suspended to deliver the whole performance for the owner of the machine.

Condor defines different execution environments as universes. When submitting the job, a submit file must be created that usually contains the name of the executable and the universe, the name of files to store standard output, error and log and parameters if needed. Job requirements can also be defined by ClassAds that is taken into account by condor when searching for appropriate resource.

Each machine might define its own policy detailing how, when and what kind of jobs can be accepted and executed through the ClassAds mechanism. Based on the rules of the machine and the requirements of the jobs Condor performs matchmaking.

When submitting a PVM job, Condor searches and allocates the minimum number of available nodes defined in the submit file. Condor usually allocates only one process per computing resource i.e. processor. When all the nodes are ready to execute job, the job is started on the master node where the master process of the job is expected to spawn the required number of PVM processes defined in the submit file.

Starting a new PVM process on a remote node requires the initialisation of the PVM daemons. Condor is integrated into PVM in order to perform resource related activities on behalf of the PVM environment. When a PVM process terminates its node is cleaned up and the master process is expected to spawn new ones if necessary. In case a node becomes unavailable because - for example - its owner starts using it, PVM process running on it is notified to terminate.

When the job has finished its execution, every allocated node is deallocated and the output of the job is available in its working directory.

This short overview is required to understand how the migration mechanism of GRAPNEL application works under the Condor job-scheduling system.

4.1.3 Self-coordinated migration in the GRAPNEL application

4.1.3.1 Assumptions

Assumptions for the self-coordinated migration are the following ones:

1. Resource management related operations (e.g. pvm_addhost) of PVM is correctly handled and served by the job scheduler.
2. Node vacation in case of resource removal is realised by soft termination of the user processes issued by the job scheduler. Soft termination lets the process handle the event and execute any procedure before exit.

These assumptions are required for the migration procedure to be executed in a correct way. Since these assumptions are fulfilled by the previously introduced Condor job-scheduler, in the rest of the section the Condor system is considered to demonstrate the process migration mechanism.

4.1.3.2 Key components

Naturally, the migration procedure is based on the support of the GRAPNEL checkpointing framework integrated in the application developed using the PGRADE programming environment. The migration mechanism is based on the active participation of four elements. They are introduced briefly, details can be found in section 3.1.3.2:

- GRAPNEL Server (GS): Since this component is managing the state of the processes and coordinates the checkpointing procedure, the migration is also coordinated by this component.

- GRAPNEL library (GL) with the checkpoint extension: checkpoint extension of this library performs checkpointing and resumption of the migrating user process.

- Dynamic checkpoint library (CL): loaded at process start-up and activated by receiving a checkpoint event, reads the process memory image and passes this information to the Checkpoint Server

- Checkpoint Server (CS): a logical component (can be part of GS) that receives checkpoint data via a communication channel, stores it into a file or vice versa.

4.1.3.3 Preparation of the migration procedure

First an instance of the Checkpoint Server (CS) is initiated in order to transfer checkpoint files to/from the dynamic checkpoint libraries (CL) linked to each processes of the application. After starting the application, each process of the application automatically loads CL at start-up that checks the existence of a previous checkpoint file of the process by connecting to CS. If it finds appropriate checkpoint file for the process, the resumption of the process is automatically initiated by restoring the process image from the checkpoint file otherwise, it starts from the beginning.

When the application is launched, the first process that starts is the GRAPNEL Server (GS) performing the coordination of the client processes. It starts spawning the client processes to create the topology of the parallel application. Whenever a process becomes alive, it first initiates its integrated checkpointing support, checks for checkpoint file and gets contacted to GS in order to download parameters, settings, etc. When each process has performed its initialisation, GS instructs them to start execution and hence, the application is running.

4.1.3.4 The migration procedure

Figure 42 shows the main steps of the migration protocol applied between the clients and the GS. The migration protocol is overlapped with the checkpointing protocol since the latter one is part of the first one.

While the application is running and the processes are doing their tasks the migration mechanism is inactive. Migration is activated when a client process detects that it is about to be killed (‘Termination’ for client A in Figure 42). The client process immediately informs GS (REQUEST_for_chkpt) that in turn initiates the checkpointing of all the client processes of the application. Time for executing this protocol is assumed to be ensured by Condor through its configurable properties.

For a client process checkpointing is initialised either by a signal or by a checkpoint message (DO_chkpt) sent by GS in order to make sure that all processes are notified regardless of performing calculation or communication. Details of this interruption are described in sections 3.1.3.4, 3.1.3.5 and 3.1.3.6. When notified processes are prepared for checkpointing (READY_to_chkpt), they are instructed by GS to initiate synchronisation (DO_sync) of messages. The synchronisation ends up with all the in-transit messages stored in memory of the client processes. Finally, client processes send their memory image to the checkpoint server to store.

Then all checkpointed processes wait (DONE_chkpt) for further instruction from GS whether to terminate or to continue its execution. GS terminates the clients to be migrated by the appropriate (DO_exit) message (Client A in Figure 42) and then GS initiates new node allocation through PVM for each terminated processes.

When host allocation is performed, a new instance of the terminated processes is spawned on the newly allocated nodes. Each migrated process automatically loads CL that checks for the existence of checkpoint file of the process by connecting to CS. This time the migrated processes will find their checkpoint file and hence their resumption is automatically initiated by restoring the process image from the checkpoint file.

The migrated processes execute post-checkpoint instructions before resuming the real user code. The post-checkpoint instructions serve for initialising the message-passing layer and for registering at GS (DONE_restoration). When all the checkpointed and migrated processes are ready to run, GS allows them to continue their execution (DO_continue).

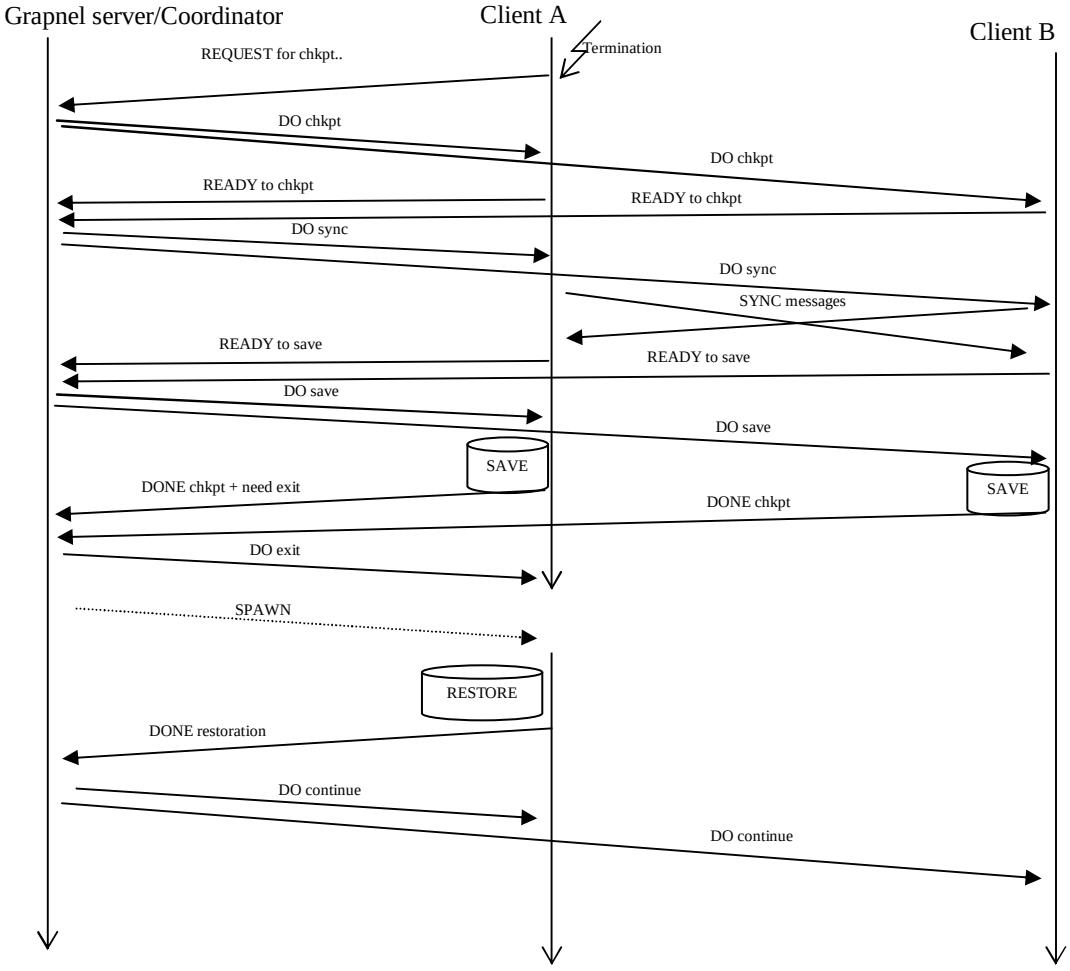

Figure 42 Migration protocol in the GRAPNEL application

4.1.3.5 Migration among nodes under the Condor job scheduler

The GRAPNEL checkpointing system has been integrated with PVM as it is detailed in section 3.1. The original purpose/motivation of this work is to make GRAPNEL applications capable of migrating its processes from one host to another relying only on PVM support where a native PVM environment is built on the nodes of a cluster.

Since jobs and resources in a cluster are supervised by a local job-scheduler the flow of migration integrated in the GRAPNEL application must be able to operate in a transparent way.

In the Condor job-scheduling system the execution of PVM applications is performed as a Master-Worker (MW) mechanism [73]. The basic principle of the Condor MW model is that the master process spawns workers to perform the calculation and continuously checks whether the workers successfully finish their calculation. In case of a failure the master process simply spawns new workers passing them the unfinished work.

The situation when a client process fails to finish its calculation usually comes from the fact that Condor removes the task because the assigned node is no longer available. This action is called vacation of the client process. In this case the master node receives a notification message through PVM indicating that a particular node has been removed from the PVM machine. As an answer the master process tries to add new PVM host(s) to the virtual machine with the help of Condor, and gets notified when host inclusion is done successfully. At this time it spawns new worker(s).

For running a GRAPNEL application, the application continuously requires the minimum amount of nodes to execute the processes. Whenever the number of the nodes drops below the minimum, the GRAPNEL Server (GS) tries to extend the number of PVM machines above the critical level. It means that the GS process behaves exactly the same way as the master process does in the Condor MW system.

Under Condor the master PVM process is always started on the submit machine (Step 1 on Figure 43) and is running until the application is finished. It is not shut down by Condor, even if the submit machine becomes overloaded. Condor assumes that the master process of the submitted PVM application is designed as a work distributor. The functionality of the GRAPNEL server process fully meets this requirement, so GRAPNEL applications can be executed under Condor without any structural modification and the GS can act as the coordinator of the checkpointing and migration mechanisms as it was described previously.

Whenever a process is to be killed (Step 2 on Figure 43) (e.g. because its node is being vacated), an application-wide checkpoint must be performed and the exited process should be resumed on another node. The application-wide checkpointing is driven by GS, but it can be initiated by any client process which detects that Condor tries to kill it. In this case the client process notifies GS to perform a checkpoint. After this notification GS instructs every process to perform checkpointing (Step 3 on Figure 43). After checkpointing, all the client processes wait for further instruction from the server whether to terminate or continue the execution. GS sends a terminate notification to those processes that must migrate.

At this point GS waits for the decision of Condor that tries to find underloaded nodes (Step 4 on Figure 43) either in the home Condor pool of the submit machine or in a friendly Condor pool. Friendly Condor pool of a pool is the one from which or to which the pool accepts/sends jobs. This is called flocking [33]. Flocking can be uni- or bi-directional depending on the friendliness relation. When two pools are flocked to each other it means that they share resources and jobs. In Figure 43 migrations is done to a friendly pool. However, migration to a node of a local or friendly pool does not differ technically.

The resume phase is performed only when the PVM master process (GS) receives a notification from Condor about new host(s) connected to the PVM virtual machine. When every terminated process is migrated to a new node allocated by Condor, the application can continue its execution.

This working mode enables the PVM application to continuously adapt itself to the changing PVM virtual machine by migrating processes from the machines being vacated to some new ones that have just been added. Figure 43 shows the main steps of the migration between friendly Condor pools. Notice that the GRAPNEL Server and Checkpoint Server processes remain on the submit machine of the home pool even if every client process of the application migrate to another pool.

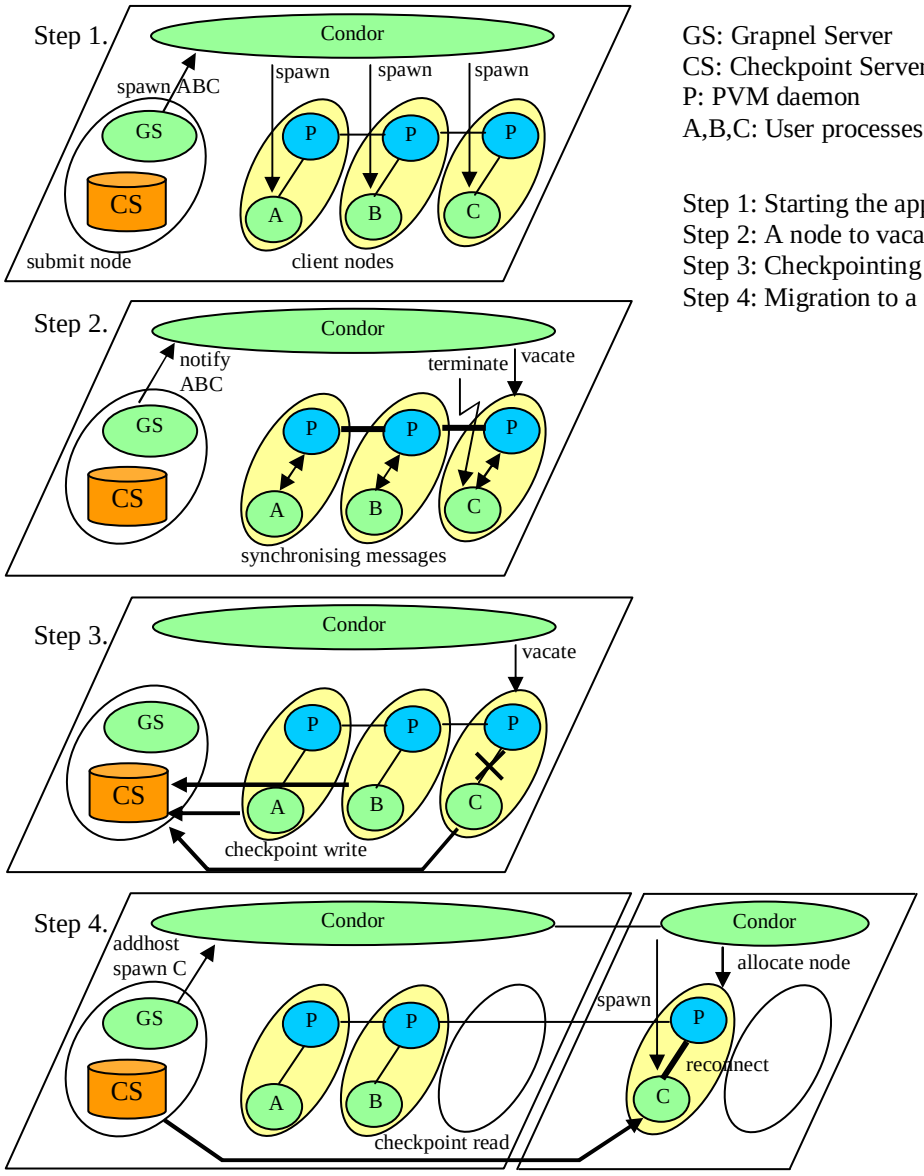

Figure 43 Migration of GRAPNEL application among Condor nodes

It should be noticed that Condor does not provide checkpointing for any kind of PVM applications it only provides application level support for fault-tolerant execution of MW type PVM applications as it is detailed in section 1.3.2 and 2.1.6.2.

4.1.4 Modelling

Process migration introduced in section 4.1 is explicitly part of all the three elaborated (CP_{ground}, $CP_{grapnel}$ and CP_{tckpt}) models. The corresponding ASM rules make sure that the migrating processes are going through the necessary phases in the model. Migration of a process consists of two execution segments: termination and resumption. Termination segment ensures that the process is checkpointed and terminated while resumption segment starts a new process and rebuild its previous state. Each process must be driven by the coordinator through the two segments when migration happens. The corresponding way of process migration is highlighted in Figure 44.

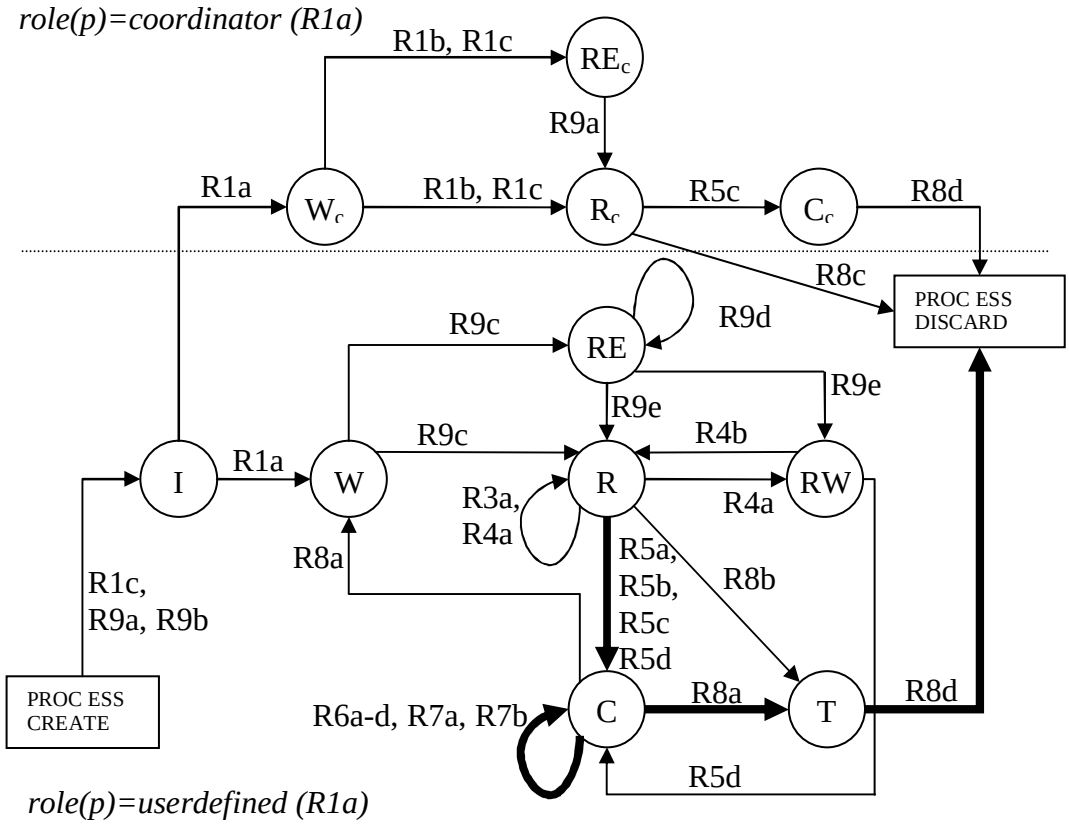

Figure 44 Process checkpointing and termination in $CP_{grapnel}$

Initially a process is in Running (R) phase (see Figure 44). When notification for migration arrives, interruption and checkpointing is performed causing the process to change its phase to Checkpointing (C). Rules R5* perform the proper interruption of the userdefined processes, while rules R6* and R7a, R7b performs the correct saving of the internals. After the process saved its internals successfully, it simply changes its phase to Terminating (T) in order to indicate the end of checkpointing for the coordinator. Finally, the rule R8d ensures that the process is discarded.

Similarly to the procedure, the corresponding way of process initialisation is highlighted with thick lines in Figure 45. Process creation leads the process to Initialisation (I) then Waiting (W) phases through R1* rules. The process is then instructed by the coordinator to perform resumption (RE) with rule R9c. When all the processes have successfully performed resumption, coordinator instruct them to continue running (R).

The migration mechanism realised by the $CP_{grapnel}$ ASM model is also analysed in section 3.1.5.2, where the relevant phases of the migrating processes are discussed and compared to the original CP_{ground} model.

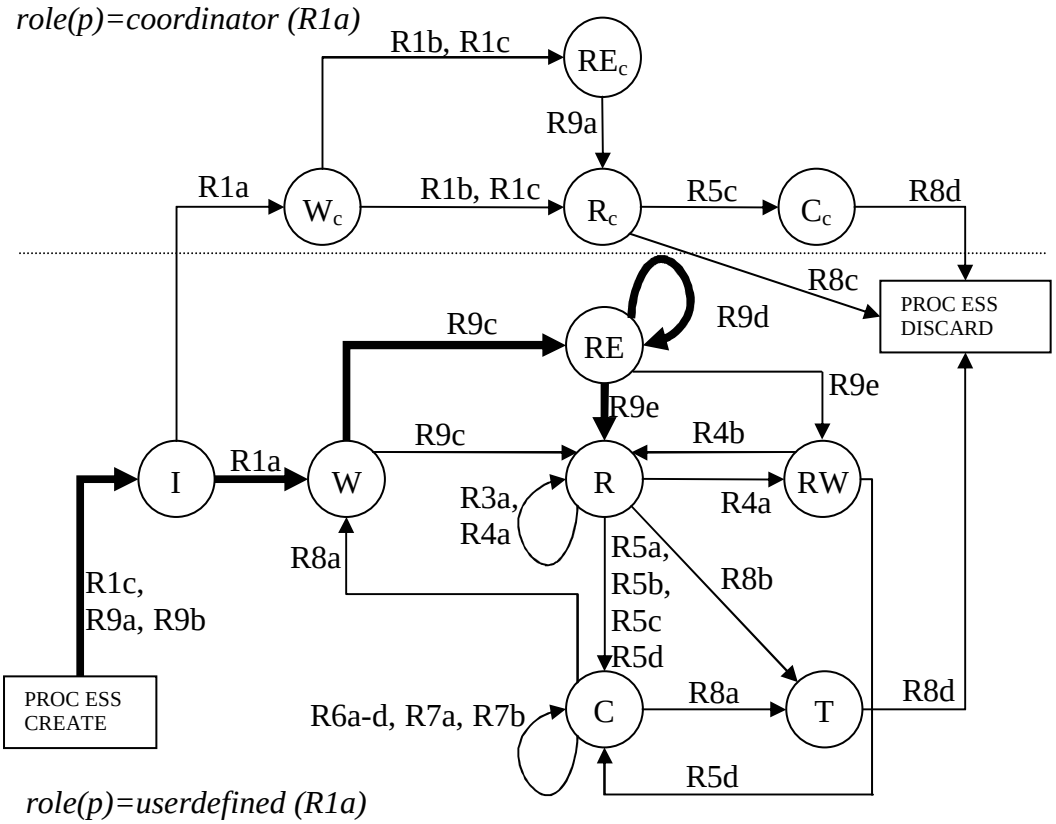

Figure 45 Initialisation and resumption segment in $CP_{grapnel}$

4.1.5 Summary

Since Condor supports only fault-tolerant execution of MW applications without checkpointing support the resulted mechanism of the combined Grapnel-PVM-Condor triumvirate enables Condor to provide execution with fault-tolerant and migration support for PVM applications developed by P-GRADE programming environment.

The most significant features of the migration solution demonstrated in this section are as follow:

- automatic detection of resource loss
- automatic checkpointing in case of resource loss
- allocation of resources on demand
- automatic resumption of terminated processes

Based on the features above the Grapnel application in the Condor environment is able to provide the following novelties:

- automatic self-healing in case a process aborts
- automatic self-adoption for the continuously changing resource availability
- process migration transparently for the cluster middleware
- process migration transparently for the programmer

4.2 Application migration on ClusterGrid

4.2.1 Overview

In the migration solution presented in thesis 3.1 the procedure is driven by a special coordinator process – integrated in the GRAPNEL application – where the termination of this process causes the entire application to be shutdown. However, this coordination process must be terminated during the application migration since link between the clusters in a ClusterGrid are usually not exists. To eliminate this problem I elaborated an application migration solution and then I mapped it to the $CP_{grapnel}$ model. Based on the results thesis 3.2 is stated.

> **Thesis 3.2:** *The GRAPNEL application implements a consistent, global state-space migration for message-passing parallel applications by saving and restoring the integrated coordination process. It enables the migration of the entire application among independent (not using each others' resources) clusters. In addition, the elaborated solution adapts itself to the internal rules of the $CP_{grapnel}$ ASM model.*
>
> Related publications are [1][10][13][15].

4.2.2 Motivation

Process migration in distributed computational environments can solve the problem of balancing the load of the nodes within a cluster, or removing a process of a parallel job from a node. However, process migration is not enough to move processes among clusters since in some cases the application may suffer from weak resource supply within a cluster. To overcome this limitation application migration is needed.

In that special cases when the whole ClusterGrid is built based on Condor job-manager it is possible to allocate the migrating task to another cluster in case the two pools are flocked to each other. This is a special case when migration of the whole application is not necessary. Running a Grapnel application under Condor and migrating client processes among nodes can be performed by tight cooperation of the Grapnel Server and the checkpoint library. While the application is using only the resources that are under the authorisation of the executor Condor pool and friendly Condor pools, Grapnel Server can coordinate everything from the submit machine.

However, Condor flocking cannot be applied in generic Grid systems where the pools (clusters) are separated by firewalls and hence global Grid job managers should be used.

In such systems if the cluster is overloaded, i.e., the local job manager cannot allocate nodes to replace the vacated nodes; the whole application should migrate to another less loaded cluster of the Grid. It means that not only the client process but even the Grapnel Server should leave the overloaded cluster. This kind of migration is called application migration opposing the process migration where the Grapnel Server does not migrate.

4.2.3 Design issues

In order to leave the pool - i.e. migrate the whole application to another pool – some extra capabilities are needed for the GRAPNEL application. These are as follow:

- Deciding when to initiate application migration

- Performing self checkpointing of the GRAPNEL server process
- Performing checkpointing of opened file descriptors

The migration among the clusters is mapped to checkpointing, termination, resubmission and recovery steps which are detailed in the next few sections. In order to resubmit the application an external Grid component (e.g. broker) must contribute to the file movement and resubmission procedure since the application is not running. The following issues must be addressed in order to realise the support of application migration:

- Deciding whether to initiate resubmission
- Performing automatic resubmission by Grid Application Manager

4.2.3.1 Initiation of application migration

Based on the reason for application migration, two different initiation methods are distinguished:

- Explicit initiation, when external participant notifies the application to perform shutdown with application wide checkpointing
- Implicit initiation, when the application decides so under predefined circumstances

As it has been detailed in section 4.1 process migration is initiated by the local scheduler by a terminate notification when the node is about to be vacated. Since the master process – GRAPNEL server in this case – is performing the checkpoint coordination, termination of this component implicitly causes the whole application to be shut down. Based on the concept, application migration can be initiated by terminate notification sent to the master process of the GRAPNEL application. This way of initiation is called the explicit service request.

Alternatively, an implicit service request can be automatically generated by the GRAPNEL server itself. Decision on automatic application migration is based on the available and required resources provided by the job-scheduler for the application. GRAPNEL server continuously monitors the state of its user processes. When available nodes drop below the minimum required by the application, some of the processes are checkpointed and shutdown by the coordinator, application suspends its execution and the coordinator is waiting for new machines to be allocated for its pending processes. In case the application is suspended for a predefined time, the coordinator decides to shutdown all its processes and exit. This way of initiation is called the implicit service request.

In case explicit service request is performed, the initiator of application migration is usually the Grid Broker component (detailed below) of the ClusterGrid.

In a computational Grid various resources are collected where typically one Grid Broker component coordinates the utilisation of the underlying resources i.e. clusters. Grid Broker usually monitors the aggregated performance and availability of the clusters and performs the mapping of applications to resources based on the application requirements and the actually measured performance. The broker selects the cluster which the application must be assigned to or removed from. In case of dynamic resource allocation policy the broker may decide to remove a job from its

cluster and resubmit it to another one due to resource availability conditions in the clusters.

The Grid Broker in this context is not aware of the checkpointing capability of the application, so removing the application and resubmitting is the same as a restart of the job. However, based on the automatic application-wide checkpointing and restart support integrated into GRAPNEL application, execution is going to be continued from the point where it was terminated. The integrated support can be considered as an optimisation for execution.

4.2.3.2 Self-checkpoint capable server process

In order to support application migration the basic requirement for the application is to produce a consistent, global checkpoint. For a GRAPNEL application, checkpointing of the user processes is done as it is described in section 4.1. To leave the cluster, the server process of the application must be checkpointed additionally.

The server process coordinates the creation of user processes and communication links among them i.e. the topology. It also manages file operations of the user processes and the whole checkpoint mechanism. Therefore it has all the required knowledge to rebuild the application at resumption phase once the server has come to life. The one and only issue to be addressed is how to checkpoint the coordinator (server) process.

When the initiation of application migration happens, all the user processes are checkpointed and shutdown. In this state:

- there are no pending messages among the processes, since the last message from the user processes towards the server is the notification about the termination
- there are no communication links, since only one server process remained in the application
- consistent checkpoint information for all user processes is stored

The coordinator at this point performs the following steps:

- disconnects the message-passing layer i.e. exits from PVM
- as a standalone process, stores the status of the open file descriptors and closes them
- finally, activates the checkpointing library to create a checkpoint of the server process and then exits

In case the application is submitted, the first process to come alive is always the server one. This is ensured by the application itself. The first instance of process checks whether it is the first one and invokes the entry procedure of the server code if that is the case.

After resubmission to a new cluster, when the server code is started to be executed, it first checks for checkpoint information. If it is found, reopening working files and re-enrolling into PVM is performed. At this point server process is in the same state as it was on that cluster where user processes finished checkpointing and shutdown.

Based on the previous mechanism, the migration of server process of the parallel application is converted to the migration of a single standalone process. Checkpointing and resumption of user processes are performed before and after the migration of the server process, respectively.

4.2.3.3 Migration of working files of the application

In the GRAPNEL application files are handled by the GRAPNEL server process. Whenever a user process is about to open a file, a service is redirected for the server process which then performs the actual opening of the files and returns a reference number. Similarly, reading, writing and closing are also realised in a central way.

While process migration does not affect the handling of opened files, for correct migration of the entire application, migration of the working files are also necessary since the server also exits upon application migration. This mechanism requires the following steps to be elaborated:

- checkpointing of the opened file descriptors at the time of server shutdown
- mapping of referenced full file names to actual file names
- reopening of working files

Checkpointing of the opened file descriptors does not require saving the content of the files since no rollback of the execution is realised. In GRAPNEL server, since all opened files are registered by the server process, a file table is managed. For each user process opened file descriptors are stored in the memory of the server. When saving the states of the opened files only the file table must be archived with file position information added as an extra parameter.

Mapping of full file names referred by the user code to file names created by the program is required for the case when working directory differs on the new cluster after migration performed. When a job is submitted to a cluster the job-scheduler allocates a working directory for it and executes the job. Opening a file can be realised by using absolute or relative names. In the former one, the process usually queries the pathname of the working directory once at its initial phase. Using the same directory name after a migration would cause the job to be aborted. To resolve this conflict the GRAPNEL server performs the mapping of the directory names and the filenames as well.

Reopening working files are done at the resumption phase of the GRAPNEL server. When the application is resubmitted to a new cluster and the GRAPNEL server process is recovering itself, reopening the files and reassignment of their file descriptors are done as a final step in its own resumption phase, but before resumption of any user process. Reopening working files can be realised by system calls, the required parameters are stored by the file table of the GRAPNEL server process.

4.2.4 Flow of migration

The central Grid Broker component of a ClusterGrid has access to the clusters and takes care of the execution of the parallel application. The Broker tries to optimize the execution time of the application and the throughput of the ClusterGrid presumably. In order to support this optimisation application migration can be a solution.

The Broker must be able to detect the need for application migration to submit and re-submit the application, to force the application to checkpoint itself, to identify working files and to transfer them among the different clusters as shown in Figure 46. In case of application migration the following scenario is executed:

1. User submits his/her application
2. Grid Broker allocates a target cluster
3. Grid Broker generates submit file for the cluster
   Submit file for an application stores the definition of executable, input- and output files, parameters and resource requirements among others. The format of the submit file depends on the actual cluster job-scheduler the application is about to be submitted.

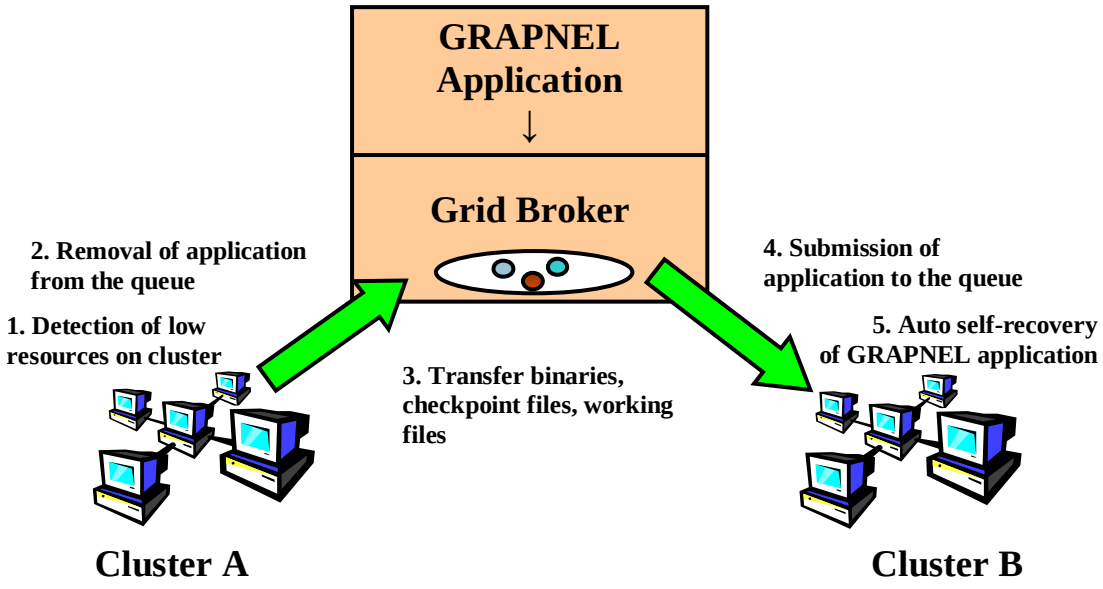

Figure 46 Phases of application migration in ClusterGrid

4. Grid Broker submits the application to the target cluster
5. Grid Broker decides to perform total migration (Step 1 on Figure 46)
   The Broker component continuously monitors the status of the cluster and initiates application migration in case the load of the cluster is inadequate (i.e. overloaded or underloaded).
6. Termination of the application is initiated by the Broker (Step 2 on Figure 46)
   Broker notifies the local scheduler to terminate the job and the local scheduler notifies the application by sending a terminate notification to the server process.
7. Grid Broker stores the content of the working directory of the application
8. Grid Broker reassigns the application to a new cluster
9. Copying the content of the working directory to the newly allocated cluster is performed by the Broker (Step 3 on Figure 46)
   Working directory contains the collection of all checkpoint files, application working files and executable. The files represent a consistent state with the application at this point.
10. Grid Broker regenerates the submit file for the new cluster
11. Resubmission is done (Step 4 on Figure 46)

Broker submits the job for execution to the job-scheduler of the newly allocated cluster.

12. At this point, GRAPNEL application recovers its execution from the point where it was terminated last time (Step 5 on Figure 46)

Based on the scenario of application migration, it can be seen that only minimum effort is needed by the Grid Broker to utilise the service provided by the integrated GRAPNEL checkpointing and migration framework.

4.2.5 Modelling

Application migration introduced in section 4.2 is explicitly part of all the three elaborated (CP_{ground}, $CP_{grapnel}$ and CP_{tckpt}) models. The corresponding ASM rules make sure that the each of the application processes (including the coordinator) are going through the necessary phases in the model. Migration of an application consists of two execution segments: termination and resumption. Termination segment ensures that the application is checkpointed and terminated while resumption segment starts a new process, it becomes coordinator, rebuilds itself, respawns and rebuilds all userdefined processes. Each userdefined process must be driven by the coordinator through the two segments when migration happens. The corresponding way of application migration is highlighted in Figure 47.

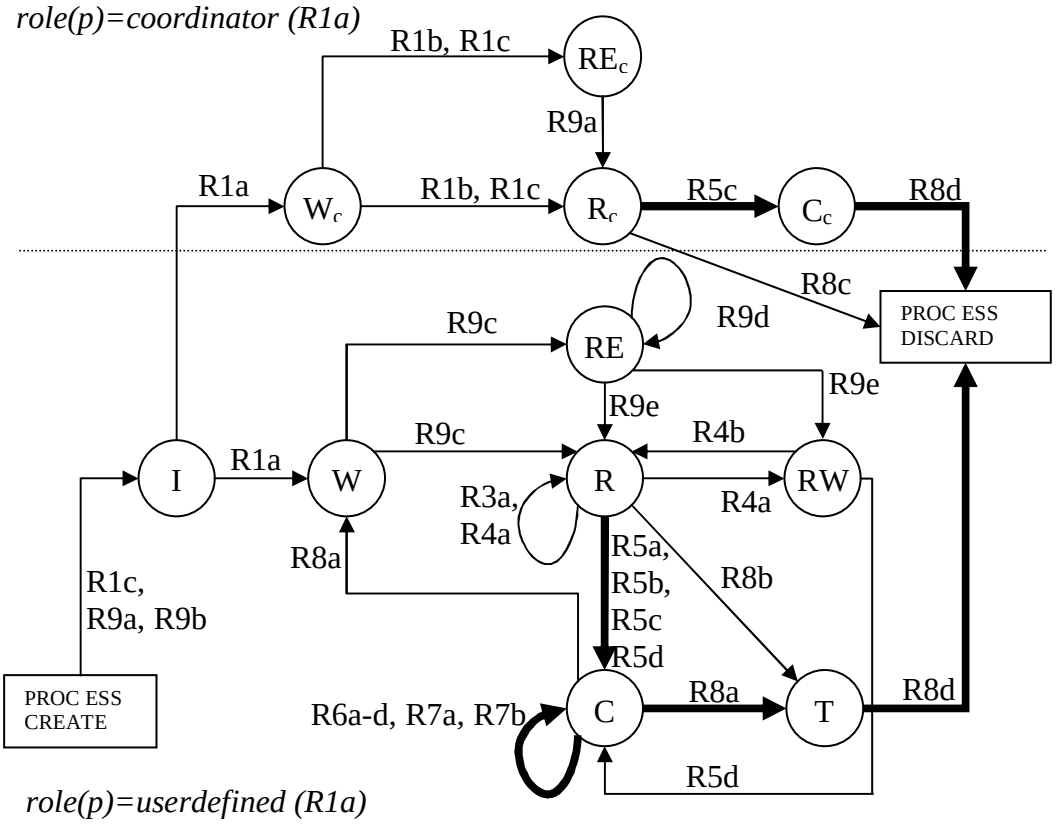

Figure 47 Application checkpointing and termination in $CP_{grapnel}$

Initially each process of the application is in Running (R) phase (see Figure 47). When decision for migration happens, coordinator instructs each process to checkpoint and to terminate itself similarly to the case when process migration happens (see section 4.1.4). When userdefined processes are all discarded, coordinator

initiates its self-saving mechanism by changing to the phase Checkpointing (C_c) and discards itself.

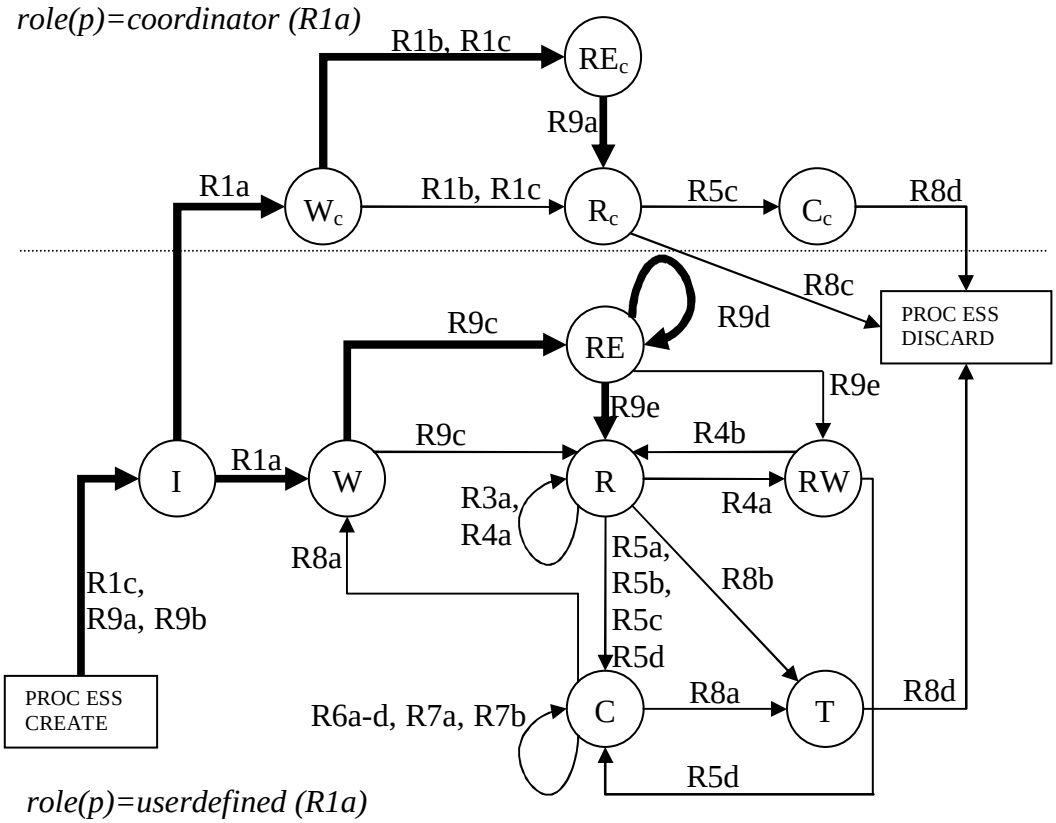

Figure 48 Application resumption in $CP_{grapnel}$

Initialisation of coordinator ensured when the application is started by the scheduler next time. Coordinator is lead through the phases called Initialisation (I) and Waiting (W_c) by the rules R9* and R1* (see Figure 48). Coordinator resumes itself (RE_c) through rules R1* and finally, respawns and rebuilds userdefined processes similarly to the initialisation procedure in case of process migration (see section 4.1.4). When ready, it continues in normal mode (R_c).

4.2.6 Summary

In this section the final result of the work is presented, since application migration mechanism described and demonstrated in this section is defined as one of the main goals of the whole dissertation.

Looking into the Use Cases defined in section 2.1.3, one of the most significant scenarios in a ClusterGrid environment is the migration of the entire parallel application from one cluster to another without active participation of the middleware components of the source and target clusters. The outcome of the work presented in this section is that middleware and application transparency is achieved, since the whole migration fits in the normal operation of the local job-schedulers and the user is neither forced to alter the source code of the algorithm.

As the scenario shows, to utilise application migration Grid Brokers must be slightly adjusted, since the Broker must be aware of the checkpointing capability of the application. However, it was not among the goals of this dissertation.

The main features of the application migration can be summarised by the following points:

- built-in coordinator checkpoints its processes when being terminated
- self-checkpoint of the coordinator is performed
- built-in coordinator discovers the existence of checkpoint information
- application rebuilds its global state by itself with the help of a built-in coordinator

The novelties of the solution presented in this section can be summarised by the following points:

- automatic self-checkpoint and recovery mechanism derived from library-level parallel checkpointing
- generation of checkpoint information by the application itself derived from application-level parallel checkpointing
- built-in checkpointing service migrates among Grid sites, i.e. clusters

As a summary, the required use cases defined in section 2.1.3 are supported successfully in a way that no modification of the migrating algorithm and no modification of the cluster components are required.

5 Discussion and Conclusion

Implementation status

The process migration mechanism introduced in the dissertation has been developed for both P-GRADE and TotalCheckpoint tools. The P-GRADE version was also demonstrated in several events like EuroPar'2003 conference (Klagenfurt, Austria) [16], Hungarian Grid Day (Budapest, Hungary), SuperComputing'2003 (Phoenix, USA), and the IEEE Cluster Computing'2003 (Hong-Kong, China) [14][15].

In these demos, three clusters were connected (MTA SZTAKI, Technical University of Budapest, and University of Westminster) to provide a friendly Condor pool system. A parallel urban traffic simulation application was submitted on the SZTAKI cluster. Then the cluster was artificially overloaded P-GRADE migrated all the processes except the Grapnel Server to Westminster. After resuming the application at Westminster, the procedure was repeated and the application migrated to the cluster in Technical University of Budapest. Since the migration framework, the Mercury Grid monitor [1] and the PROVE visualization tool [1] were integrated into the P-GRADE Grid run-time environment, the migration of the application was on-line monitored and visualized.

For application migration mechanism prototypes has also been implemented for P-GRADE and TCKPT, by which successful removal and resubmission of a parallel application were performed on Condor clusters. The enhanced version of TCKPT only reached the design phase.

Performance

Regarding the performance of checkpointing, the tasks of migration are checkpoint writing, reading, allocation of new resources and some coordination overhead. The time spent for writing or reading the checkpoint information through a TCP/IP connection definitely depends on the size of the process to be checkpointed and the bandwidth of the connection between the nodes. However, writing and reading checkpoint files can be done locally and the files can be spread among the nodes through a networked file system.

The overall migration time of a process includes the responding time of the resource scheduling system e.g. while Condor vacates a machine, the matchmaking mechanism finds a new resource, allocates it, initialises PVM daemons and notifies the application. Finally, cost of synchronisation of messages among the processes and some cost used for coordination are both negligible, since it is measured to be less than one or two percent(s) of the overall migration time.

Future work

There are two main directions that have been already considered. The first one is to apply the introduced methods and techniques on MPI applications. The main challenges are comes from the lack of support for dynamic process creation, termination and signalling. While the specification for MPI-2 contains dynamic process creation, there is still no widely accepted implementation (like MPICH for MPI-1). Another future direction is to integrate P-GRADE and TCKPT checkpointer into a Grid Checkpointing Architecture (GCA) developed by PSNC (Poznan

Supercomputing and Networking Center) through standardised interfaces. There are already some steps done towards this direction [6][7][22][23][23].

Conclusion

The work described in this dissertation aims at providing a solution for consistent, global state transfer of message-passing parallel algorithms in Grid environments.

The initial steps of the dissertation aim at defining the environment (components), applications and their features. After the proposed environment is detailed definition of some use cases is introduced. These use cases determine what services are intended to be supported by the forthcoming work described in this dissertation.

Subsequently the compatibility and integrity conditions are defined to be fulfilled by the checkpoint and migration service in order to reach middleware transparency. Conditions are formalised by using Abstract State Machines (ASM) in order to give a precise definition. An analysis of the related work is performed to compare the various checkpointing and migration tools. Analysis concluded that none of the examined works satisfy every condition at the same time.

A new ClusterGrid checkpointing method is introduced that defines the most important design criteria in order to provide consistent, global state transfer of unmodified algorithms in a Grid environment without any additionally required support on cluster level. The CP_{ground} ASM model has been elaborated by defining universes, signatures, initial state and the corresponding rules. This model has been validated against the definitions of the ClusterGrid checkpointing method to show that the defined CP_{ground} is a correct model of the method.

As a proof of the concept, two different tools are introduced. The first one is the GRAPNEL checkpointing and migration framework which is an integrated tool that resides in parallel applications developed by the P-GRADE. The main principles, structures of the framework and the flow of checkpointing have been introduced. The $CP_{grapnel}$ ASM model has been elaborated. As a next step, it has been shown that $CP_{grapnel}$ model is a correct refinement of the CP_{ground} model.

As a second solution, the TotalCheckpoint checkpointing framework is introduced that is independent from the P-GRADE programming environment and supports native PVM applications. Similarly to the previous model, CP_{tckpt} has also been elaborated and the equivalence to $CP_{grapnel}$ has been shown, too.

Finally, based on the GRAPNEL checkpointing framework process and application migration is elaborated.

6 Acknowledgement

The work behind in this dissertation covers approximately a decade in my life. During this time I was working with numerous people in lots of national and international projects to whom I am really grateful for their cooperation, support and help. Without giving an endless list of people I would like to express my thanks to several of them who contributed the most to my dissertation.

First of all, I would like to thank my advisor, Péter Kacsuk, for offering me a research position first in MTA KFKI-MSZKI and later in MTA SZTAKI. His guidance and support has helped me to become a researcher. He taught me to be able to express and write down my thoughts. During the last decade, he always found the way how to motivate me when required. As a leader of the laboratory, he ensured the background for my research and encouraged me to publish and to work on new solutions.

I would like express my thanks to Zsolt Németh, my colleague, who gave important support in elaborating the theoretical background of my dissertation. Without his support, it would have taken twice as much time as it took me to reach the end of the dissertation. He continuously stayed behind me as a consultant and he was the one who led me into the secrets of modelling by Abstract State Machines (ASM). He continuously revised my papers, my thoughts and ideas. I am really grateful to him for listening to me and commenting my ideas even when he could hardly keep the deadline of his actual work.

I would like to thank all the members of the Laboratory of Parallel and Distributed Systems in MTA SZTAKI to their ideas and comments, support and cooperation. Special thanks to Róbert Lovas, Norbert Podhorszki, Gábor Dózsa, Dániel Drótos who devoted huge amount of time to develop the P-GRADE parallel programming environment and to Zoltán Farkas, Attila Csaba Marosi, Gábor Gombás who helped me developing the TotalCheckpoint system. Several years of research and development ensured that the work presented in this dissertation works in practice as well.

My father and my mother contributed far most for this dissertation. Supporting my life and my studies for more than 20 years is something for which I will always be grateful in my life. Giving a safe background let me focus on my studies and later on my work in computer science.

Finally, I would like to thank my wife, Andrea Kovácsné Szabó. She gave me a stable background by surrounding me with a loving family including with three kids. Here I have to mention that ensuring calmness and silence with three quick and smart little children is not an easy task. She continuously encouraged me and was patient when necessary. Without her support I would not be able to achieve my PhD.

7 References

[1] P. Kacsuk, G. Dózsa, J. Kovács, R. Lovas, N. Podhorszki, Z. Balaton and G. Gombás: “P-GRADE: a Grid Programming Environment”, Journal of Grid Computing Vol. 1. No. 2, pp. 171-197. 2004.

[2] Kovács, J., Kacsuk, P.: “The DIWIDE Distributed Debugger”, Quality of Parallel and Distributed Programs and Systems, special issue of Journal of Parallel and Distributed Computing Practices, PDCP Vol.4, No. 4, Eds: P.Kacsuk, G.Kotsis, pp. 331-347, 2001

[3] J. Kovacs: "Transparent Parallel Checkpointing and Migration in Clusters and ClusterGrids", International Journal of Computational Science and Engineering, IJCSE, 2006, (to appear)

[4] József Kovács, Peter Kacsuk, Radoslaw Januszewski, Gracjan Jankowski: "Application and Middleware Transparent Checkpointing with TCKPT on ClusterGrid", Future Generation Computer Systems, selected papers of DAPSYS2006, (accepted)

[5] J. Kovacs, R. Mikolajczak, R. Januszewski, G. Jankowski: "Application and middleware transparent checkpointing with TCKPT on Clustergrid", Proceedings of 6th Austrian-Hungarian Workshop on Distributed And Parallel Systems, DAPSYS 2006, Innsbruck, Austria, September 21-23, 2006, pp. 179-189.

[6] G. Jankowski, J. Kovacs, R. Mikolajczak, R. Januszewski, N. Meyer: "Towards Checkpointing Grid Architecture", Parallel Processing and Applied Mathematics – Conference on Parallel Processing and Applied Mathematics, PPAM2005, Poznan, Poland, Lecture Notes in Computer Science, Vol. 3911/2006, pp. 659-666, Springer, 2006, ISBN 978-3-540-34141-3

[7] G. Jankowski, R. Januszewski, J. Kovacs, N. Meyer, R. Mikolajczak: "Grid Checkpointing Architecture - a revised proposal", Proc. of the 1st CoreGRID Integration Workshop, pp. 287-296, Pisa, 28-30, November, 2005

[8] Kovács József, Farkas Zoltán, Marosi Attila: "Ellenőrzőpont támogatás PVM alkalmazások számára a magyar ClusterGriden", Networkshop, Szeged, 2005

[9] Jozsef Kovacs: “Making PVM applications checkpointable for the Grid” Proc. of the Microcad 2005 Conference, Section N, Miskolc, 2005, pp. 223-228

[10] József Kovács: „Process Migration in Clusters and Cluster Grids”, Distributed and Parallel Systems: Cluster and Grid Computing, Kluwer International Series in engineering and Computer Science, Vol. 777, Dapsys 2004, Budapest, Hungary, pp. 103-110.

[11] József Kovács, Péter Kacsuk: “A migration framework for executing parallel programs in the Grid”, In: Grid Computing – Second European AcrossGrids Conference, AxGrids 2004, Nicosia, Cyprus, Lecture Notes in Computer Science, Vol. 3165, pp. 80-89, Springer-Verlag, 2004

[12] József Kovács, Péter Kacsuk: ”Improving fault-tolerant execution for parallel applications under Condor”, microCAD International Scientific Conference, University of Miskolc, Miskolc, Hungary, March 18-19, 2004, pp. 251-256

[13] József Kovács, Péter Kacsuk: ”Párhuzamos programok vándorlása a Grid-en”, University of Miskolc, Doktoranduszok fóruma, Gépészmérnöki kar szekciókiadványa, 2003, pp. 158-164

[14] R. Lovas, J. Kovács, G. Gombás, N. Podhorszki, Z. Balaton, P. Kacsuk, I. Szeberényi, T. Delaitre, A. Gourgoulis: ”Migration and Monitoring of P-GRADE Parallel Jobs in the Grid”, IEEE International Conference on Cluster Computing, Hong Kong, 2003. pp 8-11.

[15] P. Kacsuk, R. Lovas, J. Kovács, G. Dózsa, N. Podhorszki: “Metacomputing support by P-GRADE”, GGF8 Workshop on Grid Applications and Programming Tools, 2003

[16] P. Kacsuk, R. Lovas, J. Kovács, F. Szalai, G. Gombás, N. Podhorszki, A. Horváth, A. Horányi, I. Szeberényi, T. Delaitre, A. Terstyánszky, A. Gourgoulis: “Demonstration of P-GRADE job-mode for the Grid”, EuroPar 2003 Parallel Processing, Lecture Notes in Computer Science, Springer-Verlag, LNCS 2790, pp 1281-1286, Klagenfurt, Austria, 2003

[17] Jozsef Kovacs, Peter Kacsuk: “Server based migration of parallel applications”, 4th DAPSYS Conference, Linz, Austria, 29th September-2nd October 2002, pp: 30-37

[18] József Kovács: “Párhuzamos programok checkpointolása és migrációja klasztereken”, Networkshop’2002, Eger, Eszterházy Károly Főiskola, 26th-28th March 2002

[19] Jozsef Kovacs: "Formal analysis of existing checkpointing systems and introduction of a novel approach", CSCS 2006, Szeged, Hungary, June 2006 (honored with Best Talk Award)

[20] Jozsef Kovacs: "PVM & Condor checkpointing", Condor Week 2004, April 14-16, 2004, University of Wisconsin, Madison

[21] G. Jankowski, R. Januszewski, R. Mikolajczak, J. Kovacs: "Scalable multilevel checkpointing for distributed applications - on the integration possibility of TCKPT and psncLibCkpt", CoreGRID Technical Report, TR-0019, March 2006

[22] G. Jankowski, R. Januszewski, R. Mikolajczak, J. Kovacs: "Scalable multilevel checkpointing for distributed applications - on the possibility of integrating Total Checkpoint and AltixC/R", CoreGRID Technical Report, TR-0035, May 2006

[23] G. Jankowski, R. Januszewski, R. Mikolajczak, J. Kovacs: "Grid Checkpointing Architecture - a revised proposal", CoreGRID Technical Report, TR-0036, May 2006

[24] V. S. Sunderam: "PVM: A Framework for Parallel Distributed Computing", Concurrency: Practice and Experience, 2, 4, pp.:315-339, December, 1990.

[25] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek and V. Sunderam, "PVM: Parallel Virtual Machine – a User's Guide and Tutorial for Network Parallel Computing." MIT Press, Cambridge, MA, 1994.

[26] D.A. Bader and R. Pennington, "Cluster Computing: Applications", The International Journal of High Performance Computing, 15(2):181-185, May 2001.

[27] C. Catlett and L. Smarr, Metacomputing, Communications of the ACM, 35 (1992), pp. 44{52.

[28] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid. Enabling Scalable Virtual Organizations", International Journal of Supercomputer Applications, 15(3), 2001

[29] Vic Zandy's checkpointer: www.cs.wisc.edu/~zandy/ckpt

[30] The esky checkpointing tool by David Gibson: http://esky.sourceforge.net

[31] J.S. Plank, M.Beck, G. Kingsley, and K.Li, "Libckpt: Transparent checkpointing under Unix", In Proc. of Usenix Technical Conference 1995, New Orleans, LA, Jan. 1995

[32] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System", Technical Report #1346, Computer Sciences Department, University of Wisconsin, April 1997

[33] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the Grid", in Fran Berman, Anthony J.G. Hey, Geoffrey Fox, editors, Grid Computing: Making The Global Infrastructure a Reality, John Wiley, 2003

[34] Condor homepage: http://www.cs.wisc.edu/condor

[35] D. Drótos, G. Dózsa, and P. Kacsuk, "GRAPNEL to C Translation in the GRADE Environment", Parallel Program Development for Cluster Comp. Methodology, Tools and Integrated Environments, Nova Science Publishers, Inc. pp. 249-263, 2001

[36] E. Börger, "High level system design and analysis using abstract state machines", ASM Workshop, Magdeburg, September 1998

[37] E. Börger, "High Level System Design and Analysis using Abstract State Machines", in D. Hutter et al. (eds.), Current Trends in Applied Formal Methods (FM-Trends 98), LNCS 1641, Springer, pp. 1–43, 1999.

[38] E. Börger, "Why use evolving algebras for hardware and software engineering?", SOFSEM'95, LNCS, 1995

[39] Y.Gurevich, "Evolving algebras 1993: Lipari guide", E.Börger, editor, Specification and Validation Methods, pages 9-36, Oxford University Press, 1995

[40] Y.Gurevich, "Abstract state machines captues sequential algorithms", Technical Report MSR-TR-99-65, Microsoft Research, 1999

[41] E. Börger and D. Rosenzweig, "The WAM – definition and compiler correctness", Logic Programming: Formal Methods and Practical Applications, 1994

[42] E. Börger and I. Durdanovic, "Correctness of compiling occam to transputer code", Computer Journal, 39(1):52-92, 1996

[43] E. Börger and W. Schulte, "Programmer friendly modular definition of the semantics of java", Formal Syntax and Semantics of Java, LNCS, Springer, 1998

[44] E. Börger and U. Glasser, "A formal specification of the pvm architecture", B. Pehrson and I. Simon, editors, IFIP 13th World Computer Congress, volume1, pages 402-409, 1994

[45] Y. Gurevich, "May 1997 draft of the asm guide", Technical Report CSE-TR-336-97, University of Michigan, EECS Department, 1997

[46] Y. Gurevich, "Evolving algebras: An attempt to discover semantics", G.Rozenberg and A.Salomaa, editors, Current Trends in Theoretical Computer Science, pages 266-292, World Scientific, 1993

[47] Egon Börger and Robert Stärk, "Abstract State Machines: A Method for High-Level System Design and Analysis", Springer-Verlag, 2003.

[48] P-GRADE Parallel Program Development Environment, http://www.lpds.sztaki.hu/pgrade

[49] Elnozahy, E. N., Alvisi, L., Wang, Y., and Johnson, D. B. 2002. A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34, 3 (Sep. 2002), 375-408. DOI= http://doi.acm.org/10.1145/568522.568525

[50] S.Kalaiselvi and V.Rajaraman, A Survey of Checkpointing Algorithms for Parallel and Distributed Computers, Sadhana, Vol.25, Part 5, October 2000, pp.489-510

[51] M. Treaster. A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems. ACM Computing Research Repository (CoRR), (cs.DC/ 0501002), January 2005. http://citeseer.ist.psu.edu/treaster05survey.html

[52] Zomaya A Y H 1996 Parallel and distributed computing handbook (New York: McGraw-Hill)

[53] Ralston A, Reily E D 1993 Encyclopedia of computer science 3rd edn (New York: IEEE Press)

[54] Gracjan Jankowski, Rafal Mikolajczak and Radoslaw Januszewski "Checkpoint restart mechanism for multiprocess applications implemented under SGIGrid project." In Proceedings of the CGW2004, 2004.

[55] A. Beguelin, E. Seligman, and P. Stephan. "Application level fault tolerance in heterogeneous networks of workstations." Journal of Parallel and Distributed Computing, 43(2):147155, 1997.

[56] K.M. Chandy and L. Lamport. "Distributed snapshots: Determining global states of distributed systems", ACM Transactions on Computer Systems, 3(1):63-75, February 1985.

[57] R. Strom and S. Yemini. Optimistic recovery in distributed systems. ACM Transactions on Computing Systems, 3(3):204–226, 1985.

[58] L. Alvisi. Understanding the message logging paradigm for masking process crashes. PhD thesis, Cornell University, Department of Computer Science, 1996.

[59] E. N. Elnozahy and W. Zwaenepoel. Manetho: fault tolerance in distributed systems using rollback-recovery and process replication. PhD thesis, Rice University, Department of Computer Science, 1994.

[60] Bhargava B, Lian S-R, Leu P-J 1990 Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms. Proc. IEEE 6th Int. Conf. on Data Eng. pp 182-189

[61] Elnozahy E N, Zwaenepoel W 1992 Manetho: Transparent rollback recovery with low overhead, limited rollback and fast output commit. IEEE Trans. Comput. 41: 526-531

[62] William Gropp, Ewing Lusk, and Anthony Skjellum: "Using MPI", 2nd Edition, MIT Press, ISBN 0-262-57132-3

[63] Al Geist, Ewing Lusk, William Gropp, William Saphir, Steve Huss-Lederman, Tony Skjellum, Andrew Lumsdaine, and Marc Snir.: "MPI-2: Extending the Message-Passing Interface", In EuroPar96, February 1996

[64] George Stellner, "Consistent Checkpoints of PVM Applications", In Proc. 1st Euro. PVM Users Group Meeting, 1994

[65] J. Leon, A. L. Fisher, and P. Steenkiste, "Fail-safe PVM: a portable package for distributed programming with transparent recovery". CMU-CS-93-124. February, 1993

[66] Iskra, K. A., van der Linden, F., Hendrikse, Z. W., Overeinder, B. J., van Albada, G. D., and Sloot, P. M. 2000. The implementation of dynamite: an environment for migrating PVM tasks. SIGOPS Oper. Syst. Rev. 34, 3 (Jul. 2000), 40-55. DOI= http://doi.acm.org/10.1145/506117.506123

[67] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole, "MPVM: A Migration Transparent Version of PVM", Technical Report CSE-95-002, 1, 1995

[68] C.P.Tan, W.F. Wong, and C.K. Yuen, ”tmPVM - Task Migratable PVM”, In Proceedings of the 2nd Merged Symposium IPPS/SPDP, pp. 196-202, 1999.

[69] Pawel Czarnul: ”Programming, Tuning and Automatic Parallelization of Irregular Divide-and-Conquer Applications in DAMPVM/DAC” in International Journal of High Performance Computing Applications, 2003, Vol.17, No.1

[70] Dan Pei, Wang Dongsheng, Zhang Youhui, Shen Meiming, ”Quasi-asynchronous Migration: A Novel Migration Protocol for PVM Tasks.” ACM SIGOPS Operating Systems Review, 33(2): 5-15 (April 1999).

[71] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In Proceedings of the International Parallel Processing Symposium, pages 526--531, Honolulu, HI, April 1996. IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1264. http://citeseer.ist.psu.edu/stellner96cocheck.html

[72] Georg Stellner and Jim Pruyne. Resource Management and Checkpointing for PVM. Proc EuroPVM95, pp. 130-136, Hermes, Paris, 1995. http://citeseer.ist.psu.edu/stellner95resource.html

[73] Goux, J., Kulkarni, S., Yoder, M., and Linderoth, J. 2000. An Enabling Framework for Master-Worker Applications on the Computational Grid. In Proceedings of the Ninth IEEE international Symposium on High Performance Distributed Computing (Hpdc'00) (August 01 - 04, 2000). HPDC. IEEE Computer Society, Washington, DC, 43.

[74] Chen, Y., Plank, J. S., and Li, K. 1997. CLIP: a checkpointing tool for message-passing parallel programs. In Proceedings of the 1997 ACM/IEEE Conference on Supercomputing (Cdrom) (San Jose, CA, November 15 - 21, 1997). Supercomputing '97. ACM Press, New York, NY, 1-11. DOI= http://doi.acm.org/10.1145/509593.509626

[75] Osman, S., Subhraveti, D., Su, G., and Nieh, J. 2002. The design and implementation of Zap: a system for migrating computing environments. SIGOPS Oper. Syst. Rev. 36, SI (Dec. 2002), 361-376. DOI= http://doi.acm.org/10.1145/844128.844162

[76] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In LACSI Symposium, October 2003.

[77] J. Duell, P Hargrove, snd E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart, 2002.

[78] Zhang, Y., Wong, D., and Zheng, W. 2005. User-level checkpoint and recovery for LAM/MPI. SIGOPS Oper. Syst. Rev. 39, 3 (Jul. 2005), 72-81. DOI= http://doi.acm.org/10.1145/1075395.1075402

[79] Gropp, W., Lusk, E., Doss, N., Skjellum, A., "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard", Parallel Computing, North-Holland, vol. 22, pp. 789-828, 1996. http://citeseer.ist.psu.edu/gropp96highperformance.html

[80] G.F.Fagg and J.J.Dongarra, "FT-MPI:FaultTolerant MPI Supporting Dynamic Applications in a Dynamic World", EuroPVM/MPI User's Group Meeting 2000, Springer-Verlag, Berlin, Germany, 2000, pp.346-353. http://citeseer.ist.psu.edu/fagg00ftmpi.html

[81] Beck, Dongarra, Fagg, Geist, Gray, Kohl, Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. Scott, V. Sunderam, "HARNESS: a next generation distributed virtual machine", Journal of Future Generation Computer Systems, (15), Elsevier Science B.V., 1999.

[82] Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. Adnan Agbaria and Roy Friedman. In the 8th IEEE International Symposium on High Performance Distributed Computing, 1999.

[83] Graham E. Fagg, Keith Moore, Jack J. Dongarra, "Scalable networked information processing environment (SNIPE)", Journal of Future Generation Computer Systems, (15), pp. 571-582, Elsevier Science B.V., 1999.

[84] Zhang Youhui, Wang Dongsheng, Zheng Weimin, "Checkpointing and Migration of parallel processes based on Message Passing Interface", The 3rd Linux Clusters Institute (LCI) Conference, St. Petersburg, Florida, October 23-25, 2002.

[85] NIIF ClusterGrid Project, http://www.clustergrid.niif.hu

[86] G. Coulouris, J. Dollimore and T. Kindberg, Distributed Systems: Concepts and Design. Addison-Wesley, Pearson Education, 2001.