An uncoordinated asynchronous checkpointing model for hierarchical scientific workflows

Rafael Tolosana-Calasanz, José Ángel Bañares, Pedro Álvarez, Joaquín Ezpeleta, Omer Rana

Abstract

Scientific workflow systems often operate in unreliable environments, and have accordingly incorporated different fault tolerance techniques. One of them is the checkpointing technique combined with its corresponding rollback recovery process. Different checkpointing schemes have been developed and at various levels: task- (or activity-) level and workflow-level. At workflow-level, the usually adopted approach is to establish a checkpointing frequency in the system which determines the moment at which a global workflow checkpoint – a snapshot of the whole workflow enactment state at normal execution (without failures) – has to be accomplished. We describe an alternative workflow-level checkpointing scheme and its corresponding rollback recovery process for hierarchical scientific workflows in which every workflow node in the hierarchy accomplishes its own local checkpoint autonomously and in an uncoordinated way after its enactment. In contrast to other proposals, we utilise the Reference net formalism for expressing the scheme. Reference nets are a particular type of Petri nets which can more effectively provide the abstractions to support and to express hierarchical workflows and their dynamic adaptability.

1. Introduction

Since the computing potential of distributed systems is often hindered by their susceptibility to failures, many different techniques have been developed and integrated into them accordingly, in order to improve both their reliability and availability and to reduce re-computations. To achieve fault tolerance, a combination of two of these techniques, checkpointing and rollback recovery, can be exploited: checkpointing allows a system to periodically store its state during normal execution (this stored state is called checkpoint or snapshot), whereas rollback recovery, upon a failure, provides a way of restoring a system to a previously saved state. Checkpointing and rollback recovery have been studied to support various objectives (such as debugging, fault detection or improving start-up time and memory footprint) and in connection with many fields of research in both software and hardware. The benefit obtained from them is of particular relevance for the fault tolerance of long-running applications.

Similarly, scientific workflow systems have incorporated different checkpointing and rollback recovery schemes [20], though tailored to their particular features and at different granularity levels. At task-level (or activity-level) checkpointing involves saving intermediate states of running tasks, so that in case of failure a task can be re-started from a previously...
saved intermediate state. Several scientific workflow systems have incorporated checkpointing [4]. At workflow-level checkpointing involves capturing the state of the workflow as a whole, involving a data-oriented view. Indeed, scientific workflows generally involve enacting large and complex scientific activities by mapping tasks onto heterogeneous, stand-alone and distributed resources and typically they also have data flowing through the activities in the workflow, examples include Pegasus, Askalon, Triana and Kepler [4,5,24,6]. Depending on how the intermediate data from a workflow is stored, we can distinguish between light-weight workflow checkpointing or heavy-weight workflow checkpointing [5]. In a heavy-weight checkpointing scheme a copy of the intermediate data is stored into the checkpoint, whereas in light-weight workflow checkpointing a reference to the location where intermediate data can be retrieved is stored into the checkpoint. The latter requires the storage to be secure and non-volatile for allowing the system to accomplish the rollback process, whereas the former produces more overhead due to data transfer and storage.

On the other hand, depending on how the snapshots of a workflow are taken, we can find global-coordinated [5] or local-uncordinated [6] workflow checkpointing. In the global-coordinated approach, a checkpointing frequency determines when a snapshot of the whole workflow process is taken. In contrast, in a local approach, independent partial snapshots of the workflow tasks are taken in an uncoordinated way. One limitation with global workflow checkpointing is determining how to adapt the checkpoint frequency based on the current task(s) being enacted and the system's deployment environment. Taking more checkpoints than actually required leads to an increase in the checkpointing overhead, whereas taking fewer may lead to a loss of important computation, affecting the overall execution performance as well. Some work has already been undertaken to identify the optimum checkpointing frequency [7]. Additionally, in this type of approach, the workflow enactment is typically suspended in order to avoid potential race-conditions while taking the snapshot.

In this paper, we formally describe a local-uncordinated checkpointing strategy for hierarchical scientific workflows. This eliminates the problem of adjusting the checkpointing frequency, thereby reducing the associated overhead. In our checkpointing model, every workflow node in the hierarchy accomplishes its own local checkpoint autonomously and in an uncoordinated way after the enactment of the activity: a finished node sends a checkpointing event asynchronously while the enactment of the remaining workflow continues. Thus the checkpointing process is overlapped with the execution of the workflow. Nevertheless, the key challenge is to re-build a global workflow state from autonomously stored local checkpoints during the rollback recovery process. In order to tackle this problem, our model is described in terms of the Reference net formalism – a particular High-level Petri net. The Reference nets formalism plays a twofold role: first, it provides the abstractions to support and to express the behaviour of hierarchical workflows and their dynamic adaptability with clear and precise semantics. Unlike other similar approaches which achieve the required flexibility by means of the runtime system in order to include implicit fault management tasks [13], our approach expresses the exception handling mechanisms within the workflow model. Second, based upon a formal theory for determining global states of distributed systems [18], using ordering of events [15] and with the formal semantics of Reference nets, our model allows us to provide a proof that demonstrates that the rollback recovery can be consistently achieved.

The proposed model has two main parts: a net representing the workflow enactment engine, and a composite of nets forming a valid workflow. The workflow enactment engine is centralised and a workflow in the model has a hierarchical non-DAG structure. Each intermediate workflow node has a primary set of descendants (called a sub-workflow) which follow a modular approach: a workflow node gives the input data to its set of descendants and waits for the output data, having a low coupling between node and sub-workflow. Consequently, the primary sub-workflow can be replaced easily by alternative sub-workflows with equivalent functionality as long as the signature – input and output I/O interfaces – remain the same. This hierarchical approach is the one followed by workflows made available at the myexperiment.org repository. In our model, data is represented by typed tokens, allowing the model to abstract both light-weight and heavy-weight workflow checkpointing mechanisms.

The rest of the paper is organised as follows: in Section 2, a brief overview of Reference nets is given. In Section 3, we describe the characteristics of the proposed model. In Section 4, the model is formally presented. Section 5 provides simulation results and what these results demonstrate. Section 6 discusses and compares previous related work with our approach. Finally, conclusions are provided in the subsequent section.

2. Background: Reference nets

An ordinary Petri net can be defined informally as a bipartite directed graph which consists of places, transitions, arcs and tokens (see [17] for a formal definition and a general introduction to the formalism). There are many extensions to ordinary Petri nets – such as High-level Petri nets and timed Petri nets – which provide many additional features such as higher levels of abstraction. Ordinary Petri nets and their extensions have been widely used for the specification, analysis and implementation of business workflows [2]. In the scientific workflow community, High-level Petri nets have also been utilised and GWorkflowDL [29,19], Grid-Flow [9] or FlowManager [3] are representative examples of this. Unlike in ordinary Petri nets, which have just black tokens, in High-level Petri nets, tokens can model scientific data moving through a workflow.

In this work, we use a specific type of High-level Petri nets, called Reference nets [14,28], for modelling scientific workflows. Reference nets' tokens can be either Java objects – facilitating the modelling of scientific data flowing through the workflow – or sub-nets – allowing for the modelling of nested structures. Nevertheless, the main distinctive aspect of Reference nets is their dynamic nature: in Reference nets, a net itself can express the creation of new net instances explicitly,
2.1. Expressing scientific workflows – alternatives

Existing workflow description languages may be grouped into two classes [13]: script-like descriptions that specify a workflow by means of a textual programming language. These descriptions possess complex semantics and an extensive syntax. Graph-based description languages that specify the workflow with only a few basic graph elements. Examples of script workflow descriptions are GridAnt [16] and Karajan [23]. These languages contain specific workflow constructs, such as sequence or while do, in order to build up the workflow. Purely graph-based workflow descriptions generally utilise directed acyclic graphs (DAGs). Nonetheless, as Petri nets are specific graphs (bipartite graphs) with additional elements such as tokens, workflow descriptions based on Petri nets can also be included into the graph-based languages class.

Compared with script-based descriptions, graphs are easier to use and more intuitive for the unskilled user: communication between different services is represented as arcs going from one service to another. DAG-based languages offer only a limited expressiveness, so that it is often hard to describe complex workflows (e.g. loops cannot be expressed directly). However, Petri net based workflows are more general in scope, and capable of expressing more complex workflow structures (they are cyclic graphs). A commonly used script-based approach to describe workflows, mainly in the business workflow community, is the Business Process Execution Language (BPEL) and its recent version for Web Services that builds on IBM’s Web Services Flow Language (WSFL). In comparison with Petri nets, BPEL has two main disadvantages. First, BPEL possesses complex and rather informal semantics, which makes use of formal analysis methods on BPEL difficult, and also leads to difficulty for modelling workflows, especially for the unskilled end user. Second, it has a limited expressiveness, as it does not directly support some workflow patterns, such as arbitrary cycles. Additionally, BPEL does not support reflection properties as in Reference nets so that a workflow can manipulate its structure itself.

There is often a compromise between ease of use and expressiveness when choosing a workflow representation scheme. While some representations provide ease of use, by supporting a simple “drag-and-drop” capability for composing workflows, others require users to develop scripts (with annotations) that are subsequently processed by an interpreter. We recognise that Petri net-based representations as used in this work, require end users to have some understanding of Petri net semantics – and are therefore more complex to use than alternative graph-based representations. However, we believe that the additional analysis capability that is offered by such representations render the benefits of this approach greater than the costs. In previous work [27] we have demonstrated how workflows in another graph-based representation (Taverna) can be mapped to Reference nets.

3. A checkpointing and rollback recovery scheme for hierarchical scientific workflows

Our Reference nets-based model consists of two main parts: the workflow enactment engine and the workflow description. The workflow enactment engine is centralised, and workflows are described using a hierarchical non-DAG structure. Each intermediate workflow node has a primary set of descendants (called primary sub-workflow) and an optional list of alternative sub-workflows. In analogy with programming languages, this approach follows a modular approach: workflow nodes know nothing about their descendants, they only agree on the interface (input and output data and the signalled exceptions). Consequently, the primary sub-workflow can be replaced easily by alternative sub-workflows with equivalent functionality as long as the signature – input and output I/O interfaces – is preserved. Additionally, each workflow leaf node is a simple task.

It is assumed that faults can be detected at the level of the workflow hierarchy at which they occur, and lead to the signalling of an exception. An exception here must be understood as an unusual event – detectable either by hardware or software – and which requires special processing. Thus, consider a single fault (hardware/software) $f_i$, and $\{f\}$ a set of
faults, leading to a known event $e_i$. The event causes a single action $a_i$ or a set of actions (executed in some sequence) $\{a\}$ to be invoked. This can be expressed as: $(f_i||f) \rightarrow e_i \rightarrow (a_i||a)$.

In the model, unlike a global checkpoint scheme whereby a snapshot of the entire workflow is taken, each workflow node in the hierarchy stores its local snapshot independently in an uncoordinated way, and without considering the checkpoint activities of other nodes in the workflow. In general terms, the global state of a scientific workflow system consists of the execution state of its tasks (unexecuted, in execution or executed), the state of its control dependencies (loop conditions, branch conditions, etc.) and the state of its data dependencies (the produced intermediate data).

We do not checkpoint the control dependencies between nodes, because whenever an exception is signalled by a child node in a sub-workflow to its parent node, the parent node aborts the enactment of the failed sub-workflow. This sub-workflow is then replaced by an alternative instance whose enactment is started from the beginning. Additionally, in case none of the sub-workflows (neither the primary sub-workflow or the alternative ones) are valid, the current node signals the exception to its parent which will repeat the same process again or terminate.

On the other hand, the execution state of the workflow nodes is given by the Reference net marking, so that a firing history of transitions represents the execution state of workflow nodes. The most challenging problem is storing the intermediate data. Each workflow node in the hierarchy only stores its input data just before its enactment, so that this action guarantees that in case of failure, a node can re-start the enactment itself. This strategy guarantees that only the intermediate data that is being consumed within the workflow is actually going to be checkpointed [5].

Our rollback recovery is exception-driven: the propagation of exceptions up in the workflow hierarchy triggers the rollback recovery mechanisms as it was described above. In the worse case scenario, the exception moves up to the top hierarchy level and the workflow enactment engine is responsible for re-starting the enactment of the workflow from the beginning, trying an alternative workflow or aborting the enactment completely. On failure, an exception will cause a failed node to re-start its enactment – leading to “backward recovery” [8]; though there may be other parallel and neighbour workflow nodes which can continue their execution.

4. A checkpointing and rollback recovery model for hierarchical scientific workflows

The model can be expressed by three different nets: (i) the workflow enactment engine (the top hierarchy system net), (ii) the intermediate workflow node and (iii) the leaf workflow node.

**Definition 1.** An LRN, Leaf Reference net, is an instance of the net pattern in Fig. 1.

Fig. 1 shows a leaf node which specifies a simple task. The task receives the input data and all the specifications required for the remote resource in Transition $t_1$ (Variable input). In Transition $t_2$, a request to the workflow engine for enacting the simple task is accomplished. This is carried out by means of the Synchronous Channel :send. Notice that it is an uplink and the net’s parent will take the request which will be propagated up in the hierarchy. The semantic of Channel :send is that the request is performed in an asynchronous and non-blocking way and a unique identifier of the request (jobid) is returned as a result. The state of the net then moves forward to Transition $t_3$, where the result of the request will be obtained by Channel :receive. This is also an uplink, but the semantics are different: synchronous and blocking, until the simple task finishes its execution remotely. The returned result can be due to normal execution – enabling Transition $t_4$ – or to abnormal execution – enabling Transition $t_5$ – which will lead to an exception being sent to the parent node in the hierarchy.

**Definition 2.** An Intermediate Node Reference Net (INRN), is an instance of the net pattern in Fig. 2, where SubWfModel and the elements of the set $\text{setOfAlternativeWF} = \{wf_1, \ldots, wf_n\}$, $n \geq 0$ are either a WRN (Definition 3) or an LRN.

When an INRN node $I$ is defined, the designer considers all the involved elements in $I_{wf}$, $I_{AWF}$ to be functionally equivalent, so that they can be interchangeable (having compatible I/O signature) and can produce similar outputs.

As it can be seen in Fig. 2, an INRN receives its input data in Transition $t_1$, Variable input, which is a tuple formed by a pair ‘argument names’ and ‘argument values’. After firing Transition $t_1$, a new instance of the primary sub-workflow
is created (creation inscription \( w:\text{new SubWfModel} \) of Transition \( t_1 \)) and then its enactment is started (Synchronous Channel \( w:\text{begin(args)} \) of Transition \( t_1 \)). As a consequence of firing Transition \( t_1 \), an instance of the primary sub-workflow (\( \text{primary} \)) will be stored in Place \( E \), and the names of the arguments required will be stored in Place \( A \). Therefore, the token in Place \( E \) is a WRN or an LRN. For a given INRN instance (\( I \)), \( \text{SubWfModel} \) and \( \text{setOfAlternativeWF} \) are called the primary and alternative set of sub-workflows, respectively, denoted as \( I_{wf} \), \( I_{AWF} \). It should be noted that the input data is also checkpointed by Synchronous Channel \( w:\text{send(id, \{"checkpoint", [argsNames, argsValues]\})} \) in Transition \( t_1 \). The sub-workflow can terminate normally by firing Transition \( t_2 \) and returning the output (Variable \( \text{result} \)).

The sub-workflow can also terminate abnormally by signalling an exception that will be caught by means of the Synchronous Channel \( w:\text{exception} \) of Transition \( e_1 \). In Transitions \( e_2 \) an alternative candidate is requested and in Transition \( e_3 \) an alternative candidate instance (from \( I_{AWF} \)) might be received (Variable \( nw \)), in that case, Transition \( e_5 \) will be fired and in Transition \( e_6 \) the enactment of this new sub-workflow instance will be started. In contrast, when Variable \( nw \) is set to null, the exception will be propagated up in the hierarchy by Synchronous Channel \( w:\text{exception(ex)} \) in Transition \( e_4 \).

Transitions \( r_1 \) and \( r_2 \) are responsible for requesting and obtaining, respectively, previously checkpointed input data, required for enacting the alternative sub-workflows in Place \( E \). Each intermediate data in the workflow is identified uniquely by its name. Thus, in Place \( A \), the names of the arguments are stored, so that in case of rollback the input can be retrieved. The purpose of Transitions \( ch_1 \) and \( ch_2 \) will be explained later on. Data movement is represented by typed tokens in the model. The Reference net formalism can use Java primitive data types and Java Classes for this purpose.

**Definition 3.** A *Workflow Reference net* (WRN), is defined as follows:

- An INRN is a WRN.
- The Reference net obtained from the composition of the WRNs (\( \text{wf}_1, \text{wf}_2, \ldots, \text{wf}_n \)), \( n \geq 1 \), according to the patterns in Fig. 3, is a WRN.\(^1\)

It is important to note that parallel tasks in the workflow are completely independent, even in case they manipulate the same input data. In such a case, a copy of the input data is generated for each parallel task.

**Definition 4.** A *Hierarchical Workflow Checkpointing Reference net* (HWCRN), is defined as the Reference net \( \text{HWCRN} = (SN, WF) \) where:

\[^1\] It should be noticed that the proposed set of patterns is a subset of the workflow patterns defined in [22].
1. SN is the system net which is an instance of Fig. 4, where the value of the variable \( w \) in the creation inscription of Transition \( t_1 \) belongs to the set WF.
2. WF is a non-empty set of WRN.

The net in Fig. 4 abstracts the enactment engine of the workflow system. Transition \( t_1 \) receives a tuple \([w, \text{input}]\), a new instance of a workflow \( w \) and its corresponding input data \( \text{input} \). Both elements are provided by the workflow system environment which can be seen as a set of components able to perform specific activities namely, creating new workflow instances and providing workflow inputs, collecting workflow outputs, destroying workflow instances, storing data checkpoints, supporting the retrieval of checkpointed data, communicating with external resources and selecting alternative sub-workflows/workflows. When Transition \( t_1 \) fires, \( w \) moves to Place \( E \), indicating that its enactment is ready to start (the input is sent to \( w \) by Channel \( w: \text{begin} (\text{input}) \)). Transition \( t_2 \) is responsible for obtaining the workflow enactment result (Variable \( \text{wfOutput} \)). Transition \( e_1 \) is responsible for dealing with a failed workflow whose execution cannot progress any more. For these cases, and similar to the intermediate workflow nodes, Transitions \( e_2 \sim e_6 \) respond to workflow failure by looking for an alternative workflow \( nw \) which can provide the required output data. In case this is not possible, the enactment of a workflow is aborted (Transition \( e_4 \)).

Intermediate nodes in the hierarchy need to perform checkpoints, select alternative child sub-workflows and retrieve previously checkpointed data; whereas simple tasks need to only execute tasks remotely. All these actions are modelled by the corresponding interactions between each intermediate node or leaf node and the workflow engine (the system net).
The system net then delegates the different actions to the environment. An interaction is accomplished by Synchronous Channels, send and receive, which appear in every node and which are used to propagate the message (either send or receive) from any part in the workflow hierarchy to the system net and vice versa. For instance, let us consider a leaf node, in Transition $t_2$ of Fig. 1, by means of Synchronous Channel send, the message arrives at its parent intermediate workflow node at the other part of the channel which corresponds to $w$:send(id,msg) of Transition ch1 of Fig. 2. Additionally, Transition ch1 in the intermediate node also has a channel which propagates the message to the parent node of the intermediate workflow node. In the end, the message arrives at Transition ch1 (in Fig. 4) of the system net (or workflow engine). The system net redirects this message to the Synchronous Channel this:send(id,message) of Transition ch1 for processing. The mechanism of Transition ch2 of the system net and of the intermediate workflow nodes is analogous.

An HWCRN is a special case of Reference net. The current system state is given by the Reference net marking, $m$, and the sequence, $\sigma$, of transitions fired from the initial state, $m_0(\sigma)m$ is the history. As usual in Petri nets, a marking is a mapping associating items to each place of the Petri net. Depending on the subclass of nets, these items can be as simple as black tokens (ordinary Petri nets), typed data (the case of Coloured Petri nets) or even references to marked Petri nets, in a recursive way, as is the case of Reference nets. In this last case, for a place $p$, $m(p)$ can contain, for instance, a pair $(N, m^N)$, where $N$ is a (Reference) Petri net and $m^N$ is its current marking.

Definition 5. Let $H = (SN, WF)$ be an HWCRN, let $m_0(\sigma)m$ be its current state, let $w \in WF$ be a WRN being enacted in $H ((w, m^w) \in m(\Sigma^{SN}))$, and let $I$ be an INRN in $w$. Marking $m^w$ is said to be a local rollback state for $I$ if, and only if, the last transition of $I$ appearing in $\sigma$ is either $t_1^I$ or $e_6^I$.

For a given INRN, local consistent checkpoints are those markings reached just after Transition $t_1$ or $e_6$ fires, which corresponds to states where the corresponding sub-workflow (either primary or alternative) has been created, but whose execution has not started yet.

On the other hand, the checkpoint activity in the model is carried out by each INRN node, where the input data is stored just before the beginning of the execution of the node. The state of execution of the workflow corresponds to the marking of the nets, and it is maintained by the nets. For this reason, in order to recover the system from a failure of the workflow interpreter itself, the Petri net interpreter should store the firings of its nets, albeit we are not considering this in the model.

The descendants of a given INRN node can be seen as its refinements. It is possible that any intermediate data is checkpointed more than once i.e. by a node and later by its descendants. In order to avoid unnecessary storage overheads, the checkpointing database should store intermediate data only once (notice that this can be easily done as intermediate data is assigned a unique name by the workflow designer). See Section 5 for an analysis of the performance overhead of the model. Considering the special hierarchical structure of an HWCRN, a firing sequence can be mapped into a tree structure, so that every time a $:new SubWFModel$ is executed in an INRN node, a new child is created. The subtree is said to be open until the associated $:end()$ transition fires (then the subtree is said to be closed). Formally, a data checkpoint for a given WRN $w$ can be defined as follows.

Definition 6. Let $H = (SN, WF)$ be an HWCRN, let $w$ be a WRN belonging to $WF$ and let $m$, $m_0(\sigma)m$, be a reachable marking for $w$.

1. For a given INRN $I$, involved in $\sigma$, $I$ is said to be open for $\sigma$ if $t_1^I$ appears in $\sigma$, but $t_2^I$ does not appear in $\sigma$. $I$ is said to be closed for $\sigma$ if both $t_1^I$ and $t_2^I$ appear in $\sigma$.
2. A data checkpoint for $\sigma$ is the set of the intermediate data produced by all the closed INRNs appearing in $\sigma$ that have been checkpointed by any involved INRN $I$ at its Transition $t_1^I$.

4.1. Checkpointing in the model

Data in the workflow is represented by typed tokens moving from one INRN node to another. This can be the data value, or references to the location where the data can be found. In this last case, there are two checkpointing techniques, namely heavy-weight workflow checkpointing scheme and light-weight workflow checkpointing scheme. In both cases, the checkpointing mechanism in the model is the same. We also assume that there is no data streaming and that every data input or output can be uniquely identified and retrieved by a name.

In order to illustrate how our proposed checkpointing activity is performed in our model, let us consider the WRN schema of Fig. 5(a). The top hierarchy level (1st level) enacts two parallel INRN nodes $wf1$ and $wf2$. In the figure, $wf2$ has a set of descendants (at a 2nd level) which are the sequential composition of the INRN node $wf21$ and the INRN node $wf22$. The three bars between $wf2$ and its descendant nodes $wf21$ and $wf22$ represent three Synchronous Channels exception, ch1 and ch2, whereas the two remaining dotted bars between $wf2$ and its descendants represent two synchronous channels for the transfer of the input and output data.

From the sequence diagram in Fig. 5, it can be seen that Node $wf2$ sent the input data to its descendants (by means of Synchronous Channel begin). Then, Node $wf21$ received the input and finished its execution providing an output. At this point, Node $wf21$ sends an event (by using the Synchronous Channel send, see Transition $t_2$ of Fig. 2) to its parent.
(Node $w_2$) indicating that the output data obtained have to be checkpointed. As a consequence of this, Transition $t_2$ of $w_{21}$, Transition $ch_1$ of Node $w_2$ and Transition $ch_1$ of the System Net were synchronised and fired. The request event from Node $w_{21}$ to the workflow engine was processed in a heavy-weight workflow checkpointing scheme, the corresponding component in the System Net’s environment would store the data in a secure repository; whereas in a light-weight checkpointing scheme, a reference to the location of data is stored. Analogously, Node $w_{22}$ finished normally and also checkpointed its output data. It should also be noticed that the output data of Node $w_{22}$ and Node $w_2$ are the same, we assume that the component processing the checkpointing events can reuse previously stored data. An alternative to checkpointing output data is checkpointing input data. The difference between the two approaches is discussed in [5]: checkpointing output data stores all the produced intermediate data, some of which may not be needed by the subsequent processes, whilst checkpointing input data stores only the produced intermediate data that are required by the subsequent processes. Nevertheless, sometimes all the intermediate data are stored for tracking the provenance of data [6]. Moving Synchronous Channel $:send$ from Transition $t_2$ of Fig. 2 to Transition $t_1$ of the same figure, allows us to modify the behaviour from checkpointing output data to checkpointing input data.

4.2. Rollback recovery in the model

When a descendant signals an exception to its parent the parent applies a rollback operation to all of its descendants (sub-workflow). The example of Fig. 6 illustrates this rollback recovery process. In Fig. 6(a), a three-level branch of a WRN is shown. The execution state of the branch can be represented by the hierarchical structure of the schema of Fig. 6(b), which illustrates that in Place $E$ of Node $I_{12}$ there is a sub-workflow $w_{12}$ being enacted which has two children: $I_{121}$ that finished successfully and $I_{122}$ in execution. Node $I_{122}$ has sub-workflow $w_{122}$ in its Place $E$. As an evolution of the enactment, in Fig. 6(c), a failure in $w_{122}$ meant that Node $I_{122}$ had to replace it by $w'_{122}$. Nevertheless, $w'_{122}$ also failed and there were no more alternatives, leading to the state of Fig. 6(d) in which $I_{12}$ had to replace the set of descendants $w_{12}$ by $w'_{12}$.

4.3. Uncoordinated rollback

Uncoordinated checkpointing schemes generally apply to a fixed number of distributed processes which interact with one another by exchanging messages to achieve a particular outcome. Each process periodically saves (or checkpoints) its state independently in a stable storage, though the local checkpoints of a process and the messages exchanged with other processes can be interleaved. For this reason, when a process fails and rollbacks to a previous local checkpoint, the other processes also have to rollback accordingly.

In order to accomplish this coordinated rollback activity, such systems must examine many combinations of local checkpoints to establish and restore a consistent global checkpoint in rollback: a correct global state that can be reached from the system’s initial state, and that allows the execution to reach the desired outcome. Determining when individual local
checkpoints can be combined with others to form a consistent global checkpoint is an important problem which has been studied in the past. Netzer and Xu introduced the zigzag path relation among checkpoints and the necessary and sufficient conditions for global snapshots [18]: zigzag path relations are a generalisation of Lamport’s happened-before relation [15]. A global snapshot is said to be consistent if no local checkpoint happens before another; that is, if a message (or sequence of messages) sent after one checkpoint is (are) received before another. Therefore, a set of local checkpoints must be unordered for having a consistent global snapshot.

Nevertheless, a process $p$ could rollback independently without requiring other processes in the system to act accordingly, providing that $p$ is independent, i.e. does not have any interaction with other processes. The notion of independence of a process in an interval of time can be defined in the following way.

**Definition 7.** Given a distributed system $S$ which consists of a set of processes that interact with each other, a process $P$ of $S$ is said to be independent in a period of time $[t_0, t_k]$ with $t_k > t_0$, if $P$ has a local checkpoint, $C_P$, at $t_0$ such that there is no event of interaction between $P$ and the rest of processes of $S$ from $t_0$ to $t_k$.

Thus, upon a failure in the interval $[t_0, t_k]$, process $P$ can rollback independently without requiring any rollback of the other processes. Unlike distributed systems where processes can inter-communicate and store local checkpoints without being constrained to a specific pattern, in our model, workflow tasks (processes by analogy) must interact, and their checkpoints need to be made in accordance with the control flow imposed by the Petri net model.

**Theorem 1.** Let $H = (SN, WF)$ be an HWCRN, let $m, m_0(\sigma)m$, be the marking corresponding to the current system state. Let $w \in WF$ be a WRN enacted at $m, m(\sigma^{SN}) = (w, m^k)$. Let $I$ be an INRN node of $w$ open at $\sigma$ such that $t_1^I$ fires at time $t_0$. Let $s \in I_{of} \cup I_{AWF}$ be a descendant in execution in Place $E^I, m(E^I) = (s, m^k)$. If $s$ signals an exception to $I$ at time $t_k > t_0$, then $I$ is independent in the period $[t_0, t_k]$.

**Proof.** According to Definition 2, $s$ is either an LRN or a WRN. In case $s$ is an LRN, by Definition 1, an LRN only interacts with the resource where the task is mapped and with its parent. Therefore, in this case, INRN $I$ is independent in the period of time $[t_0, t_k]$. In case $s$ is a WRN, by Definition 3, a WRN is an INRN node or a composition of INRN nodes. By Definition 2, an INRN node receives its input data, starts the execution of its descendants (INRN nodes) and either finishes successfully generating output data, or finishes abnormally by signalling an exception. In consequence, the INRN nodes in $s$ did not interact with other INRN nodes outside the hierarchy of $s$ in the interval of time $[t_0, t_k]$. Therefore, INRN $I$ is independent in the period of time $[t_0, t_k]$. □

As a direct consequence of Theorem 1, an INRN node in our proposal can rollback to time $t_0$ which corresponds to its local rollback state as defined in Definition 5, without requiring the rest of the nodes to rollback accordingly.
5. Analytical analysis and experiments

In this section, we analyse the performance of our local, uncoordinated checkpointing model and compare it with a global checkpointing strategy.

5.1. Overhead analysis of our model

The overhead of a checkpoint at an INRN node in our model can be characterised as follows:

- Checkpointing overhead $C_o = T_{checkpointEvent}$, where $T_{checkpointEvent}$ corresponds to the message sent to the checkpointing database in Transition $t_1$ of an INRN node (see Fig. 2). It should be noticed that once the message is sent, the enactment continues while the data is stored: in case of light-weight checkpointing, a reference is stored, but in case of heavy-weight checkpointing, a complete dataset may have to be transferred from a distributed resource to the checkpointing database.
- The overall checkpointing overhead $OC_o = T_{checkpointEvent} * K$, where $K$ is the number of INRN nodes in the hierarchy. In this approach, incremental data checkpointing is followed, i.e. intermediate data are only stored once.
- Recovery overhead $R_o = T_{rollback} + T_{restoreCheckpoint} + T_{resume}$, where $T_{rollback}$ is the time required to propagate the exception up the workflow hierarchy and replace the component. $T_{restoreCheckpoint}$ is the time to retrieve and propagate the intermediate data and re-establish control, and $T_{resume}$ is the time to resume the enactment of the failed sub-workflow. In the worst case, however, an additional time penalty arises due to a checkpoint that has not finished storing it's state.
- Recovery overhead with penalty $R'_o = T_{rollback} + T_{resume} + T_{checkpoint}$, where $T_{checkpoint}$ is the time required to complete a checkpoint that was started previously and which has not finished yet. This time is primarily due to the overhead of transferring and storing the data into the checkpoint. It should be noticed that this is more likely to appear with the heavy-weight checkpointing strategy rather than with the light-weight one. The rest of variables are as in $R_o$ above.

The overall rollback overhead will also depend on the number of failures that arise during execution.

5.2. Overhead analysis of a global workflow checkpointing

An overhead analysis is provided based on Askalon's global checkpointing model [5] for comparison. Typically, a global workflow checkpointing strategy stops the enactment of a workflow, takes a global snapshot, waits for the checkpointing database to store the snapshot and resumes the enactment. In case of failure, the last global snapshot is retrieved, the data are moved to the corresponding place if required (only for the heavy-weight checkpointing scheme) and the enactment of the workflow is resumed. The overhead of these activities can be characterised as follows:

- Checkpointing overhead $C_o = T_{gsfstop} + T_{data} + T_{resume}$, where $T_{gsfstop}$ is the time required to suspend all workflow enactment, $T_{data}$ is the transfer and the storage of data synchronously, and $T_{resume}$ is the time to resume workflow enactment.
- The overall checkpointing overhead $OC_o = C_o + C_1 + \ldots + C_k$, with $k > 0$, depends on the checkpoint frequency. It should be noticed that an incremental data checkpointing is followed, that is, intermediate data are only stored once.
- Recovery overhead $R_o = T_{restoreCheckpoint} + T_{resume}$, where $T_{restoreCheckpoint}$ is the time to retrieve and propagate intermediate data and re-establish control. $T_{resume}$ is the time to resume workflow enactment.

The overall rollback overhead will also depend on the number of failures that arise during execution.

5.3. Comparison: simulation experiments

We conducted simulation experiments for the global and local strategies described on a hierarchical workflow comprising: (i) 5 INRN nodes at the top level, (ii) each INRN at the top level has 4 INRN nodes (at the 2nd level) and (iii) each INRN node at the 2nd level has an LRN (simple task). Overall, the workflow has 20 tasks, with each task producing an output dataset of 100 MB. Each simple task is mapped to a distributed resource, and these resources are connected to each other by a network with a 100 MB/s latency (on average), with a negative exponential distribution. In the workflow engine, sending a message to the checkpointing database is modelled by a negative exponential distribution with an average value of 40 ms. Stopping and starting a workflow task in a remote resource is modelled by a negative exponential distribution with 5000 ms on average.

In Fig. 7, the overall overhead of global and local checkpointing strategies are compared. On the left, the experiments were accomplished with light-weight checkpointing scheme and in the case of the global strategy with 5 different scenarios of 1, 5, 10, 15 and 20 checkpoints, respectively. The 5 different scenarios are displayed with the overhead time calculated in seconds. Similarly, on the right, the experiments were accomplished with heavy-weight checkpointing scheme and with 5 scenarios of 1, 5, 10, 15 and 20 checkpoints for the global strategy.
Fig. 7. Comparison between the overall overhead of local and global checkpointing strategies. Light-weight checkpointing on the left and heavy-weight checkpointing on the right.

Fig. 8. Comparison between the overhead of local rollback without penalisation and local rollback with penalisation (left). Comparison between global rollback with light-weight checkpointing and global rollback with heavy-weight checkpointing (right).

The overhead of the local checkpointing strategy is the same for light-weight and heavy-weight schemes because the checkpointing activity consists in sending a message to the checkpointing database. In contrast, in the global strategy, as the checkpoint overhead depends on the size of the data stored in a checkpoint, the overhead in a heavy-weight scheme is higher. Additionally, the overhead increases with the number of checkpoints accomplished. The overhead of the local checkpoint is much lower compared to the global checkpoint overhead, but this is because the checkpoints in the local approach are accomplished asynchronously. Thus, there could be a penalty for retrieving the data, because a checkpoint transfer and storage has not yet completed.

On the other hand, in Fig. 8, on the left, a comparison between the overhead of local rollback without penalty and local rollback with penalty (due to unfinished previous checkpoint) is undertaken, considering heavy-weight checkpointing in both cases. Three scenarios are presented: a rollback involving 1 level, 2 levels (the exception was propagated two levels up in the workflow hierarchy) and 3 levels (the exception arrives at the top workflow hierarchy and the whole workflow is replaced). The overhead increases with the number of levels involved in the rollback. This is because the more levels through which the exception is propagated in the hierarchy, the bigger the workflow involved in the rollback activity. In Fig. 8, on the right, a comparison between global rollback strategies with light-weight checkpointing and heavy-weight checkpointing is presented. Three scenarios are displayed: a rollback accomplished due to an initial failure in workflow enactment, a failure during workflow enactment and a failure close to the time when the workflow is about to complete. The overhead is higher in the middle because there are more tasks in execution. Besides, the overhead is higher for the heavy-weight strategy because of the data transfers involved.

Finally, in Fig. 9, a comparison between local and global overall rollback overheads is presented, assuming 20 failures occurring at different moments: (i) most of them at the beginning of the workflow enactment, (ii) most of them in the middle and (iii) most of them at the end. For the local rollback, a 1-level rollback was considered. The overhead for the local strategy is slower because only a sub-set of the tasks of the workflow are affected, whereas in the global strategy the whole workflow is affected. The time at which failures occur affects the global strategy more, as there are less tasks to stop/start and data to move at the beginning of the workflow than at the final stages. Additionally, in the best case, the rollback overhead for the local strategy could be overlapped with the enactment of parallel branches in the workflow.
6. Related work

The problem of rollback-recovery in message-passing systems has undergone substantial study. In [8], an extensive review of rollback-recovery mechanisms for message-passing systems can be found. In a message-passing system, if each participating process undertakes checkpointing independently, then the system may be forced to restart from the initial state (called the domino effect), losing all the work performed before a failure. In analogy to those systems, our local, uncoordinated checkpointing strategy may have several parallel activities which, instead of passing messages, pass intermediate data. However, the domino effect can be minimised because i) our checkpointing model is centralised and ii) the rollback recovery is guided by our workflow patterns: as long as our intermediate nodes are able to find an alternative sub-workflow to resume the enactment.

A review of existing Grid workflow systems and their features, including fault tolerance is provided in [30]. Another more recent work [20], reviews the state-of-the-art in fault tolerance mechanisms in Grid workflow systems, and highlights that the surveyed workflow systems can recover from far fewer faults than they can detect, especially at the middleware and workflow-levels. Hwang and Kesselman [10] propose a Grid workflow framework with different fault-tolerance techniques for flexible failure handling. One of them is the checkpointing technique, although it is only considered at task-level, i.e. when a task fails, it is allowed to be restarted from the recently checkpointed state rather than from the beginning.

The workflow enactment engine in the Askalon system [5] has developed global checkpointing with both light- and heavy-weight strategies. In both cases, a checkpointing frequency is established in the system so that when the checkpoint is required, the workflow enactment is stopped and a global workflow state is stored. A limitation of this approach is the difficulty of adapting the checkpointing frequency due to changes in tasks or the operating environment. On the other hand, empirical evidence is given in [5] that shows that the most important source of checkpointing and rollback overhead is due to data transfer and storage. Data locality may be used to minimise such overhead.

The Kepler workflow system [6] also implements exception handling in hierarchical scientific workflows. As described in [6], for each node in the hierarchy, its input data is checkpointed in advance so that in case of exception the failed descendant sub-workflow can be replaced by an alternative one. Nevertheless, few details are given in [6] on the checkpointing strategy adopted, though there seem to be many similarities with our proposal. In contrast, we describe our checkpointing and rollback recovery model in a systematic way, utilising the formalism of Reference nets to specify all the activities of checkpointing and rollback recovery. Other popular scientific workflow systems such as Triana [24], GWES [12] and Pegasus [4] currently support light-weight workflow-level checkpointing (Pegasus also supports checkpointing at task-level).

Reference nets have been used for describing workflow patterns [1] and for implementing the scientific workflow system DVega [25]. In [26], exception handling in hierarchical scientific workflows is expressed in terms of workflow patterns. This paper represents an evolution of [26], as the focus there was on exception handling.

7. Conclusions

A model is described which represents a local, autonomous and uncoordinated checkpointing strategy and its rollback recovery scheme for hierarchical scientific workflows. Other proposals require a workflow-wide checkpoint to be carried out, requiring a checkpoint frequency to be determined. The main problem that they have to address is how to adjust this frequency to limit impact on the overall system performance. In contrast, in our approach, every workflow node in the hierarchy accomplishes its own local checkpoint autonomously and in an uncoordinated way after its enactment. The main difficulty then is to determine how to re-build a globally consistent state from local, autonomous checkpoints. For this reason, we describe the model in terms of the Reference net formalism – a particular High-level Petri net that provides the abstractions to support and to express hierarchical workflows and their dynamic adaptability. Using such a model, we can utilise the constraints of Reference nets to ensure consistency of stored data associated with the workflow, from locally
generated checkpointing of nodes. A proof is presented to demonstrate that local checkpointing undertaken as proposed above does lead to globally consistent state. We also compare, through simulation, local and global checkpointing strategies, identifying the overhead incurred in each case based on the number of checkpoints and the time at which a checkpoint needs to be made (at the state, middle or end of workflow enactment).

The use of Reference nets and their associated inscriptions may be viewed as a workflow language. Using such a language, it is possible to express exception handling and checkpointing requirements formally, providing an engineer/technician greater control over the system. This leads to a decoupling between the behaviour of the system and the overall system architecture. We therefore also see a need for two workflow language levels as mentioned in [11] – a high level workflow language for the scientist and its low-level version for the technician/expert computer engineer.

References