Nets-Within-Nets Paradigm and Grid Computing

Marco Mascheroni, Fabio Farina

Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano Bicocca
Viale Sarca, 336, I-20126 Milano (Italy)**

Abstract. Grid is one of the most effective new paradigms in large scale distributed computing. Only recently Petri nets have been adopted as a formal modeling framework for describing the specific aspects of the Grid. In this paper we describe a Grid tool for High Energy Physics data analysis, and we show how modeling its architecture with nets-within-nets has led us to identify and solve a number of defects affecting the current implementation.

1 Introduction

In the last decade the Grid computing [10, 9] approach to parallel and distributed computing has defined a new path to enable high performance and throughput applications. Grid infrastructures expose computational and storage resources provided by different computing centers as uniform families of services that can be coordinated to create large scale e-Science workflows.

Grand-challenge experiments, like those related to High Energy Physics, life-science, and environmental science adopted the Grid as the tool for implementing their software. In this paper we will consider a Grid distributed data analysis tool developed to serve the community of the Compact Muon Solenoid (CMS) [19] experiment at the CERN Large Hadron Collider (LHC) [20]. A specific software tool has been developed to analyze physics data over the Grid, so that the users are protected from the architectural complexities of the distributed infrastructure itself. This application, called CMS Remote Analysis Builder (CRAB) [7] is released as open source software and has been adopted by the physics community since 2005. Even though the code quality is being continuously improved thanks to code analyzers (e.g., lint), the overall architecture has never been validated with formal tools like Petri nets.

The aim of this work is to validate some relevant parts of the CRAB tool using nets-within-nets [23]. In this paradigm the tokens of a Petri net can be Petri nets themselves. As we will see, the hierarchical structure of the system components is particularly suited for investigation with this formal framework. The Renew tool [17] has been chosen as modeling platform, as it is the only nets-within-nets tool that is mature enough to describe a real system like CRAB.

** Partially supported by MIUR (Italian Ministry of Education, University and Scientific Research)

In particular, the features of Renew used to model the system are such that the obtained model is very similar to a hypernet [2].This is a class of high level Petri nets which implements the nets-within-nets paradigm using a dynamic hierarchy, and a bounded state space [3]. As detailed in Section 4, this approach allowed us to isolate some problems in the CRAB implementation. Our approach do not cover analysis yet: modeling and step-by-step simulation are the two means used to unveil these problems.

In the literature high level Petri nets have been applied to different contexts related to Grid computing technologies. Most of the works in this field focus on the usage of Petri nets as a tool for workflows specification and execution [1, 13, 11]. A different application of Petri nets to Grid is reported in [5]. Here the resources exposed by the distributed computing infrastructure are modeled directly with the aim of validating both properties like the soundness and the fairness of their sharing for a process mining workflow. As far as we know, high level Petri nets, and in particular hierarchical nets, have been applied neither to the Grid infrastructure, nor to the study of a classical Grid application pattern like the distributed data analysis.

The remainder of the paper is organized as follows: Section 2 introduces the basic notion of nets-within-nets we refer to, and the Renew tool. Section 3 describes the Grid architecture we are considering, while in Section 4 the modeling of the system and the bugs found thanks to the formal approach are presented. A discussion about the modeling choices used in our approach is made in Section 5. Finally, some conclusions are reported in Section 6.

2 The Nets-Within-Nets Paradigm and Renew

According to the nets-within-nets paradigm, the tokens of a Petri net can be structured as Petri nets themselves. This idea is due to Valk (see [21]), who defined and studied the class of *Elementary Object Nets (EOS)* in [22]. Later on, properties of EOS were studied in [15], and other classes of high level Petri nets which uses the nets-within-nets paradigm were defined, like for example [12, 2, 14, 24, 18].

In all these models a system is usually modeled as a collection of nets. One net is designated as the *system net*, the top level of the net hierarchy. All other nets are assigned to an *initial place*, a place in which they reside initially. This distribution of nets induces a hierarchy. The system evolves by moving tokens from place to place through the firing of autonomous transitions, or by synchronizing transitions between nets at different levels. The hierarchical structure of the model is usually static, but in some models there can be interactions between nets at different levels in the hierarchy which can dynamically change the hierarchy itself. For example, in hypernets a net N can be moved from a place belonging to a net A, to a place belonging to a distinct net B. The interaction between nets A and B is only possible if they are close in the hierarchy.

The development of the Renew software tool [17], a Java-based high-level Petri net simulator that provides a flexible modelling approach based on Reference nets [16], allows the use of this paradigm to model real systems. RENEW is not only a nets-within-nets editor and simulator: it allows the use of high level net concepts like arc inscriptions, transition guards, and coloured tokens. However, we only use a subset of the features of RENEW. In particular, we choose to model the system with a hypernet-like model [2] (we will discuss in section 5 why the system is not a proper hypernet). The system is modeled as a collection of *net instances*. Tokens are *references* to net instances. Therefore it is possible that a net has more than one reference (token) in the system which refer to it. Arc inscriptions contain single variables. When a transition is fired tokens are bound to these variables. Transition inscriptions may contain channel names, used by two or more nets when they need to synchronize. An *uplink* is used when a net wants to synchronize with the net above it in the hierarchy, a *downlink* is used when a net wants to synchronize with one of the reference tokens it contains.

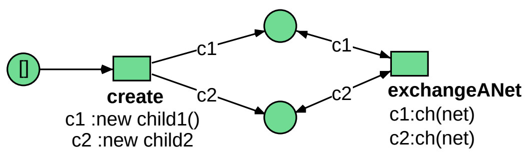
From a syntactical point of view the RENEW constructs we used in our model are the following:

– A net instance is created by a transition inscription of the form *var : new netname*, which means that the variable *var* will be assigned a new net instance of type *netname*.
– An *uplink* is specified as a transition inscription *:channelname (expr)*. It provides a name for the channel and a variable which is used for *vertical* communication between nets.
– A *downlink* has the form *netexpr :channelname (expr)* where *netexpr* is an expression that must evaluate to a net reference.

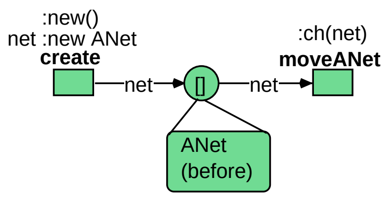
To fire a transition that has a downlink, there must be an input arc labelled with a proper variable name (*netexpr* for the previous downlink example), and this variable must evaluate to a net instance. The referenced net instance must provide an uplink with the same name,and it must be possible to bind the variables suitably so that the channel expressions evaluate to the same values on both sides. The parameter is bound to a variable present in one of the input arcs of the up(down)-link, and then it is bound to the parameter in the corresponding down(up)-link. Then the transitions can fire simultaneously.

The exchange of (structured) tokens between nets, typical of hypernets, is possible by means of parameters. Figure 1 shows an example. The only transition enabled at the beginning is *create* (Figure 1(a)), which creates an empty *child1* net, and a *child2* net (Figure 1(b), and Figure 1(c) respectively). The difference between using the parenthesis or not using the parenthesis in creating a new net is that, if you use them, then the transition that is being fired must synchronize on the channel *new()* in the child net. Therefore, transition *create* in the system net synchronizes with transition *create* in the child1 net, which creates the *ANet* net. Afterwards, transitions *exchangeNet, moveANet, receiveANet* can fire, moving *ANet* to *child2*.

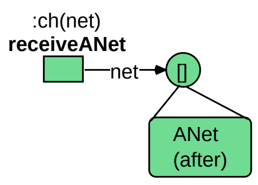
Let us notice that in our model the exchange of tokens between the two children nets, *child1* and *child2*, is made under the supervision of the system net. This means that the *system net* in some way observes the token exchange between its children.



(a) The system net



(b) The child1 net which creates a net of type *ANet* and sends it upward



(c) The child2 net which receives a net from above

Fig. 1. A simple example

3 The Application Context: Grid distributed analysis

The CMS experiment at CERN produces about 2 Petabytes of data to be stored every year, and a comparable amount of simulated data is generated. Data needs to be accessed for the whole lifetime of the experiment, for reprocessing and analysis, from a worldwide community: about 3000 collaborators from 183 institutes spread over 38 countries all around the world.

The CMS computing model uses the infrastructure provided by the Worldwide LHC Computing Grid (WLCG) Project [6] through the supporting projects EGEE, OSG and Nordugrid. Grid analysis in CMS is data driven. A prerequisite is that data is already distributed to some remote computing centers, and correspondingly published in the CMS data catalogue, so that users can discover available datasets. Parallelization is provided by splitting the analysis of large data samples into several jobs. The output data produced by the analyses are typically copied to the storage of a site and registered in the experiment specific catalogue. Small output data files are returned to the user. In the CMS experiment the CRAB tool set has been developed in order to enable physicists to perform distributed analysis over the Grid. The role of CRAB is to allow the user to run over distributed datasets the very same analysis she/he ran locally, and collect the results at the end. CRAB interacts with the distributed environment and the CMS services, hiding as much of the complexity of the system as possible. CMS community members use CRAB as a front-end which provides a thin client, and an Analysis Server which does most of the work in terms of automation, recovery, etc. with respect to the direct interactions with the Grid. The Analysis Server enables full workflow automation among different Grid middlewares and the CMS data and workload management systems. Indeed, the main reasons behind the development for the Analysis Server are:

- automating as much as possible the whole analysis workflow;
- reducing the unnecessary human load, moving all possible actions to server side, keeping a thin and light client as the user interface;
- automating as much as possible the interactions with the Grid, performing submission, resubmission, error handling, output retrieval, post-mortem operations;
- allowing better job distribution and management;
- implementing advanced use cases for important analysis workflows

The server architecture adopts a completely modular software approach. In particular, the Analysis Server is comprised of a set of independent components (purely reactive agents) implemented as daemons and communicating asynchronously through a shared messaging service supporting the "publish & subscribe" paradigm. Most of the components are themselves implemented as multi-threaded systems, to allow a multi-user scalable system, and to avoid bottlenecks. The task analyses are completely handled during their lifetime by the server through different families of components: there are components devoted to monitoring the Grid status of the single jobs in a task, other groups of agents coordinate to manage the output retrieval and the recovery of the failed jobs by scheduling their resubmission automatically. A relevant part of the agents is designed in order to handle the submission chain of user tasks to the Grid. As the Analysis Server internal architecture is a natural candidate for being analyzed with the nets-within-nets paradigm, as aforementioned, we decided to model and study the Grid submission chain. The aim of this study is to check that the involved agents behave correctly and efficiently with respect to the foreseen submission workflow. We decided to consider the system at the component-task-job level, as it represents a good compromise between the effects perceived by the tool final users and the large number of technical details that a complete representation of the Grid would require.

4 Modeling the submission use-case

In this Section we describe in detail the process of submitting jobs to the Grid through the CRAB Analysis Server. For each relevant component of the system its net representation is discussed. In addition, the bugs that have been discovered thanks to the net models are presented with the solutions that the actual code has adopted in order to solve the issues. The CRAB analysis suite was modeled using nets in a hierarchical fashion, as shown in Figure 2. A vertical line with multiplicity n, indicates the presence of n nets in the higher one (e.g.: the CRABClient net contains from 1 to N Task nets); a horizontal dashed line indicates that the linked nets are references to the same net. In our modeling we consider one client just for the purpose of simplicity. Of course, the discussed functionalities and use cases still hold when a larger number of clients

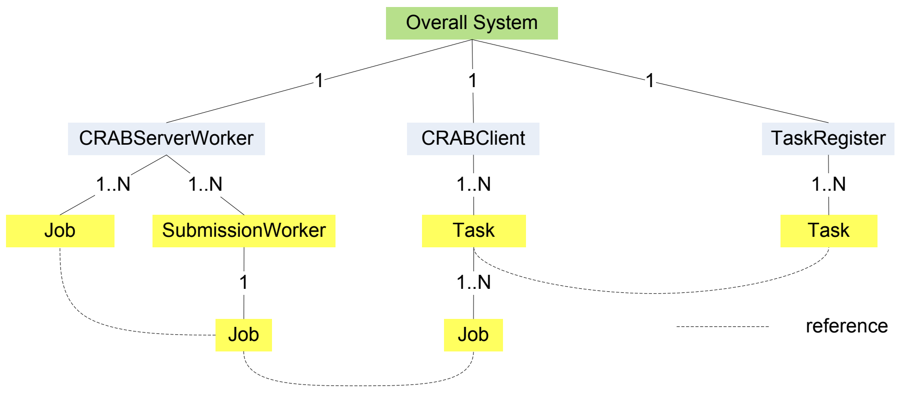


Fig. 2. The Nets hierarchy for the CRAB suite.

is considered, as the client server model assumes no direct interactions among the clients. In addition, for the use case that will be discussed, the server code separates properly the session of work for every task.

The OverallSystem net, which is the system net, contains three nets which respectively model the behavior of the client who is using the CRAB server (*CRABClient net*), the TaskRegister component which is a thread running on the CRAB server (*TaskRegister net*), and the CRABServerWorker which is also a thread running on the server (*CRABServerWorker net*). *Tasks* are the objects a client creates, and deals with. They are composed of *jobs*, the single units of work that need to be performed. The TaskRegister component is responsible for registering tasks, i.e. creating some data structures on server disks, checking if each task has all the inputs it needs to be executed, and checking if the Grid can access the proper security credentials to execute it. The CRABServer-Worker component continuously receives jobs, schedules them for execution on the Grid infrastructure, and creates a SubmissionWorker thread which monitors the lifecycle of each job on the Grid. The clients interact with the server, and can initiate some operations like: submitting jobs, killing them if needed, and asking for the results.

4.1 CRABClient, Tasks, and Jobs

The first component we are going to discuss is the CRAB client, which is modeled with the net in Figure 3. This component is what enables all the action sequences that the users can do on their Grid analyses.

The first thing a client does is to create a new task on the client machine. The typical usage pairs a unique task with a CRAB analysis session. For this reason we assume that the *tasksPool* can contain a finite number of tokens. After the task has been locally created on the client machine, the client can perform a submit operation, which is of course the most important one as it starts the submission chain. The first time a task is submitted to the server, it is also regis-

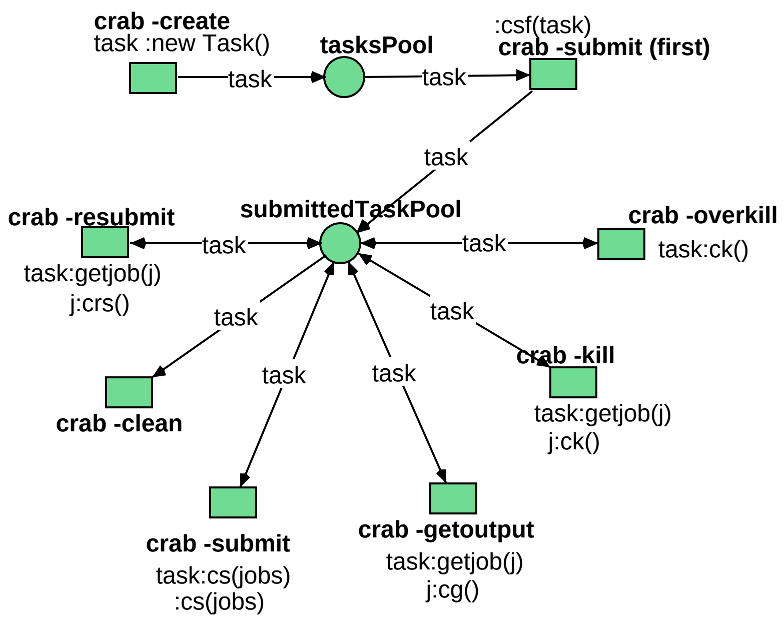


Fig. 3. The CRABClient net.

tered by the TaskRegister component. Subsequent submits are handled directly by the CRABServerWorker component. In our model the difference between the two types of submits is modeled as two different transitions. In particular *crab -submit(first)* transition has an uplink (*:csf(task)*), which means that it must be synchronized with the upper level. As a result the task reference is copied to the TaskRegister component by the Overall System net. After creation, the main operations a user can do are submit, resubmit, kill, getoutput, and clean. All these operations require an interaction with the server, but since we have focused on the submission use case, these interactions have not been explicitly modeled. For example the *getOutput* command is modeled as an interaction between the client and the job by means of two inscriptions. Handling all the possible interactions between the actors involved in the system would have resulted in a very big model, making it impossible to describe in this paper.

A task, see Figure 4, is a bag of jobs (the system allows to collect up to 4000 jobs into a singe task) and it is a representation that CRAB uses to perform collective actions on the Grid processes. Places *notRegistered, registering, registered* of the Task net contain information about the state of a task itself. These places control the enabledness of transitions *crab -submitFirst*, and *taskRegistered*, which are respectively called by the CRABClient when a job is submitted, and by the TaskRegister component when the task has been successfully registered after a *submit first* operation. The submit transition is called when a

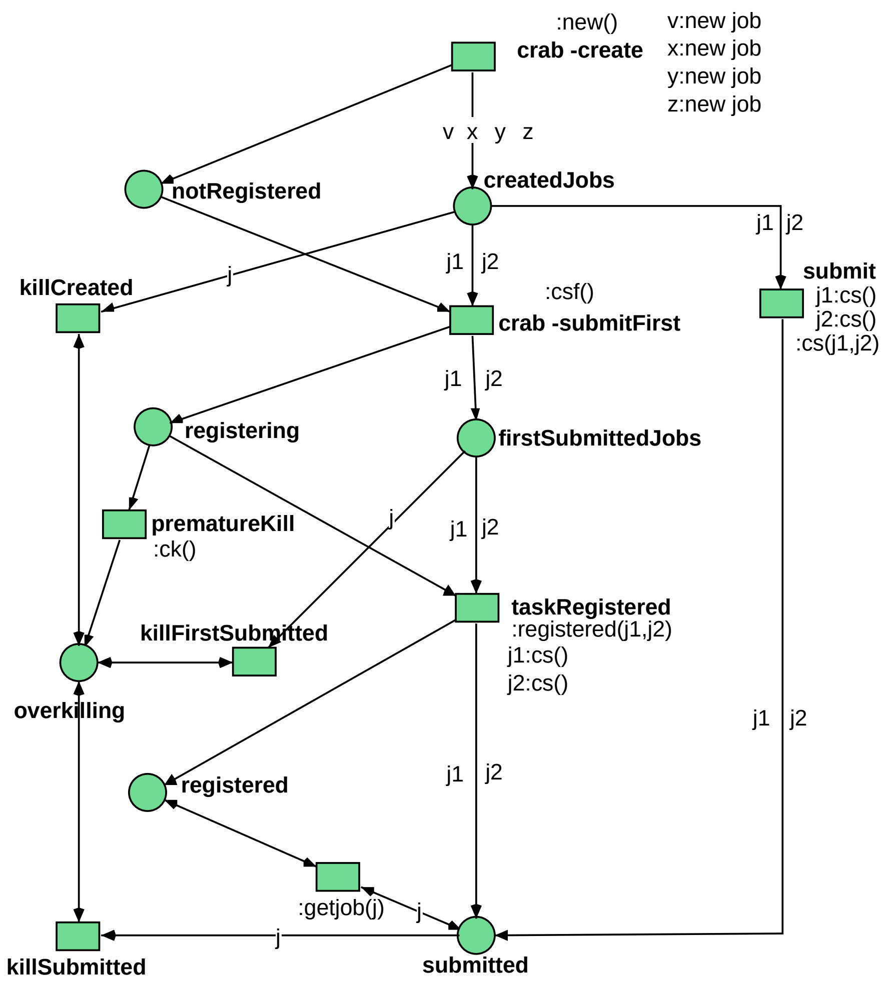


Fig. 4. The Task net. Only four jobs are considered in order to exemplify the relation with the job net.

CRABClient performs a *submit subsequent* action. In our model both *taskRegistered*, and *submit* transitions send upward two jobs through a synchronous channel, and make the job move to the submission request state.

The net representing the state of Grid jobs and their allowed actions is reported in Figure 5. This net has been modeled combining the finite state machine reported in the CRAB official documentation with the information extracted directly from the portion of code devoted to the Grid job state handling. Several transitions of this net contain uplinks, and therefore have to be synchronized with some other net. Transitions with a *:crs()* uplink (CRAB Resubmit) are

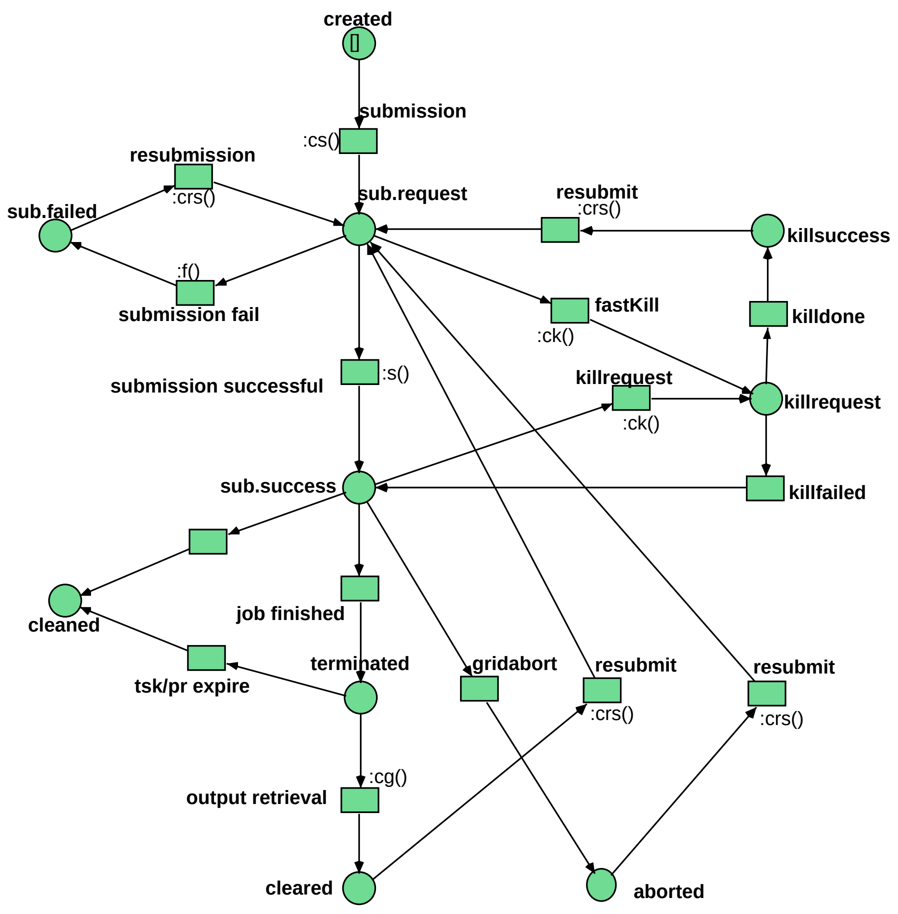


Fig. 5. The Job net.

transition enabled only if the job is in a state where a resubmit is possible, and are synchronized with the *crab -resubmit* transition of the CRABClient net, or the *resubmit* transition of the SubmissionWorker net. In the same way killings (channel *:ck()*), failures (channel *:f()*), submission (channel *:s()*), and output retrieving (channel *:cg()*), have to be synchronized with a correspondent transition in another net.

The integration of the documentation and the code with the formalism of the nets has allowed us to identify a bug in the way job states are modified. In particular, the net allows some transitions that are not actually activated by any event observed by the system (bug 1, b1). For example let us consider the unlabeled transition between the *sub.success* and the *cleaned* places in Figure 5: the latter denotes that a job has been abandoned because the user security credentials are expired and the Grid will not manage processes whose owner cannot be recognized. A malicious code interacting with the clients in place of the proper server could move jobs arbitrarily to this terminal state. The fix for this bug consisted in a review of the code managing the job state automata in accordance with what is stated by the presented Job net. Also, the pre-conditions that allow a client to perform a *kill* request over the jobs are not granted properly (b2): jobs can be killed when they are in states where the killing is dangerous. For example, a user could run into a condition where a failed job cannot be resubmitted as the system requires to kill it. That means the job is in a deadlock, as a failed job cannot be killed on the Grid.

4.2 TaskRegister

The TaskRegister component, shown on the left of Figure 6, duplicates the task and jobs structures that have been created at the client side and alters all the object attributes in order to localize them with respect to the running environment of the server, taking care also of security issues (like user credentials delegation) and files movement (check the existence of input). We modeled this cloning by means of the *reference semantics*: the TaskRegister component receives from the client a copy of the reference which points to the Task.

The component is able to handle more tasks simultaneously thanks to a pool of threads implementing the net of Figure 6. The first transition that is fired is *submission*, which is synchronized with the transition in the system net that receives the task reference from the CRABClient. Then four operations which can fail are executed on the task. These include local modification of the task with respect to the server environment, the user's credential retrieval (also known as delegation), the setting of the server behavior according to what the credentials allow to do and, finally, the checking that the needed input files are accessible from the Grid. If the registration fails the only possible operation available is *archiveTask* which deletes the reference to the task from the task register component. If the user has the privileges to execute the jobs in the task, and if the inputs needed by the task are available, then a range of jobs is selected from the task and passed to the CRABServerWorker by firing the *toCSW* transition (again under the supervision of the system net). The modeling and the simulation of the TaskRegister net has highlighted some relevant defects and bugs. In case of failure the TaskRegister component was not able to set properly the status of the jobs in a task to fail. This macroscopic lack in the system design implied different side effects. The server was not able to discriminate whether to retry automatically the registration process or to give up and notify the user about the impossibility to proceed (b3). In addition, the system could not tell if the registration has been attempted previously. This implies that the client transfers the input data every time a registration failure appears, with a waste of network resources (b4). Both the defects have been solved by introducing the proper synchronization between the *fail* transition in the component with *submission failed* in the job net. Mapping the synchronization into the server code has granted that the status of the jobs is set to the correct failure state and that

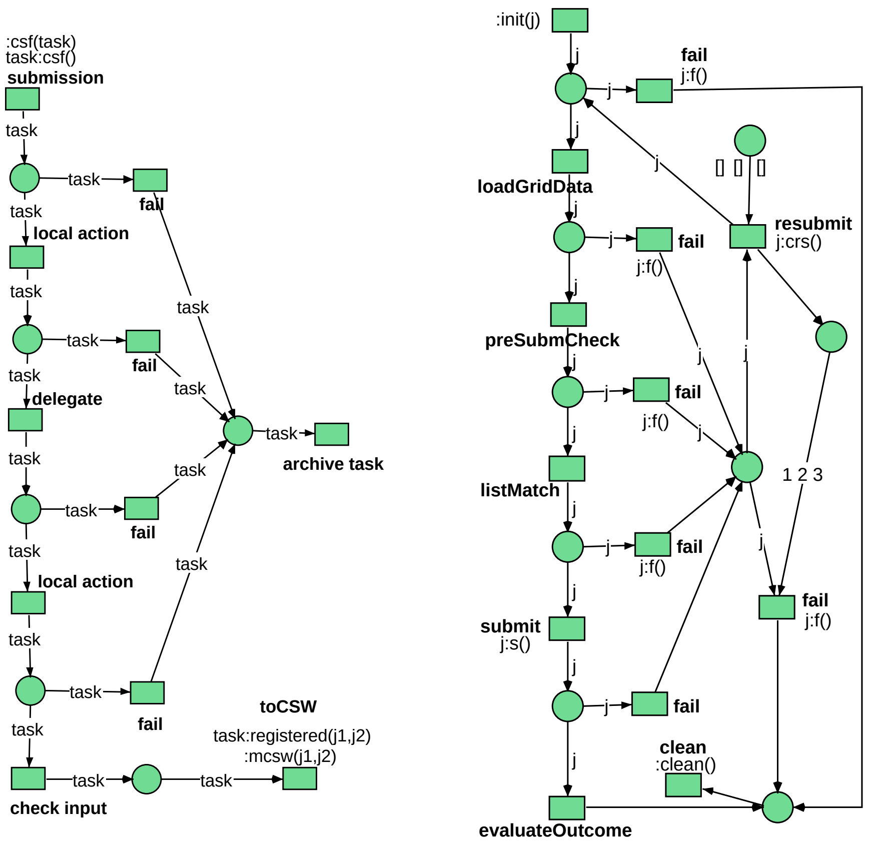


Fig. 6. TaskRegister and SubmissionWorker nets respectively

the submission counters are properly incremented (being implementative details the counter is not reported in the Job net). With this modification the server becomes aware that a first try has been executed and also network transfers are exploited more efficiently. A second bug has been identified thanks to the study of the synchronization among the transitions for the client, the jobs and the TaskRegister nets. In detail, the handling of the *kill* commands presents some issues. If a user requires to kill some jobs while the task is being registered, the system cannot distinguish properly which jobs have to be killed and therefore it applies an over-killing strategy by halting the whole task (b5). This happens because the code performs some sort of synchronization with the Task net instead of having rendezvous with the related transitions into the lists of killing jobs.

The killing of Grid jobs is a demanding action, both in terms of network communications and in terms of coordination among the different services involved in a Grid. Furthermore the killing of an analysis job is a permitted but infrequent action. For these reasons the CRAB developers have decided to suppress this early job termination feature in order to avoid the bug. Now users are allowed to kill jobs only once they have been actually submitted to the Grid.

4.3 CRABServerWorker, and SubmissionWorkers

In our model the result of a submit operation is that the CRABServerWorker component, shown in Figure 7, receives a structured token in the place *accepted*. If the submit was the first, transition *newTaskRegistered* is fired after the task has been registered by the TaskRegister component by means of transition *toCSW*, which is synchronized with transition *newTaskRegistered* through the overall system. If the submit is not the first, the task has been already registered, therefore transition *subsequentSubmission* is fired. After receiving the range of jobs, the CRABServerWorker component schedules these jobs for the execution on the Grid infrastructure. The practical effect of this component is to break the task into lists of jobs in order to improve the performance thanks to bulk interactions with the Grid middleware. The Submission Worker thread spawned by the component monitors the actual submission process of the jobs. We have modeled this fact by creating a Submission Worker net for each one of the jobs in the list. Indeed, transition *triggerSubmissionWorker* creates a new Submission Worker assigned to the variable *sw* and synchronizes it with a transition labeled *init*.

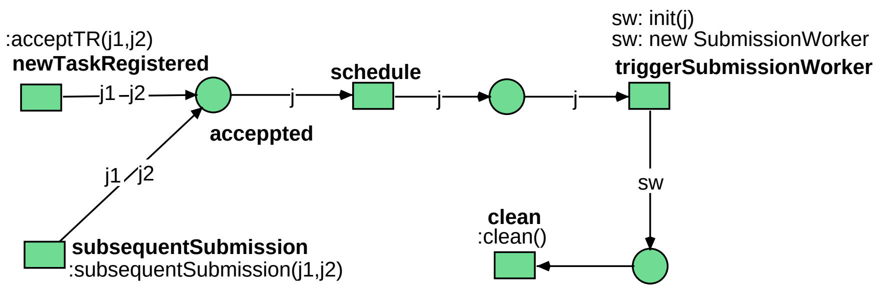


Fig. 7. The CRABServerWorker Net

The thread is responsible both for tracking the submission to the Grid infrastructure, and for resubmitting jobs when a failure occurs. Failures can occur for different reasons: network communication glitches, unavailable compatible resources, etc. Some types of failures are recoverable and in those cases the Submission Worker automatically tries to resubmit the job a three times. This value can be configured in the code, but in the model we only used the actually employed value of three. If the failure persists the job is permanently marked as failed. The net shown on the right in Figure 6 is our model of the submission worker component.

The study of the synchronization between the job and the Submission Worker nets allowed us to identify another bug in the code. The *submission success* transition in the job net (Figure 5) synchronizes with the *submit* Submission Worker's transition (right of Figure 6). This means that the CRAB Server marks the submission as successful just after the interactions with the Grid. Actually the network latencies could delay the propagation of the job failure message (b6) and, therefore, the correct rendezvous should be enacted between *submission success* and *evaluateOutcome*.

It is relevant to observe that the approach followed for the modeling of the CRAB Server submission chain is a particular case for a quite general class of Grid systems. All the Grid middlewares rely on jobs that are represented by finite state automata and that are concurrently managed by the different services involved in the Grid. In addition, the intermediate action of a broker like the CRAB Server is becoming a common pattern with the diffusion of scientific gateways: programmatic portals that abstract the user applications from the complexities of the distributed infrastructures acting as back end.

The adoption of the nets-within-nets paradigm has provided a natural and effective way to model subtle interactions among the different net levels. It would have required a significantly greater effort to discover the same problems with a flat net approach. In the following subsection details about the process of deriving the models from the documentation and the code are given.

4.4 Details on the model derivation process

The model was derived from the code by analyzing both the official documentation and the source code of the system. The Job net is directly built from the documentation. A finite state automata which describes the Job is reported explicitly. After that, simply by using pattern matching we analyzed the source code relevant for the submission use case by searching for interaction with jobs. Each source module is modeled as a net (e.g.: CRABClient, TaskRegister, CRABServerWorker etc), and the interactions with the Job nets are modeled using the RENEW uplink/downlink mechanism. A modification of the status of a job in the code is modeled as a pair of synchronized transitions in the model itself: one in the job net and one in the net that models the component changing the job status.

To ensure that the model is an accurate representation of the software, we made several task submissions with the CRAB tool and monitored the status of the jobs during the evolution. The request parameters were set up so that different behaviours of the system are tested. For example, jobs lacking of input files, job submitted by users with expired credentials, and jobs killed before the completion of task registration process are test cases that have been considered. After that, we simulated each submission on the model, taking care that the simulation of the status of the job net was consistent with the actual job status in the system.

5 Discussion

In the study we have just presented, a formal approach was used to validate a system that has already been implemented. Simulating the behavior of the system by means of a computer aided tool was what allowed us to find problems in the implementation of the CRAB server. However, another great advantage of modeling a system with formal methods is the possibility to apply *automatic* analysis techniques to extract information about the system, like invariant analysis, and model checking.

In order to apply some of these techniques, the formal model must respect specific prerequisites. For example, most algorithms for model checking a concurrent system require a bounded state space. Nets-within-nets models which satisfy this last requirement are hypernets [2] and their generalization [4], which can both be expanded to 1-safe Petri nets [3, 18]. This expansion guarantees the possibility of applying all the analysis techniques of this well known class of Petri nets to hypernets.

The first idea was to use such a class of nets to model the CRAB server, but because of the absence of modeling limitations and verification features in Renew, and because of the high complexity of the system, we preferred to use a slighty more powerful version of hypernets. To come back to the class of hypernets, having therefore the certainty that the state space is limited, the following fixes are necessary:

- Transitions which create or delete tokens must be deleted in some way. For example, transition *crab -create* of the CrabClient net cannot create an unbounded number of tasks anymore, but an input place which contains as many tokens as the maximum number of allowed tasks must be added.
  This is not a big problem. As a matter of fact the computers disks space is limited, and consequently so are the number of tasks which can be created by a user.
- Hypernets use a *value semantics*, which means that a net cannot have two references to it. Nevertheless, in our model some transitions duplicate the references to a net. Duplication of references is somehow dangerous if the intention is to keep the state space bounded. Loosely speaking, the risk is of an uncontrolled grow of the references of a net without a corresponding deletion of these references. In our model the use of the value semantics can be achieved by deleting these duplications of references, and using simple tokens to communicate the intention to modify the referenced net.

Even though analysis of properties is not available with the current version of the model because of the issues just discussed, the more practicality of the reference semantics from a modeling point of view helped us finding several design defects in the implementation of the CRAB server. In the future we plan to restrict the model to a hypernet in order to be able to verify properties like invariants, or to do model checking [1]. In our opinion, as a first step it was

[1] Restricting the model to hypernets is not the only way to have a limited state space, but a formal proof is available using hypernets thank's to the 1-safe expansion

important to use a powerful formalism to avoid getting lost in the details of the model, even though that meant sacrificing the analysis capabilities.

6 Conclusions

In this paper, we discuss a large scale Grid application used to perform distributed data analysis in High Energy Physics experiments. Because of the complexity of the architecture, the software tool has been modeled using the nets-within-nets paradigm in order to validate the correctness of its behavior using simulation. In particular we considered the fundamental use case of the submission of user data analysis to the Grid. Every component of the CRABServer involved in this use case has been modeled in the hierarchy of the nets and compared to the behavior expected by its users.

From the simulation of the model a number of bugs and design defects emerged. This has led the developers to improve the overall quality of system implementation in the subsequent releases that the users now adopt. Two groups of bugs have been identified: bugs related to wrong coding of the expected behaviors and bugs where the specific adoption of nets-within-nets formalism has highlighted synchronization problems among the entities .

In addition, the approach followed to model the CRAB tool set has shown its generality in order to model most of the Grid applications in which an orchestration entity drives the nets representing both the finite state machines of the jobs running on the distributed infrastructure and the services exposing the resources themselves.

The class of nets used to model this system is a more powerful version of hypernets, using the reference semantics instead of the value semantics, and allowing creation/deletion of tokens. As discussed in Section 5, it is possible to restrict the model to a proper hypernet by sacrificing its readability (some places and transitions must be added). Then, by means of hypernets and their expansion to 1-safe nets, it will be possible to use all the techniques defined for the class of 1-safe nets for analyzing the system.

A plugin of RENEW that allows to draw and to analyze a hypernet is being developed. We plan to use this plugin to make automatic verification of properties of the system.

References

1. Martin Alt, Andreas Hoheisel, Hans Werner Pohl, and Sergei Gorlatch. A Grid Workflow Language Using High-Level Petri Nets. In *Procs of the 6th Int. Conf. on Parallel Processing and Applied Mathematics: PPAM05*, pages 715–722, 2005.
2. Marek A. Bednarczyk, Luca Bernardinello, Wiesław Pawłowski, and Lucia Pomello. Modelling mobility with Petri Hypernets. In *Recent Trends in Algebraic Development Techniques*, volume 3423/2005 of *Lecture Notes in Computer Science*, pages 28–44. Springer Berlin / Heidelberg, 2005.

3. Marek A. Bednarczyk, Luca Bernardinello, Wiesław Pawłowski, and Lucia Pomello. From Petri hypernets to 1-safe nets. In *Proceedings of the Fourth International Workshop on Modelling of Objects, Components and Agents, MOCA'06, Bericht 272, FBI-HH-B-272/06, 2006*, pages 23–43, June 2006.
4. Luca Bernardinello, Nicola Bonzanni, Marco Mascheroni, and Lucia Pomello. Modeling symport/antiport p systems with a class of hierarchical Petri nets. In *Membrane Computing*, volume Volume 4860/2007 of *Lecture Notes in Computer Science*, pages 124–137. Springer Berlin / Heidelberg, 2007.
5. Carmen Bratosin, Wil van der Aalst, and Natalia Sidorova. Modeling Grid workflows with Coloured Petri nets. In *Procs. of the 8th Workshop on Practical Use of Coloured Petri Nets and CPN Tools: CPN 2007*, pages 67–86, 2007.
6. CERN. Worldwide LHC Computing Grid. http://lcg.web.cern.ch/lcg/public/. Accessed May, 2010.
7. Giuseppe Codispoti, Mattia Cinquilli, Alessandra Fanfani, Federica Fanzago, Fabio Farina, Carlos Kavka, Stefano Lacaprara, Vincenzo Miccio, Daniele Spiga, and Eric Vaandering. CRAB: a CMS Application for Distributed Analysis. *IEEE Transactions on Nuclear Science*, 56(5):2850–2858, 2009.
8. Jordi Cortadella and Wolfgang Reisig, editors. *Applications and Theory of Petri Nets 2004, 25th International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004, Proceedings*, volume 3099 of *Lecture Notes in Computer Science*. Springer, 2004.
9. Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.
10. Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. Grid services for distributed system integration. *Computer*, 35:37–46, 2002.
11. Zhijie Guan, Francisco Hernandez, Purushotham Bangalore, Jeff Gray, Anthony Skjellum, Vijay Velusamy, and Yin Liu. Grid-Flow: a Grid-enabled scientific workflow system with a Petri-net-based interface: Research Articles. *Concurr. Comput. : Pract. Exper.*, 18:1115–1140.
12. Kathrin Hoffmann, Hartmut Ehrig, and Till Mossakowski. High-level nets with nets and rules as tokens. In Gianfranco Ciardo and Philippe Darondeau, editors, *ICATPN*, volume 3536 of *Lecture Notes in Computer Science*, pages 268–288. Springer, 2005.
13. Andreas Hoheisel and Uwe Der. Dynamic Workflows for Grid Applications. In *Procs. of the Cracow Grid Workshop 03*, page 8, 2003.
14. Michael Köhler and Berndt Farwer. Object nets for mobility. In Jetty Kleijn and Alexandre Yakovlev, editors, *ICATPN*, volume 4546 of *Lecture Notes in Computer Science*, pages 244–262. Springer, 2007.
15. Michael Köhler and Heiko Rölke. Properties of object Petri nets. In Cortadella and Reisig [8], pages 278–297.
16. Olaf Kummer. *Referenznetze*. Logos-Verlag, 2002.
17. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Jörn Schumacher, Michael Köhler, Daniel Moldt, Heiko Rölke, and Rüdiger Valk. An extensible editor and simulation engine for Petri nets: Renew. In Cortadella and Reisig [8], pages 484–493.
18. Marco Mascheroni. Generalized hypernets and their semantics. In *Proceedings of the Fith International Workshop on Modelling of Objects, Components and Agents, MOCA'09, Bericht 290, 2009*, pages 87–106, September 2009.
19. The CMS Collaboration. The CMS Experiment at CERN LHC. *J. Inst.*, 3:S08004, 2008.

20. The TLS Group. The Large Hadron Collider Conceptual Design. Technical report, CERN, 1995. Preprint hep-ph/0601012.
21. Rüdiger Valk. Nets in computer organization. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume Volume 255/1987 of *Lecture Notes in Computer Science*, pages 218–233. Springer Berlin / Heidelberg, 1987.
22. Rüdiger Valk. Petri nets as token objects: An introduction to elementary object nets. In Jörg Desel and Manuel Silva, editors, *ICATPN*, volume 1420 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 1998.
23. Rüdiger Valk. Object Petri nets: Using the nets-within-nets paradigm. In *Lectures on Concurrency and Petri Nets*, volume 3098/2004 of *Lecture Notes in Computer Science*, pages 819–848. Springer Berlin / Heidelberg, 2004.
24. Kees M. van Hee, Irina A. Lomazova, Olivia Oanea, Alexander Serebrenik, Natalia Sidorova, and Marc Voorhoeve. Nested nets for adaptive systems. In *ICATPN*, pages 241–260, 2006.