Formal Verification of a Grid Resource Allocation Protocol

Mathias Dalheimer and Franz-Josef Pfreundt
Fraunhofer Institut für Techno- und Wirtschaftsmathematik
Kaiserslautern, Germany
{dalheimer,pfreundt}@itwm.fhg.de

Peter Merz
Department of Computer Science
University of Kaiserslautern, Germany
peter.merz@ieee.org

***Abstract*—As the adoption of grid technology moves from science to industry, new requirements arise. In todays grid middlewares, the notion of paying for a job is a secondary requirement. In addition, the concept of selling computational power on a market is not established. On the other hand, the lack of billing capabilities hinders the commercial adoption.**

In this paper, we present our resource allocation protocol that suits the needs of commercial solution providers. We have developed an auction-based resource broker which uses a distributed agent infrastructure to communicate the user's requirements to resource providers and monetary prices back. The protocol has been formally verified and guarantees certain properties - for example, we can guarantee that the right stakeholder is billed for a job.

I. Introduction

Job schedulers for distributed systems usually aim at the optimization of metrics like throughput or response time. With the advent of grid computing, other metrics such as file transfer times are used during scheduling. We argue that privacy issues and price are important in commercial grid use cases and need to be incorporated as well.

For example, consider a service provider which offers simulation software as a service. The user defines the job and transfers the input data to the service provider. It is up to the service provider to find suitable resources and forward the data - however, since the input data might be confidential, some resource providers might not be acceptable to this specific user.

In addition, different resource providers compete on a market for computational resources in terms of prices and speed. From the viewpoint of a resource provider, the user must be able to pay for the job execution. The service provider fulfills the role of a broker which ensures that payments will be made. In the case of execution failure, the service provider needs to determine which entity bears the responsibility and arrange appropriate penalty payments.

In this paper, we present the scheduling protocol of Calana. It distinguishes the roles of a resource provider, a user and a broker that coordinates requests and offers. We argue that it is not possible to do scheduling completely without user interaction because commercial computations often deal with confidential data - only the user can decide whether to trust a resource provider or not. The protocol finds a contract between user and resource provider. It ensures the accountability of the whole computation.

The paper is structured as follows: In the first section, we present the use cases with more details and derive the requirements for a scheduler. We then discuss the scheduling protocol in sections III and IV. The verification of the protocol is presented in section V. Related work is given in section VI and we conclude in section VII.

II. Grid Use cases in Commercial Applications

We describe two use cases more detailed and derive the requirements for scheduling in similar scenarios. For the discussion of the use cases, we use the following terminology:

1) A user is the consumer of a grid service.
2) A resource provider is an organization that provides computational and data resources to users for money. Typically, resource provider and users are not part of the same organization.
3) A service provider provides services to users while utilizing the resources of resource providers.

Platforms like Amazons Elastic Compute Cloud (EC2) show the great interest in using remote resources dynamically for a certain fee [1]. Regarding the quality of the resource two metrics are used today: monetary price and performance, i.e. computation speed or network bandwidth.

In the German project PartnerGrid, we cooperate with simulation software providers. Our requirements are derived from their usage scenarios, which we outline below.

A. Metal Casting Simulations

MAGMASOFT is a simulation tool chain for casting simulation. The user defines the geometry of the cast along with material and casting parameters and simulates the casting process. Based on the simulation results, the mold might be adjusted until certain properties are fulfilled - then, the real part will be casted. The application consists of a front end with built in visualization capabilities. The simulation is encapsulated in a backend which can be executed on the user's desktop, on a compute cluster or in a grid environment.

The specification of the mold contains a lot of knowledge of the user and is critical for the successful casting of highly specialized parts. A competitor could learn a lot from these input files. It is therefore not acceptable to the user to have

a job running “somewhere” on the grid - confidentiality must be guaranteed for the job. A potential resource provider must provide a security certification. These may be issued by a third party which ensures that certain standards are met.

B. Numerical Forming Simulations

GNS provides users from mainly the automotive industry with forming simulation software in order to calculate stresses and other material properties after the forming process. This allows the user to predict the behavior of the manufactured parts in the application. As in the MAGMASOFT use case, the way of manufacturing the parts contains a lot of knowledge and must not be revealed to competitors.

In addition, users demand control over price and performance. If a project deadline is close, users are willing to spend more money for a computation. If the job completion is not critical, they want to save money. Resource providers have the opportunity to define different pricing schemes. For example, if a grid resource is underutilized, the resource provider might offer a discount rate to cover the availability cost without a margin.

C. Requirements for scheduling

As the scenarios above show, it is important to users to be able to identify the provider actually executing their jobs. A scheduler suitable for these use cases must provide the following features:

1) The user must be able to identify the resource provider and approve it.
2) The user must be able to give his job different priorities and exploit the underlying market for resources.

In addition to the user’s requirements the resource and service provider have additional requirements. When resources are provided on a per-usage basis, the provider needs to be able to examine the credit rating of the user for a specific job before accepting it. The broker needs to account for the jobs and needs an audit trail in order to be able to pin problems.

As Cheliotis et al. recommend, we use an auction to determine prices dynamically for each job [2]. A broker maps current job requests to allocations using an auctioning scheme. The broker incorporates pricing and performance information in its decision. In addition, the user can specify a trade-off between price and performance. We discussed the market mechanism in an earlier paper [3].

We have developed a protocol that satisfies these needs and formally verified several properties. In the next sections, we present the core ideas, while the verification is presented in section V.

III. Overview of the protocol

Calana is a grid scheduling framework. In contrast to other systems, it is not composed of an information system and a scheduling daemon but uses a distributed agent-oriented architecture. Figure 1 shows the typical environment of Calana. We distinguish scheduling from job management in the sense that scheduling decides where to run a job, while job management is concerned with how the job is executed, e.g. when to initialize data transfers.

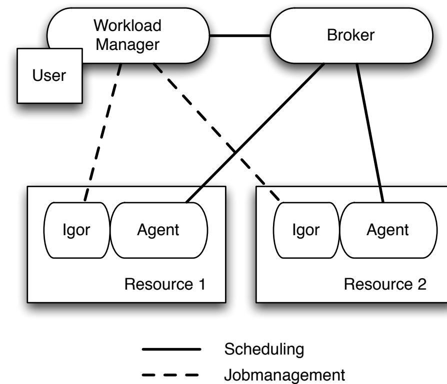

Fig. 1. The environment of a Calana deployment.

A *workload manager* is responsible for the execution of a task and serves as a “proxy” for the user. It contacts the *broker* which starts an auction for the job. The *agents* receive the auction request and decide to bid. When the auction has timed out, the best bid is offered to the workload manager. If the bid is accepted, the workload manager contacts the agent in order to get the endpoint address for the corresponding *igor*[1]. Igor is the entity actually executing the job – this can be a single node or a cluster system. The agent and the igor are closely coupled and exchange job and resource states frequently. The workload manager now contacts the igor directly in order to run the job.

We divide a job execution in three phases: First, an allocation needs to be made. This involves negotiating with resource providers, in our case through an auction. The second phase encapsulates the execution of the job. Finally, we need to account for the job – depending whether the job was successfully executed.

In addition to the requirements in section II, we have these nonfunctional requirements:

1) The messages need to be exchanged asynchronously. Since the events occur asynchronously, this is a natural requirement.
2) Eventually, the job state must be the same for all stakeholders.
3) Both user and provider must be able to cancel the allocation at all times.

The purpose of the broker is to mediate between the workload manager and the providers. It handles the negotiation and decides which provider offered the best bid. After the allocation is set, it forwards certain messages to the contract parties, i.e. a ProviderCancel message to inform the workload manager that a job cannot be handled by a provider (of course, a penalty for the provider occurs). All messages are sent to the broker

[1]“Igor, would you mind telling me whose brain I did put in? “ — Dr. Frederick Frankenstein

and forwarded to its final destination if necessary - the broker forms a messaging bus. Please note that the broker daemon can be distributed itself to prevent bottlenecks.

It is not advisable to have more than one logical broker implemented. It would be very difficult to prevent collusion effects like information trade in such a system [4]. In addition, the broker can record audit information easily.

We develop a set of state models in order to keep the job state consistent in the whole grid. Please note that we try to represent the job state - in all subsequent models, we omit the fact that a service itself has a state as well (it may be running, provisioned or down).

A. The Job State Model

Since the job state model is the common ground for all daemons, we adhere to the upcoming OGSA-BES standard in order to preserve the compatibility to other infrastructures [5]. The OGSA-BES model is shown in figure 2. A successful job starts in the state `Pending`, changes to `Running` while the job is executed, and finishes afterwards. If the job crashes, its state is `Failed`. A user can abort the job, updating the state to `Terminated`.

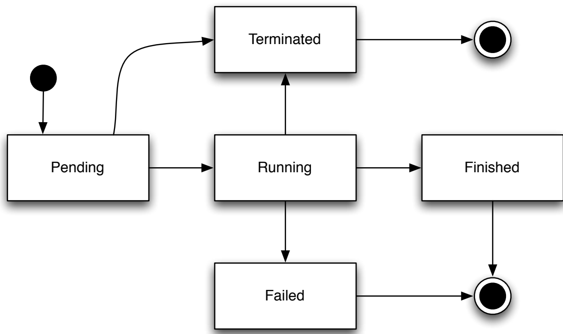

Fig. 2. The OGSA-BES job model as given in [5].

This simple model represents a common ground for many grid infrastructures. It is possible to extend the model by introducing sub states: for example, one might model file staging by adding two sub states `Running:Stagein` and `Running:Stageout` to the `Running` state. Another entity not aware of these sub states will perceive them as the state `Running`, thus preserving the meaning.

Calana extends the BES model as shown in figure 3. The `Pending` state reflects the negotiation process which must happen before the execution can start. In addition, the `Finished` state is divided in two parts: `Finished:Closing` reflects that the job has ended successfully while `Finished:Closed` is only entered if both workload manager and provider have acknowledged.

B. The Resource State Model

For a meaningful coupling of the Calana resource broker with the execution layer, we need a model of the resource's behavior, see fig. 4. We assume that each resource is capable of running one job at any time. The resource starts in the `Down` state. It then may be booted, a middleware component may need to be started. Once it is fully usable, it reaches the state `Ready` where it is capable of executing jobs. If a job can be started successfully, the resource is in the state `Running`. A job may then terminate either successfully or fail - in either case, the resource will be in the `Ready` state again.

If the resource is not executing a job, it can be shut down administratively. In the case of a resource failure, the state will change from either `Running` or `Ready` to `Down`. Please note that this state change might not be visible from the execution layer point of view - if the kernel crashes, a middleware component cannot change the state any more. But for another system, this change can be perceived - e.g. heartbeats might not be received any more.

If a job runs, there are three possible transitions: Either a job fails due to resource failure, then the job is in the state `Terminated` - it is the responsibility of the job management to get the resource back into the state `Free` (not shown in the diagram). Another possibility is a user cancellation - the resource cancels the job and returns to `Free`. Usually, the job terminated just because it is finished - so the resource is in the state `Free` as well.

It turns out that a monolithic `Ready` state is not sufficient for the coupling of the execution layer with the management layer. We introduce sub states as follows:

1) A resource is in the sub state `Ready:Free` if it is currently not processing a job - there is also no reservation for a job. Therefore, an administrative shutdown is possible in this state.
2) If the resource is reserved by the management layer, the resource is in the state `Ready:Reserved`. If the reservation is canceled, the state changes to `Ready:Free` again.
3) Otherwise, the job may be started. In this case, the resource changes to the `Ready:Starting` state. Note that the transition "startJob" is the attempt to start the job, involving all necessary operations. The start might fail, e.g. because the input data staging failed - the resource changes to `Ready:Free` in this case. Otherwise, the job has started successfully and the resource is in the state `Running`.

Please note that the resource model is an important part of the Calana protocol, but not discussed below explicitly.

IV. The Protocol

The protocol specification consists of two parts: The definition of the individual messages as an XML Schema and the definition of the behavior of the daemons. The latter is described using PROMELA, the input language of the protocol verifier SPIN [6]. It is possible to use linear temporal formulae to prove properties of the overall system automatically. We will discuss some properties later in section V.

The complete model is given in appendix A. We introduce the protocol by giving some examples of possible mes-

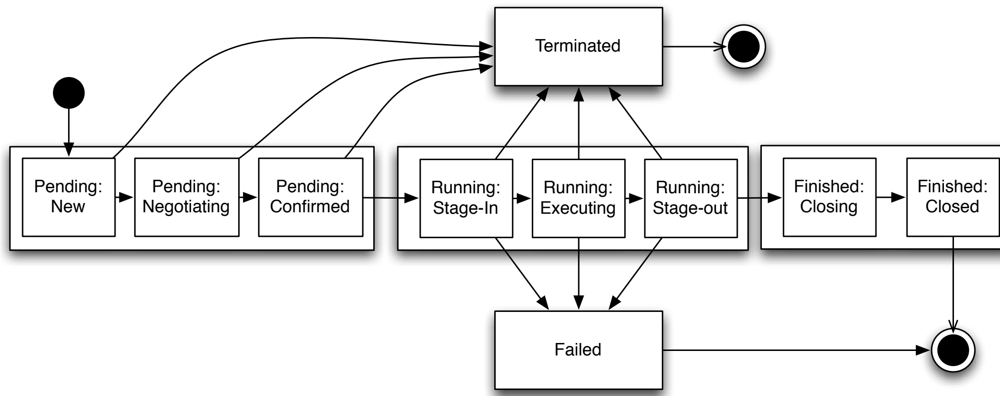

Fig. 3. The Calana job model. It extends the previously shown OGSA-BES job model by adding an advance reservation stage, file staging operations and finished acknowledgments.

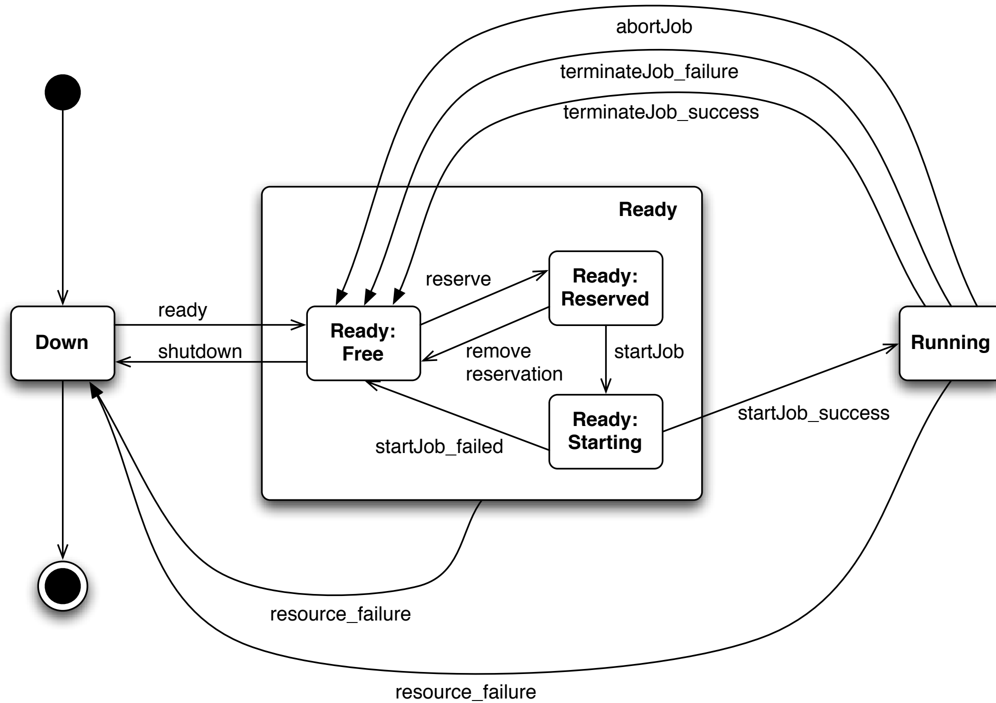

Fig. 4. The state model of a resource as seen from other entities in the grid.

sage exchanges. The protocol has two parts: The workload manager-broker protocol and the broker-provider protocol. Both are nested: The interaction between the workload manager and broker triggers interaction between the broker and the providers.

A. Use case: successful execution of a job

In this first use case we demonstrate the successful execution of a job. The allocation is made and the corresponding igor runs the job without any problems, see fig. 5.

The first message is sent from the igors to their agents in order to notify them that the igor is available for jobs. The workload manager starts the scheduling process by sending a BookingRequest to the broker. The latter then starts an auction by broadcasting a BookingRequest to all attached providers. The providers may now consider this request and choose to answer with a bid. Before the bid is sent, a reservation message tells the igor that it might be running a job in the next time, thus preventing it to shut down. The broker selects the best bid and sends an AuctionAccept message to the winning provider and broadcasts a AuctionDeny message to the others. If a provider receives the AuctionDeny message, it sends a

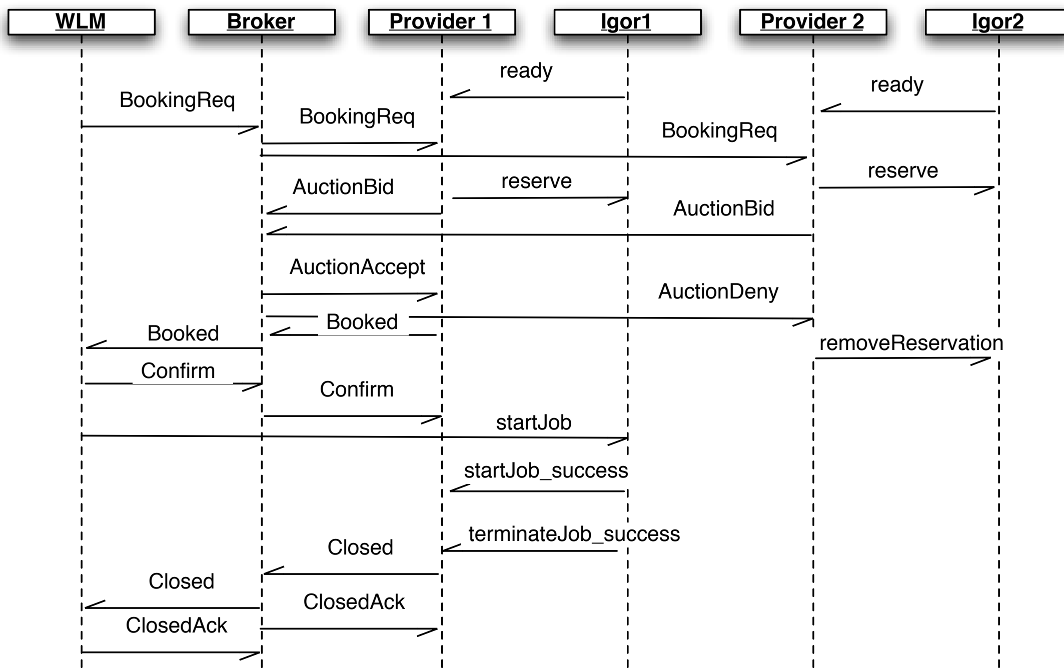

Fig. 5. Example of a successful execution.

removeReservation message to its igor.

So far, there is no binding contract between the workload manager and the provider. This is achieved in the next phase: the broker sends a Booked message to the workload manager. The workload manager can now decide whether to accept the offer or not, possibly interacting with its user. At this point, the user can decide whether the proposed resource is acceptable, i.e. from a security viewpoint. When accepting the bid, the workload manager will respond with the Confirm message. As soon as the broker has received it, there is a binding contract between the provider and the workload manager. The terms of the contract will now be billed.

The broker now forwards this message to the provider. The workload manager and provider systems may now interact directly in order to process the requested job, which is out of the scope of the broker. In this example, the workload manager sends a "startJob" message directly to the igor. In real implementations, this would be replaced by several calls, e.g. to transfer files and various job control commands. These invocations must always occur between the workload manager and the igor directly.

When the job finishes, the igor sends a "terminateJob_success" message to the provider. The broker is notified by receiving a Closed message from the provider. Although this could also be done directly between the workload manager and the provider, the broker needs to be notified for accounting purposes. The broker then forwards the message to the workload manager. Both Closed messages are acknowledged by a CloseAck message.

B. *Use case: The provider cancels*

During each phase of the protocol, it is possible that both workload manager and provider cancel the process. In order to account correctly for the failure, the broker needs to be notified as well. An example of provider cancellation is shown in fig. 6.

In this case, the provider cancels the execution during the lifetime of the agreement of the job: The auction has finished and the workload manager agreed to the scheduling result. The job was started, but the igor failed to start the job. It notifies the provider. The provider itself sends a ProviderCancel message which may contain some information why the execution has failed, e.g. due to resource outtakes or segfault of the software.

The workload manager is notified, and both workload manager and broker acknowledge the result. The broker can then account for the failure and move the corresponding money back. If a reliability metric is used during scheduling, it would also update the reliability score of the provider.

C. *Use case: The workload manager cancels*

In this use case, the workload manager cancels before he has confirmed a reservation, see fig. 7. This might be the case because the proposed resource doesn't fulfill the user's non-technical requirements. This means since there was no commitment, no payment needs to be made. Again, the workload manager sends out a ConsumerCancel message. The broker forwards this to the winning provider, which responds with an ConsumerCancelAck message. It is forwarded to the workload manager. In the meantime the reservation of the igor is removed.

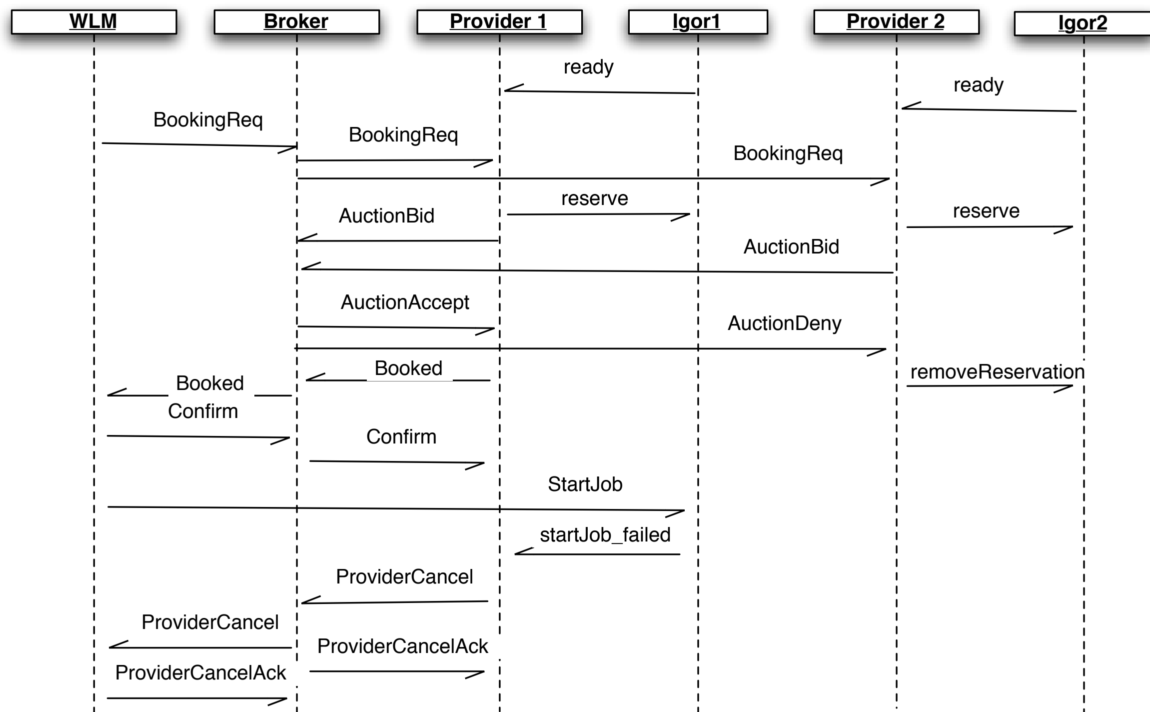

Fig. 6. The provider cancels the execution after the auction phase is finished.

V. Verification of the Protocol

Since we intend to charge users real money for used services, we need to be certain that the protocol works as intended. In general, one can distinguish simulation and testing, deductive verification and model checking methods for the validation of complex applications [7]. Both simulation and testing observe the output of a system under a given input. In general, it is not possible to test or simulate all possible inputs, thus no complete verification is possible. In contrast, deductive and model verification allow a complete verification of a given model. Deductive verification is mostly a manual process and therefore time-consuming.

For our work, we choose *model verification* as verification technique. Model checking takes a model and a specification as input and checks whether the model fulfills the specification at all times [**?**]. If not, a trace is given which allows the user to reconstruct the behaviour of the system leading to the specification violation.

We use SPIN to verify our model [6]. The models are specified in a C-like language calle PROMELA. A PROMELA model consists of the definition of processes and the messages which are exchanged through FIFO pipelines. This makes SPIN a good choice for the verification of distributed systems - SMV [8] for example employs data transfer via shared variables, which is not a good match for communication protocols. SPIN translates the model in finite state automata and enumerates all possible states of the whole model. It is possible to check for liveliness and deadlocks. In addition, one can check specifications written as linear temporal logic formulae.

The model has been verified completely without any restriction of the state-space. In order not to overlook any potential problems, we implemented the model in a generic way: For example, the provider winning an auction is chosen randomly. It is also possible to change the number of participating providers (and igors). However, even for an simplified version of the model, we were not able to verify models with more than five provider-igor pairs due to the state-space explosion. The verification was executed on a 2.4 Ghz Opteron machine with 8GB RAM. The model took approx. 6 GB of RAM to run completely in 12 minutes. We were able to prove that the model is deadlock-free and alive for all system states.

We proved several specification properties with regard to the requirements of our customers (see section II-C). By construction of the protocol, the user can identify the provider and vice versa. In addition, they can use different metrics for the selection of the resource: price and performance are currently supported during the auction phase. Although not implemented at the moment, the broker could integrate a credit ranking for each user request so that providers can assess the user's request. Another option would be to integrate payment services with the broker directly. In the following sections, we present the verification of several important properties of the protocol itself. Since the broker needs to be able to provide accounting information the protocol must guarantee to deliver a unified view for all stakeholders.

The properties, given as linear temporal logic formulae, can

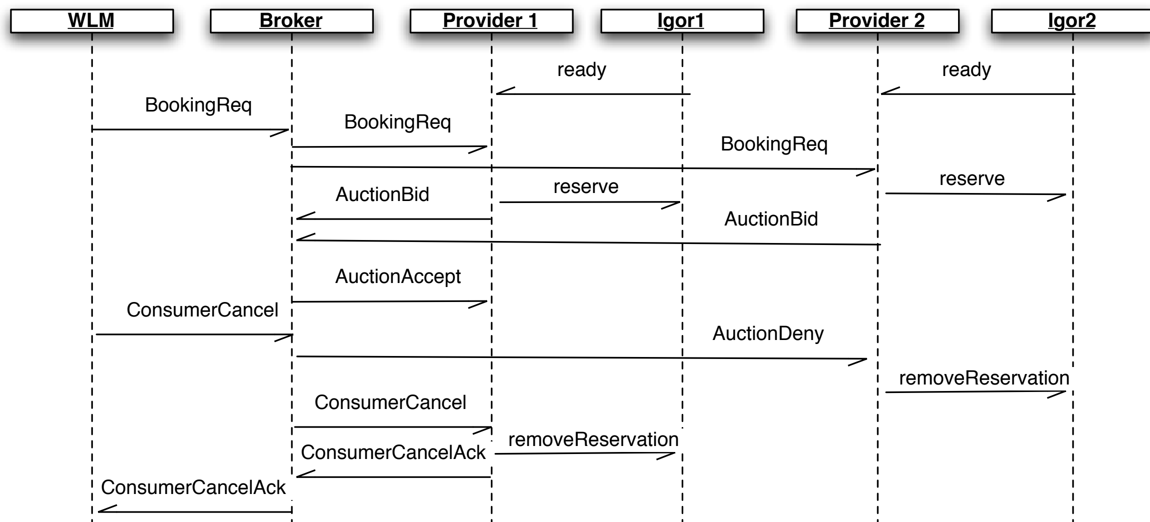

Fig. 7. The workload manager cancels the BookingRequest after the broker has received some bids, no agreement is made.

Outcome	Description
U	Undefined - this is the initial value.
BR	The booking was rejected.
SE	Successful execution of the job.
FE	Failed execution of the job.
UC	User canceled the job.
PC	Provider canceled the job.

Job state	Description
JFC	The job is in state "Finished:Closed"
JPC	The job is in state "Pending:Confirmed"

TABLE I
THE OUTCOME DEFINITIONS FOR THE PROTOCOL (UPPER PART) AND JOB-RELATED PREDICATES (LOWER PART).

be converted to Buechli-automata automatically. The Buechli-automata are then evaluated for all states of the model - usually, the property holds if the negated form of the corresponding Buechli-automaton does not terminate during the verification. An example of an LTL formula and the corresponding automaton is given in appendix B.

In order to simplify the verification, we introduced *outcomes* for all daemons, see table I. Each daemon updates the value of its outcome variable according to the messages it receives. It is now easy to verify properties based on the values of these variables. We also use the predicates defined in table I to refer to certain states of the underlying job. We use the symbol $\mathcal{W}$ for the workload manager, $\mathcal{B}$ for the broker and $\mathcal{P}$ for the provider which has won the auction. The other providers are not discussed here.

Amongst others, we have proven these properties for the protocol [2]:

1) The outcome of the workload manager is always defined: $\Box\,(\mathcal{W} \in \{\mathtt{U}, \mathtt{BR}, \mathtt{SE}, \mathtt{FE}, \mathtt{UC}, \mathtt{PC}\})$. Similar properties hold for the broker and provider daemons.
2) If the job has finished successfully, the broker records the outcome to be successful: $\Box\,(\mathtt{JFC} \rightarrow (\mathcal{B} = \mathtt{SE}))$
3) Eventually, the workload manager has the same outcome as the broker: $\Box\,(\Diamond\,(\mathcal{W} = \mathcal{B}) = \mathcal{P}))$. The broker and the provider have eventually the same outcome if the booking was not rejected: $\Box\,((\neg\mathtt{BR}) \rightarrow \Diamond\,(\mathcal{P} = \mathcal{B})))$.
4) If the consumer cancels a job, he must pay for it - except the provider canceled earlier. The provider cancellation has a higher priority than the consumer's cancellation: $\Box\,((\mathtt{JPC} \wedge \mathtt{UC}) \rightarrow$ $\Diamond\,(\mathcal{B} = \mathtt{UC}) \vee \Diamond\,((\mathcal{B} = \mathtt{PC}) \vee (\mathcal{B} = \mathtt{SE})))$
5) If the provider cancels an already confirmed job, it needs to pay for it - except the job finished earlier: $\Box\,((\mathtt{JPC} \wedge \mathtt{PC})$ $\rightarrow \Diamond\,((\mathcal{B} = \mathtt{PC}) \vee (\mathcal{B} = \mathtt{SE})))$

[2] We use the "standard" symbols for temporal logic: $\Box p$ means "always p" - the term p is an invariant. $\Diamond p$ means "eventually p" and describes the guarantee that p will hold in the end.

VI. RELATED WORK

Commercial offerings of computational resources are still niche products. Amazon's EC2 is amongst the popular ones, but SUN and IBM offer on-demand access to their datacenters as well [1]. So far, these vendors address mostly business consumers and maintain a closed-environment infrastructure - it is not possible to access all resources using e.g. Globus as a middleware. On the other hand, scientific grid initiatives like the German D-Grid are opening towards the commercial use of their infrastructures. We believe that computing power will be easily accessible in the future using standardized middlewares.

We envision a market for computational resources with a need for brokering services such as Calana. Cheliotis et al. share this vision and strongly vote for using markets in order to coordinate the demand and offer of computational power [2]. They also advocate the use of real money instead of an artificial intermediate currency as a way of simplifying the

valuation of grid resources. Buyya et al. suggest an adaptive scheduling algorithm to prioritize the jobs based on their desired deadlines given a fixed budget [9], while Ernemann et al. focus on agents with strategies [10]. Early works include Waldspurgers Spawn architecture [11]. We have presented our market architecture in an earlier paper [3].

There is a lot of literature available concerning the design of electronic markets. Auctions provide an easy way of coordinating demand and supply in various markets, ranging from flowers to next generation mobile network frequencies [12] [4]. The discussion of the economic theory is out of the scope of this paper.

For a market to be successful, it needs to show correct behaviour under all circumstances. Various ways of collusion can destroy the trust in a market quickly. The field of agent-based computational economics provides a lot of information regarding the construction of markets and strategies for agents [13] [14].

There are first steps towards the adoption of market-based use of computational resources. The GRAAP-RG of the Open Grid Forum has published the "Web Services Agreement Specification" which specifies how to use webservice calls to find agreements between two parties [15]. However, there is no integration with the execution layer or a market. This makes it difficult to provide billing services since the real outcome of a job is not known.

On the other hand, the software systems implementing the market need to be solid as well. We found the model checking approach to be very beneficial to the development of our protocol. There are several systems for model checking available, including SMV [8] and SPIN [6] [16]. Other systems like Bandera [17] or Java Pathfinder [18] aim at the automatic extraction of models from source code, e.g. given in Java. Since we developed the model before we actually implemented it, these tools do not provide any advantage for us. SPIN has been used in many applications, e.g. the verification of a space craft controller [19]. Kuo et al. presented a verification of a resource allocation protocol for grid environments [20]. We are not aware of any other formal verification in the field of grid scheduling.

VII. Conclusion

The Calana protocol has been implemented in both simulations and real implementations. We allow users to identify the resource providers and exploit competition through market prices. Both users and providers can cancel the job at any time - the broker will assign penalties for this. The broker can provide a complete audit trail of a job - we can guarantee that all stakeholders eventually assume the same state for the job.

There are still open tasks, mainly from the area of bidding strategy and job estimates. For example, even when the application runtime is known in advance, it is difficult to orchestrate file transfers such that all input files are available for the job on time. In addition, it will be necessary to incorporate software licenses in the scheduling process.

We also intend to couple the broker with a billing and accounting service that can provide user accounts. This way, the broker could check the financial standing of a user before making contracts.

Acknowledgment

The authors would like to thank the Fraunhofer Institut für Techno- und Wirtschaftsmathematik and professor Peter Merz for their continuing support. This work was supported by the German Federal Ministry of Education and Research under the contract 01G07009A-D.

References

[1] (2007) Amazon Elastic Compute Cloud. [Online]. Available: http://aws.amazon.com/ec2

[2] G. Cheliotis, C. Kenyon, and R. Buyya, "Grid Economics: 10 Lessons from Finance," in *Peer-to-Peer Computing: Evolution of a Disruptive Technology, Ramesh Subramanian and Brian Goodman (editors), Idea Group Publisher*, Hershey, PA, USA, 2004.

[3] M. Dalheimer, F. Pfreund, and P. Merz, "Agent-based Grid Scheduling with Calana," in *Proceedings of the Second Grid Resource Management Workshop (GRMW 2005), Poznan, Poland*, 2005.

[4] W. E. Walsh, M. P. Wellman, P. R. Wurman, and J. K. MacKie-Mason, "Some Economics of Market-Based Distributed Scheduling," in *International Conference on Distributed Computing Systems*, 1998, pp. 612–621.

[5] (2007) OGSA Basic Execution Service Version 1.0. [Online]. Available: http://www.ogf.org/documents/GFD.108.pdf

[6] G. J. Holzmann, "The Model Checker SPIN," *Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA: The MIT Press, 1999.

[8] K. L. McMillan, "Symbolic Model Checking: An Approach to the State Explosion Problem," 1992, PhD Thesis, Carnegie Mellon University, 1992. CMU-CS-92-131.

[9] R. Buyya and M. Murshed, "A Deadline and Budget Constrained Cost-Time Optimize Algorithm for Scheduling Parameter Sweep Applications on the Grid," 2001.

[10] C. Ernemann, V. Hamscher, and R. Yahyapour, "Economic Scheduling in Grid Computing," in *Job Scheduling Strategies for Parallel Processing*. Springer, LNCS 2537, 2002.

[11] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta, "Spawn: A Distributed Computational Economy," *Software Engineering*, vol. 18, no. 2, pp. 103–117, 1992.

[12] M. Wellman, W. Walsh, P. Wurman, and J. MacKie-Mason, "Auction Protocols for Decentralized Scheduling," 1998.

[13] W. B. Arthur, J. H. Holland, B. LeBaron, R. Palmer, and P. Tayler, "Asset Pricing under Endogenous Expectations in an Artificial Stock Market," in *Preprint from: The Economy as an Evolving Complex System II, Santa Fe Institute Studies in the Sciences of Complexity, Vol. XXVII, Addison-Wesley*, 1996. [Online]. Available: http://www.econ.iastate.edu/tesfatsi/ahlpt96.pdf

[14] J. O. Kephart, J. E. Hanson, and A. R. Greenwald, "Dynamic pricing by software agents," *Computer Networks (Amsterdam, Netherlands: 1999)*, vol. 32, no. 6, pp. 731–752, 2000.

[15] (2007) Web services agreement specification. [Online]. Available: http://www.ogf.org/documents/GFD.107.pdf

[16] G. J. Holzmann, *The Spin Model Checker*. Amsterdam: Addison-Wesley Longman, 2003.

[17] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng, "Bandera: extracting finite-state Models from Java Source Code," in *International Conference on Software Engineering*, 2000, pp. 439–448.

[18] K. Havelund and T. Pressburger, "Model Checking Java Programs using Java PathFinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, Apr. 2000.

[19] K. Havelund, M. R. Lowry, and J. Penix, "Formal Analysis of a Space-Craft Controller Using SPIN." *IEEE Trans. Software Eng.*, vol. 27, no. 8, pp. 749–765, 2001.

[20] D. Kuo and M. McKeown, “Advance Reservation and Co-Allocation Protocol For Grid Computing,” University of Manchester, Tech. Rep., 2005.

APPENDIX A
THE PROMELA MODEL OF CALANA

We provide the full PROMELA model we used in this appendix. It is structured as follows:

1) Global variables and helper routines are defined in the first section. Beside adjustable parameters, we define the message types, state variables and other globally used variables. Starting with line 76, we define some functions to encapsulate the transitions of the job finite state machine.
2) The workloadmanager process definition starts in line 200. The process is structured as a state machine with labels corresponding to the states of the process.
3) In line 270, the broker process definition starts. Please note that we choose the best bid randomly, see lines 290-305. There is no simplification of the selection process, so no agent will be prefered.
4) The provider definition starts in line 442. Note that each provider process is given an ID which is used to determine which channels to use. For each ID, there is a corresponding igor.
5) The igor is defined in line 590. We do not model the startup and shutdown of the job explicitly, so this process definition is a very simple state machine.
6) Finally, in line 648, we give the init process. This process is started by SPIN at the very beginning of the verification process. Here, we start all other processes and wait for their completion. After the other processes are finished, we investigate some state variables.

Note that the definition of the LTL formulae and the corresponding Buechli automata is stored in separate files. The inline comments provide more information on the model.

```
1  /* Number of attached agents */
2  #define N 2
3  /* Number of messages that can be in a channel simultaneously */
4  #define chanSize 2
5
6  /* Messages */
7  mtype={BookingReq, AuctionBid, AuctionAccept, AuctionDeny,
8  Booked, Confirm, Close, CloseAck, BookingRejected, BookingRejectedAck,
9  ProviderCancel, ProviderCancelAck, WLMCancel, WLMCancelAck}
10
11 /* Communication to and from the Igors */
12 mtype={ready, shutdown, reserve, removeReservation, startJob,
13 startJob_success, startJob_failure,
14 running, terminateJob_success, terminateJob_failure,
15 abortJob, terminate, resourceFailure, }
16
17 /* The agreed outcomes */
18 #define UndefinedOutcome 0
19 #define BookingRejectedOutcome 1
20 #define SuccessfulExecutionOutcome 2
21 #define FaultedExecutionOutcome 3
22 #define WLMCanceledOutcome 4
23 #define ProviderCanceledOutcome 5
24
25 /* Define job states: JobState_Substate */
26 #define JobPending_New 1
27 #define JobPending_Negotiating 2
28 #define JobPending_Booked 3
29 #define JobPending_Confirmed 4
30 #define JobRunning_StageIn 5
31 #define JobRunning_Executing 6
32 #define JobRunning_StageOut 7
33 #define JobFinished_Closing 8
34 #define JobFinished_Closed 9
35 #define JobFailed 10
36 #define JobTerminated 11
37
38 /* Define the Job State machine as a sequence of inlines
39  * which check whether a transition is allowed. They all take the
40  * JobStateVariable (WLM, Broker, Provider) to chanke and apply
```

```
41  * assert statements before changing the state.
42  */
43 #define isJobFSMEndstate(state) (state == JobFinished_Closed \
44   || state == JobFailed || state == JobTerminated)
45
46 /* Predicate to check whether the given ID has won the auction. */
47 #define isWinner(id) (id==winnerID)
48
49 /* Global variables */
50 byte WLMOutcome, BrokerOutcome, ProviderOutcome = UndefinedOutcome;
51 byte WLMJobFSM, BrokerJobFSM = JobPending_New;
52 byte providerJobFSMCollection[N] = 0;
53 byte wait=0;
54 bool isConfirmed = false;
55 bool wlmCanceled = false;
56 bool providerCanceled = false;
57
58 /*A number of channels to and from the provider */
59 chan a2b[N] = [chanSize] of {mtype};
60 chan b2a[N] = [chanSize] of {mtype};
61
62 /*A number of channels from the igor to the provider and back */
63 chan p2i[N] = [chanSize] of {mtype};
64 chan i2p[N] = [chanSize] of {mtype};
65
66 /*A number of channels from the wlm to the igor */
67 chan c2i[N] = [chanSize] of {mtype};
68
69 /* Store the ID of the auction winner - we need to share this between
70  * wlm and broker. */
71 short winnerID=-1;
72
73 /* A global variable used as index in loops. don't use elsewhere, be
74  * careful with initialization. */
75 byte i;
76
77 inline startNegotiation(jobFSM) {
78   d_step {
79     assert(jobFSM == JobPending_New);
80     jobFSM = JobPending_Negotiating;
81   };
82 }
83
84 inline booked(jobFSM) {
85   d_step {
86     assert(jobFSM == JobPending_Negotiating);
87     jobFSM = JobPending_Booked;
88   }
89 }
90
91 inline confirm(jobFSM) {
92   d_step {
93     assert(jobFSM == JobPending_Booked);
94     jobFSM = JobPending_Confirmed;
95   }
96 }
97
98 inline runJob_StageIn(jobFSM) {
99   d_step {
100     assert(jobFSM == JobPending_Confirmed);
101     jobFSM = JobRunning_StageIn;
102   }
103 }
104
105 inline runJob_execute(jobFSM) {
106   d_step {
107     assert(jobFSM == JobRunning_StageIn);
```

```
108      jobFSM = JobRunning_Executing;
109    }
110  }
111
112  inline runJob_StageOut(jobFSM) {
113    d_step {
114      assert(jobFSM == JobRunning_Executing);
115      jobFSM = JobRunning_StageOut;
116    }
117  }
118
119  inline closeJob_closing(jobFSM) {
120    d_step {
121      assert(jobFSM == JobRunning_StageOut);
122      jobFSM = JobFinished_Closing;
123    }
124  }
125
126  inline closeJob_closed(jobFSM) {
127    d_step {
128      assert(jobFSM == JobFinished_Closing);
129      jobFSM = JobFinished_Closed;
130    }
131  }
132
133  inline terminateJob(jobFSM) {
134    d_step {
135      /* We can terminate a job as long as it is not JobFinished_Closing
136       * or JobFinished_Close or JobFailed. */
137      assert(jobFSM != JobFinished_Closing && jobFSM != JobFinished_Closed
138        && (!isJobFSMEndstate(jobFSM)) );
139      jobFSM = JobTerminated;
140    }
141  }
142
143  inline failJob(jobFSM) {
144    d_step {
145      /* A job can fail only if the job is in a running substate. */
146      assert(jobFSM == JobRunning_StageIn || jobFSM == JobRunning_Executing
147        || JobRunning_StageOut);
148      jobFSM = JobFailed;
149    }
150  }
151
152  /* Macro that sends a message to all providers */
153  inline broadcast2P(msg) {
154    d_step {
155      i = 0;
156      do
157      :: (i<N) -> b2a[i]!msg; i=i+1;
158      :: else -> break;
159      od;
160      i = 0;
161    }
162  }
163
164  inline broadcast2PExcept(msg, exceptID) {
165    d_step {
166      i=0;
167      do
168      :: ((i<N) && ! (i==exceptID)) -> b2a[i]!msg; i=i+1;
169      :: (i==exceptID) -> i=i+1;
170      :: else -> break;
171      od;
172      i=0;
173    }
174  }
```

```
175 
176 /* Removes all messages from the given channel */
177 inline cleanup_channel(current) {
178     do
179     :: a2b[i]?ProviderCancel -> b2a[i]!ProviderCancelAck;
180     :: empty(a2b[i]) -> break;
181     :: else -> a2b[i]?_ ;
182     od;
183 }
184 
185 /* Remove all messages from the provider-to-broker channels */
186 inline cleanup_a2b_channels() {
187   d_step {
188     i=0;
189     do
190     :: (i<N) -> cleanup_channel(i); i=i+1;
191     :: else -> break;
192     od;
193     i=0;
194   }
195 }
196 
197 /*##################################################################*/
198 
199 
200 proctype wlm (chan out, in) {
201   WLMJobFSM = JobPending_New;
202 StInitial:
203   startNegotiation(WLMJobFSM);
204   out!BookingReq -> goto StBooking;
205 StBooking:
206   if
207   :: in?Booked -> booked(WLMJobFSM); goto StBooked;
208   :: in?BookingRejected -> goto StRejecting;
209   :: out!WLMCancel -> goto StWLMCancelling;
210   fi;
211 StBooked:
212   if
213   :: in?ProviderCancel ->
214      terminateJob(WLMJobFSM); goto StProviderCancelling;
215   :: out!Confirm -> confirm(WLMJobFSM); goto StConfirmed;
216   :: out!WLMCancel -> goto StWLMCancelling;
217   fi;
218 StConfirmed:
219   /* Run the job! */
220   c2i[winnerID]!startJob;
221   runJob_StageIn(WLMJobFSM);
222   runJob_execute(WLMJobFSM);
223   runJob_StageOut(WLMJobFSM);
224   if
225   :: in?Close -> goto StClosing;
226   :: in?ProviderCancel -> goto StProviderCancelling;
227   :: out!WLMCancel -> goto StWLMCancelling;
228   fi;
229 StClosing:
230   closeJob_closing(WLMJobFSM);
231   out!CloseAck;
232   WLMOutcome = SuccessfulExecutionOutcome;
233   closeJob_closed(WLMJobFSM);
234   goto StTerminated;
235 StWLMCancelling:
236   if
237   :: in?WLMCancelAck ->
238      WLMOutcome=WLMCanceledOutcome;
239      terminateJob(WLMJobFSM);
240      goto StTerminated;
241   :: in?Booked -> goto StWLMCancelling;
```

```
242      /*The Booking was confirmed, but WLM cancelled
243             in the meantime*/
244    :: in?ProviderCancel -> goto StProviderCancelling;
245      /* If the provider cancels, it is his responsibility! */
246    :: in?Close -> goto StClosing;
247      /* It is always possible that the job has just finished */
248    :: in?BookingRejected -> goto StRejecting;
249      /* There is nothing to cancel, so we're fine... */
250    fi;
251  StProviderCancelling:
252    out!ProviderCancelAck;
253    WLMOutcome=ProviderCanceledOutcome;
254    failJob(WLMJobFSM);
255    goto StTerminated;
256  StRejecting:
257    out!BookingRejectedAck;
258    WLMOutcome = BookingRejectedOutcome;
259    terminateJob(WLMJobFSM);
260    goto StTerminated;
261  StTerminated:
262    goto End;
263  End:      atomic {wait = wait + 1;}
264  }
265
266
267  /*################################################################*/
268
269
270  /* This proctype simulates the Broker and the WLM of the resource */
271  proctype broker(chan clientOut, clientIn) {
272    BrokerJobFSM = JobPending_New;
273    short counter=0;
274    isConfirmed = false;
275  StInitial:
276    if
277    :: clientIn?BookingReq -> goto StStartAuction;
278    fi;
279  StStartAuction:
280    startNegotiation(BrokerJobFSM);
281    broadcast2P(BookingReq);
282    goto StAuctionRunning;
283  StAuctionRunning:
284    do
285    /* either we read a message */
286    :: (counter < N) ->
287      do
288       /* For all message channels , try to read a message */
289      :: (counter < N) ->
290        if
291        ::a2b[counter]?AuctionBid ->
292          if
293          :: winnerID == -1 -> winnerID = counter; /*first bid*/
294          :: else ->
295            /* This is one of the following bids - 50% chance of
296             * being best bid!
297             */
298            if
299            /* this is the best bid */
300            :: skip -> winnerID = counter;
301            :: skip; /* its not... */
302            fi;
303          fi;
304        :: skip /*else -> skip; /* skip is redundant here.*/
305        fi; counter = counter + 1;
306      :: else -> break;
307      od; /* Tried to read from all channels */
308    /* or we end the auction.      */
```

```
309     :: else -> counter = 0; break;
310     od;
311   progress:    goto StAuctionEnd;
312   StAuctionEnd:
313     /* Do we have a bid? */
314     if
315     :: (winnerID == -1) -> goto StNoBid;
316     :: else -> skip;
317     fi;
318     /* We broadcast the result of the auction. However, for */
319     /*    the state, it is not necessary to evaluate the contents */
320     /*    of the Broadcast message. */
321     b2a[winnerID]!AuctionAccept;
322     broadcast2PExcept(AuctionDeny, winnerID);
323     if
324     :: a2b[winnerID]?Booked -> goto StBooking;
325     :: a2b[winnerID]?BookingRejected -> goto StProviderRejecting;
326     :: clientIn?WLMCancel -> goto StWLMCancelling;
327     fi;
328   StNoBid:
329     if
330     :: clientOut!BookingRejected ->
331       goto StRejectingWaitForClient;
332     fi;
333   StBooking:
334     if
335     :: clientOut!Booked -> booked(BrokerJobFSM); goto StBooked;
336     :: clientIn?WLMCancel -> goto StWLMCancelling;
337     fi;
338   StBooked:
339     if
340     :: clientIn?WLMCancel -> goto StWLMCancelling;
341     :: a2b[winnerID]?ProviderCancel -> goto StProviderCancelling;
342     :: clientIn?Confirm -> goto StConfirmed;
343     fi;
344   StConfirmed:
345     isConfirmed = true;
346     confirm(BrokerJobFSM);
347     b2a[winnerID]!Confirm;
348     /* The job will be executed by the other entities here... */
349     runJob_StageIn(BrokerJobFSM);
350     runJob_execute(BrokerJobFSM);
351     runJob_StageOut(BrokerJobFSM);
352     if
353     :: a2b[winnerID]?Close ->
354       closeJob_closing(BrokerJobFSM); goto StClosing;
355     :: a2b[winnerID]?ProviderCancel -> goto StProviderCancelling;
356     :: clientIn?WLMCancel -> goto StWLMCancelling;
357     fi;
358   StClosing:
359     b2a[winnerID]!CloseAck;
360     clientOut!Close;
361     BrokerOutcome = SuccessfulExecutionOutcome;
362     closeJob_closed(BrokerJobFSM);
363     StClosingJumpback: if
364     :: clientIn?CloseAck -> goto StTerminated;
365     :: clientIn?WLMCancel -> goto StClosingJumpback;
366       /*WLM wants to cancel, but we have already completed the job*/
367     fi;
368     goto StTerminated;
369   StWLMCancelling:
370     wlmCanceled = true;
371     b2a[winnerID]!WLMCancel;
372     if
373     :: a2b[winnerID]?WLMCancelAck ->
374       BrokerOutcome=WLMCanceledOutcome;
375       terminateJob(BrokerJobFSM);
```

```
376       clientOut!WLMCancelAck;
377       goto StTerminated;
378     :: a2b[winnerID]?Booked -> goto StWLMCancelling;
379       /*The Booking was confirmed, but WLM cancelled
380             in the meantime*/
381     :: a2b[winnerID]?AuctionBid -> goto StWLMCancelling;
382       /* There is a bid in the channel, but we don't need
383             it any more */
384     :: a2b[winnerID]?ProviderCancel -> goto StProviderCancelling;
385       /* If the provider cancels, it is his responsibility! */
386     :: a2b[winnerID]?Close ->
387       closeJob_closing(BrokerJobFSM); goto StClosing;
388       /* It is always possible that the job has just finished */
389     :: a2b[winnerID]?BookingRejected -> goto StProviderRejecting;
390       /* There is nothing to cancel, so we're fine... */
391     fi;
392 StProviderCancelling:
393     providerCanceled = true;
394     b2a[winnerID]!ProviderCancelAck;
395     clientOut!ProviderCancel;
396     BrokerOutcome=ProviderCanceledOutcome;
397     failJob(BrokerJobFSM);
398     goto StProviderCancellingWait;
399 StProviderCancellingWait:
400     if
401     :: clientIn?ProviderCancelAck -> goto StTerminated;
402     :: clientIn?Confirm -> goto StProviderCancellingWait;
403     :: clientIn?WLMCancel -> goto StProviderCancellingWait;
404     fi;
405     goto StTerminated;
406 StProviderRejecting:
407     b2a[winnerID]!BookingRejectedAck;
408     clientOut!BookingRejected;
409     do
410     :: clientIn?BookingRejectedAck ->
411       BrokerOutcome = BookingRejectedOutcome;
412       break;
413     :: clientIn?WLMCancel -> skip;
414     od;
415     goto StTerminated;
416 StRejectingWaitForClient:
417     if
418     :: clientIn?BookingRejectedAck ->
419       BrokerOutcome = BookingRejectedOutcome;
420       terminateJob(BrokerJobFSM);
421       goto StRejectingWaitForProvider;
422     :: clientIn?WLMCancel -> goto StRejectingWaitForClient;
423     fi;
424     goto StTerminated;
425 StRejectingWaitForProvider:
426     broadcast2P(BookingRejected);
427     goto StTerminated;
428 StTerminated:
429       /* The decision of the broker is made. It may occur that we did not
430        * read some messages - read them now to keep the channel empty. */
431       cleanup_a2b_channels();
432       /* Same with client channel. */
433       cleanup_channel(clientIn);
434       goto end;
435 end:    atomic { wait = wait + 1;}
436 }
437
438
439 /*#################################################################*/
440
441
442 proctype provider(byte id) {
```

```
443    byte ProviderJobFSM = providerJobFSMCollection[id];
444    ProviderJobFSM = JobPending_New;
445    chan in = b2a[id];
446    chan out = a2b[id];
447    chan toIgor = p2i[id];
448    chan fromIgor = i2p[id];
449  StInitial:
450    fromIgor?ready;
451    if
452    :: in?BookingReq -> goto StAuctionRunning;
453    :: in?AuctionDeny -> goto StTerminated;
454    :: fromIgor?resourceFailure -> goto StTerminated;
455    :: fromIgor?shutdown -> goto StTerminated;
456    fi;
457  StAuctionRunning:
458    if
459    /* We choose to bid or not to bid */
460    :: skip ->
461      startNegotiation(ProviderJobFSM);
462      out!AuctionBid;
463      toIgor!reserve;
464      goto StBooking;
465    :: skip -> goto StTerminated;
466    fi;
467  StBooking:
468    if
469    :: in?AuctionAccept -> skip;
470    :: in?AuctionDeny -> toIgor!removeReservation; goto StTerminated;
471    :: fromIgor?resourceFailure ->
472      out!ProviderCancel;
473      toIgor!terminate;
474      goto StProviderCancelling;
475    :: in?WLMCancel -> goto StWLMCancelling;
476    :: in?BookingRejected -> goto StAuctionBookingRejecting;
477    /* Our bid may arrive to late: The auction might have timed out. */
478    /*:: timeout -> goto StTerminated; */
479    fi;
480    /* Now, we have the result of the auction... */
481    if
482    :: out!Booked -> booked(ProviderJobFSM); goto StBooked;
483    :: out!BookingRejected -> toIgor!removeReservation; goto StRejecting;
484    :: in?WLMCancel -> goto StWLMCancelling;   fi;
485  StBooked:
486    if
487    :: out!ProviderCancel -> toIgor!removeReservation; goto StProviderCancelling;
488    :: fromIgor?resourceFailure ->
489      out!ProviderCancel;
490      toIgor!terminate;
491      goto StProviderCancelling;
492    :: in?Confirm ->
493      confirm(ProviderJobFSM);
494      goto StConfirmed;
495    :: in?WLMCancel -> goto StWLMCancelling;
496    fi;
497  StConfirmed:
498    if
499    :: fromIgor?startJob_success -> goto StRunning;
500    :: fromIgor?startJob_failure ->
501      out!ProviderCancel;
502      toIgor!terminate;
503      goto StProviderCancelling;
504    :: in?WLMCancel -> goto StWLMCancelling;
505    fi;
506  StRunning:
507    runJob_StageIn(ProviderJobFSM);
508    runJob_execute(ProviderJobFSM);
509    runJob_StageOut(ProviderJobFSM);
```

```
510   if
511   :: fromIgor?terminateJob_success ->
512     out!Close;
513     closeJob_closing(ProviderJobFSM);
514     goto StClosing;
515   :: out!ProviderCancel -> toIgor!abortJob; goto StProviderCancelling;
516   :: fromIgor?resourceFailure ->
517     out!ProviderCancel;
518     toIgor!terminate;
519     goto StProviderCancelling;
520   :: in?WLMCancel -> goto StWLMCancelling;
521   fi;
522 StClosing:
523   if
524   :: in?CloseAck ->
525     ProviderOutcome = SuccessfulExecutionOutcome;
526     closeJob_closed(ProviderJobFSM);
527     goto StTerminated;
528   :: in?WLMCancel -> goto StClosing;
529     /*WLM wants to cancel, but we have already completed the job*/
530   fi;
531 StWLMCancelling:
532   toIgor!abortJob;
533   out!WLMCancelAck;
534   ProviderOutcome = WLMCanceledOutcome;
535   terminateJob(ProviderJobFSM);
536   goto StTerminated;
537 StRejecting:
538   if
539   :: in?BookingRejectedAck ->
540     ProviderOutcome = BookingRejectedOutcome;
541     terminateJob(ProviderJobFSM);
542     goto StTerminated;
543   :: in?WLMCancel -> goto StRejecting;
544   :: fromIgor?resourceFailure -> toIgor!terminate; goto StRejecting;
545   fi;
546 StAuctionBookingRejecting:
547   toIgor!abortJob;
548   out!BookingRejectedAck;
549   ProviderOutcome=BookingRejectedOutcome;
550   terminateJob(ProviderJobFSM);
551   goto StTerminated;
552 StProviderCancelling:
553   if
554   :: in?ProviderCancelAck ->
555     if
556     :: (isWinner(id)) ->
557       ProviderOutcome = ProviderCanceledOutcome;
558       failJob(ProviderJobFSM);
559     :: else -> skip;
560     fi;
561     goto StTerminated;
562   :: timeout ->
563     if
564     :: (isWinner(id)) ->
565       ProviderOutcome = ProviderCanceledOutcome;
566       failJob(ProviderJobFSM);
567     :: else -> skip;
568     fi;
569     goto StTerminated;
570   :: in?Confirm -> goto StProviderCancelling;
571   :: in?AuctionAccept -> goto StProviderCancelling;
572   :: in?AuctionDeny -> goto StProviderCancelling;
573   :: in?WLMCancel -> goto StProviderCancelling;
574   :: in?BookingRejected ->
575     out!BookingRejectedAck;
576     ProviderOutcome = BookingRejectedOutcome;
```

```
577      goto StTerminated;
578    fi;
579  StTerminated:
580    toIgor!terminate;
581    goto end;
582  end:
583    atomic {wait = wait + 1;}
584    skip;
585  }
586
587  /*###################################################################*/
588
589  proctype igor(byte id) {
590    chan toProvider=i2p[id];
591    chan fromProvider=p2i[id];
592    chan fromWLM=c2i[id];
593  StDown:
594    toProvider!ready;
595    goto StReadyFree;
596  StReadyFree:
597    if
598    :: fromProvider?reserve -> goto StReadyReserved;
599    :: fromProvider?terminate -> goto end;
600    :: toProvider!resourceFailure -> goto StFailure;
601    :: toProvider!shutdown ->
602          goto end;
603    fi;
604  StReadyReserved:
605    if
606    :: fromWLM?startJob -> goto StReadyStarting;
607    :: fromProvider?removeReservation -> goto StTerminated
608    :: toProvider!resourceFailure -> goto StFailure;
609    :: fromProvider?abortJob -> goto StReadyFree;
610    :: fromProvider?terminate -> goto end;
611    fi;
612  StReadyStarting:
613    if
614    /* usually, the igor will also send the successful start to */
615    /* its wlm, but we don't model this. */
616    :: toProvider!startJob_success -> goto StRunning;
617    :: toProvider!startJob_failure -> goto StReadyFree;
618    :: toProvider!resourceFailure -> goto StFailure;
619    fi;
620  StRunning:
621    /* Here, the implementation would start the job. */
622    /* toProvider!running; */
623    if
624    :: toProvider!terminateJob_success -> goto StReadyFree;
625    :: toProvider!terminateJob_failure -> goto StReadyFree;
626    :: toProvider!resourceFailure -> goto StFailure;
627    :: fromProvider?abortJob -> goto StReadyFree;
628    :: fromProvider?terminate -> goto end;
629    fi;
630  StTerminated:
631    if
632    :: fromProvider?abortJob -> goto StTerminated;
633  /*  :: fromProvider?failed -> goto StTerminated;*/
634    :: fromProvider?terminate -> goto end;
635    fi;
636  StFailure:
637    if
638    :: fromProvider?terminate -> goto end;
639    :: fromProvider?reserve -> goto StFailure;
640    :: fromProvider?removeReservation -> goto StFailure;
641    :: fromProvider?abortJob -> goto StFailure;
642    fi;
643  end:
```

```
644     skip;
645 }
646
647 init {
648   /* Set two channels for the broker - client connection*/
649   chan c2b = [chanSize] of {mtype};
650   chan b2c = [chanSize] of {mtype};
651
652   /* startup*/
653   atomic {
654     run wlm (c2b, b2c);
655     /* Start the Broker */
656     run broker(b2c, c2b);
657     /* Start the Providers */
658     byte j=0;
659     do
660     :: (j<N) -> run provider(j); run igor(j); j=j+1;
661     :: else -> break;
662     od;
663     j=0;
664   }
665
666   if
667   :: (wait == N+2) -> /* We have reached the end of all processes */
668     /* 1. We have the same outcome */
669     assert(WLMOutcome == BrokerOutcome);
670     if
671     :: (BrokerOutcome != BookingRejectedOutcome) ->
672       assert(BrokerOutcome == ProviderOutcome);
673     :: else -> assert(1==1);
674     fi;
675     /* 2. We have only specified outcomes */
676     assert (0 <= WLMOutcome <= 5);
677     assert (0 <= BrokerOutcome <= 5);
678     assert (0 <= ProviderOutcome <= 5);
679   fi
680 }
```

APPENDIX B
MODELLING A LTL CLAIM

Here, we demonstrate how SPIN checks whether an LTL formula holds for the given model. We want to check that the provider eventually reaches the same outcome as the broker. There is one exception to this: If the job was not scheduled, they may have different outcomes.

The corresponding LTL formula is:

$$\Box\left(\left(\neg \mathtt{BR}\right) \rightarrow \Diamond\left(\mathcal{P} = \mathcal{B}\right)\right)$$

The corresponding automaton is shown in the listing below. It has been created with the LTL property manager of XSPIN, a graphical frontend to SPIN.

First, we need to define the predicates used in the LTL formula. $\neg\mathtt{BR}$ simply compares the brokers outcome variable with the constant value `BookingRejectedOutcome`. A second predicate checks whether the provider's outcome value equals the broker's. The LTL formula can now refer to these predicates.

The LTL formula is negated and converted to the Buechli automaton below. The property holds if the combined execution of the Calana model and the automaton produce the automaton not to enter any acceptance cycle [6]. The acceptance cycle starts with the label `accept_S4` in the generated code below. It is straightforward to create other LTL claims. SPIN can be instructed to compose the model with an automaton as below and report any acceptance cycles, which means that the property is violated.

```
#define notBookingRejected      (BrokerOutcome != BookingRejectedOutcome)
#define providerEqBroker        (BrokerOutcome == ProviderOutcome)

        /*
         * Formula As Typed: [] ((notBookingRejected) -> (<> (providerEqBroker)))
         * The Never Claim Below Corresponds
         * To The Negated Formula !([] ((notBookingRejected) -> (<> (providerEqBroker))))
         * (formalizing violations of the original)
         */

never {    /* !([] ((notBookingRejected) -> (<> (providerEqBroker)))) */
T0_init:
        if
        :: (! ((providerEqBroker)) && (notBookingRejected)) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: (! ((providerEqBroker))) -> goto accept_S4
        fi;
}
```