Mapping of Software Model to Simulation Model for Performance Requirement Verification

Ronaldo Arias
Instituto de Pesquisas Espaciais (INPE), Aerospace Electronic Division, Onboard Data Handling Group, São José dos Campos, SP, Brazil,
ronaldo@dea.inpe.br

Celso Massaki Hirata
Instituto Tecnologico de Aeronautica (ITA), Department of Computing Science, São José dos Campos, SP, Brasil, hirata@ita.br

Keywords: Software performance requirement verification, model driven and performance simulation model**.**

Abstract
This paper describes the mapping of a software model to a simulation model in order to support the performance requirement verification. More specifically, we describe a mapping of a software model, based on UML Deployment, and State Machine diagrams annotated with performance information, to a simulation model that is specified in Activity Cycle Diagrams. The simulation model is translated to a simulation program so that verification of performance requirements can be made. The mapping is part of a framework based on the UML Profile for MARTE (Modeling and Analysis of Real-Time and Embedded Systems) employed for performance requirement verification of real time computers systems. An example is presented to show the feasibility of the mapping.

1. INTRODUCTION

In real time systems computation, the correctness of the system depends not only of the logical results of the computation but also the time the results are produced [Stankovic 1988]. In practice, the verification of performance requirements is made in the final stages of the software development cycle using test and code optimization techniques. Code optimization techniques are rather restrictive in terms of performance improvement. If the performance requirements are not met, a considerable effort is required, including the re-definition of the architecture, detailed design update, addition of computer resources, and even the re-definition of the requirements. The term verification is used in this work with the meaning of check the performance requirements in the software architecture model.

There is an understanding that the verification of performance requirements should be made as early as possible in the software development life cycle. The Software Performance Engineering methodology (SPE) [Smith and Williams 2002] is the first comprehensive approach used to integrate the performance analysis in all the software development processes [Balsamo at al., 2004].

There has been some research effort to address the verification of performance requirements in the RUP software development process [Paes and Hirata, 2008], however the approach specifies only what and when the tasks should be made, not how they are accomplished. The Rational Unified Process [IBM 2010] specifies activities and tasks that should be accomplished to produce the software artifacts. The activities and tasks require the collaboration of different roles. As pointed out in [Paes and Hirata 2008], the verification of performance requirements in the software development process involves the collaboration of simulation analysts and software specialists. Verification also entails iterations in the software development life cycle, so that feedbacks between activities are required.

The Object Management Group has defined a set of standards, named UML Profiles in order to aid the process of modeling and analysis of non-functional requirements in the initial phase of the software development process. This paper focuses on the UML Profile for MARTE (in short MARTE) [OMG 2009]. It replaces the UML Profile for Schedulability, Performance and Time [OMG 2005]. MARTE adds capabilities to UML for model-driven development of real time and embedded systems. It provides support for specification, design, verification and validation activities. It also extends the software-driven analysis to system-driven (hardware and software) analysis. This characteristic is used to model the hardware environment in which the software shall be executed.

The general performance analysis cycle based on models is composed of four phases: software model specification, performance model generation, performance model analysis, and generation of feedback on the software model. Temporal information is required for the software model in order to generate the performance model. One important factor to the performance analysis is the ability to automatically generate the performance model from the software model.

Model-driven engineering (MDE) is a software development methodology which focuses on creating

models, or abstractions, closer to some particular domain concepts rather than computing concepts. It is intended to increase productivity by increasing compatibility between systems, simplifying the process of design, and promoting communication between individuals (roles) working on the system. The Unified Modeling Language (UML), with its supporting tools, is the most used language to describe models in MDE. [Kordon et al., 2008] claim that MDE is incomplete and there is a need to include verification. In their view, the verifications, including of performance requirements, should be made in the early phases using MDE.

Software performance analysis can be used to compare design alternatives, check physical deployments and scalability, and identify system bottlenecks [Marzolla 2004]. Performance analysis is accomplished by using analytical models [Di Marco, 2005] and simulation [Balsamo and Marzolla 2003]. Simulation requires a considerable effort for modeling and experimentation. Nonetheless, it provides modeling flexibility and can be refined to provide accurate results.

This paper describes a mapping of a software model to a simulation model in order to enable the verification of performance requirements in an iterative manner. The software model is represented by UML Deployment and State Machine diagrams annotated with performance information, according to the MARTE profile. The simulation model is represented by Activity Cycle Diagrams (ACD), which is a Timed-Petri nets variant. ACD provide a simple graphical representation to model discrete event simulation systems. The ACD model is translated to a simulation model, which is used for verification of performance requirements. We claim that the usage of ACD models provides easier iteration in the verification of performance requirements and generation of feedback on the software model since the simulation analyst is more familiar with those models rather than software models.

Section 2 presents a short description of UML State Machine diagrams and ACDs used in the mapping process. Section 3 describes the related work in the area of verification of software performance requirements applied to real time computer systems. Section 4 describes the mapping of UML diagrams to a performance simulation model, through an example. The conclusions and final remarks are presented in Section 5.

2. BACKGROUND

This Section describes some characteristics of UML state machine diagrams and ACDs applied in the mapping of software model to simulation model. The mapping algorithm utilizes the information from the *states* and *transition flows* of the UML state machine diagrams. A *state* is a situation in which some invariant condition holds. The condition may represent a static situation such as an object waiting for some external event, or a dynamic condition, such as the process of performing some behavior, where the model element enters the state when the behavior starts and leaves it as soon as the behavior is completed [OMG 2004]. A *transition flow* represents the response of a state machine to events. The transition is composed by an event trigger, a guard and an activity, as described as follows: *Event Trigger [Guard] / Activity*.

The *event trigger* indicates the occurrence of a specific event. A *guard* (optional) is a Boolean condition that provides control over the transition. The guard is evaluated when the state machine dispatches an event occurrence. If the guard is true, the transition may be enabled, otherwise, it is disabled. A transition with a guard happens only if the triggering event occurs and the guard is evaluated to true. The *activity* (optional) may be an action sequence, including actions that explicitly generate events, such as sending signals or invoking operations. The activity is executed if the transition fires. The output condition of a state can be an external event trigger or an internal event *end of processing*.

Activity Cycle Diagrams [Pidd 1992] provide a graphical representation to model discrete event simulation systems. Its power is based on its simplicity. It identifies and describes graphically the behavior of each entity in the system and shows their iterations. The original proposal makes use of only two states: active and dead. The active state, represented by a rectangle, usually involves the co-operation of different entity classes. The duration of an active state is always determined in advance, usually by using a probability distribution if the simulation model is stochastic. The dead state, represented by a circle, involves no co-operation between different entity classes and is generally a state in which the entity waits the occurrence of same event, i.e., wait for same resource. Dead states can be thought as wait queues. Dead states and queues are dealt equally in this work. The time that an entity spends in a dead state cannot be defined in advance.

The life cycle of each entity class is represented by a sequence of active and dead states. The transition from state to state is represented by arrows. In general, these states are drawn as alternate dead and active states. The complete diagram consists of a combination of all the individual entity cycles.

The entities can be temporary or permanent. The temporary entity enters and leaves the system according to its life cycle and the permanent entities stay in the system and can be thought as resources that provide service to the temporary entities. In order to improve the representation of more complex systems, which is required for real time system development, this work uses an extended ACD notation [Hirata and Paul 1996]. The extended ACD corresponds to a richer set of active states and it is intended for object-oriented implementation. It includes states for generation (named Generate), internal transformation (named Activity), composition and decomposition (named

Router), interruption (named Interrupt), and destruction (named Destroy) of entities and of activities.

3. RELATED WORK

This section describes the related work in the area of verification of software performance requirements.

[Balsamo et al., 2004] present a review in the field of model-based software performance prediction. They describe a set of approaches that propose the use of performance models to characterize the quantitative behavior of software systems during all the phases of the software development life cycle. The main points emphasized in the study are:

- Lack of software performance requirement validation is mostly due to the knowledge gap between software engineers and quality assurance experts, and the short time to market rather than due to foundational issues;
- There is no approach, which is fully supported by automated tools, and, at the same time, there is no approach that does not provide or foresee some kind of automatic support.
- Most of the approaches make use of UML or UML-like formalisms to describe behavioral models;
- Queuing Networks are the preferred performance models;
- Few approaches provide feedback information. Methods based on simulation techniques can provide more easily feedbacks because they can have a direct correspondence between the software specification abstraction level and the performance model evaluation results;
- Complexity problems related to the generation of the performance model can be addressed by using simulation techniques. Analytical methods can provoke the problem of space state explosion.

[Arief and Speirs 2000] present a simulation framework (Simulation Modeling Language - SimML) used to automatically generate a process-oriented simulation program in Java from an UML model. This framework is used to predict the system performance from an UML design. The UML model is composed of class and sequence diagrams. An XML file is used to store the design and the simulation data, and it is generated directly from the UML diagrams. The performance simulation tool uses the XML file to generate the simulation program. There is no automatic feedback to the software model.

[Balsamo and Marzola 2003] propose an approach for software performance modeling, where the simulation model is derived from the software architecture represented by Use Case, Activity and Deployment UML diagrams, complemented with additional performance information based on a sub-set of the OMG standard UML Profile for Schedulability, Performance and Time Specification. The approach uses a software tool, named UML-PSI, to generate automatically a process oriented simulation model. It also provides a simple feedback to the software model. An XMI (XML Metadata Interchange) file is the interface between the software and performance model.

Different from previous solutions, our approach combines UML State Machine and Deployment diagrams annotated with performance information according to MARTE, and maps them to an intermediary performance simulation model in ACD, which is translated in a simulation program. With these models the software and the simulation specialists can interact with each other in an iterative process of modelling and performance verification until a design solution is reached.

We claim that the ACD model can help in the process of software performance analysis. It reduces the problem related to the technological gap between the software and the simulation specialists [Balsamo at al., 2004, Di Marco 2005, and Sancho at al., 2005]. The simulation specialists is more used to concepts of verification, such as bottlenecks and hot spots, of his/her domain, so he/she can exploit better the possibilities to improve the simulation (ACD) model. Similarly the software specialist is more concerned and familiar with design concepts to build the software (UML) model. In our approach, both specialists can work on their own domains and interact with each other on compatible models.

4. MAPPING FROM SOFTWARE MODEL TO SIMULATION MODEL

This section describes the mapping of a software model based on UML diagrams to an ACD model. The mapping is employed in the simulation framework for verification of software performance requirements of real time computer systems illustrated in Figure 1. In the framework, the analysis of simulation results can lead to simulation model changes, which in turn can lead to software model changes.

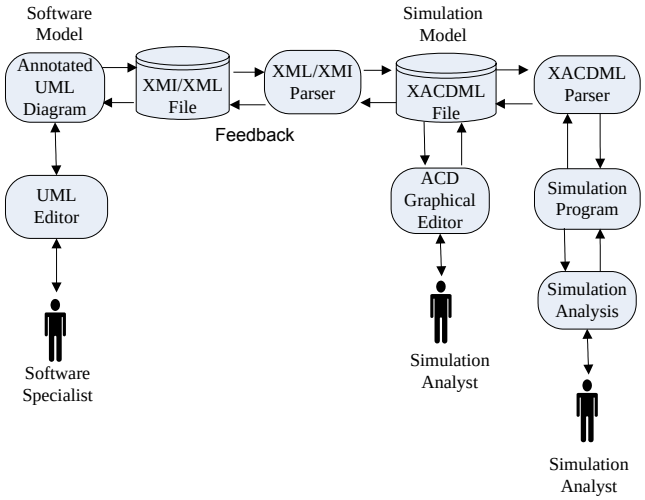

Figure 1. Framework for Software Performance Analysis

The software model is represented by UML Deployment and State Machine diagrams. The deployment diagram is

used to identify the physical resources/nodes in which the computations take place. The state machine diagrams are used to describe the behavior of the system entities and their iterations. Each entity is represented by a state machine diagram. The UML diagram model is annotated with performance information according to the MARTE Profile. An UML editor tool generates the corresponding XMI file, a parser program converts it to an ACD model and another parser translates the ACD model to the simulation program. The ACD model is described by an XML for ACD, called XACDML [Gil and Hirata 2003]. XACDML is designed to provide interoperability between development tools used for specification, design, verification, code generation, and performance analysis.

Our approach is presented through the example of a web-based video-streaming application derived from [OMG 2005]. The software model is represented by the deployment diagram, as shown in Figure 2, and the state machine diagrams, as shown in Figures 3, 4 and 5.

In the example, a user at a Client Workstation requests a video to the centralized remote Web Server. Based on the request, the Web Server chooses an appropriate Video Server, which initiates a Video Player on the user's site and then sends it a stream of video frames. In what follows we list the main parameters of this example.

- A client request a new video with an exponential distribution with mean of 6 seconds (workload intensity) as represented by the *SelectService* state in Figure 5;
- The time to send/receive the Initiation Transmission message (*InitTxMsg*) and the Confirmation message (*ConfMsg*) from Web Server to Video Server has an exponential distribution with mean of 100 ms as illustrated in the *VideoReqMsg* transition in Figure 3;
- The video frame transmission/reception time has an exponential distribution with mean of 200 ms as represented by the *ReceiveFrame* state in Figure 5 and the *SendFrame* state in Figure 4;
- The Client Workstation processing demand per frame has a uniform distribution with minimum of 175 ms and maximum of 225 ms. The frames are received and stored in a video interface buffer to be displayed as represented by the *ProcessFrame* state in Figure 5;
- The time to send/receive the Terminate Player message (*TerminatePlayerMsg*) from the Video Server to the Client Workstation has an exponential distribution with mean of 100 ms as indicated by the *SendTerminate* transition in Figure 4;
- Each video is composed of N frames. The time to show each video frame on the video is 1 second;

The UML stereotypes, tagged values, and constraints extensibility mechanisms are used to annotate the performance information in the UML diagrams. The "<<PAStep>>" stereotype and the "{open (interArrTime (Exp, 6, s)}" tagged value of Figure 5 are examples of performance information added to software model. The "<<PAStep>>" stereotype indicates a basic sequential execution on a host processor and the "{open (interArrTime (Exp, 6, s)}" tagged value is associated with the video request workload. It indicates the time between successive client video requests.

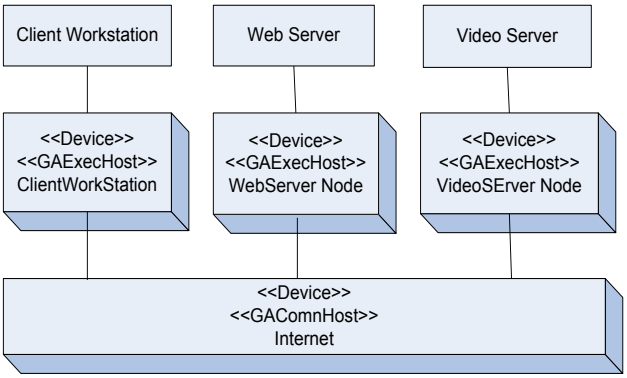

Figure 2. Web-Based Video Deployment Diagram

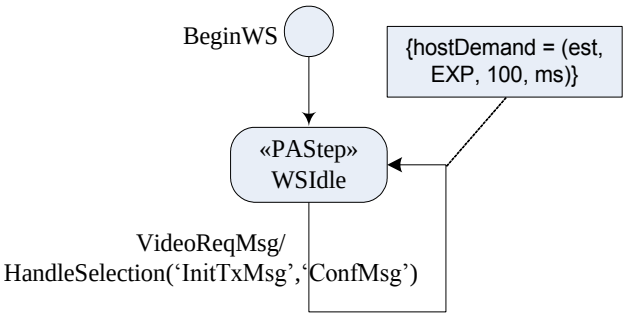

Figure 3. Web Server State Machine Diagram

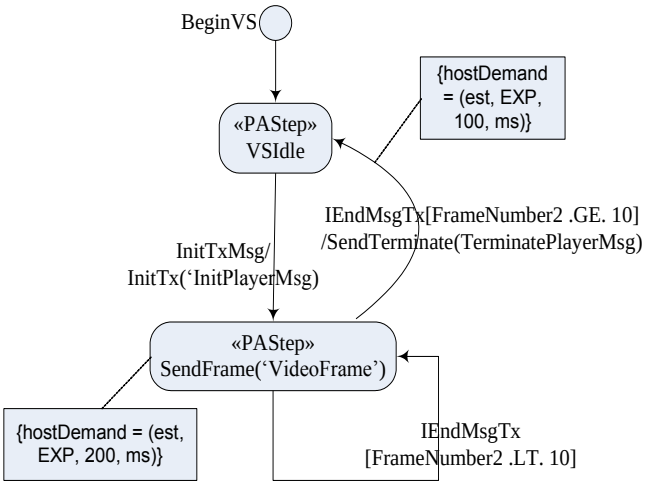

Figure 4. Video Server State Machine Diagram

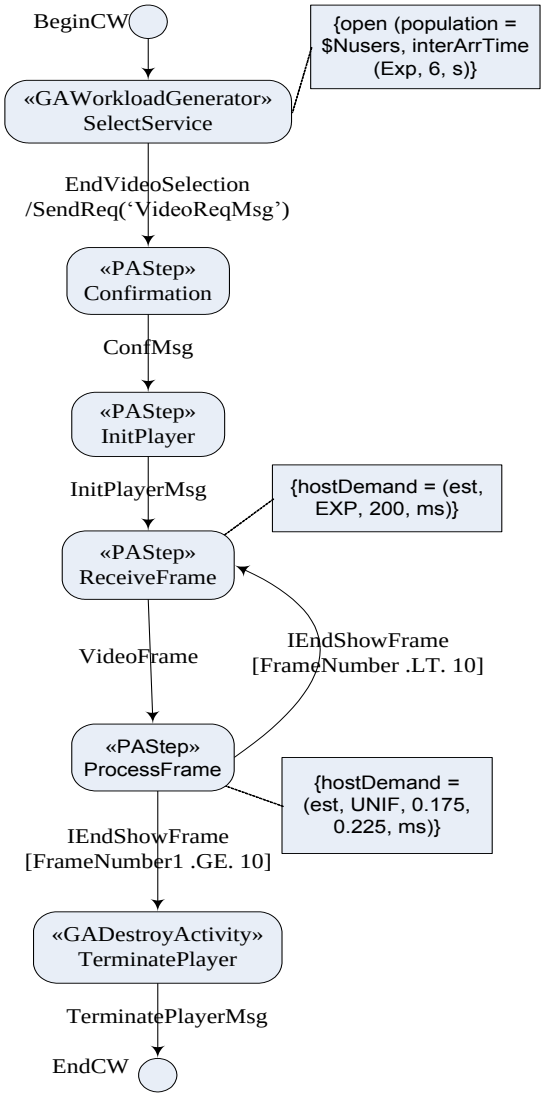

Figure 5. Client Workstation State Machine Diagram

4.1. Mapping of UML Diagrams to ACD

This section describes the mapping of UML models annotated with performance information to ACD models.

A commercial UML modeling tool is used to add the performance and simulation information to the UML diagrams, and to generate an XMI file. We implement a parser to analyze the XMI file and convert it to an XACDML file. The parser shows the feasibility of the automatic mapping.

The mapping starts with the identification of the ACD entities (permanent or temporary). The deployment diagram is used to identify the physical resources/nodes on which the computations take place. A resource represents a physically or a logically persistent entity that offers one or more services [OMG 2004]. For computer system modeling, CPU is an example of active resource, and printer, network and disk are examples of passive resource.

As described in Figures 3, 4 and 5, each entity is represented by an UML state machine diagram, which is mapped to an individual ACD.

The main guidelines and the algorithm for the mapping of the state machines and the deployment diagram to the individual ACD are listed below.

- The deployment diagram is used to identify the physical resources in which the computations take place (ACD entities).
- For each UML state in the state machine diagram, a pair of dead and active state in the ACD is generated, except for the initial and final states.
- The first state of a temporary entity shall be represented by the *Generate* active state in the ACD, and it shall describe the entity arrival information. The last state of a temporary entity shall be represented by the *Destroy* active state in the ACD.
- The initial UML state of a permanent entity shall be converted to a Generate active state, which indicates the creation of a fixed number of permanent entities;
- According to the attributes of the UML state, the ACD dead or active state can be made dummy. An UML state with no processing associated generates a dummy ACD active state, and an UML state with no external output trigger generates a dummy dead state. In this case, the output condition is end of internal behavior processing, where the entity enters the state when the behavior commences and leaves it as soon as the behavior is completed.
- The UML state output condition shall be associated with the output condition of the dead state in the ACD.
- Active and dead states dummies are included in the ACD essentially to maintain the alternating sequence of dead and active states.
- UML transitions with an activity associated also generate a pair of dead and active states in the ACD. The dead state is always dummy.

```
- UML to Individual ACD Mapping Algorithm

for (each UML State Machine) {
  for (all states) do
    if (temporary entity)
       then if (initial state)
                then {Create a Generate active state;
                      Check arrival time rate;
                      If (there is processing associated
                            with the output transition flow)
                         then Create a dummy queue and an
                              active state;
                     }
                else if (final state)
                       then {if (output condition is external event)
                                then Create a queue and
                                     a destroy active state;
                             else Create a dummy queue and
                                     a destroy active state;
                           Check for performance information;
```

```
                        }
                    else // neither initial nor final state
                        Create dead and active state ();
    else {// permanent entity
        if (initial state)
            then Create a Generate active state;
            else if (idle state)
                then {Create a resource queue;
                    for (all state transitions) do
                        if (there is processing associated)
                            then {Create an dummy dead and
                                an active state;
                            for (all external events) do
                                Create a queue and
                                an active state ();
                            }
                    Check for performance information
                    }
                else // not idle state
                    Create dead and active state ()
        }
    Update dead and active state links;
}
```

```
- Create dead and active state () Algorithm
{
  if (output condition is an external event)
     then Create an event queue
     else Create a dummy queue
  if (there is processing associated with the state)
     then Create an active state
     else Create a dummy active state;
  Check for performance information and update links;
  for (all state transitions) do {
    if (there is an activity associated with the transition)
       then for all (external events associated with the transition)
            do { Create a dummy queue and an active state;
                 Check for performance information;
                 Update links;   }
}
```

Figure 6 describes the individual ACD for the temporary *ClientWorkstation* entity. The Generate active state *GSelectService* defines the entity arrive rate. The *SendReq* state requests a video to the WebServer (indicated by the message denominated *VideoReqMsg*). Then the entity waits from the WebServer a *ConfMsg* message indicating the reception of the video request and the message *InitPlayerMsg* from the *VideoServer* indicating the beginning of the video transmission. The *VideoServer* starts the video frame transmission to the *ClientWorkstation* (*VideoFrame* message). At the end the *VideoServer* sends a message *TerminatePlayerMsg* to the *ClientWorkstation* indicating the final of the video transmission. Finally the *ClientWorkstation* entity is destroyed by the destroy active state *DTerminatePlayerD*.

The following convention is adopted to assign the names of the dead and active ACD states: the dead state name starts with the character "Q", the active states Generate, Router and Destroy start respectively with the characters "G", "R" and "D", and the dummy dead and active state name end with the letter "D".

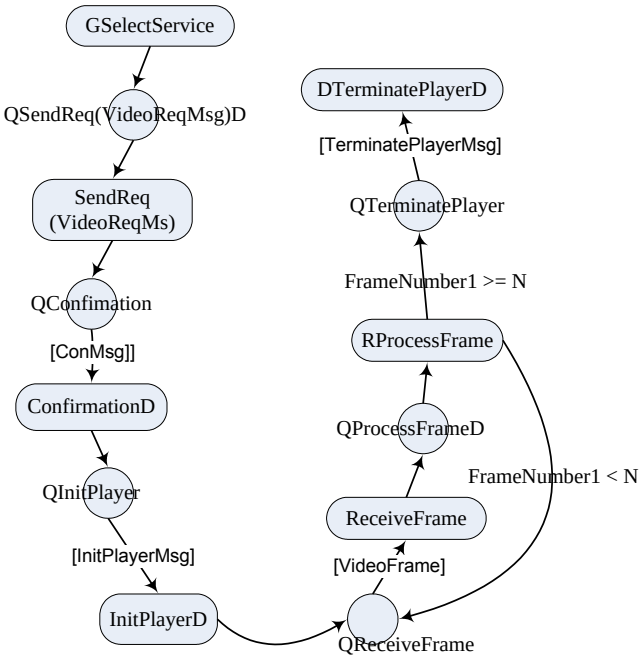

Figure 6 – Client Workstation Entity ACD

After the generation of the individual ACDs, they are integrated in a global ACD. In what follows we list the main steps to generate the global ACD.

- Assemble a table that indicates all the external events for all the entities. In our computer system, the message exchange shall be dealt as a synchronization/communication external event. The producer shall be the message sender and the consumer shall be the message receiver.
- For each message exchange event between two entities, create a cooperative active state. The individual ACD indicate which entity sends and which entity receives the message. For example, the active state *SendReq/HandleSelection* of Figure 7 is an example of cooperation active state. It was created by the interaction of the *SendReq(VideoReqMsg)* active state of *ClientWorkstation* entity and *HandleSelection* active state of *WebServer* entity.
- Insert in the global ACD the activities that do not have communication with other entities in the system.
- The active state that interacts with more than one entity can be broken in more than one cooperative active state in the global ACD.
- Update the links of the global ACD according to the individual ACD.

Figure 7 depicts the global ACD generated from the individual ACDs.

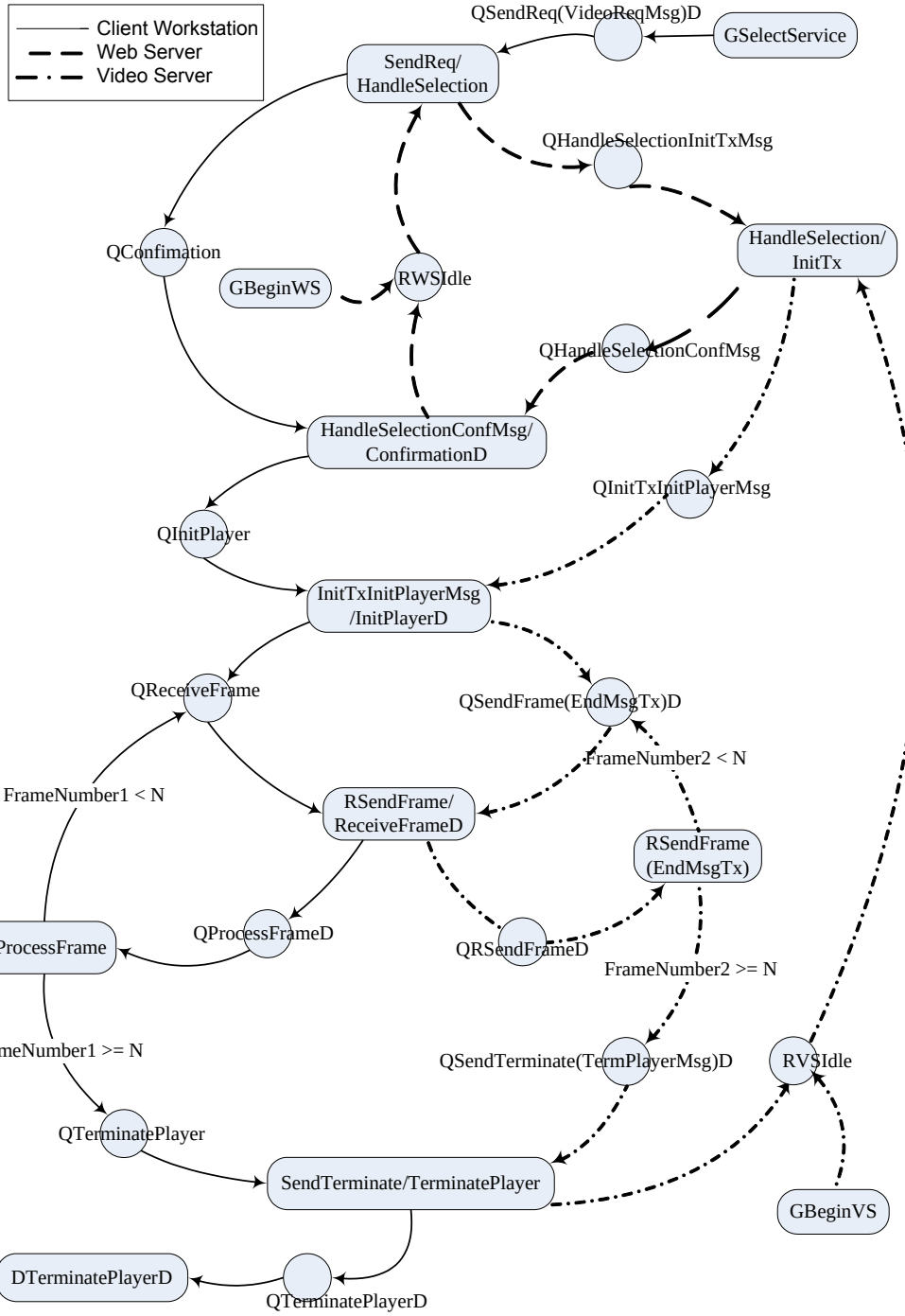

Figure 7. Web Based Video Application ACD

The system entities are identified by different types of lines. The *WebServer* entity waits for a client request at the *RWSIdle* resource queue and the *VideoServer* entity waits for a new video transmission at the *RVSIdle* resource queue. The user makes a video selection (*GSelectService* active state) and sends it to the *WebServer*. The iteration between the *ClientWorkStation* and the *WebServer* is represented the active state *SendReq/HandleSeleciton*. Based on this selection, the *WebServer* chooses an appropriate *VideoServer* entity by sending a command to initiate the video transmission (*HandleSelection/InitTxMsg* active state), and after send a command to the *ClientWorkstation* entity to confirm the reception of a video request message (*HandleSelectionConfMsg/ConfirmationD*

active state). The *VideoServer* transmits the video frames to the *ClientWorkstation* up to the end of the video (*RSendFrame/ReceiveFrameD* active state). At the end the *VideoServer* transmit the *TerminatePlayerMsg* message to the *ClientWorkstation (SendTerminate/TerminatePlayer* active state*)*.

Figure 8 presents a short fragment of the XACDML model generated from the software model. It shows three elements of the global XACDML model: the *QterminatePlayer* dead state, the *GSelectService* Generate active state and the *RProcessFrame* Router active state. They are elements of the Web Based Video ACD shown on Figure 7.

```
<?xml version="1.0"?>
<acd id= "AUTOMATICALY_GENERATED XACDML">
        …
<dead id="QTerminatePlayer" classe="a_3">
<type struct="QUEUE" size="10" init="0" />
</dead>
        …
<generate id="GSelectService" classe="a_3">
<stat stype="EXP" parm1="6.0" />
<next ndead="QSendReq(VideoReqMsg)D" />
<persistence persistence = "temp" />
<entitynumber entitynumber = "0" /></generate>
        …
<router id="RProcessFrame" classe="a_3">
<stat stype="UNIFORM" parm1="0.175" parm2="0.225" />
<prev pdead="QProcessFrameD"/>
<whennext cond="FrameNumber1 .GE. 10">
<next ndead="QTerminatePlayer"/>
</whennext>
<whennext cond="FrameNumber1 .LT. 10">
<next ndead="QReceiveFrame"/>
</whennext>
</router>
        …
</acd>
```

Figure 8. XACDML Representation

4.2. Generating and Executing the Simulation Program

The global ACD model, represented by an XACDML file is used as input to generate the simulation program. The XACDML Parser, shown in Figure 1, analyzes the global XACDML file, generates an internal data structure with all the information about the system entities and simulation parameters, and then executes the simulation.

Table 1 shows the results of the simulation experiments for two operation scenarios. The only difference between the two experiments is the number of video servers.

For this example it is assumed that all videos have a constant number of frames equal to 10, and the time to show each frame on the video window is 1 second. The simulation experiments show that with only one video server and a video request with exponential distribution of mean 6 seconds, it is not possible to meet the requirement of not exceeding the video exhibition time limit of 10.3 s for at least 90% of the video requests. With one video server, 47.57% of the requests exceeded the video exhibition time limit. For the simulation experiment with two video servers, this value is 6.19%, i.e. less than the 10% defined by the requirements.

Table 1. Simulation Results of the Video Server Example

	Experiment 1	Experiment 2
Simulation run time (s)	36000	36000
Warm-up period (s)	1800	1800
Exponential video request mean time (s)	6	6
Frame number per video	10	10
Video Server Number	1	2
Average processing time for each video request (s)	12.2	4.98
Percentage of video requests that exceeded video exhibition time limit (<10%)	47.57	6.19

4.3. Analysis of the mapping

The example illustrates the feasibility of automatic mapping for a rather complex application. The mapping of UML to ACD elements described in Section 3.1 was used to implement the parsers described in Figure 1

UML state machine diagram is a widely used representation for real time modeling. It provides a powerful representation to model the dynamic behavior of the real time system entities and their iteration. It also has similarities with ACD which facilitates the conversion, compared with other UML diagrams used for dynamic behavior modeling. Since there is straight correspondence between state machine diagrams and ACDs, the interaction between the simulation and software specialists is facilitated. Through the experimentation by running the simulation program and through the analysis of the ACD model, the simulation specialist may suggest possible alternatives to improve the performance of the software model. The performance annotations derived from the MARTE profile were used in the web-based video example.

5. CONCLUSIONS AND FINAL REMARKS

This paper presents a mapping of a software model to a performance model in order to allow the verification of performance requirements. The software model is represented by UML Deployment and State Machine diagrams annotated with the performance information according to the MARTE profile and the simulation model uses ACD.

The mapping is employed in a simulation framework for verification of software performance requirements of real time computer systems. The framework uses two parsers.

The first one translates the software model into an ACD performance model and the second one translates the ACD model to the simulation program.

We claim that the intermediate ACD model can help in the process of software performance analysis. It minimizes the problem related to the technological gap between the software and simulation specialists in the process of software performance analysis. Both specialists can work on their own domains (UML and ACD models) and interact with each other in an iterative process of modelling and performance verification until a design solution is reached. The mapping helps the feed forward and feedback interaction. Currently, we are employing the mapping in other real time software applications. We are using it to verify performance requirements of embedded system for satellites. For future work we intend to improve the automation of the feedback process of the framework described in Figure 1. We also intend to make the framework available for download in a near future.

Acknowledgments

We would like to thank FINEP for the financial support.

References

Arief, L.B., and N.A. Speirs. 2000. "A UML Tool for an Automatic Generation of Simulation Programs." In ACM Proceedings of Second Int'l Workshop Software and Performance, 71-76.

Balsamo, S., A. Di Marco, P. Inverardi, and M. Simeoni. 2004. "Model-Based Performance Prediction in Software Development: A Survey." IEEE Transactions on Software Engineering, Vol. 30, Number 5, 295-310.

Balsamo, S., and M. A. Marzolla. 2003. "Simulation-Based Approach to Software Performance Modeling." ESEC/FSE'03, ACM, Finland, 363-366.

Di Marco, A. 2005. "Model-Based performance Analysis of Software Architecture." Dissertation, PH.D. Thesis, Dipartimento di Informatica, Università di L' Aquila.

Gil, J. N., and C. M. Hirata. 2003. "XACDML - Extensible ACD Markup Language." In Proceedings of the 36th Annual Simulation Symposium, IEEE, 343-50.

Hirata, C. M., and R. J. Paul. 1996. "Object-Oriented Programming Architecture for Simulation Modelling." International Journal in Computer Simulation, v. 6, n. 2, 269-287.

IBM. 2010. *Rational Unified Process*. Available via <http://www-01.ibm.com/software/br/rational/rup.shtml> [accessed April 11, 2010].

Kordon, F., J. Hugues, and X. Renault. 2008. "From Model Driven Engineering to Verification Driven Engineering." In Lectures Notes in Computer Science, Volume 5287/2008, 381-393.

Marzolla, Moreno. 2004. "Simulation-Based Performance Modeling of UML Software Architecture." PH.D. Thesis, Università CA Foscari Venezia.

OMG. 2004. *UML 2.0 Superstructure Specification*. Version 2.0**,** OMG.

OMG. 2005. *UML Profile for Schedulability, Performance and Time*. Version 1.1, OMG.

OMG. 2009. *UML Profile for Marte: Modeling Analysis of real-Time and Embedded Systems*. Version 1, OMG.

Paes, C. E. B., and C. M. Hirata. 2008. "RUP Extension for Software Performance." 32nd Annual IEEE International Computer Software and Applications Conference, 732-738.

Pidd, M. *Computer Simulation in Management Science*. John Wiley & Sons Ltd, Chichester, 3rd ed, 1992.

Sancho, P. P., C. Juiz, and R. Puigjaner. 2005. "Automatic Performance Evaluation and Feedback for MASCOT Designs." In Proceedings of the 5th international Workshop on Software and Performance. WOSP '05. ACM, p.p. 193-194.

Smith, C.U., L. G. Williams. 2002. *Performance Solutions, A Pratical Guide to Creating Responsive, Scalable Software*. Addison-Wesley.

Stankovic, J. A. 1988. "Misconception About Real Time Computing: A Serious Problem for Next-Generation System." IEEE computer, Vol. 21, no. 10, 10-19.

Biography

Celso Massaki Hirata is an associate professor at the Computer Science Department, Instituto Tecnologico de Aeronautica (ITA), Brazil. He obtained his Ph.D. degree in Computer Science from Imperial College, UK, M.Sc. degree in Operations Research and B.Eng. degree in Mechanical-Aeronautical Engineering from ITA. His research interests include distributed systems, simulation modeling, and software engineering. He has over 70 publications in these areas. His email address is <hirata@ita.br>.

Ronaldo Arias is a member of the technical staff of the On-Board Data Handling Group, Aerospace Electronic Division (DEA), Instituto National de Pesquisas Espaciais (INPE), Brazil. He is a PhD student in Computer & Electronic Engineering division from Instituto Tecnologico de Aeronautica (ITA). He obtained his M.Sc. degree in Computer Science from ITA and his B.S. degree in Computer Science from Universidade Federal de Sao Carlos. His research interests are real time systems, software engineering, simulation, and fault tolerance. His email address is <ronaldo@dea.inpe.br>.