Architecture Analysis in Software Factories

Sandeep Neema
Vanderbilt University/ISIS
P.O. Box 351829
Nashville, TN 37235, USA
+1 615 343 7472
sandeep.k.neema@vanderbilt.edu

Jason Scott
Vanderbilt University/ISIS
P.O. Box 351829
Nashville, TN 37235, USA
+1 615 343 7472
jscott@isis.vanderbilt.edu

Gabor Karsai
Vanderbilt University/ISIS
P.O. Box 351829
Nashville, TN 37235, USA
+1 615 343 7472
gabor.karsai@vanderbilt.edu

ABSTRACT
In this paper, we argue for the incorporation of architecture analysis techniques in Software Factories. While software factories often rely on a domain-specific (or product-line specific) architecture, frequently there are some architectural alternatives left open. In these situations, architectural analysis, static or dynamic is essential. The paper elaborates these concepts and shows an example from the domain of avionics systems how this can be accomplished.

Categories and Subject Descriptors
D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering, Object-oriented design methods

General Terms
Design, Performance.

Keywords
Model-integrated Computing, Software Factories.

1. INTRODUCTION
Software Factories configure tools, processes, and content for building software product lines [1]. Often, the product lines are built using a reference architecture that is instantiated in different ways in specific products. The reference architecture may rely on a number of common architectural patterns (e.g. publish/subscribe, etc.), or it could be based on a very strict framework for component composition. In any case, there could be architectural decisions that are made by the designers (i.e. the "operators" of the factory) and that ultimately effect the functional and non-functional properties of the product. For instance, allocating components in a different way or relying on proxy techniques for local data distribution in real-time embedded systems can greatly influence the end-to-end latency in avionics systems.

The difficulty is that the impact of architectural decisions could be hard to determine at design time. Especially, if the system is large (has, let us say, >1K components), it is very difficult to predict whether the resulting system satisfies expectations or not. We argue that it is essential to incorporate analytical techniques in software factories to assist with the architectural decisions. We assume that the components one uses in the factory are not necessarily available, but at least some approximations of their relevant properties are known (e.g. worst-case execution time). Given these assumptions, architectural analysis tools could be used to validate architectural decisions.

In this paper we introduce an illustrative example from the domain of distributed (soft) real-time systems. The problem was to determine some non-functional properties of a particular instance of a domain-architecture. We have developed: (1) a domain-specific design modeling language for capturing the variability in the domain architecture, (2) a modeling language for capturing the analytical model of a given execution platform, (3) a translator from the design language into the analysis language, (4) the simulation components that approximated the behavior of components and the execution platform, (5) a dynamic analysis framework that allowed quantity studies via using a discrete-event system simulator. We were able to show the impact of architectural decisions using this tool, in a quantitative manner. We were using our metaprogrammable tools from our work on Model-Integrated Computing [2].

The approach as presented here makes key contributions to the following focal points of Software Factories research:

1) Verification and Validation: The paper demonstrates an analytical technique to help assess architectural compliance with system requirements. The architecture of the system is described with a domain specific modeling language that incorporates the architectural invariants in the form of well-formedness constraints as well as in the form of transformations that capture the dynamic operations semantics of the architecture.

2) Automatic Configuration, Code Generation, and Model Transformation: The paper also demonstrates the use of model transformations for derivation of analysis models from the architecture models. A technique based on graph rewriting has been employed for specifying model transformations, which is formal in addition to being efficient and scalable.

3) Product line element verification: The techniques demonstrated in the paper can be used to address questions highly relevant to the users of software factories. For instance: "If we include certain features in a particular

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA Workshop Software factories, 2005, October 16, 2005, San Diego, CA, USA.

instance application (i.e. product line element), will the system exhibit the right performance or not?"

We also note that the general approach of deriving analysis models from design models using a transformation is applicable beyond the realm of distributed real-time embedded systems, including information systems. Moreover, the analytical formalism of Discrete Event Systems is very general and has been employed in modeling and simulation of a large variety of complex systems [9].

2. CHALLENGE PROBLEM

Real-time Operating Systems based on the ARINC 653 specification, are rapidly becoming the operating platform of choice for safety critical real-time avionics applications [7]. The focal point of the specification is *robust partitioning* – a mechanism for strong isolation of applications, both in time (temporal partitioning) and in space (memory partitioning), for assuring a high degree of fault containment, ease of verification, validation, and certification. A robustly partitioned system allows partitions (or application software) with different criticality levels to execute on the same processor core without affecting each other spatially or temporally.

Partitions are scheduled on a fixed-cycle basis, with a major time frame that is repeated throughout the runtime. Within the major time frame each partition may be allocated one or more windows – defined by offset from start of major time frame, and duration. Thus each partition is assured dedicated and uninterrupted[1] access to execution resources during its execution window. The static schedule must be designed by taking into account periodic computational requirements of each partition. Please note that within each partition multiple processes may execute concurrently and are governed by a priority-based preemptive scheduling scheme that is local to the partition, and sharing resources that are allocated to the partition. Similar to temporal windowing, predetermined areas of memory are allocated to partitions, with at most one partition having write access to any particular area of memory.

One of the challenges in systems implemented atop an ARINC 653 compliant RTOS is in inter-partition communication. For the reasons of isolation, and fault containment, the specification does not allow any direct/memory-mapped communication across partitions. The inter-partition communication must be done using the ARINC 653 communication services, and the actual communication is performed by an I/O service which executes as a system partition governed by the overall partition scheduling. The direct implication of this requirement is that the static partition schedule, an architectural decision, has a significant impact on the communication latencies. We present a small subset of an avionics application as a challenge problem[2] that illustrates this concern, and demonstrates the value of an analytical approach in understanding and addressing this concern.

[1] Bus-acknowledgements and time-outs may interrupt one partition's execution even though the events relate to a different partition. We note this here since it affects the analysis.

[2] The authors thank Northrop Grumman and USC for the avionics problem cited here.

Consider an avionics system that consists of a number of subsystems. Each subsystem is implemented as a separate ARINC 653 partition. The avionics system has a number of sporadic events that need to be processed. A key design objective is to establish an upper bound on the arrival rate of such events. The particular event processing scenario is to process New Page request events from the Multi-Function Display (MFD). By menu selection the pilot can request a different page from a particular functional unit to be shown on the display. The processing of the request involves several subsystems: the MFD display hardware, the DisplayManager (DM) subsystem, the PageContentManager (PCM) subsystem, the FlightManager (FM) subsystem, and the FlightDirector (FD) subsystem.

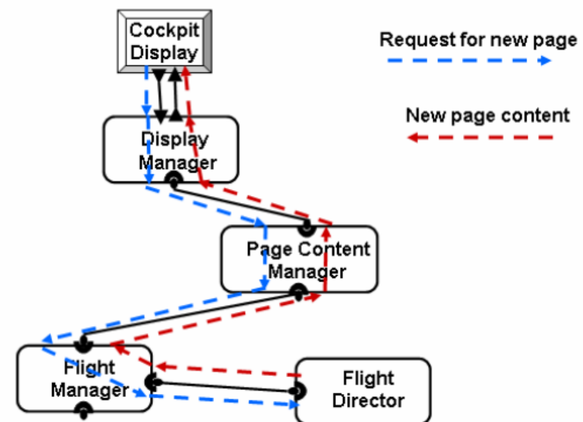

Figure 1: System Latency Flow

The maximum arrival rate of the New Page request is bounded by the minimum latency between the time the pilot selects the menu item to request the page and the time the new page has been displayed. The minimum latency is determined by the amount of time it takes to process the flow of the page request including the display of the new page. The lower bound of this processing latency is determined by the fact that subsystems execute as separate partitions and that inter-partition communication incurs a partition offset delay as well as frame delays at the partition execution rate.

3. TOOL CHAIN

Analogous to manufacturing assembly lines, a software factory needs to be configured to support a particular software product line. Configuration involves customizing meta-programmable tools and integrating them in "tool chains" to support the design process for a particular software product line. A *tool chain* is an integrated suite of tools, where the integration is accomplished by providing automated transformations from the output of a tool to input of another tool. In prior research at ISIS we have developed a meta-programmable integration framework, labeled Open Tool Integration Framework or OTIF that facilitates creation and deployment of tool chains (see [10] for details). Utilizing this infrastructure, we have set up a tool chain to support design, analysis, and synthesis of the class of systems described above. Figure 2 shows a logical architecture and design flow in the toolchain. The details of the tools and transformations are provided below.

3.1 Design Modeling

In this section we briefly describe Avionics Systems Modeling Language (AvSML), a multi-aspect Domain-Specific Modeling Language that supports modeling of component based Avionics applications architected to be deployed on an ARINC 653 compliant RTOS. This language has been instantiated in the meta-programmable GME tool, resulting in a domain-specific modeling environment that can be used by System Integrators and Designers to model avionic systems.

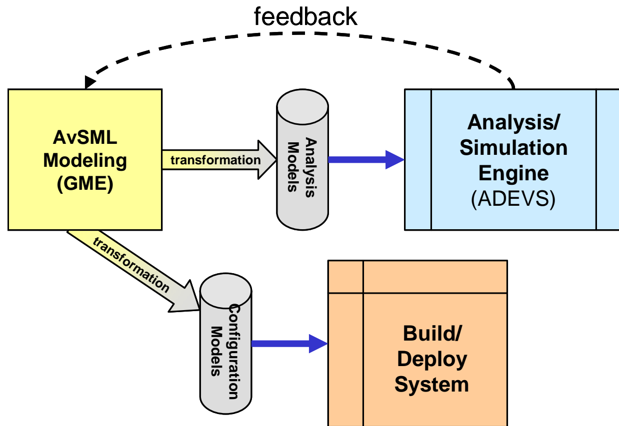

Figure 2: Tool Chain

The key modeling concepts of AvSML can be summarized into following categories:

1. Component (Subsystem) Modeling
2. Component Interaction Modeling
3. ARINC 653 Platform Modeling
4. Deployment Modeling

Figure 3 shows a (partial) metamodel of Component Modeling sub-language of AvSML. Components are defined by modeling their interfaces (*Connector*)[3], and their internal behavior. The internal behavior of a component is modeled by defining a workflow (data and control flow) that includes *Connectors*, *PrimitiveFunction* and [Compound]*Functions*, and *State* variables. This in effect captures the internal processing flow of the component, from when data/event arrives to a Connector, till when results are sent to a Connector. The workflow language (not shown in the figure below) is intentionally not Turing complete for reasons of analyzability. Please note that the internal behaviors of the *PrimitiveFunctions* can be implemented in a Turing complete programming language, however the models abstract out those details with a data interface (*DataInput*, *DataOutput*), and a characterization with respect to execution properties. The execution property characterization allows modeling the execution time in the form of a statistical distribution, and parameters specific to the distribution, for example the mean (μ) and variance (σ) in case of a Gaussian distribution.

[3] The little curved arrow at the bottom of the class icon, indicates a proxy to a class that has been defined in a different paradigm sheet (class diagram).

The Component Interaction Modeling sub-language of AvSML allows composing systems by instantiating components, and defining their interactions (we refrain from detailing the metamodel in the interest of brevity). The interaction semantics are defined by the type of Component Connectors participating in an interaction. Two types of interaction mechanisms are supported: a) Sampled interaction, and b) Queued interaction. The sampled interaction implies a form of asynchrony in which the source component produces output on its connector which stays persistent till a new value is written. The destination component samples the connector and reads the most recent value. The queued interaction on the other hand implies a loose form of synchrony in which output of the source component is queued till it is consumed by the destination component. Please note that his particular choice of interaction semantics is driven by the platform. The ARINC 653 specification supports these two types of inter-partition interaction mechanisms, and the challenge problem requires implementing components (subsystem) in individual partitions, thereby constraining the interaction semantics. One may argue here for the need for platform independence in the DSML; however we note that the ARINC 653 specification influences the domain architecture, which in turn dictates the DSML.

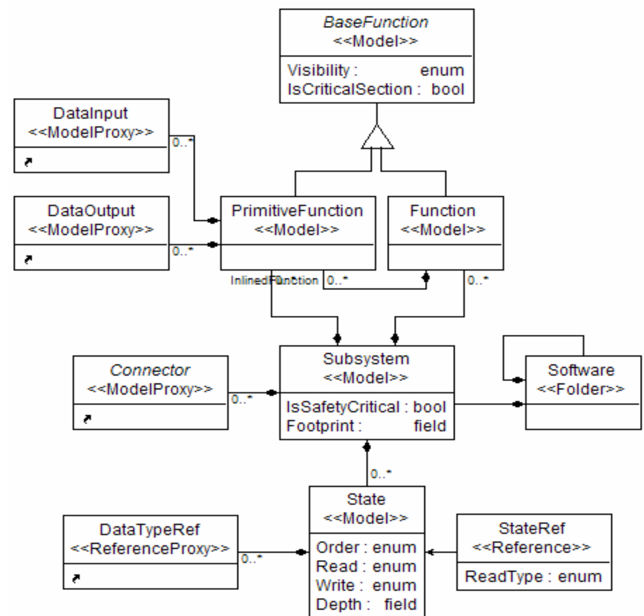

Figure 3: Component (Subsystem) Modeling

Figure 4 shows a metamodel of the Platform and Deployment Modeling sub-language of AvSML. This sub-language includes concepts to model the *Partitions*, time slots or *Windows* assigned to a partition, *InterPartition*, and *IntraPartition* communication ports, and inter-partition communication *Channels*. The deployment of application components or subsystems is modeled with *SubsystemRef* , which refers to a *Subsystem*. For reasons of fault-tolerance the deployment modeling also allows specifying backup deployments i.e. in addition to the primary deployment, a component is also deployed in a different partition/processor as a backup (*SubsystemBackup*) where it stays inactive till a fault causes the primary deployment to fail. The sub-language allows

associating partitions with *ComputationalUnits* which represent processing resources.

In summary, AvSML allows representation and characterization of a component-based Avionics application architected in accordance with ARINC 653 specification.

3.2 Analysis Models

As illustrated in the challenge problem, our analysis objectives are to establish non-functional properties such as end-to-end latency, resource utilization etc. Experience indicates that static analysis techniques such as RMA fails to adequately address the analysis needs of class of systems represented by the challenge problem, due to both an overly conservative approximation with worst-case execution time, as well as inability to incorporate data dependencies. This motivates the need for dynamic analysis techniques based on Simulation. Discrete Event Simulators with the ability to program the event processing behaviors of simulation components have been utilized in a variety of performance analyses [9].

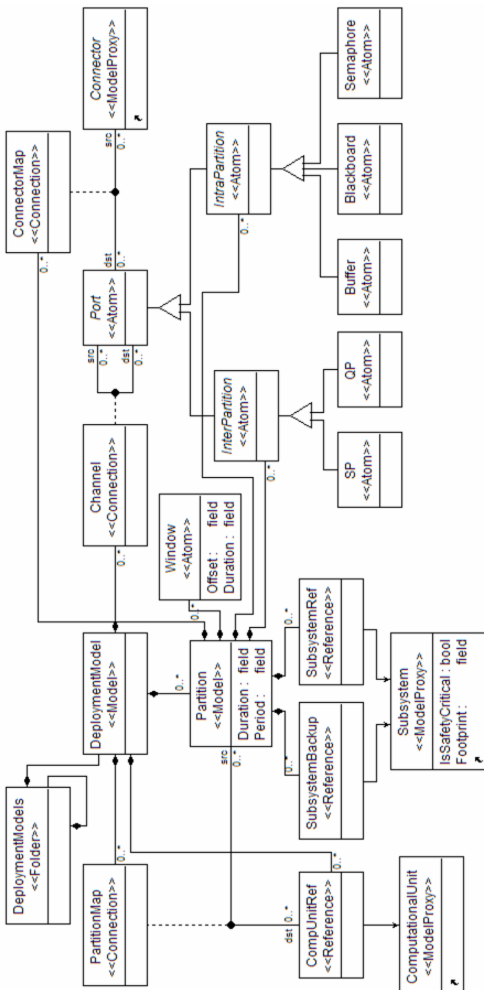

Figure 4: Platform and Deployment Modeling

We utilize one such DES based performance simulation tool that has been developed at ISIS in another research. In the rest of this section we describe the input language of this tool, which defines the analysis model. Figure 5 shows a (partial) metamodel of the analysis model. As can be expected the analysis metamodel abstracts away some of the details of the software engineering artifacts, but explicates the details of all computation activities that consume processing resource, all communication activities that consume bandwidth, and the workflow across these.

The key concepts here can be summarized as follows. A *Partition* represents allocation of processing resources, and can be associated with a *ProcessorElement* with more than one *Window*, characterized with a *Duration*, and *Offset*. A *Function* represents processing activities, and the processing time requirements are characterized as a statistical distribution (*DistFunc*), and a set of parameters specific to the distribution (*Mean*, *Min*, *Max*, *Mode*, *StdDev*). Please note here that the Function construct captures, processing activities in general, whether it is processing activity in application (see Component Functions in AvSML), or processing activity in the platform services which are not explicitly represented in the design models but are incorporated via the design model to analysis model transformation. The *DataElement* construct facilitates specification of communication, and is associated with a *NetworkElement* (not visible in the figure). Please note that the ARINC 653 inter-partition communication of necessity involve a memory copy, and may often take place across a bus. The *Functions* produce or consume *DataElements* in the workflow.

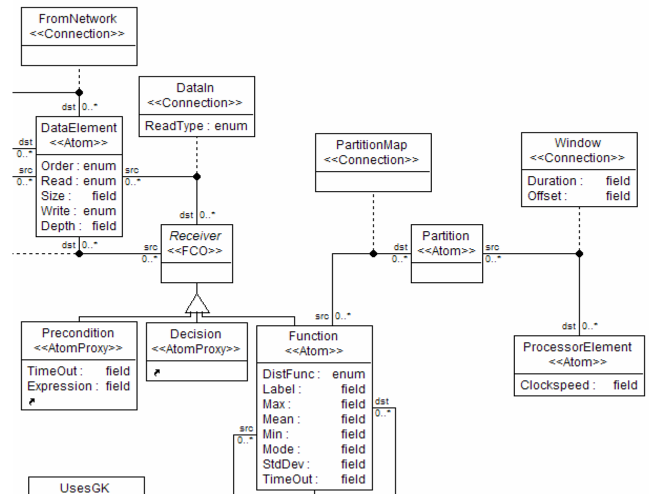

Figure 5: Analysis Metamodel

Next, we briefly describe the derivation of analysis models from design models as expressed in a transformation that we have developed. Some elements of the mapping are one-to-one and fairly straightforward:

- ComputationUnit in AvSML maps to ProcessorElement
- Partition, Windows and their association with ComputationUnit in AvSML maps to Partition and its association with a ProcessorElement through a Window. Note here that the syntactic representation of these concepts differs in AvSML and the Analysis metamodel, for obvious reasons of user convenience (relevant to AvSML) vs. conciseness (relevant to Analysis metamodel).

The component workflow in AvSML maps analogously, however, some elaboration is performed during mapping. The elaboration happens at the communication endpoints. A *Function* is inserted to incorporate the processing overhead of the platform API, as well as a *DataElement* is introduced to capture the communication resource requirements.

3.3 Analysis Engine and Component Models

Prior to describing the analysis engine and simulation components, it is worthwhile to note here that the analysis (or simulation) engine is responsible for operationalization of the analysis metamodel described above. Thus, from a language viewpoint, one can consider the description of the simulation components as defining the operational semantics of the analysis metamodel.

Our simulation engine is built atop ADEVS, a public domain discrete event simulator developed at University of Arizona [8]. ADEVS is implemented as a C++ library that supports the construction of discrete event models based on the Parallel DEVS formalism [9]. While the details of the DEVS formalism are clearly outside the scope of this paper, it can be summarized as a collection of concurrently interacting components, the dynamic behavior of which is defined in terms of events. Thus, each DEVS component implements an event-driven reactive behavior. ADEVS designates such components as 'atomic' components. The behavior of an atomic component is defined by its state transition function, its output function, and its time advance function. The interface for these behaviors is defined in an abstract base class labeled 'atomic'. All simulation components derive from atomic and provide implementation for the behaviors listed above. In addition to reactive components, ADEVS also allows modeling passive components that represent data or tokens.

The behavior of some key simulation components can be summarized as follows:

Function – The function simulation components has three internal states: Idle, InvokedWaiting, InvokedProcessing. The Function can receive three types of events: recv_control, proc_req_grant, proc_grant_cancel, and can generate three types of events: relinquish_control, proc_req, proc_req_done. Additionally, it also handles timer events (time advance behavior) from the simulator. Initially, a Function is in an Idle state, and is waiting to receive control, which can be passed from a prior element in the workflow (routed by the workflow engine), or from an external generator (mimicking environmental or user stimuli). Once it receives the 'recv_control' event it emits a 'proc_req' event and enters the 'InvokedWaiting' state. In the 'InvokedWaiting' state it waits for a 'proc_req_grant' event. The processor request can be granted depending on whether the partition to which the function is mapped is currently sliced in. In case the 'proc_req_grant' event is received it transitions to the 'InvokedProcessing' and notifies the ADEVS engine through the time advance interface of the required processing time. The ADEVS engine then notifies back the Function when the required processing time is complete. The Function then emits a 'relinquish_control' and a 'proc_req_done' event and transitions to the 'Idle' state. It is also possible that before the processing time of the Function has expired, the Partition to which the Function is mapped can get sliced out in which case the Function receives a 'proc_grant_cancel' event. In response to this event the function computes the remaining processing time, emits a 'proc_req' event and transitions to 'InvokedWaiting' state.

Please note that the required processing time is computed based on the statistical distribution that characterizes the execution time of the function code. In our current implementation data-dependent execution times can not be handled, however, an extension is in progress which would allow characterizing the data tokens with additional meta-data, and incorporating user-defined computation of processing time based on data token characteristics.

Partition – The partition simulation components has three internal states: SlicedIn, SlicedInProcessing, and SlicedOut. Partition handles four types of input events: slice_in, slice_out, proc_req, proc_req_done, and it generates two types of events: proc_req_grant, proc_grant_cancel. Partition starts in a 'SlicedOut' state. On receiving a 'slice_in' event it transitions to the 'SlicedIn' state. From the 'SlicedIn' state, in response to a 'proc_req' event, it emits a 'proc_req_grant' event and transitions to a 'SlicedInProcessing' state, and marks the Function that is currently active. From 'SlicedInProcessing', in response to a 'proc_req_done' event it transitions to 'SlicedIn' state, while in response to a 'slice_out' event it emits a 'proc_grant_cancel' event and transitions to 'SlicedOut' state. From 'SlicedIn' state it transitions to 'SlicedOut' state in response to 'slice_out' event.

Processor – The processor simulation component simply implements a time slicing scheduler. It cycles through the list of windows sorted by their start offset, sends a 'slice_in' event to the respective partition at the start of the window, notifies the simulation engine to wake it up after the window duration, at which point it sends a 'slice_out' event to the corresponding partition. This cycling behavior is repeated for the entire duration of the simulation.

The result of the simulation is an event trace which captures the timing of occurrence of each event. A secondary, Matlab-based data-mining and post-processing tool bundled with the simulation tool performs processing of the event traces to compute end-to-end latency, communication buffers, and processor utilization.

4. RESULTS

The challenge problem briefly described in section 2 was subjected to modeling and analysis using the tool chain that we described above.

Figure 6 shows models of the challenge problem in AvSML. We modeled each of the subsystems (MFD, DM, PCM, FM, and FD) including their interfaces, and their internal workflows. Each component (except for MFD, and FD), have two input queued ports and two output queued ports. A request that comes in to the input port is processed by the workflow, and the results of the processing are dispatched via the output ports. For example the DM component has one input port through which it receives a page request, which it processes and dispatches via its output port to the PCM component. The DM component has another input port on which it receives NewPage, which it then processes and dispatches via another output port to the MFD component. The MFD component has one output port through which it generates page requests, and an input port through which it receives new page. We characterized the execution time requirements of the

processing functions based on our best understanding of the computational requirements of the system, as well as modulated these requirements to derive interesting analytical observations. We also modeled the interactions between these components as described in the challenge problem. The components were deployed on a single processor but were distributed across multiple partitions i.e. one component was mapped to one partition. The analysis question stated in the challenge problem can be reformulated with respect to the models as: *estimate the latency between generation of an output on the PageRequest port of the MFD component, and arrival of an input on the NewPage port of the MFD component.* As we noted earlier that the partition schedule can have significant impact on the latency, we experimented with different partition schedules in the deployment models to substantiate this hypothesis.

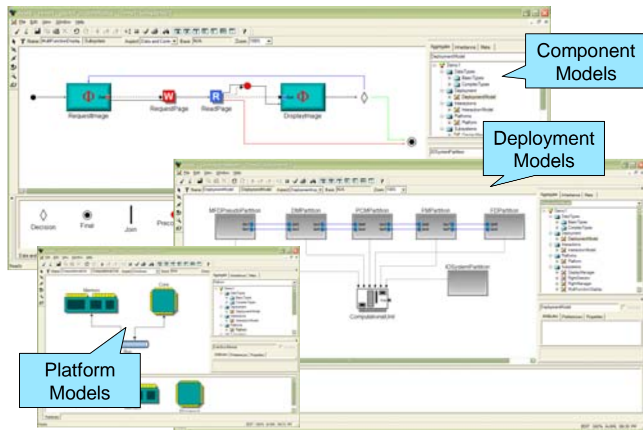

Figure 6: AvSML Models of the Challenge Problem

Figure 7 shows the analysis models for the challenge problem, derived automatically from AvSML models through a model transformation. Two views are shown: a) an aggregated workflow including functions, and data elements, and b) a partition-window/processor mapping view. In the aggregated workflow the iconic objects with symbol φ are functions, whereas the other iconic objects represent data elements.

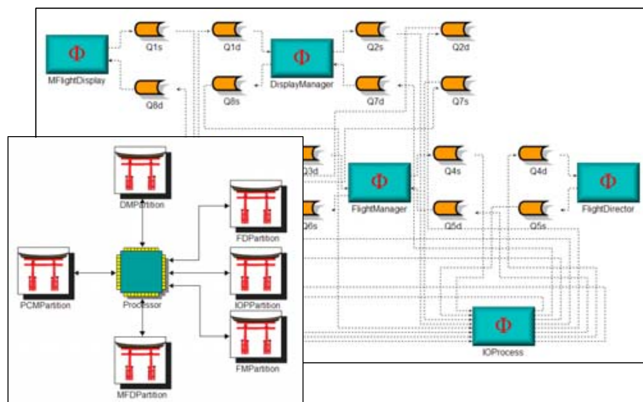

Figure 7: Analysis Model derived from AvSML Models

Data elements are characterized with the size requirement for communication. Please note the connection topology in this view. All the outputs from a component go to a data element, which goes into an IOProcess, which then routes it to another data element that goes into the input of a component. As we mentioned earlier this type of routing is a consequence of the ARINC 653 specification that mandates that all inter-partition communications are performed by an IO Process running in a system partition. Please note that the design models do not capture these details, and this routing is an example of automatic synthesis of the platform details through the transformation. The partition mapping view shows all the partitions mapped to a single processor, and the connections between Partition/Processor capture the windowing details such as offset, and duration.

These analysis models were generated from the AvSML model and were subjected to another transformation, which is purely syntactical in nature and as such we refrain from describing its details. This transformation takes the analysis model and generates a textual configuration file that is sent to the ADEVS engine which executes the simulation model, and generates an event trace. A Matlab-based post-processing utility tool computes end-to-end latency estimates by examining the timestamps of the events corresponding to the generation of an event by the MFD function into the data-element at its output port, and corresponding to the receipt of an event into the data-element at its input port.

In all of the partition scheduling configurations the major time frame was 200 ms, and the simulation was run for 100 seconds (500 schedule cycles). The chart shown plot the latency on the y-axis, and the simulation time on x-axis.

Figure 8 shows latency plots from our first partition scheduling configuration. This could be considered a most optimal (with respect to latency) partition schedule. The partitions were time ordered according to their data dependencies, and an IO partition was scheduled between every communicating partitions. Thus the entire roundtrip from the MFD to FD and back completes in two schedule cycles, and the worst case latency is ~380 ms which is under two scheduling cycle time (400 ms). The variances in latencies are due to the statistical distribution of the execution times.

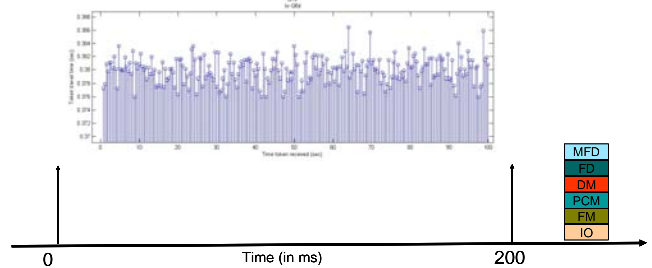

Figure 8: End-to-End Latency Results (Configuration 1)

Figure 9 shows the result from the second partition scheduling configuration. Here we chose a more realistic partition schedule which allocates two windows to the IO process two windows to the FD process (owing to its periodicity requirements), and one window to each of the rest. The windows are not specifically arranged to follow the data dependency order. The latency results in this configuration demonstrate a 4.5 cycle (~900 ms) round trip delay. The histogram chart of latency also conveys a Gaussian type of distribution which is what we chose for the execution times of processing. The worst case latency in this configuration is observed to be 5 cycle (~1000 ms).

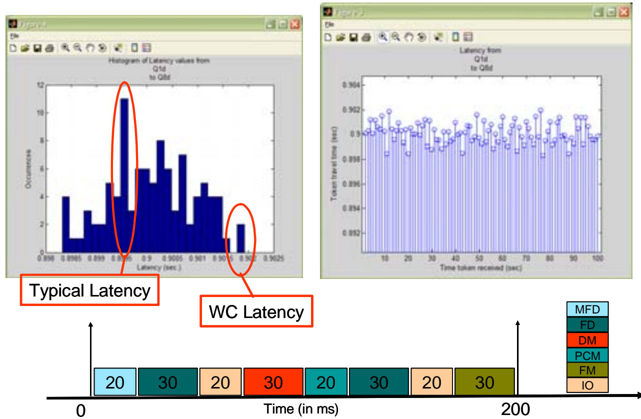

Figure 9: End-to-End Latency Results (Configuration 2)

Figure 10 shows the latency results from our third partition scheduling configuration. The scheduling configuration in this case is the same as the second configuration; however, we experimented with the distribution of execution time. We increased the variance in the execution time of the FD component. Note that while this variance is artificial, this simulates a real scheduling jitter that is acknowledged in the specification as mentioned in the footnote on page 1. Please note here that due to the spread in the execution time of the FD component, it may not always finish its execution in a single window, and the rest of the processing is finished in the next window allocated to the FD partition. The consequence as can be understood from the latency plots is a one cycle (200 ms) jitter. Since, the FM partition follows the second window of IO partition, which in turn follows the second window of the FD partition. If the execution of the FD partition is completed in the second partition then the IO partition transfers the output to the FM partition in the same scheduling cycle. However, in case the execution of the FD partition does not complete in the second window, then it is postponed till the first window of the next cycle. Thus the FM partition gets the result in the next cycle instead of the same cycle which results in a 1 cycle jitter.

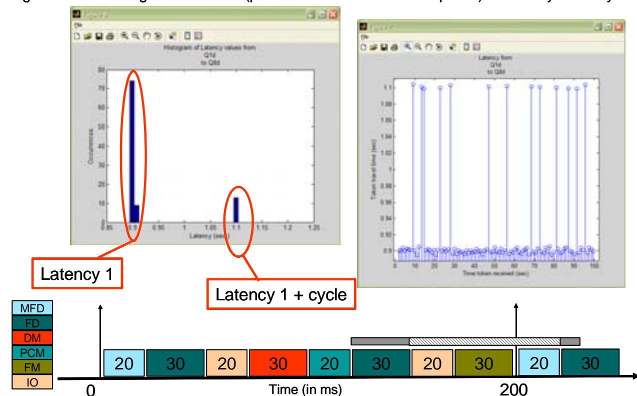

Figure 10: End-to-End Latency Results (Configuration 3)

This type of jitter in latency may have significant consequences depending on the nature of the application. The results shown above highlight the ability of the simulation based performance analysis in drawing such interesting observations. One caveat has to be made: the quality of the results is dependent on the richness of the scenarios/configurations posed. This is where arguments are typically made for static analytical techniques since they are not necessary subject to limitations of the tested scenarios. Unfortunately, such techniques can not be applied to the type of complex applications represented by the challenge problem without gross approximations, and oversimplifications.

5. RELATED WORK

Architecture analysis is a well-known concept in the field, but often it is restricted to static analysis to check architectural compliance [4]. As it is presented in the literature, it assumes the use of an Architecture Description Language (ADL), with additional tool support. We argue that for the Software Factory world, the problem is better solved by using a two-level approach: namely using a (1) domain-specific modeling language for capturing the design, and a (2) dedicated analysis modeling language (with automatic transformation from the first to the second). The reason for this is that the design language could hide platform-specific details that are irrelevant for the designer (and all these can be incorporated in the transformation), and thus analysis language could be optimized for the analysis environment/tool that is used.

Previous work has been done on modeling architectural dynamics using Statecharts (see, e.g. [2]), and the importance of using simulations has been pointed out in [5]. Here we propose something different: the dynamic analysis of the *entire* architecture, with automatically generated analysis models. In this respect, our work is related to [6], but differs in using domain-specific design models (instead of UML).

6. CONCLUSIONS

We have illustrated how architectural analysis can be done in a Software Factory setting, through an example worked out using our MIC tools. We claim that analysis, especially quantitative analysis should be part of a software factory in order to validate architectural decisions in the design. Obviously, a number of properties could be checked using other techniques (e.g. architectural compliance, interface compliance between components, etc.) that we did not elaborate here, but must be considered as well.

Note that early in the design component models could be very approximate and the worth of architectural analysis could be questioned. In this case, the approximate models may serve as the source of requirements for component implementers, because if the components are indeed built according to these expectations, we have a high degree of confidence in the final product.

Architectural analysis provides a number of research challenges. The issues of scalability, performance modeling of software components, the modeling of their dynamics, modeling of platforms, are just a few examples. However, we believe answers to these questions will lead to better software engineering processes that allow us to build complex, large-scale systems that will work in real-life as they were expected to work in the design phase.

7. ACKNOWLEDGMENTS

The authors would like thank all government agencies (DARPA, NSF, and NASA) and industrial partners (Boeing, Lockheed-

Martin, and others) who provided support for the infrastructure tools used in the effort described in this paper.

8. REFERENCES

[1] J. Greenfield, and K. Short: *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, John Wiley, 2004.

[2] Karsai G., Agrawal A., Ledeczi A.: A Metamodel-Driven MDA Process and its Tools, WISME, UML 2003 Conference, San Francisco, CA, October, 2003.

[3] A. Egyed, D. Wile: Statechart Simulator for Modeling Architectural Dynamics, *Proceedings of 2nd International Working Conference on Software Architecture* (WICSA), Amsterdam, The Netherlands, August 2001.

[4] Abowd, Gregory, Robert Allen and David Garlan, "Using Style to Understand Descriptions of Software Architecture," *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'93)*, Los Angeles, CA, 7-10 December 1993, in SIGSOFT Software Engineering Notes, Vol. 18, No. 5, December 1993, pp. 9-20.

[5] Alexander Egyed: Dynamic Deployment of Executing and Simulating Software Components, *in Proceedings of the 2nd IFIP/ACM Working Conference on Component Deployment (CD)*, Edinburgh, Scotland, UK, May 2004, pp. 113-128.

[6] T. Verdickt, B. Dhoedt, F. Gielen, P. Demeester: Automatic Inclusion of Middleware Performance Attributes into Architectural UML Software Models, IEEE Transactions on Software Engineering, Aug. 2005, pp 695-711.

[7] ARINC 653 standard, www.arinc.com.

[8] ADEVS: http://www.ece.arizona.edu/~nutaro/index.php

[9] Bernard P. Zeigler, Tag Gon Kim, Herbert Praehofer: Theory of Modeling and Simulation, AP, 2000.

[10] Karsai, G., Lang, A., Neema, S.: Design Patterns for Open Tool Integration, Vol 4. No1, DOI: 10.1007/s10270-004-0073-y, Journal of Software and System Modeling, 2004.