AVICPS 2010 Workshop

The First Analytic Virtual Integration of Cyber-Physical Systems Workshop.

Nov. 30, 2010. San Diego, California.
Co-located with RTSS 2010.

Sponsored in part by:

LO K H E E D M A R T I N

Edited by Sibin Mohan and Dionisio de Niz

©Copyright 2010 by the authors
Program Chairs

- Dioniso de Niz. SEI, Carnegie Mellon University.
- Sibin Mohan. University of Illinois at Urbana-Champaign (UIUC).

Program Committee

- Lui Sha. University of Illinois at Urbana-Champaign (UIUC).
- Raj Rajkumar. ECE, Carnegie Mellon University.
- Bruce Krogh. ECE, Carnegie Mellon University.
- Ken Butts. Toyota.
- Marco Caccamo. University of Illinois at Urbana-Champaign (UIUC).
- Peter Feiler. SEI, Carnegie Mellon University.
- Alessandro Forin. Microsoft Research.
- Russell Kegley. Lockheed Martin Corporation.
- Steve Vestal. Adventium Labs.
- Bill Milam. Ford.
- Tom Fuhrman. General Motors.
- Sagar Chaki. SEI, Carnegie Mellon University.
- Jorgen Hansson. Chalmers University.
- Rodolfo Pellizzoni. University of Waterloo.
## Contents

<table>
<thead>
<tr>
<th>Title</th>
<th>Authors</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Model-Based Design Approach for Wireless Sensor-Actuator Networks.</td>
<td>A. Rowe G. Bhatia and R. Rajkumar</td>
<td>1</td>
</tr>
<tr>
<td>An Adaptive Discrete Event Model for Cyber-Physical System.</td>
<td>K. Yue L. Wang S. Ren X. Mao and X. Li</td>
<td>9</td>
</tr>
<tr>
<td>Analytical System Composition.</td>
<td>L. Sha and J. Meseguer</td>
<td>17</td>
</tr>
<tr>
<td>Anytime Algorithms for GPU Architectures</td>
<td>A. Abrahimi Saba and R. Mangharam</td>
<td>31</td>
</tr>
<tr>
<td>Formal Verification of Real-Time Embedded Software for Multicore Platforms.</td>
<td>S. Chaki and A. Gurfinkel</td>
<td>39</td>
</tr>
<tr>
<td>Multi-domain Modeling of Cyber-Physical Systems Using Architectural Views.</td>
<td>A. Bhave B. Krogh D. Garlan and B. Schmerl</td>
<td>43</td>
</tr>
<tr>
<td>Need for Strong Semantic Language for Integrating Cyber-Physical Systems.</td>
<td>Bruce A. Lewis</td>
<td>51</td>
</tr>
<tr>
<td>Virtual Integration for Improved System Design.</td>
<td>D. Redman D. Ward J. Chilenski and G. Pollari</td>
<td>57</td>
</tr>
<tr>
<td>Virtual Integration of Cyber-Physical Systems by Verification.</td>
<td>P. Manolios and V. Papavasileiou</td>
<td>65</td>
</tr>
</tbody>
</table>
Message from the Chairs

It is our pleasure to present these proceedings from the first Analytic Virtual Integration for Cyber-Physical Systems (AVICPS) workshop, co-located with the 31st Real-Time Systems Symposium (RTSS).

The complexity and scope of embedded and real-time systems has grown significantly in the last decade or so. While it was possible to design such systems in the past by considering the software and physical constraints separately, this is no longer the case. The advent of cyber-physical systems (CPS) brings with it an increase in complexity that requires the use of sophisticated tools and models in a distributed development environment where different parts of the system are designed by independent teams. Due to the inherent difficulty of designing such systems, many problems are discovered late during the integration of the different parts of the system, where it becomes extremely costly to fix. Hence, there is a need to find most such problems ahead of time and also model the various parts in a way that aids in the development as well as the reduction of integration problems. Such models would also enable the development and use of analysis techniques to improve the understanding and efficiency of such systems.

The study of such integration techniques, models, analyses, etc. would itself require the dedication and effort of the larger research community. This workshop aims to bring together researchers from both, academia and industry, to address these very issues. In its first incarnation we find a good mix of (a) solid technical papers that outline current research work that will help solve some of the problems as well as (b) position papers that aim to outline future research directions for the community. We hope that this workshop and the interesting discussions that will result from it enables the community to improve its understanding of the CPS domain and also lay down research directions that will drive future development in this area.

We hope that the efforts of this workshop will spur increased research activities in the areas analysis, modeling, tools, etc. for the CPS community and that these research efforts will be sustained over many years to ensure its success.
A Model-Based Design Approach for Wireless Sensor-Actuator Networks

Anthony Rowe Gaurav Bhatia Raj Rajkumar
Dept. of Electrical & Computer Engineering
Carnegie Mellon University, U.S.A.
ag@ece.cmu.edu, gnb@ece.cmu.edu, raj@ece.cmu.edu

Abstract—In this paper, we propose a model-based design approach for developing wireless sensor-actuator networks that can map multiple sets of application-level interactions onto a single networking substrate while still enforcing individual requirements. We use a top-down design approach where the functional requirements for each application are graphically modeled using a tool called SysWeaver. Sensor networking applications add unique challenges for model-based design frameworks because the system deployment view is tightly coupled to an installation-specific network topology and link characteristics. Wireless devices can also be mobile and hence may not easily map to standard deployment views. We introduce a SysWeaver plugin called SenseWeaver that is able to capture live topology data from an instrumentation deployment and feed the topology and link characteristic information to the system model. A developer can then use SenseWeaver to specify the functional requirements of multiple applications, analyze communication and task scheduling requirements based on actual topology data, and automatically generate customized code for each sensor network node.

I. INTRODUCTION

Wireless sensor networks provide a versatile and simple deployment platform for sensing and interacting with the physical environment. These devices can support multi-hop communication forming mesh networks capable of self-configuration, self-healing and automatic management. These properties make sensor networks suitable for various cyber-physical system applications like industrial control, critical infrastructure monitoring and building heating and cooling systems. Much work has been done in addressing a variety of challenging sensor networking topics including: network stacks, energy management, simulation, application task design etc. All of these components at each layer in the stack need to work tightly together for a deployed system to operate correctly and efficiently. Systems are now emerging where multiple sets of application requirements are sharing the same underlying infrastructure. For example, many home automation systems have transducers that should be shared with heating and cooling systems in order to optimize building energy consumption. In this paper, we present a plugin called SenseWeaver for the SysWeaver model-based design tool that helps model, synthesize, analyze and automatically generate code for complex wireless sensor-actuator applications.

One of the main advantages for using wireless mesh networking is the ability to rapidly deploy devices and have them automatically setup communication paths. For control applications, this makes it difficult to estimate communication latencies without having information about the underlying deployment topology which may be difficult to anticipate at design time. For example, an HVAC system might have a control loop designed around reading temperature and CO2 values in multiple rooms that need to be processed in order to actuate heaters, coolers or blowers in different parts of the building. Without predictable system components and runtime information, it is hard to estimate reliability and to tune control loop update rates based on timing parameters. Model-based design of distributed applications provides the ability to use analysis frameworks that can ensure correct system operation while satisfying these types of para-functional requirements.

In general, model-based design holds the promise of (a) capturing rich behavioral descriptions along multiple concerns (or aspects), (b) offering an interoperable toolset to analyze and verify both functional and para-functional requirements of a system, and (c) supporting the ability to generate executable code directly from these models. While measurements from run-time environments may need to be fed back to calibrate the models (with parameters such as execution times, network topologies and system overheads), model-based design can in principle be independent from the specific hardware, operating systems or programming languages.

SenseWeaver uses a top-down approach where functional requirements of applications are graphically modeled using a tool called SysWeaver. SysWeaver enables the capturing of para-functional behaviors (e.g. timing, fault-tolerance, security, etc.) of an embedded real-time system and their interactions with the functional behavior of the system [1]. SysWeaver also has the ability to (a) analyze the para-functional properties of the system (e.g. timing properties) either internally or by exporting an appropriate subset of the model to external analysis tools, (b) automate design choices (e.g. mapping of software to hardware entities) [2], and (c) generate code for distributed embedded platforms [3]. This paper addresses the unique considerations required for applying this design cycle to sensor-actuator networks.

Modeling wireless networks has unique challenges as compared to existing systems that utilize component-based design. The physical environment and specific placement of devices heavily influences timing and reliability of communication links in sensor networks. In systems like automotive
body electronics, the network topology and placement of hardware is well under the designers control. This is not always the case when deploying a reconfigurable control system using wireless components. Sensor networks tend to have highly redundant segments in the network layer. Instead of modeling this and other similar characteristics on an individual node-by-node basis, primitives are needed that capture aspects of the network as a whole. Finally, wireless networking models need to capture properties like mobility, self-healing and self-configuration.

To meet the unique challenges of sensor networks, SenseWeaver provides a WSN physical view, a set of sensor networking primitives, an analysis framework and deployment plug-ins for SysWeaver. We use an instrumentation phase to collect information about the environment from a deployed network. We introduce a primitive that represents the sensor network as a clustered component that allows deployment across multiple nodes with a single connection. We also provide the semantics to represent mobile nodes and how they can interface with the networking cluster.

II. RELATED WORK

In the following section we will discuss various current approaches and related tools that aid in the design and deployment of wireless sensor networking applications. We will discuss existing programming language approaches, simulation tools and component based design modeling tools.

One approach to deploying sensor networking systems relies on using a high-level programming language with a single system-wide view of the application. TinyDB [4] takes a database-centric approach by accessing the network using SQL-type commands. A cross-layer design with an integrated tree routing MAC protocol facilitates communication optimized for database access patterns. This allows for energy-efficient network-wide querying of sensors with data aggregation. Though extremely efficient at accessing whole sensor networks, TinyDB does not support custom application-specific logic. The built-in routing protocol does not easily support arbitrary node-to-node communications. In many applications like control and automation, nodes need to communicate autonomously without explicit gateway control.

The Regiment Macro-programming System [5] is an example of a high-level programming language that describes an application as a set of spatially distributed data streams. Regiment contains primitives that facilitate processing data, manipulating regions and aggregating information across regions. The high-level program goes through a de-globalization process where code is compiled from a network-wide application into a set of node-specific executables. Regiment is compiled down to an intermediate token machine language that passes information over spanning trees constructed across the network. This approach provides great flexibility when it comes to application-specific logic; however, it is less efficient at providing short-lived queries like TinyDB. The token-machine-based approach does not easily lend itself to highly dynamic behavior with multiple modes of operation and changing data paths. The tight coupling between language and network protocol makes any-to-any communication as well as low-level adjustment of MAC protocols difficult.

Multiple research groups have developed wireless sensor networking simulators that tend to specialize in a particular layer of the system. ns-2 is an open-source discrete event simulator widely used in networking research. Primarily designed for simulation of IP networks, various projects like UCB Daedalus and CMU Monarch have extended the framework to support wireless communication and mobility. SensorSim [6] extends ns-2 by adding sensor network-specific models, supporting hybrid simulation and providing a graphical user interface. The OPNET Modeler wireless suite is a commercial tool designed for modeling various different wireless networking technologies ranging from 802.11 to mobile ad-hoc networks. The software focuses on the protocol stack with the ability to model RF propagation, interference, transmitter/receiver characteristics, node mobility and the interconnection with wired transport networks.

OMNET++ [7] is an open-source discrete event simulator that shares many of the same features as OPNET. Tossim [8] is a discrete event simulator that emulates the lowest layer of TinyOS primitives. Tossim allows source to be compiled either for simulation or for real deployment on nodes. Em* [9] is a Linux-based framework that can run applications on embedded X-Scale or mote class devices. Em* software can operate in simulation or on real hardware. Simulators like ATEMU [10] and Avrora [11] attempt to simulate the network at the cycle accurate machine code level. Machine-code simulation allows any operating system to be simulated and is not limited to homogeneous source files. Most of these simulators are designed to aid users in developing and evaluating network protocols rather than looking at end-to-end application development. Our work is complementary in that it tries to generate the system using a top-down approach that would utilize an underlying network layer that can be fine-tuned using one of the many existing network simulators.

Various modeling tools have emerged in order to address the challenges associated with end-to-end application design for sensor networks. GRATIS [12] is a graphical framework built on top of the General Modeling Environment (GME) [13] that allows designers to connect different TinyOS components together. GME provides a meta-modeling framework where domain-specific models can be integrated with analysis and synthesis algorithms. It supports multiple views and supports most, if not all, phases of the development process. GME does not provide an automatic multi-view synchronization that reinterprets changes in one view in the semantics of the other as SysWeaver does. GME is based on meta-models that have offline interpreters, while SysWeaver provides modeling blocks (couplers) that validates the model as it is being built. GRATIS and its predecessor GRATIS II are able to statically analyze, validate and translate the models of TinyOS programs into NesC executables. It does not provide a way of modeling interactions between applications,
VisualSense is a modeling and simulation framework that builds upon Ptolemy II [14] for wireless sensor networks. Ptolemy provides models of computation with which the user can construct a system. Most of these models of computation support actor-oriented design. Actors are software modules that communicate with other actors through events. Actors have ports, and the port connections specify the communication parties. The execution of a model in a system is defined by a director. Each model has a director which specifies the semantics of the actor graph. A model can, in turn, be encapsulated in an actor by defining an interface. The execution of this model is then controlled by the director of the model into which it is inserted. VisualSense provides a means for defining the channels for sensor node communication as well as sensor node attributes. The framework permits the integration of additional node and channel models written in Java. Though useful for modeling sensor systems, VisualSense does not have a direct path towards code generation that can run on a real hardware platform.

Viptos [15] connects VisualSense with TinyOS and Tossim allowing graphical models of sensor networking applications to be automatically generated and deployed on real hardware. Viptos maintains the ability to connect Ptolemy II components with the TinyOS network which allows the introduction of non-TinyOS nodes. Viptos focuses on design and simulation of a single application system while SenseWeaver allows the modeling and composition of multiple applications that share a common network. SenseWeaver also introduces the notion of target instrumentation in order to provide its model with information from the real network.

III. WORKFLOW

In this section, we introduce the steps in our proposed workflow for the design and implementation of a WSN system. Figure 1 shows the various components of SenseWeaver and their interactions. The workflow iterates between the model and various actions that iteratively add detail to the model. For example, a developer might capture topology information and then test to see if application timing parameters are met. If they fail, the designer can either re-visit the timing parameter, or try adjusting the topology by adding more nodes and then re-run the analysis tools.

At a high level, the SenseWeaver workflow consists of the following steps:

1) Model functional requirements of applications,
2) Introduce initial network topology data from the system into model,
3) Model physical attributes of network,
4) Synthesize system parameters which achieve or satisfy requirements,
5) Analyze the system based on user-input and synthesis output,
6) Repeat steps if necessary to satisfy specified functional requirements and system constraints, and
7) Generate Code and Deploy the application.

This design cycle is shown in Figure 2 with the main steps shown in rectangular boxes. The ovals show the supporting functionality that the SenseWeaver plug-in adds to SysWeaver.

A. Modeling

Model-based software design for wireless sensor networks aims to target those key areas of embedded systems which apply to a large-scale networks. Specifically, we look to achieve 1) composability and scalability, 2) multiple behavior encapsulation, 3) usability, 4) communication infrastructure and 5) correctness by construction. A model which can satisfy the requirements for these areas results in an efficient system design workflow and serves as the central component which interacts with other components in SenseWeaver. Visualization of the model is extremely useful for helping the designer navigate and interact with the model. In our workflow, we use the SysWeaver tool [16] to help us satisfy the above mentioned model-based design objectives.
SysWeaver provides abstractions to model both functional and para-functional behaviors into separate views whose interactions are automatically handled. Each of these views emphasizes a single concern enabling different domain experts (e.g. signal processing experts, control experts, real-time experts, fault tolerance experts) to focus on the concern of their expertise leaving the interactions with the other views to be automatically handled by the tool. The interactions among views are managed by maintaining a single internally consistent model, and treating each view as a partial projection of that model using a view-specific filter designed to only show elements and abstractions relevant to that view.

We now look at the requirements of the different modeling aspects as pertaining to Wireless Sensor Network design and how the primitives in SysWeaver enable us to model these requirements.

1) Functional Modeling: The functional model consists of the different applications in the system along with their interactions. This includes representations for periodic and aperiodic tasks, as well as a description of interfacing between the various tasks. We require a representation for the tasks, which can be defined as blocks, and the interfacing between them, which we call links. A system would consist of instantiations of blocks along with links "wiring" them together. For wireless sensor networks, we would need to make sure that blocks can model periodic tasks, event-triggered tasks, as well as data-flow tasks. A block should have input and output interfaces and should be able to support multiple threads of execution. Links should contain information about what they communicate (e.g. message sizes). Blocks should be composable and reusable so that multiple instances of the same type of block can be made.

SysWeaver uses the notion of a Component to represent software modules. The main pieces of each Component are Ports, ApplicationAgents and Couplers which are used to model the system and its interactions. A Port is the interface through which components communicate with each other. There are input ports which receive data and output ports which transmit data. A data transmission is represented as an event. Events are entities which are communicated across components and they can represent any data structure. The application code is represented inside an ApplicationAgent. An ApplicationAgent is a set of functions which process and generate events. The Ports contain entities called ProtocolAgents which are responsible for communicating the data between components and contain the mechanisms to do so. For example, if components are on a single processor using shared memory, the ProtocolAgent uses function invocation as communication. Conversely, if components are on different processors, the ProtocolAgent uses inter-processor communication based on the network protocol property. ProtocolAgents can contain multiple threads of execution as deemed necessary by the designer. ApplicationAgents are reusable and can be hierarchically structured to satisfy the composability requirement.

2) Physical Modeling: Wireless sensor networks have the property where changes in the physical environment greatly impact the system. Therefore, the system model needs to be aware of these physical properties and constraints and needs to capture these attributes. For example, node location, physical obstacles and infrastructure information play an integral role in how routes and communication schedules should be designed.

The Physical View in SysWeaver is used to capture the physical properties of the system. It can be used to model the sensor nodes and their location within the infrastructure as well as information about the infrastructure itself. An editor is used to create the infrastructure layout and nodes can then be positioned within the layout. This information is communicated to the underlying semantic layer. Any changes in the layout are automatically conveyed to the semantic layer to analyze the impact on the system. For example changes in how devices interact with the networking structure on the physical view would also be reflected as changes in the deployment view. The data in the Deployment View or the Physical View can also be populated by interfacing with external tools. Figure 3 shows a screenshot of SysWeaver with the different views and components of a model. Here we see the physical view on the top, an event-flow view on the bottom left and the deployment view on the bottom right.

3) Deployment Modeling: The Deployment Model describes the hardware used in the system and needs to have all of the important details associated with the target hardware and interconnections. The communication mechanism used in the network (e.g. MAC Protocol, Link Layer Protocol) should be modeled and should support easy replacement. This applies to the underlying target platform which consists of hardware information as well as OS information, if any. The model enables architecture exploration for deployment purposes. The Deployment Model should also be able to communicate with the Functional model so that the appropriate communication and processing information as pertaining to the functional blocks is captured. This requires a mapping between functional blocks and the deployment blocks.

One unique property of WSNs is the notion of mobile devices that interact with the system and are also part of the system. These need to be part of the network, so they cannot simply be treated as system inputs, but do not have fixed locations. Many nodes have the property that the same application and networking code should run on multiple nodes within a network. The model should be able to easily support both of these properties.

Hardware information is easily captured by the Deployment model in SysWeaver using the notion of Couplers. Couplers are primitives that express relationships between entities. For example, a Network Coupler represents the network relationship between all the Node Couplers connected to it. Couplers also have properties associated with them. A Network Coupler can contain information about the underlying MAC Protocol and any changes to this are propagated to all other Couplers. The Deployment Model also contains a SensorNetwork Coupler which contains a graph and list of all the nodes on the network along with link information and topology information. This enables the
designer to deploy a group of tasks onto the SensorNetwork Coupler. This connection denotes that the group of tasks should run on all nodes in the network in addition to other tasks assigned to individual nodes. SysWeaver also has a Mobile Node Coupler which it uses to represent nodes in the network that do not have fixed locations. Depending on the designer and the underlying target platform relationship, this Coupler can be used to represent unique nodes and the properties can be used for code generation and analysis purposes.

B. Instrumentation

This part of the framework is critical for maintaining a tight coupling between the system model and the actual runtime network properties. By being able to integrate real network data into the model, the system can be optimized to accurately reflect the desired functionality and requirements. Getting hold of this data is a difficult task and requires support from the underlying target platform. The Instrumentation Phase can occur on multiple occasions. It primarily happens at the initial step in the workflow, where we collect the initial network topology. After that, instrumentation can be done after deployment to collect data while the system is actually running. It is important that the instrumentation code be very concise and non-interfering since it may execute during normal system operation.

Information required for instrumentation in SysWeaver is collected either through help from the target platform, or through custom instrumentation code which can be generated from within SysWeaver. The data can consist of network connectivity graphs as well as link strengths between nodes. The data can be used to update corresponding components that exist in the Physical View and add components which might not have existed in the view. This closed-in-the-loop design results in a tightly coupled system modeling and deployment. The instrumentation features in SysWeaver are extensible so that new data can be easily added.

C. Synthesis and Analysis

The power of model-based design is increased by the ability to do an efficient analysis of the system model. To support analysis, the model needs to encapsulate all the relevant information while effectively visualizing results. The kinds of analysis that a designer would be looking for are 1) node lifetimes, 2) end-to-end flow latencies, 3) network load hot-spots, 4) communication schedulability and 5) flow reliability.

The semantic model within SysWeaver gives us a way to store most of the information required for analysis. All Application Agents have timing and schedulability information such as deadlines and sampling periods. Based on the underlying platform, they can also be assigned priorities.
Each Application Agent has a Transition Table which defines the state machine within the component. Each entry in the Transition Table has a Trigger, an Action, and a Worst Case Execution Time (WCET) value for the entry. The Trigger indicates what causes the transition to occur, which in most cases is a result of an event arriving through an input port. A special type of trigger called a PeriodicTrigger indicates a periodic transition. The Action describes the event produced from the Trigger. The Action normally results in an event which is sent through an output port. The final event resulting from the Completion of a flow is designated as a CompletionEvent. Each Application Agent can have multiple entries in its Transition Table. The table entries have a many-to-many relationship to support all combinations of triggers and actions. Having this kind of structure within a component gives an indication of execution times within a flow as a result of interaction between components. To capture network latency, the Couplers which connect ports to each other have message sizes associated with them. This coupled with information about transmit and receive delays associated with the Node Coupler or the Network Coupler can be used to give network latencies and flow analysis. The timing information within each Application Agent along with the approximate size of receive and transmits done by each Application Agent gives us the resource usage of each Application Agent. By capturing this information in the Functional View, the Deployment View can then calculate node lifetimes since it has the mapping of the Application Agents deployed on each node. SysWeaver can export and import information from other tools which it can aggregate and provide to specific analysis engines.

Another property of the semantic model is that it can provide synthesis internally as well as by interfacing with external tools. Synthesis can be used to provide suggestions or estimate properties that the designer is trying to optimize. Coupled with the analysis engine, the synthesis engine can be a very powerful feature. It can provide insight into communication routes, node schedules, sampling periods for tasks and optimal locations for nodes. Being able to provide suggestions for system aspects can go a long way in helping the designer who may not be a WSN expert. The engine takes the model as an input along with the requirements which the designer is trying to meet and outputs the parameters that it can tweak to try and satisfy the requirements. Synthesis can invoke the analysis plug-in to verify if any of the constraints are being violated.

Figure 4 shows an example input and output file associated with a typical network topology that describes both the links and application-level communication requirements. "*" is a symbol reserved to represent the body of nodes in the system. A flow generated from "*" to a node, or from a node to "*" is an upstream or downstream communication specifically from all nodes to one or from one node to all. In this example configuration, the mobile node RSSI data could be generated from any node and must be aggregated at a single point. The details of the actual communication scheduling are beyond the scope of this paper, but in general we assume an underlying TDMA MAC protocol. The analysis engine then relies on greedy graph searching heuristics that attempt to order flows while satisfying a two-hop interference constraints. Since each task in our model was given a priority and worst-case execution time, we can use well known real-time scheduling theory to ensure feasible schedulability of tasks on each node. With TDMA-based communication, we know all communication patterns ahead of time allowing us to determine the worst-case latencies in the absence of packet loss and pre-compute blocking times. Finally, by combining the worst-case execution times of tasks along with scheduled communication, we can accurately predict the worst-case energy consumption and hence the battery life of a node. The analysis engine can check these computed values against the parameters specified as properties in the design to alert the designer of inconsistencies. Even on a small system shown in our example, the scheduling complexity becomes difficult to manage by hand, making automated synthesis essential.

D. Deployment

The Deployment phase of the workflow involves gathering the implementation of the system as modeled. Different kinds of deployments can result from a single consistent model. Code for simulation as well as for the target hardware can be generated by adjusting deployment preferences. The implementation of the deployment phase involves building library blocks for the different kinds of deployment and for different simulators as well as different target hardware. Using the SysWeaver approach, each Coupler becomes a library block, wherein the Protocol Agent is the code that handles communication between components, the Application Agent code is the interface with the user application code, and there are StateChangeEvent handlers which handle the relaying of events between the Protocol Agent and the Application Agent. SysWeaver is used to generate the "sys-code" which glues together the coupler libraries with the generated communication code, and user-specific code. Having a library built for each type of deployment enables code generation for different platforms in the same system. Simulation is represented as a Deployment target which uses the coupler libraries built for the different simulators.

The SenseWeaver plug-in generates code that can be compiled to run on the nano – RK real-time operating system (RTOS) described in [17]. Nano-RK is a fully preemptive RTOS with multi-hop networking support that runs on a variety of platforms. It supports fixed-priority preemptive scheduling for ensuring that task deadlines are met, along with support for and enforcement of CPU and network bandwidth reservations. Tasks can specify their resource demands and the operating system provides timely, guaranteed and controlled access to CPU cycles and network packets in resource-constrained embedded sensor environments. It also supports the concept of virtual energy reservations that allows the OS to enforce energy budgets associated with a sensing task by controlling resource accesses. Nano-RK provides various MAC and networking protocols including a low-power-listen CSMA protocol called B-MAC [18], an
implicit tree routing protocol and a TDMA based protocol called RT-Link [19].

Due to the energy constraints and the desire for analyzable timing properties, we opted to use the TDMA network protocol, where all packet exchanges occur in well-defined time slots. Each node in the system must be given a time slot schedule that coordinates with its neighbors. Given a network link topology, using distance two graph coloring, it is possible to generate a schedule that is collision-free and avoids the hidden terminal problem. Given information about flows in the system, it is possible to further optimize schedules such that nodes are able to sequentially forward data within a single TDMA cycle.

**IV. CONCLUSION AND FUTURE WORK**

In this paper, we introduced SenseWeaver, a SysWeaver plug-in that supports model-based design of wireless control applications. We showed how a top-down model-based design approach for building wireless sensor-actuator networks can manage complexity as well as enable the automatic integration of multiple applications. As component libraries become increasingly mature, system development will be able to cleanly reuse code. This will not only reduce the amount of hand written code required for an application, but also reduce development time and increase system reliability. Having a plug-in framework with a well-defined interface gives system designers more choices in composing, analyzing and deploying systems which will make future systems more structured and easily amenable to change.

In the future, we plan to incorporate existing simulation tools as well as develop more sophisticated synthesis and analysis engines. We also hope to expand upon the heterogeneous nature of SenseWeaver by supporting more platforms and networking protocols. For SenseWeaver, we are developing an expert system which is a rule-based design engine that provides the designer with verification and analysis capabilities to compose systems. This involves the definition of a rules interface as well as support to encapsulate possible sets for rules which can be composed together. The interface of the Component Designer is being updated to add more custom properties as required by applications. This should help to increase the number of reusable components.

**REFERENCES**


An Adaptive Discrete Event Model for Cyber-Physical System

Ke Yue, Li Wang, Shangping Ren
Department of CS
Illinois Institute of Technology
Chicago, IL 60616
{kyue.liwang64.ren}@iit.edu

Xufei Mao
Beijing Key Laboratory of Intelligent Telecommunications Software and Multimedia
Beijing University of Posts and Telecommunications
maoxufei@bupt.edu.cn

Xiangyang Li
Institute of Computer Application Technology
Hangzhou Dianzi University
xli@cs.hz.zj.cn

Abstract—Cyber-Physical Systems (CPS) often involve a wide spectrum of events, ranging from lower-level signals to higher-level abstract events. In order to compose different levels of events, a Concept Lattice-based Event (CLE) model was developed. In the CLE model, the traditional first order logic is used in specifying rules composition. However, it is possible that the rules specified by the first order logic may have inconsistency. Furthermore, unanticipated events that are not considered in the initial design may affect systems' performance, or even lead to system failure. In order to address these issues, an Adaptive Discrete Event (ADE) model is proposed in this paper. ADE model uses Discrete Event Calculus (DEC) to overcome possible inherent inconsistencies in composition rules that are specified by first order logic. In addition, we define abnormal event rules as an adaptive part in the CPS event model to handle unanticipated events. Finally, a CPS application “iLight” is developed based on the ADE model, results show that “iLight” adapts to the new environment successfully.

Index Terms—Cyber-Physical Systems; discrete event model; adaptive model

I. INTRODUCTION

Cyber-Physical Systems (CPS) often involve a large spectrum of events ranging from lower-level physical signals to higher-level abstract events. Correct system functioning relies on the understanding of these events and the ability to infer or reason about information from these events. However, due to the wide variety of events across different abstraction levels, defining an event model that not only reflects the abstraction diversity of events, but also, at the same time, provides an effective mechanism to reason about the events at different levels, becomes a challenging issue.

Commonsense reasoning [1] is an approach that enables a system to make inferences by utilizing events obtained from different levels. The functionality of commonsense reasoning is to infer, or reason about, what event will happen next, and how to react to such event. The foundation of the reasoning process lies in the rules designed in advance which are based on human everyday knowledge. For example, a human being cannot cross the wall and a broken bulb cannot light again. Both examples belong to the everyday knowledge used in people’s daily commonsense reasoning process. Adopting such reasoning process into the CPS event model will make the system more responsive to the happenings of events, and we need to take it into consideration in our design procedure.

However, commonsense reasoning using first order logic [2] may contribute to inherent inconsistency. The crux of the inconsistency problem is that multiple event composition rules may be related to the same event. In other words, the same event may result in two different conclusions. To see a simple scenario where the reasoning result of an event is not consistent due to manifold reasoning rules, consider a smart home system with two lights. One rule may state that all the lights must be on at time point \( t + 1 \) when a person comes into the room at time point \( t \). However, at the same time, to save energy, another rule states that if two lights are on at the same time at time point \( t \), one of the light must be turned off at time point \( t + 1 \). Imagine a case that a person comes into the room at time point \( t \), clearly, the two rules will result in contradiction with each other.

In a real world, environment states may not hold forever. Therefore, we need a mechanism to model environment changes and rules to specify how CPS application responds to such changes. Many existing event models, such as OWL-DL ontology based event model [3], OWL-S ontology based event model [4], CLE model [5], did not focus on the dynamic nature of environment in which a CPS usually operates. Environment changes will introduce a myriad of unanticipated events which may not be taken into consideration at the beginning of the design. These unconsidered events may impair the ability of a CPS to work properly and may even lead to system breakdown.

To overcome the issues discussed above, we have developed an Adaptive Discrete Event (ADE) model. The ADE model incorporates the Discrete Event Calculus (DEC) [6], [7], rather than first order logic, to compose events. DEC has been proved successful in solving inherent inconsistency problem [1]. Moreover, the abnormal reasoning set is adopted to dynamically have the unanticipated events. By providing corresponding mechanism for the unanticipated events, the...
CPS can be easily adapted to the new environment. The main contributions of this paper are twofold: first, we introduce DEC in our Adaptive Discrete Event (ADE) model to define composition rules across vertical and horizontal domains. Predicates in DEC form the basis of solution to the inherent inconsistency problem existed in the traditional first order logic that was used in our previous concept-lattice based event model [8]. Second, the adaptability of CPS application is improved by introducing abnormal reasoning set which enables the CPS to handle unanticipated events.

The rest of this paper is organized as follows: Section II compares the related work. Section III formally defines the ADE model. The developed model is applied in a case study called “iLight”, which is illustrated in Section IV. Finally, we conclude this paper and point out future work in Section V.

II. RELATED WORK

The event concept has been investigated from different aspects. For instance, Mikhail et al. [9] define an event as a detectable action performed during the program execution, such as a statement execution, a procedure call, or a message transmission, etc. Luckham et al. [10] refer component interactions as event. Auguston’s definition of event [11] is similar to the one in [10], but it is also used to form a behavior model for debugging and testing automation tools. As CPS applications are different from the applications above, they not only involve actions performed at the physical level, but also at network and computer level, even possibly at human level, the general event concepts mentioned above cannot accommodate such needs. One of our work in this paper is to develop a generalized event structure that can accurately specify events at different levels.

In order to handle events, work in [12] [13] uses first order logic to reason about the CPS events acquired by the system. However, first order logic cannot fully represent the semantic meaning of the commonsense reasoning. Mueller et al. [14] propose Discrete Event Calculus (DEC) to specify complex relationships among events. Although other computational approaches, such as situation calculus [15] and fluent calculus [16], are able to specify event relationships, DEC is superior because it addresses the rule inconsistency problem more efficiently. However, the mathematical definition of an event is not given in DEC.

Knowledge graph systems [17] are able to infer boolean values about the system state based on specific inputs and rules. However, knowledge graph systems do not focus on the dynamic aspects. Event models in previous work [8] [18] only focus on static and presumed events. However, in a real environment, unexpected events may happen at any time and may lead to system malfunction. So it is essential to design a dynamic and adaptive event model to model CPS applications so that they can be easily tailored to a new scenario.

III. ADAPTIVE DISCRETE EVENT MODEL

For CPS applications, when and where an event occurs are crucial information. This is because we must know precedence order of different events to get temporal relationship among them. Therefore, in our early work [8], we introduced temporal and spatial information into event structure to completely capture the characteristics of CPS events. For self-completeness, we restate the definition in the following subsection.

A. CPS Event Instance

Definition 1 (CPS Event Instance): A CPS event instance consists of an event type and a set of attributes. It is structured as:

$$E_{cps}: \Gamma \mu @ (T, L, O)$$  \hspace{1cm} (1)

where,

- \( \Gamma \) represents the type name of the event instance.
- \( \mu \) represents the attribute set of the event instances and has the form of \( \mu = (\mu_1, \mu_2, \ldots, \mu_i) \) where \( \mu_1, \mu_2, \ldots, \mu_i \) indicates the attribute of the event instance.
- \( T \) indicates the time point in reference to the event observer when the event instance happens.
- \( L \) indicates the location in reference to the event observer where the event instance occurs.
- \( O \) is an observer of the event instance. The existence of an observer is also treated as an event.
- \( O_T \) is the global observer of a CPS system. Its location is the system’s origin, and its time interval is defined as the system’s life span. There is only one global observer in a given CPS system.

In summary, Definition (1) defines an event with time and location attributes detected by the observer.

B. Simplified Discrete Event Calculus

In the definition of Discrete Event Calculus (DEC), there are three types: event, fluent, and time point. The concept of fluent used in DEC is the time-varying properties of the world. Actually, as pointed out in [19], event and fluent can be considered to be the same if temporal attribute is added to event. Moreover, as our focus is on reasoning events, we restrict ourselves to the subset of DEC. More precisely, we define a simplified DEC with only the following primitives:

- **HoldsAt(e,t):** This means that an event happens and holds at time point \( t \).
- **Initiates(e_1,e_2,t):** Event \( e_1 \) triggers another event \( e_2 \) to hold at time point \( t + 1 \).
- **Terminates(e_1,e_2,t):** Event \( e_1 \) triggers another event \( e_2 \) not to hold at time point \( t + 1 \).
- **ReleasedAt(e,t):** Event is released from the previous state at time point \( t \). That is to say, the state of event \( e \) can be changed at time point \( t \).
- **Releases(e_1,e_2,t):** Event \( e_1 \) releases another event \( e_2 \) from the previous state at time point \( t + 1 \). In other words, event \( e_1 \) will make the state of event \( e_2 \) at time point \( t + 1 \).

Among these predicates, **ReleasedAt(e,t)** and **Releases(e_1,e_2,t)** are the crux of the solution to the
problem of inconsistency. The cause of the inconsistency problem is that an event’s state may be affected by more than one rule simultaneously. By using these two predicates $\text{ReleasedAt}(e, t)$ and $\text{Released}(e_1, e_2, t)$, an event’s state can not be changed if it is not released from previous state. In other words, these predicates provide a lock mechanism for the event’s state.

By depicting an event’s state at a time point, the predicates above form the basis of our reasoning process. Based on these predicates, we define the rules set $\ell$ to compose events. CPS can make a deduction and control the system by using these rules. The example below illustrates how simplified DEC is used in our model.

**Example 1:** Let’s assume a scenario in a smart home, if a person comes into the room and the light is dim, we should turn on the light. We define three events, `PersonIn`, `RoomDim`, `TurnOnLight`. The formal definitions are listed below and spatial-temporal information is under the same observer $O_1$.

- **PersonIn**
  - $(\text{height}@O_1)$
  - $(\text{lightstrength}@O_1)$
  - $(\text{TurnOnLight}@O_1)$

Based on `PersonIn`, `RoomDim` events, we want the smart home system to generate a result event `TurnOnLight` and perform corresponding control. The reasoning rules using simplified DEC are defined as:

$$
\text{HoldsAt}(\text{PersonIn}, t) \land \text{HoldsAt}(\text{RoomDim}, t) \Rightarrow \text{Initiates}(\text{PersonIn}, \text{TurnOnLight}, t)
$$

When the circumstance in which a CPS is designed for is changed, new events which are not considered in the initial design may occur, making some predefined rules in the system invalid. Let’s take the reasoning rule (3) as an example. Assume that the left three predicates, i.e., $\text{Initiates}(\text{PersonIn}, \text{TurnOnLight}, t)$, $\text{ReleasedAt}(\text{TurnOnLight}, t)$ and $\text{HoldsAt}(O_1, t)$, are true. According to the reasoning rule (3), the predicate $\text{HoldsAt}(\text{TurnOnLight}, t + 1)$ should hold at time point $t + 1$, which indicates that we must take certain control mechanism to turn the light on at time point $t + 1$. However, what if the light is not on? It is intuitive to make an inference that there must exist some unexpected events which lead to the system failure, for instance, the bulb is broken. We call these unexpected events as abnormal events. There might be numerous events belonging to abnormal events because CPS failure can be caused by a wide variety of events. In addition, abnormal events usually happen when a system operates in a new environment. Therefore, rule (3) may be not suitable if abnormal events exist. We must change the rules to accommodate various environments and make CPS applications more adaptive. In order to achieve this goal, we introduce an abstract concept event $E_{ab}$ to generalize all the abnormal events. The concept event $E_{ab}$ means that the CPS’s output event is not consistent with initial expected event.

Initially, the predicate $\text{HoldsAt}(E_{ab}, t)$ is false and we assume that our CPS works properly. The occurrence of abnormal events will make predicate $\text{HoldsAt}(E_{ab}, t)$ to be true. So we have to take certain mechanism to make $\text{HoldsAt}(E_{ab}, t)$ false and prevent CPS from crashing. Hence, we add the predicate $\text{HoldsAt}(E_{ab}, t)$ to our previous rule (3) and get the new rule:

$$
\text{Initiates}(\text{PersonIn}, \text{TurnOnLight}, t) \land \
\neg \text{HoldsAt}(E_{ab}, t) \land \
\text{ReleasedAt}(\text{TurnOnLight}, t) \land \text{HoldsAt}(O_1, t) \Rightarrow \
\text{HoldsAt}(\text{TurnOnLight}, t + 1)
$$

In this new rule, $\text{HoldsAt}(E_{ab}, t)$ must be guaranteed to be false to get the expected event and make $\text{HoldsAt}(\text{TurnOnLight}, t + 1)$ true at next time point.

When the system detects that a light is not on when a person is coming in, from the above rule, we can infer that an abnormal event, such as, the bulb is broken, may have occurred. In other words, the abnormal event that the bulb is broken leads to system failure:

$$
\text{HoldsAt}(\text{BulbBroken}, t) \Rightarrow \text{HoldsAt}(E_{ab}, t)
$$

The new rule shows that we must ensure $\text{HoldsAt}(\text{BulbBroken}, t)$ to be false to make $\text{HoldsAt}(E_{ab}, t)$ false. In terms of implementation, we must provide a mechanism, i.e., providing backup bulb, to prevent the event that a bulb is broken from bringing the system down. We can add similar rules which may make
HoldsAt(E_{ab}, t) \text{ true and prepare corresponding mechanisms to handle those abnormal events. By adding more abnormal event rules to the system, the chance that CPS is affected by abnormal events is decreased. Thus, the system adaptability is enhanced by utilizing this methodology.}

We call the rules designed for abnormal events as abnormal rules set $\gamma$. In this case, $\gamma = \{6\}$. By combining the abnormal rules set $\gamma$ and $E_{ab}$, we get the abnormal reasoning set $\zeta$. We add $\zeta$ to the observer definition, so each observer has its own abnormal reasoning set. Now we define the observer’s formal mathematical structure as follows:

**Definition 2 (CPS Observer Event Instance):** A CPS observer event instance is composed of an event type and a set of attributes. It is structured as:

$$E_{obs} : \Gamma \mu@\langle T, L, O \rangle$$

where,

- $\Gamma$, $T$, $L$ have the same meaning with those in Definition 1.
- $\mu$ represents the attribute set of the event instances, and we add abnormal reasoning set $\zeta$ to this attribute set. $\zeta = \{\gamma, E_{ab}\}$. $\zeta$ is a set containing observer event’s abnormal event reasoning rules $\gamma$ and abnormal event $E_{ab}$.
- $O$ is an observer providing observer event, the existence of an observer is also treated as an event. For example, a sensor node can act as an observer under which all the events have the same relative geographical and temporal coordinates. This sensor is also observed by another sensor acting as an observer. Although several observers are allowed in a CPS system, the global observer $O_T$ must be unique and serves as the system’s coordinate and wall-clock.

Based on the discussions above, we provide the definition of Adaptive Discrete Event model for CPS as follows:

**Definition 3 (ADE Model):** An Adaptive Discrete Event (ADE) Model is a structure:

$$M = \langle X, Y, O, E, \ell \rangle,$$

where,

- $X$ is the set of all the input events.
- $Y$ is the set of all the output events.
- $O$ is observer event sets.
- $E$ is the set of all the events.
- $\ell$ is the event reasoning rules set.

In the next section, we will design an ADE model for the CPS application “iLight”. The events and the rules will be explained in detail in the “iLight” system.

**IV. AN APPLICATION FOR ADE MODEL**

The “iLight” is a passive tracking Cyber-Physical System. It is used to track a moving target by using multiple groups of sensors, and automatically compute the target’s moving patterns such as height, moving speed, etc. Sensors will get raw data which are deemed as input events, and based on these events, the system can make a deduction regarding whether there is a person coming in, whether he/she is an adult or child, and judge whether he/she is walking or running. When “iLight” system is migrated to a new environment, unexpected events may prevent the system from functioning properly. Therefore, for the purpose of adaptability, rules defined in the observer event can add or delete abnormal events dynamically, making “iLight” adaptive to different environments.

**A. “iLight” System Background**

The “iLight” tracking system uses light sensors and light sources to track moving objects. Light sensors are sensitive to the light change around them. When a moving object comes across the light source and the light sensor, the light level (photo value) will decrease sharply as Figure 1 illustrates. Once a person comes across at 10 second, the photo value sensed by the light sensor decreases from 50 to less than 20. By detecting such event, we can make a simple deduction that a person is coming in.

As shown in Figure 2, two groups of sensors $\{A_1, A_2, A_3, A_4\}$ and $\{B_1, B_2, B_3, B_4\}$ are deployed on either side of the wall. Moreover, $a$ and $b$ are two light resources. Once a moving object with height $h$ goes between two groups of sensors, the photo values of sensors $A_2, A_3, A_4, B_2, B_3, B_4$ will decrease dramatically and the photo value of $A_1, B_1$ will be almost the same as before. Now segments $aB_1, aB_2, bA_1, bA_2$ will intersect at point $C, D, E, F$ which form a quadrilateral. As the distance between the highest point of the person and intersection of $CF, DE$ is very small, we can regard these two points as the same. As the coordinates of vertices of $a, b, A_1, A_2, A_3, B_1, B_2$ are known, it is easy to calculate the coordinate of intersection between $CF$ and $DE$. Hence the moving object’s height can be calculated.

The “iLight”Test bed includes 41 wireless sensor nodes and one of them is chosen as sink node and global observer. Ten light sources are installed to provide light, and a base station is used to monitor the data. 40 sensors are divided into 10 groups and each group has 4 sensors respectively. One sensor is randomly chosen from the group acting as local observer. Two groups of sensors are deployed on the opposite side of the wall as shown in Figure 3. Each black node refers to one group of sensor nodes. Red nodes in the graph refer to the light sources. The red dash lines indicate the laser-like narrow
low-divergence light beams. The black curve is the trajectory of the moving object. Little black circle is the location where target is detected by the light sensors.

In addition, under certain situations, the sensor may behave abnormally, and the photo value may decrease even there is no person coming across due to the hardware constraints and background noise ($\mathcal{E}_{ab}$). Light sensor will report such event due to error reading especially when there is too much background noise. So we can use 10 groups of sensors and every two groups deployed on the opposite side of the wall to monitor the events. Only two sensors’ photo value decrease within same range, can there be a real event happening, and the alarm event will be triggered to report an intruder.

![Fig. 3. Vertical views of the monitored area](image)

Table 1 summarizes the main parameters of the “iLight” tracking system. For example, we use five people with different heights as the tracked targets. In this case, we let each target go through the monitored area and the system must provide a concrete event description from the sensed datum.

Based on the discussion above, we now begin to design the ADE model. Based on this model, CPS must be able to accurately detect an intruder based on the events derived from the light sensors. In order to be more accurate, the decision regarding whether there is an intruder must be made based on the results of two groups $G_1$ and $G_2$. One group of sensors are deployed in the way shown in Figure 2. We also define the sensors with the highest position whose light values changed as $A^1_4, B^1_4$ in group $G_1$ and $A^2_4, B^2_4$ in $G_2$. The defined events are as follows:

\begin{itemize}
  \item $A^1_4GSenDecrease (id, PValDed, coord) @ (T_0, L_0, O_{b0})$
  \item $B^1_4GSenDecrease (id, PValDed, coord) @ (T_1, L_1, O_{b1})$
  \item $A^2_4GSenDecrease (id, PValDed, coord) @ (T_2, L_2, O_{b2})$
  \item $B^2_4GSenDecrease (id, PValDed, coord) @ (T_3, L_3, O_{b3})$
  \item $G_1PDecrease (id, height) @ (T_4, L_4, O_{b4})$
  \item $G_2PDecrease (id, height) @ (T_5, L_5, O_{b5})$
  \item $G_1PersonIn (id, height) @ (T_6, L_6, O_{b6})$
  \item $G_2PersonIn (id, height) @ (T_7, L_7, O_{b7})$
  \item $PeopleWalkIn (speed) @ (T_8, L_8, O_\tau)$
  \item $PeopleIsAdult (height) @ (T_9, L_9, O_\tau)$
  \item $SinglePersonIn (speed, height) @ (T_{10}, L_{10}, O_\tau)$
  \item $Alarm (soundstrength) @ (T_{11}, L_{11}, O_\tau)$
\end{itemize}

Property id of each event refers to the id number assigned to each group. $PValDed$ is the decreased photo value and coord is the coordination needed to calculate the top point of a human being who comes across the light sensor’s detected area. Property height, speed refers to the human being’s height and speed. The observer is used to synchronize the time among the sensors.

\begin{itemize}
  \item $O_{b0}(\zeta_0, id) @ (T_0, L_0, O_{b4})$
  \item $O_{b1}(\zeta_1, id) @ (T_1, L_1, O_{b6})$
  \item $O_{b2}(\zeta_2, id) @ (T_2, L_2, O_{b5})$
  \item $O_{b3}(\zeta_3, id) @ (T_3, L_3, O_{b7})$
  \item $O_{b4}(\zeta_4, id) @ (T_4, L_4, O_\tau)$
  \item $O_{b5}(\zeta_5, id) @ (T_5, L_5, O_\tau)$
  \item $O_\tau = obs(\zeta_6)@[0, \infty), (0, 0, 0), \infty, T)$
\end{itemize}

In this case, $\tau$ denotes the system itself and $obs$ denotes this observer’s name. Property id is the id number assigned to each sensor which acts as an observer. First, we assign a value for the decreased photo value sensed by the sensors, say between 30 to 50. Now, based on these events, we define the rules as follows:

$$30 \leq value \leq 50 \land HoldsAt(O_{b0}, t) \Rightarrow HoldsAt(A^1_4GSenDecrease, t)$$  \quad (9)
\[ 30 \leq \text{value} \leq 50 \land \text{HoldsAt}(\text{Ob}_1, t) \Rightarrow \\
\text{HoldsAt}(B^1_2 \text{GSenDecrease}, t) \] 
(10) 
\[ 30 \leq \text{value} \leq 50 \land \text{HoldsAt}(\text{Ob}_2, t) \Rightarrow \\
\text{HoldsAt}(A^1_2 \text{GSenDecrease}, t) \] 
(11) 
\[ 30 \leq \text{value} \leq 50 \land \text{HoldsAt}(\text{Ob}_3, t) \Rightarrow \\
\text{HoldsAt}(B^2_2 \text{GSenDecrease}, t) \] 
(12) 
\[ \text{HoldsAt}(A^1_2 \text{GSenDecrease}, t) \land \\
\text{HoldsAt}(B^1_2 \text{GSenDecrease}, t) \land \\
\text{HoldsAt}(\text{Ob}_4, t) \land \neg \text{HoldsAt}(\zeta_4, \text{E}_{ab}, t) \Rightarrow \\
\text{HoldsAt}(G_1 \text{PDecrease}, t) \] 
(13) 
\[ \text{HoldsAt}(A^2_2 \text{GSenDecrease}, t) \land \\
\text{HoldsAt}(B^2_2 \text{GSenDecrease}, t) \land \\
\text{HoldsAt}(\text{Ob}_4, t) \land \neg \text{HoldsAt}(\zeta_5, \text{E}_{ab}, t) \Rightarrow \\
\text{HoldsAt}(G_2 \text{PDecrease}, t) \] 
(14) 
\[ \text{HoldsAt}(G_1 \text{PDecrease}, t) \land \\
\text{Initiate}(G_1 \text{PDecrease}, G_1 \text{PersonIn}, t) \land \\
\text{ReleaseAt}(G_1 \text{PersonIn}, t) \Rightarrow \\
\text{HoldsAt}(\text{Ob}_4, t) \Rightarrow \\
\text{HoldsAt}(G_1 \text{PersonIn}, t + 1) \] 
(15) 
\[ \text{HoldsAt}(G_2 \text{PDecrease}, t) \land \\
\text{Initiate}(G_2 \text{PDecrease}, G_2 \text{PersonIn}, t) \land \\
\text{ReleaseAt}(G_2 \text{PersonIn}, t) \land \\
\text{HoldsAt}(\text{Ob}_4, t) \Rightarrow \\
\text{HoldsAt}(G_2 \text{PersonIn}, t + 1) \] 
(16) 
\[ \text{HoldsAt}(G_1 \text{PersonIn}, t) \land \\
\text{HoldsAt}(G_2 \text{PersonIn}, t) \land \\
\text{ReleaseAt}(\text{SinglePersonIn}, t) \land \\
(G_1 \text{PersonIn}.\text{height} = G_2 \text{PersonIn}.\text{height}) \land \\
\text{HoldsAt}(\text{O}_\top, t) \Rightarrow \\
\text{HoldsAt}(\text{SinglePersonIn}, t) \] 
(17) 
\[ \text{HoldsAt}(G_1 \text{PersonIn}, t) \lor \\
\text{HoldsAt}(G_2 \text{PersonIn}, t) \lor \\
\text{ReleaseAt}(\text{PeopleWalkIn}, t) \lor \\
((G_1 \text{PersonIn}.\text{speed} \leq 1) \lor \\
(G_2 \text{PersonIn}.\text{speed} \leq 1)) \lor \\
\text{HoldsAt}(\text{O}_\top, t) \Rightarrow \\
\text{HoldsAt}(\text{PeopleWalkIn}, t) \] 
(18) 
Based on the rules above, we get the composition rule set:
\[ \ell = \{(9), (10), (11), (12), (13), (14), \\
(15), (16), (17), (18), (19), (20), \} \]
Consider the case that we need to migrate “iLight” to outdoor environment where CPS may malfunction due to severe environment. We may find that the CPS gives an alarm when nobody is there. To find what event makes the system malfunction, CPS begins to diagnose itself. In our example, the CPS may compare the data obtained from different sensors and find some sensor’s data is different from the majority. Hence it may make a conclusion that this sensor is fake. Then new abnormal event SensorFake is added to CPS. Because the SensorFake event happens, the \text{HoldsAt}(\text{E}_{ab}, t) will be true indicating that this sensor is broken and the alarm will not sound anymore.

\[ \text{HoldsAt}(\text{SensorFake}) \Rightarrow \text{HoldsAt}(\zeta_4, \text{E}_{ab}, t) \] 
(21) 
\[ \text{HoldsAt}(\text{SensorFake}) \Rightarrow \text{HoldsAt}(\zeta_5, \text{E}_{ab}, t) \] 
(22) 
Therefore, we get abnormal reasoning set \[ \zeta = \{(21), (22), \text{E}_{ab}\}. \]
Now we define the event set used in our ADE CPS model. The event set \( E \) is:
\[ E = (A^1_2 \text{GSenDecrease}, B^1_2 \text{GSenDecrease}, G_1 \text{PersonIn}, \\
A^2_2 \text{GSenDecrease}, B^2_2 \text{GSenDecrease}, G_2 \text{PersonIn}, \\
G_1 \text{PDecrease}, G_2 \text{PDecrease}, \text{PeopleWalkIn}, \text{Alarm}, \\
\text{PeopleIsAdult}, \text{SinglePersonIn}, \text{Ob}_0, \text{Ob}_1, \\
\text{Ob}_2, \text{Ob}_3, \text{Ob}_4, \text{Ob}_5, \text{O}_\top) \]
(18) 
The input events set \( X \) is:
\[ X = (A^1_2 \text{GSenDecrease}, B^1_2 \text{GSenDecrease}, \\
A^2_2 \text{GSenDecrease}, B^2_2 \text{GSenDecrease}) \]
The output events set $Y$ is:

$$Y = (\text{PeopleWalkIn}, \text{PeopleIsAdult}, \text{SinglePersonIn}, \text{Alarm})$$

The observer events set $O$ is

$$O = (\text{Ob}_{b_1}, \text{Ob}_{b_2}, \text{Ob}_{b_3}, \text{Ob}_{b_4}, \text{Ob}_{b_5}, \text{Ob}_T)$$

Therefore, the ADE model is $M = (X, Y, O, E, \ell)$, and the model is illustrated in Figure 4.

![ADE model diagram](image)

**V. CONCLUSION**

In this paper, we present an Adaptive Discrete Event (ADE) model for Cyber-Physical System (CPS) to deal with the rule inconsistency problem and unexpected events issue. In particular, a simplified DEC to compose different levels of events is used, and we further introduce abstract concept event to capture unexpected events caused in heterogeneous environments in which a CPS application operates. Moreover, abnormal reasoning set is added into the common sense reasoning rules to makes the system adaptive to different environments. In our current work, the abductive process for system to perform the self diagnosis is, nevertheless, not formulated in the rules, and it is our future research focus.

**REFERENCES**

Analytical System Composition

Position Paper

Lui Sha and Jose Meseguer
University of Illinois at Urbana-Champaign
lrs, mesequer@cs.uiuc.edu

1 Introduction
The convergence of sensing, control, communication and coordination in cyber-physical systems such as modern airplanes, power grid, train and medical device networks poses enormous challenges because of their complexity caused by:
• Distributed and concurrent interactions between the cyber and physical subsystems.
• Strong QoS requirements including real-time, fault tolerance, security and their composition.
• Dramatic losses, including losses of human lives, when these systems malfunction and/or penetrated by cyber–terrorism.

System integration is one of the most challenging aspects in the development of distributed CPS. This phase often consumes 70% of the total development budget. To succeed, we need to develop an analytical system composition framework based on AADL[11].

2. The Challenges
From the perspective of system integration, interaction faults dominate. Interaction faults lead to a recurring phenomenon in system development: a stormy system integration phase is often preceded by a relatively calm unit development and testing phase. This is because early in a project, most of the application functional bugs can be localized within individual modules. Unfortunately, interaction bugs in cross-cutting properties such as real-time, safety, and security are unlikely to show up at early stages. As integration proceeds, the degree of concurrency and the complexity of interaction increase exponentially. This triggers the emergence of interaction bugs, generating second and third order faults that may crash the system and leave a few clues to discover their root causes. This makes system integration phase to often consume 70% of the total budget. This is a major contributor to cost and schedule overruns. This is the reason we focus on an analytic system composition framework designed to reduce interaction complexity and to support automated analysis for composed systems.

2.1 Making Assumptions Explicit and Machine Checkable
In embedded systems, the assumptions of the operational environment are embedded in the software design. In the current software engineering practice, the assumptions are often stated informally in documents and are sometimes left implicit. When a legacy component is reused in a new environment, inconsistency between the assumptions embedded in the software and the new environment can lead to serious problems.
• The EU Ariane 5 rocket explosion during launch was traced back to the reused Ariane 4 software module, which relied on the rocket speed dependant fact that the horizontal speed variable was “physically impossible to overflow.” This assertion was correct for Ariane 4 but, unfortunately, incorrect for Araine 5. http://www.around.com/ariane.html
• Patriot anti-missile system failure during the Gulf War was caused by the incongruence between the timer module and the new application environment. http://www.fas.org/spp/starwars/gao/im92026.htm

It is important to develop an assumption management technology with the following capabilities:
• Provide an AADL based framework to encode assumptions in a machine checkable format.
• Provide a grammar for disparate component developers to encode assumptions (and provide guarantees) in a standard format.
• Provide a vocabulary for the properties of an assumption.
• Enable composition of assumptions, when systems are built from smaller sub-systems.
• Automatically validate a relevant subset of assumptions during system evolution.
• Vertical assumptions tracking: Track the assumptions across the software life-cycle.

An early work on this approach was done by Ajay Tirumala et. al. [1]. In the context of formal specifications analyzed by model checking, we can consider several possibilities for explicitly including such semantic assumptions about the physical environment into the models and using them for formal analysis. When the relevant physical quantities can be captured in the formal model as explicit parameters, such as clock skews, network delays, and execution times, the instantiation of such parameters to the appropriate values for the given physical environment yield the adequate model to be analyzed.

A stronger version of this idea may specify the semantic interface with the physical environment as a parameter theory in which not only numeric parameters, but also formal constraints between such parametric values and possibly specifications of dynamic behavior are given. Yet another possibility is to capture the behavior of the physical environment by means of formulas in temporal logic. Such formulas can then be used for assume-guarantee reasoning, and also for model checking purposes. It is an important research topic to identify which of these approaches, or new ones, is most appropriate.

2.2 Ensuring Temporal Determinism and Predictability

Nearly all the modern aircraft designs adopt the Integrated Modular Avionics (IMA) architecture. The industry standard for IMA is ARINC 653. Unfortunately, ARINC 653 specification on computation resource sharing policies is incomplete. It specifies only how the CPU is shared between partitions, but computational performance is a function of how well CPU, cache and communication buses are working together [2].

Multicore architectures present further challenges. A multicore architecture is substantially different from a single core one in that concurrently executing tasks (on the parallel cores) share critical physical resources such as caches, buses, and chip interconnect networks. This extensive sharing of physical resources on critical paths may jeopardize timing predictability and security.

• **Shared physical resources**: multicore architectures employ sharing of several physical resources, including caches, main memory, communication infrastructures, power/clock frequency. Extensive hardware resource sharing makes it extremely challenging to provide strong isolation guarantees.
• **Multiple active components**: CPU cores, coprocessors, DMA peripherals are active components can compete for access to physical shared resources and interfere with each others.
• **Security Challenges**: Concurrent resource sharing opens the door for a compromised partition in one core to mount denial of service attacks to secured partitions running on different cores. In addition, potential covert channels can be created. For example, by transmitting information by modulating one’s cache foot-print.

One specific research technique we would like to put forward is to create the technology for Single Core Equivalent (SCE) software partitions. The key idea is that the execution of a SCE partition on a multicore-based system is certifiably equivalent to the execution of the same partition on a single core system. Once we achieve this goal, large numbers of existing certified software applications developed for single core systems can be reused with standard single core (re)certification processes -- a monumental saving in engineering and certification effort.
Finally, the analysis of the timing behaviors of a modern avionics has become highly complex. When there are tight constraints on resources such as bandwidth and processor capacity, certain seemingly small changes in a few components can lead to a cascade of timing problems. Consider the case of moving a task from a processor's IMA partition to another processor's partition. The task sets need to be updated. The I/O and network traffic must be rerouted. The schedulability equations of processor, I/O and network need to be recreated, and the analysis needs to be propagated end to end. Last but not least, all the architecture specification documents have to be updated. The traditional real time performance analysis process is, to a large extent, a manual, labor-intensive process that is unable to provide timely feedback during early design stages when multiple teams are continually making changes to the design.

Minyoung Nam, Sibin Mohan et. al. have automated the performance analysis process with a system integration tool prototype called ASIIST (Application-Specific I/O Integration Support Tool) [3][4]. To move a task, we can now use a graphical interface to drag and drop a task from one processor's IMA partition to another. All the steps described above are done automatically, including the updating of the architecture specification in AADL. In this paper, we show how to use this tool to explore the design space of an IMA system architecture, so as to derive designs with systematic performance analysis. Finally, we note that one of the related efforts is the ongoing work in AADL equation generation annex. This work needs, however, to be extended to cover multicore chips.

2.3 Model Checking of AADL Models
AADL is both a modeling language for real-time embedded systems and an international standard widely used in industry. It has features to model the real-time aspects of embedded systems and to represent both the software and hardware architectures of the components making up such systems. In addition, simulators for AADL models have become available.

However, at present AADL lacks a formal semantics. This lack is important for real-time embedded systems, because many of them—in areas such as avionics, motor vehicles, and medical systems—are safety-critical systems, whose failures may cause great damage to persons and/or valuable assets. A highly desirable goal is to have a formal semantics of AADL that can be used to automatically generate formal executable specifications of AADL models [5]. These specifications can be used for automatic verification of safety and liveness properties by model checking. Furthermore, these formal methods should be supported by tools that are integrated into the AADL tool chain.

In recent work by Olveczky, Boronat and Meseguer [5], an object-based real-time concurrent formal semantics for a substantial subset of AADL has been developed in the Real-Time Maude formal specification language [9]. This semantics has then been used directly as the core of the AADL2Maude tool, which can simulate and formally analyze AADL models. Real-Time Maude support for nested objects, real-time concurrency and a wide range of formal analysis capabilities. By automating the AADL formal semantics with the AADL2Maude tool, one can automatically generate formal executable specifications of AADL models (with behaviors specified in the Behavior Annex) in Real-Time Maude for simulation, reachability analysis, and LTL model checking purposes.

2.4 Towards a Synchronous AADL Annex
Most CPS systems are distributed real time systems. A big challenge in designing, verifying, and implementing such systems is system complexity. It is hard to get distributed software right, and even harder to verify it. When concurrent interactions are asynchronous, the state space grows exponentially and model checking quickly becomes unfeasible, even for small-sized systems. Since AADL models are made up of different components and can have many threads that communicate asynchronously, this makes model checking a great challenge.
A recent breakthrough in this area is the Physically Asynchronous Logically Synchronous (PALS) architectural Pattern [6][7][8], which can drastically reduce the complexity of a distributed real time system by making it semantically equivalent to a much simpler Synchronous Embedded System (SES). Case studies in both [7] and [8] show an exponential state space reduction when model checking of a distributed system is replaced by its semantically equivalent SES.

For the AADL2Maude tool, this means that the successful model checking analyses have taken full advantage of the PALS transformation, so that we model check the AADL model of the SES, which is equivalent to an asynchronous DES model in AADL for the physical implementation. This suggests the usefulness of exploiting PALS in AADL for design, verification, and code generation purposes. The idea is to define a Synchronous AADL Annex in which we can:

1. Begin with a simpler AADL model of a SES in the AADL Synchronous Annex;
2. Use this simpler model for simulation and formal verification purposes using AADL2Maude;
3. Transform the AADL model of the SES into an AADL model of the semantically equivalent DES; and
4. Generate code for the semantically equivalent distributed system using the PALS middleware.

2.5 QoS Composition
Currently, the main stream approach is mainly a decomposition approach, which hands each specific QoS attributes such as security, schedulability and fault tolerance. The protocol is typically developed by different communities. In the context of AADL, the solution approach for each QoS attribute may be codified into an AADL annex [10]. However, solutions for individual QoS attributes may interfere with each others. For example, both fault tolerance and security consume resources and impacts schedulability. Replication used for fault tolerance purpose may facilitate the spread of intrusions. Security protocols impose challenging constraints for system reconfigurations from the perspective of fault tolerance. These are examples of these well known problems.

AADL community brings together multiple different modeling and analysis technologies and applying them to a common specification. Research programs, for example, DARPA’s Software Enabled Control [12] supported research on the use of multiple complementary models. However, QoS composition remains to be a serious challenge. There has yet to be a commonly accepted technology that is powerful enough to characterize and to analyze the interactions between selected QoS protocols embedded in AADL models.

We need to find a way to support the analytic composition of QoS attributes, so that AADL annexes for different QoS attributes can be used in a compositional way that supports tradeoff analysis. To this end, we need to develop:

- Formal models that model the semantics of interaction between different QoS protocols.
- A suite of coherent QoS attributes and protocols supported by AADL Annex with annotations to support the compositional tradeoff analysis.
- Tools to support the analysis and composition of selected QoS protocols for applications with different QoS needs.

3. Conclusion

System integration is one of most challenging aspects in the development of distributed CPS. This phase often consumes 70% of the total development budget. We need to develop an analytical system composition framework based on AADL, which includes:
1. Components annotated with explicitly stated and machine checkable assumptions regarding their physical environment.
2. Automated schedulability analysis, where the schedulability equations can be directly generated from annotated AADL models supported by the AADL equation generation annex under development.
3. A formalized AADL specification with tools to translate it to a model checking tool, supported by a Synchronous AADL annex under development.
4. Technologies to support the composition of QoS attributes.

References
Abstract—Modern automobiles are increasingly dependent on software and electronics to realize a wide range of critical system functionality. This represents a marked transition of automotive platforms from largely mechanical to progressively cyber-physical systems. Consequently, the share of software development in automotive design is expected to grow, and so is the associated complexity of specification, design, integration, analysis, and verification. This development could lead to unacceptably long design cycles and unanticipated failure scenarios in future automotive systems. Realizing this challenge, automobile manufacturers, supplier, and tool developers have developed AUTOSAR, which represents an AUTomotive Open System ARchitecture for managing complexity and improving reliability. In this paper, we provide a brief overview of the current AUTOSAR standard, and identify possible extensions. We highlight the need for an associated end-to-end integration framework for AUTOSAR, and extend an existing model-based framework called SysWeaver to address this requirement. We describe the mapping of AUTOSAR concepts to SysWeaver, and summarize the benefits of SysWeaver from an integration and analysis perspective. Finally, we also apply our end-to-end integration framework towards the development of a scaled automotive platform, and illustrate the proposed workflow in the context of a representative application.

I. INTRODUCTION

Automobiles can no longer be considered as largely mechanical/physical systems. Current estimates indicate that up to 40% of the cost to develop a premium car could be from software and electronics, with software development alone contributing to about 13% [4]. Contemporary high-end vehicles can include up to 20 million lines of code, with the number only expected to grow higher in the near future. Automotive designers are faced with the daunting task of designing, developing, and deploying such cyber-physical systems. The operational environment of such systems also dictates strict end-to-end timing requirements, which are proving to be increasingly challenging from verification and validation standpoints.

The computing infrastructure in modern automotive platforms is both diverse and distributed (see Figure 1). In 2004, on the Volkswagen Phaeton, there were 61 different processing units, of which there were 45 different kinds of units. There were 3 coupled bus systems resulting in 3860 metres of cabling which weighed 64 kilograms. Communication consisted of 2500 signals as part of 250 unique network messages. Code-size and memory was in the order of 50 megabytes. This results in a complex and highly distributed system with multiple tight closed-loop controls. There is a need for addressing this complexity and to ease the integration effort between suppliers and manufacturers. Having recognized the need to address these issues, the automotive industry formed a development partnership called the Automotive Open System Architecture (AUTOSAR) [1]. The goals of AUTOSAR include standardization of basic software functions, maintainability throughout the product lifecycle, capturing safety requirements and management of scalability. In this paper, we briefly review the AUTOSAR standard, identify possible extensions, and propose the use of an end-to-end design and integration framework to enable AUTOSAR-compliant automotive systems.

Developing a system from requirements to production now requires proven processes to reduce errors and time-to-market. The major pieces of the problem include a) System Requirements Specification, b) System Representation, c) System Analysis and d) Implementation. AUTOSAR addresses the System Requirements Specification challenge, while Representation and Analysis are delegated to supporting frameworks. In this regard, conventional processes and methods do not scale very well, while model-based design holds the promise of (a) capturing rich behavioral descriptions along multiple concerns (or aspects), (b) offering an interoperable tool-set to analyze and verify both functional and para-functional requirements of a system, and (c) supporting the ability to generate executable code directly from these models. Although measurements from run-time environments may need to be fed back to re-configure system models, model-based design can in principle be independent from the specific hardware and
operating systems. Realizing these desirable properties, we extend SysWeaver [5], an existing model-based framework, to support AUTOSAR, and illustrate its application using an end-to-end scaled automotive cyber-physical system case study.

Before introducing SysWeaver, we first provide a brief overview of AUTOSAR, identify possible extensions, and highlight the benefits that can be obtained using an end-to-end design and integration framework in the AUTOSAR context.

II. AUTOSAR OVERVIEW

AUTOSAR consists of several integrated concepts that support a multi-layered approach. For this paper, we shall focus on three major pieces - AUTOSAR Software Component specification, the AUTOSAR Operating System (OS) Specification, and the AUTOSAR Communication Model (COM). AUTOSAR Software Components describe an application that runs on the infrastructure. These components have well-defined interfaces which are standardized within AUTOSAR and include a Software Component Description. The AUTOSAR Virtual Functional Bus (VFB) serves as the integration abstraction through which software components communicate in a technology-independent fashion. The actual implementation of this abstract layer on a per Electronic Control Unit (ECU) level is provided by the Runtime Environment (RTE). The infrastructural functionality is provided by Basic Software. We now describe these core concepts in detail.

A. AUTOSAR Software Component

Each AUTOSAR Software component describes part of the functionality of an application and several interconnected Software Components provide the implementation of an application. The description scale is variable and can be small to provide reusability or can be large to provide hierarchical encapsulation. The AUTOSAR software component is atomic in the sense that a single instance of the component runs on a single ECU and cannot be divided to run on multiple ECUs. A Software Component Description is independent of any hardware mapping but can be restricted to run on certain kinds of ECUs. A special kind of Software Component is a Sensor-Actuator Component which encapsulates application-dependencies of specific sensors and actuators.

Components have well-defined interactions with other components, which are described through Ports. A port represents a specific point of interaction with other components. Ports have well-defined interfaces which can either be a Client-Server interface or a Sender-Receiver interface. Client-Server interfaces are used for defining a set of invokable operations and Sender-Receiver interfaces are used to describe data-oriented communication. Ports are either Require-Ports (RPort) or Provide-Ports (PPort). An RPort requires an interface, which is in turn provided by a PPort. In this paper, we focus only on Sender-Receiver interfaces. Client-Server interfaces introduce blocking-wait delays [12], which require more detailed analysis beyond the scope of this paper. They are also not commonly observed in task pipelines, which are the specific focus of this paper.

Atomic Software Components consist of entities called Runnables. Runnable entities are the smallest code fragments provided by AUTOSAR software components and Software Components can be composed of multiple Runnables. Runnables are the schedulable parts of the software component and are executed in the scope of an OS Task. Runnables are activated by events as part of the RTE which are called RTE Events. There are several types of RTE events and they can either activate or wakeup Runnables. Runnables also have runtime information associated with them describing their resource usage (e.g. memory usage, WCET).

B. AUTOSAR OS Specification

The AUTOSAR operating system specification is based on that of the OSEK/VDX Operating System specification [14]. OSEK/VDX is widely used in the automotive industry and has been deployed on various different types of ECUs. The AUTOSAR OS is an event-triggered operating system providing high flexibility and clock maintenance support. OS features include fixed-priority scheduling, interrupt handling, startup and shutdown interfaces.

In the AUTOSAR OS, there are two types of tasks. Basic Tasks are sequential and execute to completion. Extended tasks have various synchronization points, which result in various suspension durations initiated by the task itself. Interrupt Service Routines (ISR) in the specification are divided into two categories. Category 1 ISRs are not controlled by the OS, and do not use OS services. Category 2 ISRs use OS services, and their execution behavior can be used by the RTE. ISRs have static priorities with their priority levels strictly above those of tasks. This allows ISRs to be preempted only by ISRs of higher priority. ISRs and Tasks are also allowed to share resources. From the perspective of timing analysis, both category 1 and category 2 ISRs are accounted using minimum inter-arrival times between interrupts. Interrupts are modeled at the ECU level, and they often correspond to devices attached to the ECU such as serial, CAN, and FlexRay controllers, all of which have a minimum inter-arrival time property associated with them.

C. AUTOSAR Communication Model

The AUTOSAR communication structure has various layers. The RTE interacts with the AUTOSAR Communication Layer (COM) through which it relays data signals. The AUTOSAR COM interacts with the Protocol Delivery Unit (PDU) Router layer, which is responsible for transmitting data between different types of communication interfaces. The PDU Router can directly interact with the communication hardware interfaces. If the data packets exceed the size limits of the specific communication bus, the PDU Router can also use different communication transport layers. The communication hardware interface in AUTOSAR aims to provide a uniform interface for data transmissions across all bus systems in AUTOSAR. AUTOSAR supports a variety of communication bus specifications including CAN [3], LIN [11], and FlexRay [7].
D. Extending the AUTOSAR framework

The AUTOSAR framework has the strong fundamental integration abstraction of a virtual functional bus, which enables interoperability, reusability, and maintainability. The AUTOSAR standard is actively evolving and there are still many key issues that need to be addressed before real-world systems can take advantage of AUTOSAR. We identify the following potential areas for future extensions to AUTOSAR:

1) Timing Models: The lack of a specific execution model, and heterogeneous semantics have made it increasingly difficult to develop proper timing models in the AUTOSAR context [16]. Timing models are especially important in achieving the goal of designing predictable and analyzable systems from the timing perspective. Automotive software systems typically have multiple end-to-end control loops with strict timing constraints. The lack of a specific timing model makes it harder to guarantee the proper functioning of these control loops. There are constraints listed within the AUTOSAR specification to avoid timing interference between the heterogeneous execution paradigms. However, satisfying these constraints adds additional overhead to designers and integration engineers.

2) Fault Tolerance: Replication in terms of both hardware and software redundancy is a well-studied technique in existing dependable systems literature [2]. Currently, AUTOSAR does not have proper support for handing such fault-tolerance mechanisms. Automotive software systems are certified according to various safety standards using requirements like Safety Integrity Levels (SIL). Some subsystems require high levels of reliability, which are hard to realize at a reasonable cost without leveraging redundancy-based techniques. The concepts of backup tasks, replicas, and voters have not been a part of the AUTOSAR specification. It would be very useful to support fault-tolerant designs under the AUTOSAR model for handling safety-critical automotive subsystems.

3) Instrumentation Support: AUTOSAR is primarily targeted as an architecture to enable easy integration and interoperability. The run-time environment and basic software components are provided to support the various application tasks executing in the system. Less emphasis is placed on the development and testing phase of the software components themselves. Timing properties and behavioral characteristics should be extractable from the AUTOSAR RTE during the development phase. In order to support this process, support should be available for instrumenting the application tasks.

4) Feedback: Integration of the various software subsystems uncovers implicit assumptions and performance issues that are often present in the software design. Performance feedback from the run-time environment can provide valuable assistance during the system integration phase by monitoring the system behavior. Various issues ranging from the misconfiguration of system parameters to inefficient task allocation can be discovered through feedback support. AUTOSAR currently does not provide support for such detailed feedback.

5) Verification Support: During the system development phase, significant testing time can be saved through various verification techniques like range checking for inputs/outputs, program flow checks, and execution time bounds. Valuable fault-containment support is also provided by such mechanisms. The AUTOSAR specification does not focus on verification of system design.

AUTOSAR relies on strong support from end-to-end integration tools to address most of these issues. In the next section, we extend SysWeaver [5], a model-based design and integration framework to support AUTOSAR and address these key issues.

III. END-TO-END INTEGRATION FRAMEWORK FOR AUTOSAR

The current AUTOSAR specification will benefit from an end-to-end model-based design and integration framework, which can capture all system requirements, properties, and abstractions, leading to an efficient system development cycle. The cycle proposed in this work is shown in Figure 2. In general, model-based design holds the promise of (a) capturing rich behavioral descriptions along multiple concerns (or aspects), (b) offering an interoperable toolset to analyze and verify both functional and para-functional requirements of a system, and (c) supporting the ability to generate executable code directly from these models. While measurements from run-time environments may need to be fed back to fine-tune runtime assumptions of the models (e.g. execution times), model-based design can in principle be independent from the specific hardware, OS and programming languages. In order to leverage these advantages of model-based design in AUTOSAR, we extended SysWeaver, which is an end-to-end model-based framework for embedded systems.

A. SysWeaver Introduction

SysWeaver is a model-based design, integration, and analysis framework introduced by de Niz et. al [5] for embedded real-time systems. It explicitly captures the para-functional behaviors and their impact on the functional components of a system. It uses a view-based representation of various system aspects such as Timing, Fault-Tolerance and Deployment. Each aspect is modeled in a separate view known as a Dimension. Different domain experts can work on each of these different aspects, while SysWeaver resolves dependencies among the different views automatically. The primitive components in Sysweaver are called ApplicationAgents that
have external communication interfaces called Ports. Software state executions are handled through Transition elements, internal to ApplicationAgents, that describe the transformation of inputs into outputs. The Transitions also describe the effect of activating or triggering a software component. SysWeaver also supports hierarchical structures using Encapsulator elements. The actual relationships between Components, Ports, Encapsulators and other component interactions are managed through relationship entities called Couplers. Couplers express interactions like event and data communication, relationships among components such as timing and replication, as well as hardware abstractions such as ECUs and network buses.

System requirements are verified through Analysis and Simulation plugins. There are internal plugins which implement well-known analysis and simulation techniques. SysWeaver also provides the ability to use external plugins using the XML system description, which can also be exported to other frameworks. Framework interoperability is achieved through the same interface. The built-in plugins provide various analysis features like Network Load and Schedulability (Figure 3). CPU Task and Network Bus Simulations (Figure 4) are also available for each CPU and Network entity. This is done using a runtime simulation based on the worst-case execution times (WCET) of software components. SysWeaver also generates system code using the code generation plugins. This provides the ability to automatically generate distributed glue code which ties together the distributed functional code.

B. Mapping to AUTOSAR

Many of the AUTOSAR constructs are already captured by the existing capabilities of SysWeaver. AUTOSAR Software components map directly to ApplicationAgents in SysWeaver. R-Ports and P-Ports are captured by different Port elements; ReadPorts and WritePorts. The Transition element represents Runnable inside an AUTOSAR Software component. Transition elements can be time-triggered or event-triggered through Port interactions. The Port elements in SysWeaver have well-typed interfaces and data types which is consistent with AUTOSAR Port requirements.

AUTOSAR COM constructs are captured by Couplers in SysWeaver. ECU Couplers represent AUTOSAR ECUs and Network Couplers represent specific AUTOSAR COM interfaces. Specific Network Couplers for the different COM instances were created such as FlexRay, CANBus, and LINBus. AUTOSAR ISRs are captured within ECU Couplers and have priorities and timing properties.

The AUTOSAR OS Task representation is not explicitly present in SysWeaver. OS Tasks in AUTOSAR follow the OSEK/VDX model. They consist of static Task Schedules that correspond to a list of time-triggered or event-triggered Runtimes to be executed. To capture this requirement, an AUTOSAR Task component is created to contain the static schedules. The Task Schedule consists of the list of Transition elements to execute, along with time-triggered and event-triggered properties such as Port dependencies and periods. The schedule is termed as static since the assignment of runnables to tasks is done at design time. The time-triggered runnables are executed according to the schedule table, while the event-triggered runnables are activated by events. The event-triggered runnables are still scheduled within the context of a task, and they are executed only if they have received their corresponding activation events. Each AUTOSAR Task component is constrained to execute on a single ECU, ensuring that none of its constituent runnables execute in parallel. Task components in SysWeaver can thus model Basic and Extended Tasks as per the AUTOSAR OS specification.

C. Analytical Integration Support

AUTOSAR provides an effective abstraction of the VFB, which eases the system integration phase. However, verifying end-to-end timing properties, especially across heterogeneous networks is still a challenging problem in the AUTOSAR
context. SysWeaver adopts the approach described in [9], where applications are specified as task pipelines for analysis.

The system timing requirements are specified in terms of end-to-end deadlines for pipelines. Each pipeline can traverse a number of pipeline stages $p_i$ through $p_n$, where each pipeline stage $p_i$ is assumed to have a period $T_{p_i}$. The pipeline stage $p_i$ communicates with its succeeding pipeline stage using a message $m_{p_i}$. The worst-case response-time $R_{p_i}$ of each pipeline stage $p_i$ executing on the CPU can be computed using standard response-time tests [10]. For each message $m_{p_i}$, the message queuing delay $Q_{m_{p_i}}$ and transmission time $C_{m_{p_i}}$ can be calculated using protocol-dependent analysis. Using these values, the end-to-end response time of a pipeline $p = p_1 \rightarrow m_{p_1} \rightarrow p_2 \rightarrow m_{p_2} \ldots \rightarrow p_n$ is bounded by (see [9] for details):

$$
\Delta^G(p_1 \ldots p_n) \leq \sum_{i=1}^{n} (R_{p_i} + Q_{m_{p_i}} + C_{m_{p_i}} + T_{p_{i+1}}) + R_{p_n}
$$

This assumes that there is no time synchronization between different pipeline stages, and that the task offsets are arbitrary.

If a synchronized time base is available, then the end-to-end response time bound can be refined as:

$$
\Delta^G(p_1 \ldots p_n) \leq \sum_{i=1}^{n} (R_{p_i} + Q_{m_{p_i}} + C_{m_{p_i}}) + R_{p_n}
$$

where each pipeline stage $p_{i+1}$ is assumed to be offset by $\Delta^G(p_1 \ldots p_i) + Q_{m_{p_i}} + C_{m_{p_i}}$.

Known end-to-end response-time analysis techniques for FlexRay [8], [15], CAN [6], LIN, and heterogeneous networks have been integrated into the SysWeaver framework for calculating $Q_{m_{p_i}}$ and $C_{m_{p_i}}$ using schedulability analysis plugins. Future extensions to the analytical framework can also be made through the same interface.

D. Benefits in the AUTOSAR context

The key benefits of our end-to-end design and integration framework arise from addressing the major components lacking in the AUTOSAR specification.

- The presence of heterogeneous subsystems, and the lack of a coherent execution model make it unwieldy to develop a timing model for the AUTOSAR context. The structured methodologies employed by SysWeaver manages the complexity of developing such a timing model. In order to support the AUTOSAR context, we added the analytical framework described in Sub-section III-C to SysWeaver using plugins.

- We provide support for fault-tolerance and replication through couplers. This enables the code generated from SysWeaver to be automatically replicated. The infrastructure to manage the replicas, send heart-beat messages, and fail-over mechanisms are readily captured in the SysWeaver framework. This infrastructure is designed to be independent and transparent to the AUTOSAR Runtime Environment.

- Automatic code generation enables the SysWeaver framework to provide an extensive instrumentation infrastructure. This provides valuable feedback to the system designers during the development and integration phases. Verification checks are provided through annotations to the Ports of Application Agents.

SysWeaver, thus, augments the AUTOSAR infrastructure as an end-to-end design, analysis and integration framework, to achieve interoperable, predictable, analyzable, and maintainable automotive system designs.

We now describe our work with scaled automotive platforms, which provides the context in which the SysWeaver framework is currently being used for the end-to-end development of automotive cyber-physical systems.

IV. Case Study: Scaled Automotive Platforms

We built an experimental platform to demonstrate rapid prototyping and to validate our analysis techniques. The requirements included being able to demonstrate the real-time aspects of an automotive control system, as well as being representative of an actual system. There needs to be motion control as well as peripheral body control to illustrate the complex and integrated nature of the system. Figure 5 shows the block diagram of our platform.

1) Hardware Platform: A remote control car system of 1/10th scale size was chosen with some modifications to its mechanical platform. The basic remote-control (RC) car chassis was mounted with two programmable Electronic Speed Controllers (ESC), one for each of the rear wheels, thus offering independent forward and reverse throttle to the respective wheels. The Speed Controllers are controlled through Pulse-width Modulation (PWM) Signals. A Servo Motor was mounted close to the front-end of the car for each of the front wheels resulting in two servo motors. This construction enables independent wheel control. Thus, each of the two rear wheels can be propelled either forward or reverse independent
of the other wheel. Likewise, the front wheels can be maneuvered in independent directions with different wheel angles.

A set of five Axiom-Freescale HCS12XDT512 embedded microcontroller boards, representative of the ECUs found on modern day automotive platforms, make up the distributed automotive control system. A low-speed CAN network connects these boards. Each microcontroller is primarily responsible for controlling a different section of the car. One ECU serves as the Bus Master that sends commands for throttle and direction. The remaining four boards control the Right Wheel Motor (via the Right ESC), the Left Wheel Motor (via the Left ESC), the Right Wheel Servo, and the Left Wheel Servo respectively. The Bus Master ECU is connected to a FireFly Sensor Node [13] which serves as a wireless remote-control interface for the platform (Figure 7). On receipt of input from the Firefly sensor node, the Bus Master ECU sends commands to the individual ECUs to control the various steering and speed motors. To experiment with fault-tolerance, two different platforms were built. On one of the platforms, the ECUs that control steering are also electrically connected to control the wheel motors. The completed platforms are shown in Figure 6. The next generation of this platform will comprise of a hierarchy of FlexRay, CAN and LIN. While this version only uses CAN, the concepts and our overall approach are still applicable and can be validated.

2) Software Platform: The software environment comprises of the SysWeaver framework (for offline design, analysis and code generation), the RTA-OSEK operating system and the Freescale Codewarrior compiler. An AUTOSAR XML description is imported into SysWeaver to create the various component libraries. The system model is then constructed in SysWeaver to capture the functional aspects, the task model and the ECU and COM models. Real-time properties such as estimated worst-case execution time and task priorities are added to the various components for analysis and implementation purposes. All the ECUs in the experimental platform run the RTA-OSEK operating system. RTA-OSEK is an operating system built on the OSEK/VDX specifications and is AUTOSAR-compliant. It takes a system description in the OSEK Implementation Language (OIL) containing Software and ECU Task descriptions for each ECU. The Freescale Codewarrior Compiler is used to compile the code and generate executables for each ECU.

The functional model of the system also needs to satisfy various concurrent requirements. The front wheel servos need to be synchronized with each other as well as the rear-wheel motors. The motor control of the platform consists of various concurrent flows. The Master node needs to deliver user input to the ECUs controlling the various electro-mechanical parts. Also, a state monitor executes on some of the ECUs to detect node failures for fault-tolerance purposes. The functional view of the system in SysWeaver is shown in Figure 9. Various
AUTOSAR Task components are created in the Dynamic View of the system model in SysWeaver. Each AUTOSAR Task component contains a list of the Runnables to execute with periodic timing properties. These components are then mapped to their respective ECU's. Each ECU component is then connected to the CAN Bus coupler in the Deployment view of the model which is shown in Figure 10.

3) Analysis and Implementation: Once the system is modeled in SysWeaver, analysis is performed to verify system requirements. Using the system taskset shown in Table IV-3, we compute end-to-end response times using our CAN bus analysis plugin along with a schedulability analysis for each ECU. CAN message identifiers are automatically generated for each message on the bus. The creation of actual CAN messages is not required as SysWeaver is able to identify messages on the CAN bus using the information provided in the Functional and Deployment views. The various Runnables, Tasks and Messages used in the experimental system are provided in Table 1. The runnable parameters like WCET, Period and ECU were provided by the system designer. Message sizes and CAN configuration (125Kbps) were also specified at the design phase. CAN message identifiers are automatically generated for each message on the bus. SysWeaver automatically calculates the corresponding task WCET, task period, message waiting times and message transmission times. The basic constituent task pipelines and their corresponding detailed end-to-end delay calculations are provided in Table 2. These calculations employ the schedulability analysis developed for the CAN bus [6], which extend to the end-to-end integrated system using the approach described in section III-C. The system was constrained to have a fail-over duration within 125ms. Therefore, the corresponding parameters were chosen for task pipelines $p^1$ and $p^2$. When either ECU4 or ECU5 fails, the corresponding fail-over ECU (ECU2 and ECU3 respectively) detects the failure within 125ms, and takes over the critical task of motor control. The user-interface to system response was also constrained to be less than 500ms. The corresponding pipeline parameters are then chosen for pipelines $p^3, p^4, p^5, p^6, p^7, p^8$. The worst-case delay is encountered by pipelines $p^5$ and $p^7$, which encounter up to 129.64ms of delay. The user input runnable has a period of 200ms. The system may require up to 200ms to receive the user input and 129.64ms to respond appropriately, therefore, maintaining the end-to-end constraint of 500ms. The runnable periods were chosen to simultaneously satisfy all system pipeline constraints (including fail-over pipelines), resulting in considerable slack for the user-input pipelines.

The implementation and deployment of software on the platforms were achieved through the use of the framework as shown in Figure 2. Once the functional code for the Runnables was created, the entire process is automated in SysWeaver through the code generator plugin which links in the prebuilt SysWeaver middleware code with the application-specific Runnables code and then invokes the RTA-OSEK engine on a per ECU basis. This results in automatic generation of the runtime code for all ECUs through a single plugin invocation in SysWeaver. The code-generator plugin uses the SysWeaver model to generate the glue-code necessary to tie in the functional application code contained within the AUTOSAR Runnables. This includes automatic generation of CAN messages using the prioritized CAN message identifiers generated from the analysis for each message that is to be placed on the communication bus. Using SysWeaver’s ability to inject instrumentation code into the runtime code, calculations for various system properties are fed back into the system model. The platforms run well and showcase interesting and useful functions such as control, fail-overs,
end-to-end delay management, task pipeline synchronization, analyzability, and AUTOSAR extensibility. Testing of the platforms was conducted, showing that they worked as designed. The instrumentation code measurements were within the timing analysis calculations. Fault-tolerance was tested by disabling one of the motor ECUs exhibiting fail-over times as specified in the constraints.

V. CONCLUSIONS

Automotive software is growing in both complexity and criticality, with many essential features such as throttle and brake being already controlled by software. Emerging driver-assist features and autonomous driving capabilities can only be expected to add to this trend. In this paper, we briefly reviewed the AUTOSAR standard proposed by the automotive industry to address anticipated challenges in designing and integrating automotive software systems. We proposed key extensions to AUTOSAR, and developed an end-to-end design and integration framework based on SysWeaver to model, integrate, analyze, verify, and implement AUTOSAR-compliant automotive systems. We illustrated the basic steps involved in the proposed workflow using the development cycle of a distributed real-time small-scale automotive cyber-physical system as an example. We built a functional model, performed integration and analysis, and verified the resulting properties using a real implementation.

In the future, we intend to look at more in-depth analysis of SysWeaver in other application domains. We are also in the process of building prototypes on a larger scale to look at an even more realistic representation of modern vehicles including the increasing use of more autonomous driver-assist features using different types of ECUs to show interaction across different vehicle domains.

VI. ACKNOWLEDGEMENTS

The authors would like to thank Dhananjay Motwani for his help in the implementation effort.

REFERENCES


<table>
<thead>
<tr>
<th>Runnable</th>
<th>Task</th>
<th>Runnable Period (ms)</th>
<th>Runnable Period Tpi (ms)</th>
<th>Bound ECU</th>
<th>Runnable WCET Cpi (ms)</th>
<th>Task WCET Cpi (ms)</th>
<th>Msg. Length (Bytes)</th>
<th>Dest.</th>
<th>Msg. Wait Time Qm+1 (ms)</th>
<th>Tx. Time Cmp+1 (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserInput</td>
<td>Task1</td>
<td>200</td>
<td>100</td>
<td>ECU1</td>
<td>10</td>
<td>50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Calculate L Direction</td>
<td>Task1</td>
<td>100</td>
<td>100</td>
<td>ECU1</td>
<td>10</td>
<td>50</td>
<td>2</td>
<td>Task2 (m+1)</td>
<td>2.24</td>
<td>0.8</td>
</tr>
<tr>
<td>Calculate R Direction</td>
<td>Task1</td>
<td>100</td>
<td>100</td>
<td>ECU1</td>
<td>10</td>
<td>50</td>
<td>2</td>
<td>Task3 (m+1)</td>
<td>3.04</td>
<td>0.8</td>
</tr>
<tr>
<td>Calculate L Speed</td>
<td>Task1</td>
<td>100</td>
<td>100</td>
<td>ECU1</td>
<td>10</td>
<td>50</td>
<td>2</td>
<td>Task2 (m+1)</td>
<td>3.84</td>
<td>0.8</td>
</tr>
<tr>
<td>Calculate R Speed</td>
<td>Task1</td>
<td>100</td>
<td>100</td>
<td>ECU1</td>
<td>10</td>
<td>50</td>
<td>2</td>
<td>Task3 (m+1)</td>
<td>3.84</td>
<td>0.8</td>
</tr>
<tr>
<td>Control L Servo</td>
<td>Task2</td>
<td>50</td>
<td>50</td>
<td>ECU2</td>
<td>10</td>
<td>25</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Control L Motor</td>
<td>Task2</td>
<td>50</td>
<td>50</td>
<td>ECU2</td>
<td>10</td>
<td>25</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Control L Headlight</td>
<td>Task2</td>
<td>50</td>
<td>50</td>
<td>ECU2</td>
<td>10</td>
<td>25</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Control R Servo</td>
<td>Task3</td>
<td>50</td>
<td>50</td>
<td>ECU3</td>
<td>10</td>
<td>25</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Control R Motor</td>
<td>Task3</td>
<td>50</td>
<td>50</td>
<td>ECU3</td>
<td>10</td>
<td>25</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Control R Headlight</td>
<td>Task3</td>
<td>50</td>
<td>50</td>
<td>ECU3</td>
<td>10</td>
<td>25</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Control L Motor</td>
<td>Task4</td>
<td>50</td>
<td>25</td>
<td>ECU4</td>
<td>10</td>
<td>20</td>
<td>1</td>
<td>Task2 (m+1)</td>
<td>0.8</td>
<td>0.72</td>
</tr>
<tr>
<td>Control L Headlight</td>
<td>Task4</td>
<td>50</td>
<td>25</td>
<td>ECU4</td>
<td>10</td>
<td>20</td>
<td>1</td>
<td>Task2 (m+1)</td>
<td>0.8</td>
<td>0.72</td>
</tr>
<tr>
<td>Control L Motor</td>
<td>Task4</td>
<td>50</td>
<td>25</td>
<td>ECU4</td>
<td>10</td>
<td>20</td>
<td>1</td>
<td>Task2 (m+1)</td>
<td>0.8</td>
<td>0.72</td>
</tr>
<tr>
<td>Control R Headlight</td>
<td>Task4</td>
<td>50</td>
<td>25</td>
<td>ECU4</td>
<td>10</td>
<td>20</td>
<td>1</td>
<td>Task2 (m+1)</td>
<td>0.8</td>
<td>0.72</td>
</tr>
<tr>
<td>Control L Headlight</td>
<td>Task5</td>
<td>50</td>
<td>25</td>
<td>ECU5</td>
<td>10</td>
<td>20</td>
<td>1</td>
<td>Task3 (m+2)</td>
<td>1.52</td>
<td>0.72</td>
</tr>
<tr>
<td>Control R Headlight</td>
<td>Task5</td>
<td>50</td>
<td>25</td>
<td>ECU5</td>
<td>10</td>
<td>20</td>
<td>1</td>
<td>Task3 (m+2)</td>
<td>1.52</td>
<td>0.72</td>
</tr>
</tbody>
</table>

TABLE I

SYSTEM TASK SET AND ALLOCATION
Anytime Algorithms for GPU Architectures

Aminreza Abrahimi Saba† and Rahul Mangharam‡
†Dept. of Computer and Information Science
‡Dept. of Electrical and Systems Engineering
University of Pennsylvania
{aminreza, rahulm}@seas.upenn.edu

Abstract—Most algorithms are run-to-completion and provide one answer upon completion and no answer if interrupted before completion. On the other hand, anytime algorithms have a monotonic increasing utility with the length of execution time. Our investigation focuses on the development of time-bounded anytime algorithms on Graphics Processing Units (GPUs) to trade-off the quality of output with execution time. Given a time-varying workload, the algorithm continually measures its progress and the remaining contract time to decide its execution pathway and select system resources required to maximize the quality of the result. To exploit the quality-time trade-off, the focus is on the construction, instrumentation, on-line measurement and decision making of algorithms capable of efficiently managing GPU resources. We demonstrate this with a Parallel A* routing algorithm on a CUDA-enabled GPU. The algorithm execution time and resource usage is described in terms of CUDA kernels constructed at design-time. At runtime, the algorithm selects a subset of kernels and compositions to maximize the quality for the remaining contract time. This is an early effort to enable imprecise and approximate real-time computation on parallel architectures for stream-based time-bounded applications such as traffic congestion prediction and route allocation for large transportation networks.

I. INTRODUCTION

Performance scaling of single-thread processors stopped in 2002 and has fueled the use of multicore Graphics Processing Units (GPUs) which have been growing in transistor count by 65% annually. The current generation of NVIDIA's Fermi GPU consists of 512 cores and is capable of executing 24,576 concurrent thread kernels for efficient stream processing. GPUs will, in the year 2015, use 11nm technology and contain around 5,000 cores, which should render them capable of around 20 Teraflops [1]. In the past few years, the GPU has evolved into an increasingly convincing computational platform for non-graphics applications [2]. Our goal is to investigate time-bounded algorithms on GPUs to effectively trade-off execution time with output quality. This will enable a large class of data-dependent and dynamical applications with a large number of variables to leverage the high-throughput concurrent computation of GPUs. Algorithms in this category include real-time estimation, prediction and decision making in weather science, nation-wide traffic management and electronic trading.

As such data-dependent applications are increasingly subject to larger workloads, it is often not possible to compute results in real-time. Furthermore, it is often better to have an imprecise or approximate result within a time-bound rather than a precise answer that is outdated. Algorithms for such applications must therefore adaptively allocate resources appropriate to the goals and loads and evaluate their progress with execution time to deliver intermediate results. Such results may be in the form of estimated values, a range of answers or measurements.

This paper presents an early investigation of time-bounded algorithms on the NVIDIA CUDA General Purpose GPU programming platform [3], [4], [5]. The Compute Unified Device Architecture (CUDA) provides a programming model for general purpose programming on GPUs. The interface uses standard C code with parallel features. Using CUDA, we investigate the design of time-bounded algorithms that continually measure the quality of the output with the remaining time and dynamically adjust their execution path to optimize the outcome of the computation by a specified deadline. Such Anytime Algorithms allow for approximate and imprecise computation with an output quality metric that is monotonic with the available execution time [6], [7]. The focus of this investigation is on the construction, instrumentation, runtime progress measurement and adaptive selection of execution paths for anytime algorithms on GPUs.

A. Anytime Algorithms

Anytime Algorithms may be considered in two variations: Interruptible algorithms and Contract-time algorithms. Interruptible algorithms can be interrupted at any time to deliver the best result obtained so far. They are convenient to use as the results may be requested in an on-demand manner. Contract-time algorithms are specified an execution time a priori and decide on the best strategy to maximize the quality of the output within that duration. Contract algorithms are often more intuitive to design, and typically use simpler data structures and control structures, making them easier to implement and maintain. Our focus is on contract-time algorithms for soft real-time stream processing on GPUs. In general, Anytime Algorithms have four properties:

1. Quality & Execution Time trade-off -To measure the runtime performance, anytime algorithms generally introduce a quality function which is a monotonic function of the amount of time available to the algorithm. It is therefore necessary to construct a performance profile of output quality at design-time and select a set of operating points along the trade-off curve at runtime. A task is monotone if the quality of its intermediate result does not decrease as it executes longer. Such monotone tasks are available in many problem domains, including numerical computation, statistical estimation and prediction [7], heuristic search [8], sorting, and database query processing [9]. In this study, we will consider the

31
parallel execution of a Parallel A* search algorithm that is executed with a specified contract time.

2. Performance Predictability - Anytime Algorithms must expose control or tuning knobs to traverse the quality-time tradeoff. Such knobs may be in the form of sampling rates and iterative improvement functions which affect the quality in terms of metrics such as accuracy, coverage area, certainty, and resolution (level of detail). Quality-time tradeoffs may be achieved by several techniques such as milestone method, sieve functions and selecting between multiple versions of the algorithm [10]. With the milestone method, the algorithm executes for a minimum duration and then evaluates its progress at checkpoints within the control-flow graph. Based on the remaining time, the algorithm may decide to execute both the mandatory and optional operations, or just the mandatory operations. Sieve functions allow for computation steps to be skipped such that a minimum sampling may be achieved over a shorter duration. Alternatively, the application may be implemented with multiple versions of the same algorithm such that computationally intensive implementations may be swapped with quicker but less precise versions. For each of the above techniques, it is necessary that alternative execution approaches have the facility to measure the quality with explicit metrics in the current mode. Languages such as Flex provides language primitives with which the programmer can specify the intermediate result variables and error indicators, as well as the time instants to record them [11], [12].

3. Correctness measure - As the execution pathway of an anytime algorithm is a function of the workload and application goals, it is only determined at runtime. Thus, the outcome may be among a range of possibilities, and it is essential to ensure that outcome is correct. Consider the example in Fig. 1, where the application has a contract-time of 11s. At runtime, the algorithm may choose among any of the alternate execution paths, where paths on the left are more computationally intensive (time consuming) but more accurate than those to the right. In this example, the algorithm chooses path 1A which consumes 8.2s. As the remaining time is relatively small for the rest of the execution, the algorithm decides at intermediate checkpoints to execute less accurate and faster paths in later phases (i.e. 2C followed by 3D) to finish by the deadline. In order to check the correctness of combinations of alternate execution paths, one approach would be to verify if the string $1A - 2C - 3D$ is a subset in the language of acceptable sequences. More generally, anytime algorithms require a result evaluator to ensure the correctness during the course of execution.

4. Suspend and Resume Execution Anytime algorithms must have the capability to be interrupted either at anytime or at pre-determined checkpoints to output an intermediate result. In addition, they must be able to continue operation using intermediate results.

B. Anytime Algorithms for Parallel Computing

Anytime algorithms have been largely studied on sequential and single execution-path architectures. While early investigations have explored multi-processor architectures [13], our efforts are with parallel GPU computing architectures. This introduces three differences from earlier approaches: (a) The algorithm must be mapped spatially across multiple processing resources (grids, blocks and warps) and this introduces platform-dependent variations in the quality-time tradeoff; (b) Alternative implementations of the algorithm must be compiled into kernels at design time and appropriate kernels must be composed to select an operating point along the quality-time curve; (c) The algorithm specifies the execution of a single thread and the kernel executes hundreds to thousands of threads concurrently. Thus, it is necessary to wait until all threads complete in one kernel till the next kernel is loaded. A primary challenge of kernel composition with the current GPU architectures is to execute similar threads within one kernel and restrict the set of concurrently executing kernels on the GPU.

Our framework has four key elements: (a) profiling the algorithm by timing analysis to partition execution across multiple exploration and exploitation phases, (b) instrumentation for on-line measurement of the progress of the algorithm and the remaining time, (c) interfacing control knobs (or control dimensions) which trade-off output quality with time so that the algorithm can adapt its behavior along those dimensions when needed, and (d) policies for runtime selection of the most appropriate execution path based on (a) and (b).

C. Quality and Resource Profiles

To profile and explore the quality-time trade-off, we employ a framework similar to Quality of Service Resource Manager (Q-RAM) [14]. Q-RAM is an optimization framework to maximize the output quality, given multiple resources and multiple control knobs (control dimensions). The quality-

![Figure 1. Anytime algorithm with multiple optional paths.](image-url)
time and resource-time profiles are computed at design-time and pruned by extracting the convex hull of the quality-resource-time profile for an application. We apply this meta-approach to time-bound computation of Parallel A* searches executed in parallel (PAP*) across a mesh graph on a GPU. Our current study only explores a single resource (time) and single quality metric (map resolution) as our goal is to stream output for a large number of vehicles continually as the weights of the graph change due to congestion. Other quality metrics such as route update rate and the incremental length (e.g., 0.5, 1, 3km) for which the route must be calculated, may be included.

D. Organization

This paper is organized as follows: The next section provides a brief background on the GPU architecture and timing analysis for sample algorithms. Section 3, describes the construction of the PAP* algorithm followed by its quality-time profiling and runtime kernel composition. Section 4 presents performance results of the anytime implementation of the PAP* algorithm and its effectiveness in exploiting the quality-time trade-off. We conclude with a summary of the related work and future directions.

II. GENERAL PURPOSE GPU TIMING ANALYSIS

In recent years graphics processing units (GPUs) have provided us inexpensive highly parallel computation power [2], [5]. Harnessing this power was relatively difficult until GPU architectures started to support general purpose programming languages (e.g., C, C++, Python) in addition to graphics APIs [3], [4], [15]. So instead of expressing algorithms in terms of programmable shaders, they can be constructed using more general programming models like CUDA.

A. CUDA GPU Architecture

CUDA provides parallel programming extensions to the C programming language. It gives the programmer direct access to the virtual instruction set and explicit control of the memory in the device. The programmer specifies the CPU and GPU functions in terms of thread and thread-block layouts. The finest unit of execution in CUDA is a thread. Threads are grouped into blocks and blocks are organized into a grid (Figure 2). Each multiprocessor executes one block at a time, and each block runs on just one multiprocessor. The CPU and GPU run different code. The CPU executes the main program and sends tasks to GPU in form of kernel functions. While there can be many kernel functions defined in a program, there can be only one kernel running on the device at any point in time.

In our measurements, from the CPUs perspective, kernel invocation endures small overhead (<0.12ms). Thus, in the rest the experiments we ignore this overhead by repeating the experiments 100 times on the device with just one invocation and then divide the measured time by 100. This way, the overhead is amortized and excluded from our measurements.

![Figure 2. CUDA programming model: showing CPU (host) to GPU (device) communication. Kernels are mapped to grids on the device, which execute blocks of threads concurrently.](image)

B. An Example of GPU WCET Measurement

We now discuss the specifics of Worst Case Execution Time (WCET) measurement for an example CUDA program for vehicle routing across a street map. This step is required to construct the quality and execution time profile of an algorithm at design-time. Figure 3(a) depicts the high-level Control Flow Graph (CFG) which is largely the decision paths a vehicle takes when it reaches an intersection. The left side of the control flow graph, which consists of the outer and inner loops, depicts the worst-case execution path for the program. The outer loop determines the high level routing for all the vehicles in the grid, while the inner loops calculate the finer routing details for each vehicle in the grid. The execution time for the program is largely influenced by the behavior of these two loops, since most of the time is spent performing these calculations. Hence, there is a need to obtain good estimates of the WCETs for both of these loops.

We achieve this by instrumenting the code and observing the clock during the instrumentation points. The CUDA architecture has a special register, named %physid, that keeps track of the multiprocessor on which the current thread is executing. Each multiprocessor has its own individual clock that can be easily read from the kernel using the clock() system call. The instrumentation checkpoints (timestamp capture) are shown in Figure 3(a) by the red triangles placed at various points along the CFG. Each instance of the inner loop was forced to execute for eight iterations since each vehicle’s routing has at most eight road segments to choose from at an intersection. The outer loop was run for 100 iterations to capture the average and worst case. When we plotted the behavior of the application for a varying number of iterations of the outer loop we noticed that a run of 100 iterations captured most of the vehicle routing scenarios.

C. Timing results

Fig. 3(b) shows the execution times for the inner loops for each thread that executes on the CUDA processor. While the graph shows a fair distribution of the execution times for the inner loop (largely due to map data fetches from global memory on the device), the results are fairly well bounded.
within 0.5ms. Hence this can be considered the WCET profile for the inner loop. As was the case for the inner loop, we find an upper bound for the execution of the outer loop is approximately 3.6ms. Fig. 3(c) shows the absolute execution times from the start of the kernel for all warps. A warp is a group of 32 threads scheduled concurrently on a single stream processor. The execution profile of the thread closely follows that of the outer loop, except that the time taken is two orders of magnitude higher. The execution times for the complete thread is dominated by the times for the outer loop. We obtain an upper limit for the execution times for entire program to be 349.4ms. Since we schedule 850 threads at a time on the CUDA processor (this is a constraint of this particular processor model), the first batch takes at most 349.4ms, the second takes a further 349.4 ms, and so on. Using this approach, we can instrument the entire application and extract timing measurements (see Fig. 3(d)) for both design-time analysis and runtime adaptation.

We adopt the same measurement technique for the Parallel A* search algorithm we use as a case-study for the remainder of this paper.

III. PAP* PARALLEL SEARCH ALGORITHM

We step through the construction of a Parallel A* algorithm that is capable of trading-off execution time for output quality. To enable the contract-time capability, the algorithm must expose one or more control knobs (control dimensions) such that the execution time and quality are profiled for all operating points at design time. At runtime, the algorithm must be capable of adjusting the operating point based on the remaining time and goals of the application. This case study provides a simple walkthrough this process. A fundamental limitation of the CUDA device is that each of its multiprocessors follows a SIMD architecture. This limits the efficiency for algorithms with divergent thread processes. Our goal is to adapt the A* algorithm to run efficiently on GPU given these limitations. The adaptation of this algorithm to CUDA was inspired by the work done in [16] and [17].

A. Graph representation

Graphs are commonly represented by their adjacency matrices. However since we are mostly dealing with sparse graphs of street topologies, such a representation will be very wasteful in terms of memory space. We therefore store the graph in memory using adjacency lists which are more compact. Adjacency lists of the vertices of each graph are packed into a single array. In addition, different graphs are packed together into a single compact adjacency list. So on a higher level it may look like we have a graph with several connected components, i.e. a jungle. Each query is confined to one of these connected components. Each vertex stores the starting index of its adjacency list in the global compact adjacency list. Vertices of all graphs \( G_i(V_i, E_i) \) are represented as an array \( V \). An array \( E \) stores the adjacency lists of all vertices of all graphs. Each entry of the edge array \( E \) refers to a vertex in vertex array \( V \).

B. Parallel A* searches in Parallel (PAP*)

In PAP*, the goal is to run multiple instances of the A* search algorithms while each of them can take advantage of the parallel nature of the GPU. The problem is thus: given a weighted directed graph \( G(V, E, W) \) with non-negative weights, and a set of source vertices \( S \), and a set of the corresponding destination vertices \( D \), find the shortest path from each vertex in \( S \) to its corresponding destination in \( D \).
C. CUDA implementation of PAP*

In our implementation we use a set of arrays for each of the graphs we have: a vertex array $V$, and edge array $E$, boolean masks $F$ and $C$ of size $|V|$ which record the search frontier and the finalized nodes, and also a weight array $W$ of size $|E|$. Also we have a cost array $C$ which keeps record of the shortest path from the source to the expanded nodes.

Each thread on the device is assigned to one node of a graph. In each iteration, each vertex checks to see if it is in the frontier list. If yes, it fetches its current cost from the cost array $C$ and its neighbors weights from the weight array $W$. The cost of each neighbor is updated if greater than the cost of the current vertex plus the edge weight to that neighbor (relaxation). At the end of the execution of the kernel, a second kernel compares cost $C$ with updating cost. It updates the cost $C$ only if its cost is higher. The updating cost array reflects the cost array after each kernel execution for consistency.

The second stage of kernel execution, shown in Fig. 4, is required as there is no synchronization between CUDA blocks. To find the minimum cost vertices in the search frontier of each graph we first find the minimums per CUDA block and then find the global minimums in a separate kernel execution.

Another decision that should be made in the runtime configuration of the kernels is the way the threads are organized into blocks and the grid. The next section briefly explains the idea.

D. Multiple parallel queries

For the experiments in this paper, we use fixed and identical queries with the source and destination being the two opposite corners of the mesh (worst case), while the edge weights are chosen randomly and are different in different meshes. There are different modes in which each query can be run. For example, we can run a single query with maximum parallelism (maximum number of blocks). Another option is to pack multiple queries together and run them together as if we are running one query on a larger graph. The implementation structure of the PAP* allows us to do this on a single graph or multiple graphs seamlessly. This decreases the amount of parallelism as seen by each individual query (see Fig. 5), but as will be explained later, it might increase the overall performance of the system, as there might be more parallelism to exploit in multiple queries than in a single one.

IV. PAP* - CONTROL DIMENSIONS

For the query processing routine to adapt to the available time, we provide degrees of freedom along which the query processor can change the operating point in the algorithm to trade-off route accuracy and execution time.

A. Platform Independent Control Dimensions

In this case study, we have used a regular mesh graph, along with three lower resolution versions of it ranging from 1024 nodes down to 16 nodes, as shown in Fig. 6. Level 0, is the original graph and Levels 1, 2, and 3 are lower resolution meshes derived from the original graph. These abstractions speed up execution and provide partial results in lieu of a complete path. The higher level meshes hide the details of the graph and thus shrink the search space. Lower levels with more details lead to a larger search space, but provide more accurate search results. In this experiment, we assigned scores 6, 4, 2, and 1 for routes calculated in the layers 0 to 3 respectively. These scores are application dependent but generally illustrate that the more detailed paths are assigned higher scores to maintain monotonic quality with increasing processing. For each search query we consider these different versions of that graph. It is possible to change the level at runtime. The experiments are repeated for a number of runs to distinguish the measurements from the noise.

B. Platform-dependent Control Dimensions

The PAP* algorithm is very flexible in terms of mapping execution to the thread-block level architecture on the device. We observe that for the GPU, architecture-specific constraints provide a large dynamic range for effectively scheduling threads for a given workload. It can be run with different thread/block numbers given any search graph size. This provides us with an optimization parameters to schedule parallel searches together on the device. Fig. 7, shows the variation in execution time of a single A* search with different graph sizes and blocks numbers. The sweet spot for a single isolated search is when we have as many blocks as possible, provided that the number of threads on each multiprocessor doesn’t go below the number of cores per multiprocessor.

![Figure 4](image1.png)

**Figure 4.** PAP* consists of four CUDA kernels. Three of them, relaxation, find minimums, and accumulate/update run in a tight loop until the target node is found.

![Figure 5](image2.png)

**Figure 5.** Merging queries together and running them together in parallel versus running maximally parallelized single queries, serially.
Figure 6. Multi-level graph: we maintain 4 different layers for the graph that we are using. Each higher layer represents a coarser level approximation of the lower level graph. This figure is not to the scale and is just for demonstration of the idea.

In this figure the number of multiprocessors is 27, with 8 cores per multiprocessor. Thus, spreading the computation to maximally use all multiprocessors while running at least 8 threads per multiprocessor allows for the highest throughput for this algorithm. Fig. 8 is a plot of the algorithm’s execution time using different number of blocks. Here the number of threads per multiprocessor would be number of vertices divided by the block count.

C. Kernel Composition

For an anytime application, a pool of kernels with fixed time-quality trade-offs are aggregated in the host at design-time. At runtime, based on the load and quality-level goals of the application, the appropriate kernel is selected on-demand and queued to execute on the device. As shown in Fig. 9(a), when we compose a kernel in this manner from several smaller kernels, the running time of the composed kernel is equal to the sum of the running times of the kernels in the longest path from start to completion (critical path). So, on a device that has $M$ multiprocessors a kernel composed of $M$ parallel 1-block search kernels takes the same amount of time to complete as a single 1-block search. Thus, given the information in Fig. 7 the best way to run 32 searches is to compose a new kernel from 32 1-block searches, rather than running 32 32-block searches sequentially. However, if we have just a single query then we are better off running a single kernel with 32 blocks.

Each of these searches is one CUDA kernel and the CUDA device doesn’t allow us to run more than one kernel at a time. So, to run multiple kernels (e.g. 1-block A* searches) concurrently, we would have to make a new kernel that can effectively emulate multiple parallel kernels at runtime, while from the viewpoint of the device there is just one kernel executing [18].

In this scheme the number of blocks is equal to or less than the number of multiprocessors on the device. So, to assign blocks of a kernel to different multiprocessors, we can use block indices. So we’ll have a nested $IF...THEN...ELSE$ structure that runs specific code based on the block ID. We don’t assign more than one kernel to a single block as far as possible. More specifically, the overall structure of each thread looks like the following pseudo code:

Algorithm 1: Kernel composition example.

```
input: K_i();
bid ← blockIdx.x;
if bid < K_1.size then
    K_1();
    K_2();
    if bid < K_3.size then
        K_3();
    else if bid ≥ K_3.size then
        K_4();
        K_5();
    end
else if bid ≥ K_1.size then
    K_6();
end
```

So, assuming that $K_i.size$ is the number of blocks that kernel $K - i$ requires, blocks 0, 1, ..., $K_1.size - 1$ run kernel $K_1$ and then $K_2$ and then blocks 0, 1, ..., $K_3.size - 1$ run kernel $K_3$, and so on. Fig. 9(b) demonstrates how these base kernels are packed together. So we have effectively made a new kernel that contains at its core six basic kernels.

![Figure 7. PAP* execution time for the multilevel graph of figure 6.](image)

![Figure 8. PAP* execution time for different block numbers](image)

36
combinations of kernels incurs a static compilation cost plus a negligible constant run-time cost. So at any point of time we can invoke one of these six kernels, or any kernel composed using a subset of them. Using these combinations increases the flexibility of the scheduling. We can run kernels in parallel to increase device utilization when a single kernel cannot guarantee high occupancy of the device.

V. ANYTIME PAP* EVALUATION

A. Offline scheduling: quality vs. execution time trade-off

Based on the timing information for different operation modes as summarized in Fig. 6 and Fig. 7, we developed a dynamic CUDA thread-kernel scheduler which selects parameters for different queries based on the available time, to maximize the overall query response quality. We have designed an scheduler that given the total amount of time available, finds the best scheme to combine and schedule the required kernels to run the queries. This scheduler, named offline, creates batches of queries and runs all queries in that batch in one of the levels of the multilevel graph.

However, due to variations in actual kernel execution times and those estimated at design time we need to be able to adapt to the variation in the available time. So, after each batch completion, the scheduler optimizes the current schedule given the actual amount of time left to finish the queries. Fig. 10 shows the quality versus contract time for scheduler offline, along with the overall quality score that static schedules can achieve, while running a total of 6,144 search queries. The static level scores are consistently lower than the offline scheduler as the scheduler cannot adapt and misses the deadlines for queries with contract time below the minimum time required for the specified level of the graph. offline finds the optimal graph levels and composition to run the kernels so as to meet the deadline while maximizing the overall quality of results.

B. Online Scheduling

Since the scheduler might create very large kernels in its optimization of the quality score, variation in actual kernel execution times can be large. So if for some reason one of the batches takes much longer than was anticipated by the scheduler, then the deviation from the optimal plan might be so large that forces us to unduly decrease the quality of the subsequent batches which will hurt the overall quality of the results. So the scheduler will have more flexibility if it had the option of modifying a schedule while it is running. We design a new scheduler, named online that can do this. If the scheduler detects that a query is taking longer than the projected amount of time, it can adapt to the new situation and make sure that the system will succeed in responding to all queries in a timely manner while having a larger search space to optimize the schedule. For this purpose, online probes the system continually and determines when the projected time bounds have been violated by some predefined margin. In that case, a new schedule is composed based on the new time bounds, and put into effect immediately. So, it can change the execution time of a query batch even after it has been dispatched by the host CPU by changing the level in the graph from the current location on the map.

Fig. 11 shows the projected and the actual execution times for three query batches each consisting of 6,144 queries. We have manually forced the actual execution of the kernels to take longer to demonstrate the ability of the system to...
adapt to new time bounds. Fig. 12(a) shows the accumulated execution times of kernels running in the order proposed by the scheduler. As can be seen in Fig. 12(a), when dispatching kernel 3, offline detects that we are short of time based on the actual execution times, we make a new execution plan that finishes the query processing within the given time bounds. However, online detects the situation while running kernel 2 and changes the execution plan accordingly to meet the time bounds. As we expect, the sooner we detect an execution plan that has gone awry the better we can fix it because more time is left and we have more candidate execution plans to choose from. Thus as can be seen in Fig. 12(b), online has been able to score higher than offline.

VI. RELATED WORK

The term “Anytime algorithm” was introduced by Dean and Boddy[6] in their work on time-dependent planning during the late 1980’s. Horvitz[7] introduced the flexible computing model for time-critical decision making and planning algorithms in Artificial Intelligence. Following this, several studies in the AI community focused on composing anytime algorithms to more complex systems for sensor interpretation and path planning[19], [20], search[8], and evaluation of belief networks[21]. In the early 1990s, the real-time community explored scheduling approaches for imprecise and approximate computing[10], [13]. The Flex language was developed with time as a first-class citizen to specify timing and performance requirements within the algorithm[11], [12]. This contributed to approaches for performance polymorphism[22] such that mandatory and optional computations may be selectively executed based on the available execution time. These efforts were seldom evaluated on computing platforms or applied to complex problems.

VII. CONCLUSION

We present early efforts in time-bounded algorithm execution on a GPU’s parallel architecture. The focus of this work was on the construction of anytime algorithms which effectively trade-off output quality and execution time to provide approximate results within a contract-time. We present the construction, instrumentation, on-line measurement and runtime scheduling of such anytime algorithms. Through a case study with Parallel A* search, we demonstrate the feasibility and effectiveness of the proposed approach. We have currently considered quality-controlled algorithms and plan to include admission-controlled and auto-tuning algorithms.

REFERENCES

Formal Verification of Real-Time Embedded Software for Multicore Platforms

Sagar Chaki Arie Gurfinkel
Carnegie Mellon Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA, USA
{chaki,arie}@sei.cmu.edu

Abstract—Real-time embedded software (RTES) plays an increasingly critical role in all aspects of our lives. Ensuring that RTES behave in a predictable, safe and secure manner is an open challenge. The emergence of multicore hardware has introduced an additional level of complexity to this arena. In this paper, we take the position that formal verification is a very promising approach to find concurrency-related problems in multicore RTES. We argue that multicore RTES present unique domain-specific restrictions (and new challenge problems) that can be leveraged (and targeted) by formal verification to yield solutions that are precise, scalable, automated, and applicable to source code. We also believe that this effort will increase synergy between formal verification and real-time scheduling.

Keywords-formal verification; multicore; schedulability

I. INTRODUCTION

Real-time embedded software (RTES) permeate our technology-driven existence, controlling a wide variety of systems ranging from nuclear power plants and energy grids to cars and cell phones. Ensuring that RTES operate correctly is therefore of paramount importance to the preservation of our modern way of life.

Despite a wide body of research and development effort, ensuring the safe and secure operation of RTES remains an open challenge. Instances of RTES failure, causing varying degrees of damage and destruction, continue with unfailing regularity. The technological and economic compulsion for migrating to multicore platforms further exacerbates concurrency-related issues for RTES. Clearly, the status quo is unsatisfactory, and new insights and techniques are needed in order to advance the state-of-the-art in solving this crucial problem.

In this paper, we argue for a sustained effort in applying formal verification in a precise and targeted manner to find errors in RTES deployed on multicore hardware. In particular, we focus on concurrency-related bugs, such as race conditions and deadlocks, that are notorious to detect and eliminate via traditional validation methods like testing. We argue that multicore RTES present unique domain-specific restrictions and novel challenge problems. Further, formal verification that leverages these restrictions, and targets these challenge problems, will yield precise, scalable, and automated solutions that are applicable to source code. Finally, this effort will strengthen the interplay between real-time scheduling and formal verification.

We structure our position in the following steps: (i) background, (ii) appropriateness of formal verification; (iii) existing approaches; (iv) targeted formal verification; and (v) conclusion.

II. BACKGROUND

Ensuring the correct behavior of programs is a fundamental challenge in the computational sciences. One answer to this challenge is formal verification. The main focus of formal verification is ensuring the “functional” correctness of programs – e.g., that a program properly sorts a list of numbers, does not deadlock, interacts with another program via a specific sequence of actions, etc. Research in formal verification spans several decades, and has made tremendous progress in terms of the degree of automation and applicability to realistic programs. For example, model checking [1] is a fully automated approach to verifying temporal logic specifications over finite Kripke structure models. More recently, the SLAM project [2] has pioneered the application of abstraction, model checking, and refinement techniques to enable automated verification of C programs. In addition, there have been several instances of applying abstract interpretation [3] to the static analysis of industrial software. We believe that formal software verification – by which we mean the gamut of formal techniques for analyzing software statically – is uniquely suited and sufficiently mature for reasoning about multicore RTES.

III. APPROPRIATENESS OF FORMAL VERIFICATION

The road to practical adoption of formal verification has been long and hard. In theory, techniques like model checking and abstract interpretation are “push-button”. In practice, however, they are often implemented in tools that must be used by an expert in order to yield useful results. This makes them expensive, and less desirable than cheaper alternatives like testing. We believe, however, that several factors render formal verification to be the superior alternative in the context of finding concurrency-related bugs in RTES:

1) Cost of failure. RTES are often deployed in safety-critical situations where the cost of failure is very high or catastrophic. A typical example is a nuclear power plant, a medical device, or an airplane where a failure could lead to loss of human lives. Other situations,
where failures cause high economic damages, are cell phone towers and electric grids. High cost of failures justify the non-trivial up-front cost of applying formal verification.

2) **Non-determinism.** Concurrency-related bugs are extremely difficult to unearth via conventional code review and testing. There are known cases where bugs have remained uncovered despite years of testing. The main reason behind this is the large number of execution paths that a concurrent program is able to exercise due to non-deterministic choices between different thread interleavings. Even the best testing efforts cover only a small fraction of a program’s state space. Formal verification is exhaustive and covers all possible program executions and thread interleavings.

3) **Multicore Platforms.** The increased concurrency (and indeed the emergence of real-­concurrency where multiple threads are able to execute simultaneously) in multicore platforms amplifies the non-determinism of software, and the relative advantage of formal verification over non-exhaustive methods. The migration to multicore platforms is driven by technological and economic incentives that are unlikely to be reversed.

4) **Certification.** Finally, software verification is an important aid in certification. For example, the DO-178C standard explicitly requires the application of formal verification. Other standards, such as the ECSS-E-ST-40C [4], suggest formal verification when alternatives (such as testing) are arguably inadequate. Based on the points above, we believe that such an argument is indeed plausible in the case of multicore RTES.

IV. EXISTING APPROACHES

It is worth recalling that our goal is to develop analysis tools that target concurrency-related issues and are precise, scalable, automated, and applicable to source code. We believe that existing formal verification tools (based on static analysis, software model checking, or a combination of the two) fall short on one or more of these accounts.

There is a large body of work on formal verification of models spanning several decades. Of special relevance to us is the verification of timed automata [5] and hybrid automata [6], timed process calculi – such as RTSL [7], ACSR [8] and PARS [9] – and timed Petri nets [10]. They represent foundational ideas and results that guides the verification of multicore RTES. However, the focus of this paper is the analysis and verification of source code, which is not only complementary to that of models, but brings in its own set of challenges as well.

The problems in analyzing source code have been the main focus of commercial static static analysis tools (such as Coverity, Klocwork, Grammatech and Fortify). These tools are targeted toward finding sequential problems (such as NULL pointer dereferences and buffer overflows) over a large corpus of code. They are automated and scalable, but challenged by a lack of precision (i.e., many false warnings). A main reason behind their imprecision is that they strive for generality in terms of the programs they are able to handle. This means that they are only able to assume – and thus leverage – a limited set of domain-specific restrictions.

In the context of sequential software, there has been several success stories of applying formal verification to develop analysis that is precise, scalable, automated, and applicable to source code. Two notable examples are the SLAM [2] and the ASTREE [11] projects. We believe that the success of both projects stems in large part on their focus on specific problems, and their use of domain-specific restrictions, which enabled them to develop and refine very specific solutions.

For example, the SLAM project focuses on ensuring that Windows device drivers interact with the kernel only in prescribed ways. These properties are expressible as finite state machines and verifiable over Boolean abstractions of the source code. These restrictions enable SLAM to effectively use predicate abstraction and control-flow reachability over Boolean programs as the underlying verification engine.

The ASTREE project focuses on detecting numeric errors and overflows in avionics software. The restriction to specific programs and properties enables the researchers to develop special-purpose abstract domains, such as ellipsoid [11], that are very effective for computing the invariants needed to prove (or disprove) the target properties.

In the same spirit, we believe that novel approaches that target specific concurrency-related problems, and leverage domain-specific restrictions, must be developed for multicore RTES.

V. TARGETED FORMAL VERIFICATION

The development of targeted formal verification tools for multicore RTES is aided by unique restrictions afforded by this domain:

1) RTES have deterministic scheduling, which restricts the amount of concurrency and possible interleaving between threads. Leveraging this restriction provides a way to ameliorate the state space explosion problem, a major obstacle to scaling formal verification.

2) The scheduling in RTES is governed by a precise priority-based mechanism. In the special case of fixed priority scheduling, thread-interleaving is even more restricted. For every thread $T$, the set of other threads that might preempt $T$ is statically known. In more specific cases, e.g., for RMA-scheduled [12] RTES, even the number of times a thread may preempt another is bounded and known statically. These restrictions offer further opportunities for improving scalability and automation.

3) RTES involve restricted use of complex programming constructs. In many cases, language features like dy-
namic memory allocation and recursion are disallowed or severely restricted. While these restrictions are driven primarily to make the runtime behavior of the system more predictable, they also enable the automated extraction of precise models for the purposes of formal verification.

4) Multicore RTES also open up new challenge problems for formal verification. One source of problems is the emergence of real concurrency. In a multicore platform, two threads are able to run simultaneously on different cores, and thus access shared resources (like memory and cache) leading to problems like race conditions, deadlocks, and bus overload. Techniques for ensuring mutual exclusion, such as the priority ceiling protocol, break down in a multicore environment. Another source of problems is the added dimension of cores in terms of resource allocation. Allocating threads to cores (statically or dynamically) so as to optimize any desired utility measure is a new challenge.

We believe that these restrictions and challenge problems will aid in the development of targeted and effective formal verification for multicore RTES. At the same time, we believe that this endeavor will further strengthen the synergy between formal verification and real-time scheduling.

VI. Conclusion

Ensuring the predictability, safety, and security of RTES remains a fundamental and open challenge. The emergence of multicore hardware only increases the complexity of the problem. However, multicore RTES present unique domain-specific restrictions and new challenge problems. We argue that these restrictions and challenge problems aid the potential of formal verification to lead to precise, scalable, and automated techniques for finding concurrency-related issues in source code. Additionally, we believe that research in this direction effort will increase synergy between formal verification and real-time scheduling.

ACKNOWLEDGMENT

Several people have participated in discussions that led to the ideas and arguments in this paper. In particular, we would like to thank Russell Kegley and Ben Watson from Lockheed Martin, USA, and Mark Klein from the SEI.

REFERENCES


Multi-domain Modeling of Cyber-Physical Systems
Using Architectural Views

Ajinkya Bhave, Bruce Krogh
Dept. of Electrical & Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15217
{jinx, krogh}@ece.cmu.edu

David Garlan, Bradley Schmerl
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15217
{garlan, schmerl}@cs.cmu.edu

Abstract—Designing cyber-physical systems (CPSs) increasingly requires the use of multi-domain models throughout the development process. Ensuring consistent relationships between various system models is an important part of an integrated design methodology. This paper describes an architectural approach to reasoning about relations between heterogeneous system models. The run-time base architecture of the system is used as a unifying representation to compare the structure and semantics of the associated models. Each model is related to the base architecture through the abstraction of an architectural view, which captures structural and semantic correspondences between model elements and system entities. The use of the architectural view framework to relate system models from different domains is illustrated in the context of a quadrotor air vehicle.

Keywords—architectural views; multi-domain modeling; cyber-physical systems; component-connector;

I. INTRODUCTION

Today’s complex cyber-physical systems (CPSs) are created using models throughout the system development process, an approach referred to as model-based development (MBD) [7]. Models allow designers from different disciplines to develop and evaluate design alternatives within the context of formalisms relevant to selected aspects of the system. Each representation highlights certain features and occludes others to make analysis tractable and to focus on particular performance attributes. A particular modeling formalism typically represents either the cyber or the physical elements well, but not both. For example, differential equation models represent physical processes well, but do not represent naturally the details of computation or data communication. On the other hand, discrete formalisms such as process algebras and automata are well suited for representing concurrency and control flow in software, but are not particularly useful for modeling continuous phenomena in the physical world. Thus, the heterogeneity of elements in CPSs requires multiple perspectives and formalisms to explore the complete design space. Ensuring consistent relationships between various system models is an important part of the integrated MBD methodology.

We have developed the CPS architectural style as a system-level representation that is not prejudiced towards either the cyber or the physical side [2]. Architectures are annotated structural representations that describe systems at a high level of abstraction, allowing designers to determine appropriate assignment of functionality to elements, evaluate the compatibility of the parts, and make trade-offs between different quality attributes such as performance, reliability, and maintainability. This paper describes how the CPS architecture for a system provides a unified point of reference for multi-domain models based on heterogeneous formalisms. Our approach is to define relationships between system models at the architectural level, rather than developing a universal modeling language or a meta-modeling framework for translating between models from different formalisms. We believe that an architectural approach provides the right level of abstraction: one that captures the structure of and interdependencies in a system without attempting to comprehend all of the details of any particular modeling formalism.

The next section describes the use of the CPS architectural style to define base architectures for cyber-physical systems. The STARMAC quadrotor is introduced as a case study in this section. Section III introduces the concept of architectural views as means of relating heterogeneous models to a common base architecture. In Sect. IV, we illustrate the creation of three heterogeneous views in the context of the STARMAC quadrotor. Section V describes related work in this area, and the concluding section discusses ongoing work to extend our approach to represent and analyze multi-domain model consistency for CPSs.

II. A UNIFYING ARCHITECTURAL REPRESENTATION

Architectures are often represented using a collection of architectural perspectives, which represent a set of related concerns [3]. The component-and-connector (C&C) perspective models a system as an annotated graph of components and connectors, in which the components represent principal computational and physical elements of a system’s run-time structure, the connectors represent pathways of communication and physical coupling between components, and annotations represent properties of the elements [10]. In this work, the term ‘architecture’ is synonymous with a C&C architecture, because all the modeling formalisms of interest to us focus on analyzing properties and behavior of entities defined in the C&C architecture of the system under study. The CPS architectural style is defined in the C&C perspective.
Current C&C architectural styles, which focus primarily on software and computational infrastructures, are not comprehensive enough to describe a complete CPS. A CPS contains physical elements in addition to cyber entities, and includes elements representing interactions between these two domains. We have addressed this shortcoming through the development of a CPS architectural style [2] that augments traditional cyber architectures with elements corresponding to physical dynamics and laws. This architectural extension allows us to create a run-time representation of the complete CPS, called the system’s base architecture (BA). The BA of a CPS is an instance of the CPS architecture style, which contains all the cyber and physical components and connectors that constitute the complete system at runtime.

The BA should contain enough detail to convey the nature of information and physical quantities flowing between components. In addition, the communication mechanism between components and relations between physical variables should be defined by the appropriate connectors. For new CPSs, the BA is built during the design phase from validated requirements and system specifications. For legacy CPSs, the BA is inferred from the implemented system, existing documentation and system models, and the knowledge of the system designers. In either case, we assume that the BA evolves as the design of the CPS evolves, throughout the system development lifecycle.

The following example of a real-time, embedded, multi-loop feedback system, will be used to illustrate the concepts of architectural views and their relation to the BA. The Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC) [6] is a quadrotor platform developed to test algorithms that enable autonomous operation of aerial vehicles. The aircraft has four rotors for actuation, arranged symmetrically about its body frame. The vehicle has a sensor suite consisting of an inertial measurement unit (IMU), a Global Positioning System (GPS) unit, and sonar. It implements a hierarchical control system, with a low-level attitude controller (AC) and a high-level position controller (PC). A remote ground station controller (GSC) generates reference trajectories for the quadrotor to follow, and has joysticks for control-augmented manual flight. The two onboard controllers communicate through a serial link. Communications between the PC and the GSC are managed over a WiFi network, using the UDP protocol.

Figure 1 illustrates the BA for the quadrotor. This BA was created from the STARMAC implementation. The complete run-time architecture is modeled in the CPS style, which allowed us to represent both the cyber components (control algorithms and real-time software) and the physical dynamics (forces and torques imparted to the vehicle frame from physical sources). A more detailed description of the complete quadrotor CPS architecture is provided in [2].
III. ARCHITECTURAL VIEWS

A CPS is typically described and analysed using multi-domain models, where each model focuses on a fixed set of concerns about the underlying system. Figure 2 shows four models of the STARMAC quadrotor that represent the same system from the physical, control design, software, and hardware domain perspectives.

In virtually all analysis tools, such models are constructed as collections of interacting components or modules. Thus, each model has a structure that can be viewed as an architecture with syntax and semantics defined by the particular formalism underlying the design of the tool. Our goal is to define consistent relations between these models at some level of abstraction.

The approach proposed here focuses specifically on architectural views which represent the architectures of system models as abstractions and refinements of the underlying shared BA. In this context, well-defined mappings between a view and the BA can be used as the basis for identifying and managing the dependencies among the various models and to evaluate mutually constraining design choices. The BA thus becomes the repository for retaining results from various analyses and designs so that the interdependencies are explicit.

This gives us the ability to reason about relations between models by studying their individual mappings to the BA of the system.

Current tools do not provide insights into the relationships between such heterogeneous models of a CPS. This represents a problem for architectural modeling, since it is generally impossible to understand how design decisions or analyses in one view impact those of another. From a structural perspective, an architectural view supports the description of a derived architectural model to abstract over details that are irrelevant for a particular analysis. The following definition formalizes the concepts of the BA and architectural views that we have described informally thus far.

**Definition 1.** An architectural view \( V \) for a modeling formalism \( \mathcal{M} \) is a tuple \( < C_V, R^M_{V}, R^V_{BA} > \) where:

- \( C_V \) is the component-connector configuration of the view, with the types, semantics, and constraints defined by the modeling formalism of the view
- \( R^M_{V} \) is a relation that associates elements in the model with elements in \( C_V \)
- \( R^V_{BA} \) is a relation that associates elements in \( C_V \) with elements in the BA

Figure 3 shows the conceptual relationship between system
The relationships between elements in a model and entities in the BA will not generally be one-to-one. \( R^M_{V_B} \) is either one-to-one or an encapsulation of model entities, as defined by the modeler’s choice of grouping. It effectively creates a “componentized” version of the model and allows grouping of multiple elements in the model to a single element in the view. \( R^V_{BA} \) is an encapsulation/refinement relation, which enables the system architect to group specific components and connectors in the view and map them to subparts of the BA. Some correspondences are declared explicitly by the architect while other correspondences are inferred, based on the semantics of the underlying view formalism.

One-to-many (encapsulation) and many-to-one (refinement) maps are allowed. However, many-to-many maps are not allowed since this can lead to inconsistent connections being hidden inside the encapsulated components. The component-connector structures resulting from carrying out element encapsulations on a view and on a BA are called an encap-view and an encap-BA, respectively.

IV. ARCHITECTURAL VIEWS OF THE QUADROTOR

This section describes how heterogeneous models of the quadrotor can be related to the BA through architectural views. The choice of the modeling domains is motivated by the analysis and verification activities typically found in the design process of embedded control systems. In this case, the STARMAC design team documented the software subsystems and the hardware architecture of the vehicle [1]. We modeled the quadrotor physical dynamics in MapleSim from first principles, as well as studying the vehicle dynamics from existing control system models in Simulink.

A. Control View

From a control engineer’s perspective, the quadrotor system can be viewed as a signal flow (Simulink) model. The position and attitude controller components in the BA are represented by the Robostix and emph{Gumstix} subsystems in the Simulink model. The vehicle dynamics are represented by the star-mac_dynamics block, and the GPS and IMU sensors are defined by the Superstar_II and the 3DM blocks, respectively. Figure 4 illustrates the creation of the control view from a Simulink model. The relation \( R^M_{V_B} \) maps each top-level Simulink block to a component, and each group of signal lines between them to a connector, resulting in the control view’s \( C_V \).

The semantics for the \( C_V \) are derived from the underlying signal flow semantics of the Simulink metamodel. For example, every connector in the control view represents a (cyber or physical) signal. Hence, semantically equivalent connectors between two components in the BA can be mapped to a single connector in the control view under this particular \( R^V_{BA} \). The mapping of four cyber-physical (C-P) connectors between the attitude controller and an encapsulated component (containing VehicleFrame) in the BA to a single connector between Robostix and Starmac in the view is shown in Fig. 5.

We disallow many-to-many maps between macro elements because the following type of situation could arise. Suppose that the architect decided to group the Robostix, Gumstix, and GPS components of the control view as one macro element. The architect also groups the position controller, attitude controller, and GPS components in the BA into a single element, and associates it with the macro element in the
An inconsistent connector existing between Robostix and GPS (highlighted in Fig. 5) will be hidden away in the macro view element, and will not be detected if the control view and BA are compared for some type of consistency check.

B. Process Algebra View

Finite State Process (FSP) [8] is a process algebra that models behavior is modeled in terms of event patterns that denote sets of event traces, called processes. Each event in a trace represents a discrete transition of a system. In general, FSP captures the behavior of cyber elements fairly well, while physical elements are described by abstracting away their continuous dynamics. The components in an FSP view are those entities whose behavior can be described by an FSP primitive process. A connector between two FSP components signifies that the two processes interact with each other through events and describes the protocol for that interaction, again as an FSP process.

The FSP specification of the quadrotor currently abstracts the dynamics of the quadrotor and focuses on the communication between the ground station and position controller. The process algebra view is created by mapping each view entity to an FSP process in the specification, as shown in Fig. 6. The Gnd_Station component is mapped to the GroundStation process, which specifies how the GSC sends setpoints to the PC. The QuadRotor component is mapped to the PositionController process that describes how the ideal closed-loop quadrotor responds to position setpoints. The connector between Gnd_Station and QuadRotor is mapped to an FSP process that specifies the communication protocol between the two. The connector can be one of two types: a lossy connector represents a wireless UDP link, while a lossless connector with retry models a wireless TCP. Having alternative connector protocols allows us to compare the behavior of the overall system depending on the protocol of the connection.

The mapping between the process algebra view and the BA is shown in Fig. 7. The abstraction of vehicle dynamics is represented in the encapsulation of all the physical components in the BA into a single QuadRotor component in the view.

C. Physical View

The physical view models the dynamics of the vehicle in terms of the forces and torques applied by the rotors to the vehicle frame. The nonlinear dynamics of the quadrotor helicopter are those of a point mass \( m \) with moment of inertia \( I_b \in \mathbb{R}^{3 \times 3} \), location \( \rho \in \mathbb{R}^3 \) in inertial space, and angular velocity \( \omega \in \mathbb{R}^3 \) in the body frame. The vehicle undergoes forces \( F \in \mathbb{R}^3 \) in the inertial frame and moments \( M \in \mathbb{R}^3 \) in the body frame, yielding the equations of motion,

\[
\vec{F} = -D_B \dot{v}_V + mg \hat{e}_D + \sum_{i=1}^{4} T_i \hat{z}_B
\]

\[
\vec{M} = \sum_{i=1}^{4} T_i (\vec{r}_i \times \hat{z}_B)
\]

where \( D_B \) is the aerodynamic drag force, \( T_i \) is the thrust from the \( i^{th} \) rotor, \( r_i \) is the distance between the vehicle center of mass and the \( i^{th} \) rotor, and \( g \) is the acceleration due to gravity.

The CPS style enables the formal representation of such dynamic behavior in the overall system architecture. The dynamics model of the quadrotor is implemented in the Modelica language, and the semantics of the \( C_V \) are defined in terms of non-causal interconnections between the effort and flow variables of each attached component’s ports.

The mapping between the physical view and the BA is shown in Fig. 9. The set of view components map to a subset of the elements in the BA.
V. RELATED WORK

Multiple efforts have focused on supporting multi-view, model-based system development. The SAE AADL (Architecture Analysis and Design Language) is an international standard for predictable model-based development of real-time and embedded computer systems [5]. AADL offers a set of predefined component categories to represent real-time systems and it is capable of describing functional component interfaces like data and control flows, as well as non-functional aspects of components like timing properties. However, AADL does not support architectural representation of physical domain entities (except as generic ‘device’ components), nor does it address how heterogeneous views can reconciled.

Ptolemy II is a tool that enables the hierarchical integration of multiple “models of computation” in a single system, based on an actor-oriented design [1]. Even though Ptolemy II supports hierarchy and incorporation of multiple formalisms at the detailed simulation level, it is not possible to define architectural styles or high-level design tradeoffs. In addition, there is no support for acausal, equation-based modeling of physical systems, since the underlying formalism is event-based communication.

The Vanderbilt model-based prototyping toolchain provides an integrated framework for embedded control system design [9]. It provides support for multiple views, such as functional Simulink/Stateflow models, software architecture, and hardware platform modeling along with deployment. The toolchain’s ESMoL language has a time-triggered semantics, which restricts the functional view to Simulink blocks that can only execute periodically. There is currently no support...
for additional views (e.g., physical or verification models), nor a notion of consistency between additional system views. In contrast, our work focuses on architecture-level view comparison, not on meta-modeling or model transformations.

SysWeaver [4] is a model-based development tool that includes a flexible code generation scheme for distributed real-time systems. The functional aspects of the system are specified in Simulink and translated into a SysWeaver model to be enhanced with timing information, the target hardware model and its communication dependencies. The translation from Simulink is not completely automated if closed-loop controllers are present. Sysweaver’s computational framework semantics is restricted to tasks that exchange information via message-passing (time or event-based). There is also no support in SysWeaver for a physical plant modeling view.

VI. DISCUSSION

Once we have the ability to relate heterogeneous system models to the BA through the mechanism of architectural views, several research issues arise.

First are the rules that determine what kinds of encapsulations are permitted. For instance, while it is natural to abstract over a set of elements in constructing a view, issues arise when some of those elements are cyber and some are physical, leading to elements with properties of both. A second issue arises when combining multiple connectors: connector abstractions that combine multiple connectors into a single one in the view are necessary, but this leaves open how best to handle the respective interfaces. Additionally, questions similar to those for components arise when combining connectors representing cyber and physical interactions.

A third issue arises when a view is better represented in a different architectural style than the baseline. This can occur when a specific analysis uses an architectural model with a more specific vocabulary of types of elements, and possibly additional constraints on their interconnection. Restrictions on the relationships between such styles are needed, and some
form of style consistency will need to be prescribed.

More generally, to aid in handling complex systems it should be possible to create various compositions (e.g., intersections and unions) of views. At present there is no such view specification language or calculus of views that would permit such flexible projections. There is also the problem of view update: how to reconcile the changes (and analyses) in one view with those in other views. In general, there may be many possible updates that are consistent with the changed view. Consistency in view update and integration is particularly important in the domain that we are addressing, since we plan to use the CPS architecture to identify and resolve dependencies between the analyses and design decisions carried out through different views.

The exploration of these issues form the next steps in our approach to multi-domain modeling using architectural views.

ACKNOWLEDGMENT

This work is supported in part by National Science Foundation (NSF) under grant no. CNS0834701 and by Air Force Office of Scientific Research (AFOSR) under contract no. FA9550-06-1-0312.

REFERENCES

Position Paper: Need for Architecture Description Language with Standardized Well Defined Meaning for Architecture Centric Engineering of Cyber-Physical Systems

Bruce Lewis, US Army AMRDEC

Abstract: System complexity, especially from the perspective of the dynamics of system interaction, is rapidly accelerating as computers are used to integrate applications at each level of system execution, from subcomponents to systems to systems of systems. This new complexity is expressing itself in the cost of system integration and the issues of reliability, dependability and safety, as well as overall system performance. To avoid very significant costs and risks to the program, we must virtually integrate these systems before building them, which is not easy to do with all the complexities of computer system interaction. In addition, due to the complexity of large systems and the lack of funding to adequately test them, we must enhance reliability through additional analytical and formal verifications. We must understand these systems from a consistent model, integrating lower level models with perhaps domain specific languages of specification, into an analytical architectural framework. At any time during the lifecycle of the system, this architectural model reflects the current state of system development. From it, we drive many forms of analysis to determine architectural compliance to system constraints and requirements. This system model must be architectural and component based if we are to understand the interactions, impact of change, and the emergent properties. The Architecture Analysis & Design Language (AADL) was designed for this purpose and is being applied in the area of software reliant system development. Architectural analyses for Cyber-Physical systems will add additional complexities to the expression and analysis of systems. To be effective in addressing these systems for analysis, a similar capability will be required. Cyber-Physical system models and analyses themselves will not integrate or be consistent without being formed against a common, standardized, well defined architecture description language for understanding compositional effects, cross contractor integration, incremental development and multi-dimensional analysis. Hence the use of such a language is critical to the goal of Cyber-Physical virtual integration.

Where Are We Today

Current industry practice is primarily based on a “build then test then fix the emergent issues then test then fix...”. This has served us up to this point, especially with the help of very senior, very smart people who can find the issues that surface during integration. However, this brings up the issue of what testing is not finding in these complex systems as well as the cost and schedule impact of making the system work. The issue of cost of integration of complex systems is being recognized especially in the aviation industry. The scale and heterogeneous nature of Cyber-Physical systems seem even more complex.
The aviation industry is experiencing a growth in software cost and complexity that threatens to overrun the cost of the full system development. The System Architecture Virtual Integration (SAVI) [1] project of the Aerospace Vehicle Systems Institute developed data that shows that for the next generation aircraft development, software cost will exceed $10 Billion. This threatens the development feasibility of these systems. SAVI’s approach to reduce these costs, an industrial consortium, led by Boeing and including Airbus, Lockheed Martin, BAE Systems, Rockwell, Goodrich, the FAA, Army, and NASA, is virtual architecture integration and analysis. The reason relates to the high cost of integration due to the late discovery of system integration issues. Figure 1 illustrates the potential cost reduction impact of error discovery before system integration. The chart shows that 70% of the errors are created in the requirements and design phases, but only 3.5% are found during these phases. This indicates a significant weakness in the current process. Currently about 80% of the errors are found during and after system integration. The multipliers of 16x, 40x and 110x are cost multipliers based on the phase of discovery of the errors. If we find the error during the requirements/design phase, the cost of rework for that error is reduced by these factors. There is significant opportunity for error reduction in system integration through a virtual, predictive hardware/software integration of the architecture. Cyber-Physical systems virtual integration will further broaden our ability to find issues in the system. The published SAVI report [1] from phase 1 is available. The current SAVI demo, phase II, is looking at Cyber-Physical interactions in the aviation system.

Figure 1: Only 3.5% of errors found in requirements and design, 80% found during or after integration test

In order to understand architectural issues, the behavior of system interaction must be captured in the architecture and analysis applied to assess critical qualities required of the architecture for system
execution. Today’s systems are highly software reliant, so the interactions across the system occur through the software and computer hardware, adding significant complexity to system modeling, analyses, and simulation. These analyses must be integrated in the architecture context so that the impact of the component or change in any component can be evaluated across the system model. In addition the architectural model must be incrementally developed by the integrator and his suppliers, based on the components with sufficient meaning to understand their interaction dynamics in the context of system execution. The system architecture specification must be consistent, including semantics, data and properties that drive the models, not just linked models from multiple teams. The representation of the architecture must be machine processable with a standard, well formed set of semantics supporting engineering and formal analysis. In this context, data for multiple analyses can be extracted from the architecture specification residing in a shared system repository. See figure 2. Through the architectural context and static and dynamic semantics, missing information can be detected and analyses can be based on the level of completion of the architecture.

Figure 2 Extracting analysis parameters from the architecture model.

Now the designer is in a position to evaluate change in a specific architectural property against an architectural requirement and more powerfully to evaluate the ripple impact of a changed property or component across multiple critical qualities at multiple levels in the architecture. Without well defined semantics in the architecture description language, designers need to add semantics, or override semantics or try to integrate semantics across multiple tools and models, in effect creating a custom
language. This is a significant task, especially when integration of models and analysis is needed across multiple forms of analysis and across multiple contractors.

The Architecture Analysis & Design Language [2] is a language with the well formed semantics to capture the architecture for both static and dynamic analysis. It was developed for this purpose as well as to support the generative integration of the components of the system to the specification supported by its analysis. The language provides a standard core set of capabilities and then enables extension through property sets and annexes. The annex mechanism provides a means for extension for Cyber-Physical systems. Language semantics support both engineering analysis and formal analysis as demonstrated by the number of analysis tools being developed for the language using its semantics. The report “Software Reliant System Qualification” [3] provides an overview of analysis methods and tools across multiple domains that have employed this standardized language for its semantic tightness and architectural framework. See papers [6-9] as additional examples related to system verification and validation. It has enabled high efficiency and high integrity automated system integration [4, 10]. NASA has developed an IV&V method using it [5].

Such a standardized language is needed for the challenge presented by the additional analytical context presented by Cyber-Physical Systems. The AADL committee is exploring possible annexes to better support Cyber-Physical Systems and SAVI is experimenting with the AADL in the Cyber-Physical context.

References


Abstract—Current system development techniques still rely on integration labs and manual integration of system components, typically occurring after the design and development of the subsystems is largely complete. Discovery of design errors late in the development life cycle during system integration testing often results in large budget and schedule overruns. Technologies that enable virtual integration of embedded electronics and the mechanical systems with which they are coupled allow discovery of design errors earlier in the product development life cycle. The System Architecture Virtual Integration program within the Aerospace Vehicle Systems Institute is a collaborative effort aimed at developing these technologies.

Index Terms—virtual integration, system modeling, model-based engineering

I. INTRODUCTION

Efforts to develop major commercial and military systems have increasingly suffered from cost and schedule overruns. These trends are largely attributed to increasing system complexity, driven by requirements for increased functionality and performance.

System developers are using model-based engineering technologies to help cope with this increasing complexity. However, these technologies have been deployed as domain-specific tools within a given development domain. In the embedded system domain, several technologies have emerged including block diagram abstractions for control systems, state diagrams, auto-code generation, automated test vector generation, model-in-the-loop, and hardware-in-the-loop testing. Other domains include complex finite element modeling and computational fluid dynamic models.

The relatively isolated development of these domain-specific models has created an explosion of design and analysis models that are relatively disconnected. The problems created by this situation are often not manifest until the system is integrated across the development domains. At this point in the development life cycle, the impact of design errors created by incomplete, incorrect, or conflicting requirements often cannot be completely quantified, though the impacts are admittedly very significant. Moreover, system integrators are acutely aware that late discovery leads to excessive rework, non-optimal solutions and is associated with growth in cost and schedule.

The System Architecture Virtual Integration (SAVI) program is a program being executed within the Aerospace Vehicle System Institute (AVSI) that aims to address issues stemming from the lack of an integrated systems modeling environment. The SAVI approach of “Integrate, then Build” acknowledges the modern distributed development environment while seeking to arrest the propagation of requirements errors through the development life cycle, primarily by capturing design assumptions and shared properties of the system design in an authoritative, annotated architectural model. Such a reference model provides a common, analyzable model to confirm that the definition of the system (requirements through design) remains complete, consistent, and correct at all levels of system decomposition.

II. CURRENT SITUATION

A. System Complexity

There is no precise definition of system complexity, but suggested measures include such considerations as the number of components, the number of interconnects and the number of signals being transmitted. The desire to simultaneously optimize performance while improving safety and reliability has led to an increase in the number of sensors and signals employed in integrated systems. Control of these systems has increasingly relied on the real-time flow of information between subsystems. These factors have contributed to the growth of software content in modern systems. Therefore, we used source lines of code (SLOC) as an available and representative measure of system complexity.

Fig. 1 shows the data from several commercial aircraft development programs. The data for Airbus programs are from [1] and the Boeing data from [2]. A linear regression fit to the discrete data points indicates that SLOC count has been doubling approximately every four years. The COCOMO II [3] program was then used to estimate the costs associate with these SLOC counts. Using conservative values for the parametric inputs based on current practices, it becomes apparent that SLOC growth at this rate using existing software development practices will seriously impact affordability of the next generation of commercial aircraft.

In addition to the super-linear growth in the complexity of the systems themselves, the environments in which these systems are being developed are also becoming more complex. Large system integrators work closely with their international supply chain to produce complex systems such as modern aircraft. Specifications, requirements, and other design
data must flow across organizational, international and cultural boundaries. These environments strain the capability of traditional English language documents. Even within a single company, internal organizational boundaries can create problems. Reliance on formal documentation and data deliverables to accurately communicate design intent often masks design assumptions that may not be fully understood until a problem is uncovered during system integration.

The cost to correct requirements errors increases exponentially with development phase. Fig. 2 shows cost escalation factors for software development derived from several sources [4]-[7]. A requirement error that is detected and corrected during the requirements development phase (Req) has an escalation factor of one. The succeeding phases are design (Dsn), coding (Code), unit test (UT), software integration test (IT), system integration test (ST), final acceptance test (AT), and in-service operations (Ops). The scale is dominated by the operations data reported by Baziuk [7], but the figure still shows that requirements errors detected and corrected during system test are 25 to 90 times more costly to correct than if they were corrected during the requirements development phase.

Much of this cost escalation is derived from the expanded impact of change. The trends toward more tightly integrated and increasingly complex systems, coupled with more decentralized development environments, make understanding the full scope of change impact difficult to determine using current development practices.

B. Evolving System Engineering Methods

Methods to handle increasing complexity have evolved in recent years to significantly include modeling as part of the design, development, and verification process. Modeling is used both to analyze system behavior and to abstract details of the system design into a form that is more readily comprehended by humans. This usage is analogous to the abstraction of machine code programming to assembly language and subsequently to higher level programming languages. In the same way that compilers allowed programmers to focus on software engineering, graphical models and auto-coders have allowed more awareness of systems engineering.

Useful models have typically been developed within specific domains for system development. TABLE I lists typical domains into which an engineering organization may be segregated. A full understanding of the limits of use and assumptions applicable to a model is typically limited to the engineers within the domain. TABLE II lists examples of the types of models that may be used in the development of a commercial aircraft.

These partial lists illustrate that current technologies solve part of the complexity problem, namely the detailed design and test within a given domain, but do not necessarily support efficient integration of the models and subsystems across domains.

C. Link models vs. reference models

There have been attempts to address this explosion of
relatively unconnected models [8]-[9]. These attempts however have relied on using a “link” model of the system architecture. A “link” model makes no attempt to unify the data used to represent the system, but rather reasons about the interaction between models within different domains. This reduces the problem to managing interfaces between models.

An example of such a model is the Generic Modeling Environment (GME) developed by the Institute for Software Integrated Systems at Vanderbilt University [8]. In this case, configuring metamodels that specify the modeling paradigm of the application domain creates domain-specific modeling and program synthesis environments. Another example developed by Lockheed Martin is NAOMI, an experimental platform for distributed multi-modeling [9].

Link models afford flexibility and can accommodate multiple modeling environments found in a distributed supply chain. However, design assumptions that are specifically handled at the interface between models may remain masked. Furthermore, for an environment of N unique modeling domains, a fully conversational link model requires N(N-1) translators (Fig. 3a).

In contrast, a reference model approach uses a central data repository to maintain a single source of “truth” for the design intent of the system. Specific domain models are derived, at least in part, from the reference model. Bidirectional flow of data between the reference model and specific domain models eliminate the possibility of conflicting values for data that are used in multiple domains. In this case, a fully conversational model requires 2N translators to bridge N unique development domains (Fig. 3b).

D. Common Issue – Standard Solution

System integration problems plague the development of all types of complex systems. Examples can be found in military, commercial aircraft, space, and automotive systems. Just as standard electrical connectors are developed to allow interconnection of multiple, independently developed electronic components, a standard means to store and communicate multiple types of data in a reference model is required to allow integration of multiple models across a decentralized development environment. For OEM’s, this approach allows unambiguous communication of design specifications to multiple suppliers. Suppliers gain the

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>EXAMPLES OF SYSTEM MODELING DOMAINS</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Modeling Domains</td>
<td></td>
</tr>
<tr>
<td>Ops/Mission Analysis</td>
<td></td>
</tr>
<tr>
<td>System Design</td>
<td></td>
</tr>
<tr>
<td>Algorithm Development</td>
<td></td>
</tr>
<tr>
<td>Hardware Design</td>
<td></td>
</tr>
<tr>
<td>Software Design</td>
<td></td>
</tr>
<tr>
<td>Logistics Support</td>
<td></td>
</tr>
<tr>
<td>Manufacturing</td>
<td></td>
</tr>
<tr>
<td>Integration &amp; Test</td>
<td></td>
</tr>
<tr>
<td>Performance Simulation</td>
<td></td>
</tr>
<tr>
<td>Engineering Analysis</td>
<td></td>
</tr>
<tr>
<td>Human System Integration</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE II</th>
<th>EXAMPLES OF AIRCRAFT DEVELOPMENT MODELS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development Models</td>
<td></td>
</tr>
<tr>
<td>Operational Models</td>
<td></td>
</tr>
<tr>
<td>System Models</td>
<td></td>
</tr>
<tr>
<td>Component Models</td>
<td></td>
</tr>
<tr>
<td>Functional/Behavior Model</td>
<td></td>
</tr>
<tr>
<td>Performance Model</td>
<td></td>
</tr>
<tr>
<td>Structural/Component Model</td>
<td></td>
</tr>
<tr>
<td>Cost Model</td>
<td></td>
</tr>
<tr>
<td>Safety Model</td>
<td></td>
</tr>
<tr>
<td>Security Model</td>
<td></td>
</tr>
<tr>
<td>Reliability Model</td>
<td></td>
</tr>
<tr>
<td>Maintainability Model</td>
<td></td>
</tr>
<tr>
<td>Structural Model</td>
<td></td>
</tr>
<tr>
<td>Mass Production Model</td>
<td></td>
</tr>
<tr>
<td>Manufacturing (Assembly) Models</td>
<td></td>
</tr>
</tbody>
</table>

System Architecture Model (Integration Framework) |
| Analysis Models |
| Hardware Models |
| Software Models |
| Verification Models |

Fig. 2. Cost escalation factors for correction of errors discovered and corrected in different phases of development.
advantage of maintaining a single integrated modeling environment rather than one for each OEM customer. Given that much of the infrastructure necessary to implement such an integrated environment is precompetitive and in the realm of standards, a collaborative solution is the preferred means to develop this infrastructure. AVSI launched the collaborative SAVI project based on a mutual understanding that integration complexity will continue to increase and on the recognition that individual companies cannot solve this problem alone, industry cannot afford to solve it multiple times, and that industry cannot afford not to solve it.

III. AVSI

The Aerospace Vehicle System Institute is an aerospace industry collaborative formed in 1998 at Texas A&M University to address common, precompetitive issues facing industry members. Members propose project ideas and those interested in participating form teams to execute the projects. Current membership includes U.S. and European aircraft manufacturers, subsystem suppliers, government organizations, and academia.

Several AVSI projects have been launched to explore the reference model concept. A survey of architecture description languages (ADL’s) was performed in 2005 and 2006 to understand the state of technology as it applied to the concept of a reference model based, full aircraft system modeling environment. It was recognized that no existing technology fully addressed the requirements for such an environment. This led to a project that developed a preliminary plan for the SAVI program to address the capability gaps identified in the existing technologies.

IV. The SAVI Program

SAVI promotes an annotated architecture model as the unifying framework and integration mechanism across all models to enable inter-model analyses in a distributed, heterogeneous modeling environment. This environment is characterized by a consistent, single source of truth architecture that exposes assumptions and shared properties in each model. The program is an international collaboration between aircraft OEM’s, suppliers, government research/accreditation/certification bodies, and government systems acquisition organizations. SAVI will make use of emerging systems modeling technologies including model-based engineering, proof-based design, and component-based design.

A. SAVI Proof-of-Concept

The initial effort under the SAVI program was a Proof of Concept (PoC) project to validate the SAVI “Integrate, then build” concept. The objectives of this project were to investigate changes to the system acquisition paradigm, perform a proof-of-concept demonstration, develop a return on investment estimate, and create a roadmap for the full development of SAVI technologies.

The SAVI approach starts with a reference architecture developed by the prime system integrator. This architecture is modeled to perform system trade studies and allow validation of high-level requirements, which forms the basis of a model-based acquisition paradigm. The team examined elements of the current (“as-is”), multi-tiered acquisition paradigm to identify where integration errors are typically generated. This current approach was then compared to a future (“to-be”), model-based acquisition paradigm based on an integrated SAVI modeling environment. Fig. 4 illustrates the SAVI integrated modeling environment concept. A central repository maintains a single source of “truth” for the shared system design data. In practice there may be several repositories, each compliant with a SAVI standard format, in order to protect proprietary data. However, these repositories are logically connected to allow system analyses on hybrid models assembled from OEM and supplier repositories. Bidirectional data flow using standardized formats is implemented to maintain consistency and enable configuration management of the distributed model.

In this environment, models rather than paper documents are used to communicate design intent. Requests for proposals from OEM’s are in the form of a system architecture model with responses by suppliers in the form of subsystem
models. These subsystem models are virtually integrated with the system model to evaluate proposals and to afford early and continuous verification and validation of the requirements allocated to these subsystems and of the subsequent design models. Ambiguities associated with written requirements are eliminated through the use of a semantically strong modeling language. The future SAVI model-based acquisition paradigm is illustrated in Fig. 5.

Unification of shared modeling data must not be accomplished at the expense of protection of intellectual property rights (IPR). Fig. 6 illustrates a mechanism by which shared data can be isolated from proprietary detailed implementations. In this case, model data is segregated into public and private data repositories. The aggregated public repositories contain the shared properties that comprise the “single truth” source of the system design intent. These shared repositories, as informed by the various proprietary subsystem models, enable system level analyses by the OEM. Existing modeling languages already provide some of the constructs necessary to effect this isolation.

The SAVI Proof of Concept demonstration was structured to exercise the concepts of a model-based acquisition paradigm. It consisted of a hierarchical aircraft model comprised of three tiers: an aircraft level model, an aircraft system model, and a subsystem model. Analyses run on the full model included a roll-up weight against budget calculation and power consumption/capacity analysis. The demonstration team was also representative of an actual system analysis with team members from aircraft OEM’s and Tier 1 suppliers located in Seattle, WA, Cedar Rapids, IA, Fort Worth, TX, Pittsburgh, PA, Rochester, United Kingdom, and Toulouse, France working on a shared model hosted at Texas A&M University. The demonstration utilized a standard architectural modeling language [SAE AS5506 – Architecture Analysis & Design Language (AADL)] and open source tools. The Software Engineering Institute at Carnegie Melon University was contracted to develop extensions to these tools to facilitate the demonstration and to assist with the analyses.

The demonstration illustrated how requirements and constraints are propagated from an OEM’s higher-level model down to suppliers’ lower-level models. It further demonstrated verification of higher-level requirements and constraints by analysis of lower-level models. As this project was intended as an illustrative demonstration, only those elements required for the selected analyses were extended to more detailed designs.

The top tier aircraft level model consisted of major aircraft systems (engines, landing gear, avionics, and electrical power) and signals connecting these systems (electrical power distribution, hydraulic power distribution, and electronic signal distribution). Limited aircraft level analyses were implemented. These investigations included an analysis of total weight and verification of weight budgets allocated to subsystems. A simple power capacity/power allocation analysis was also performed. The integrated modular avionics (IMA) system was selected for detailing to lower tiers.

The IMA system model included elements of both the hardware platform and the software running on the hardware platform. Analyses included electrical power budget, MIPS budgets, RAM capacity and budgets, and end-to-end flow latency.

Implementation of a third-party flight management application running on the IMA platform was the lowest tier modeled. This implementation included allocation of tasks to elements of the IMA and analyses of execution time versus budget and partition schedulability.

The demonstration effort produced a set of models maintained in a Subversion repository. Video versions of the demonstration were produced in addition to live demonstrations at various venues.

The PoC project sought to establish a more quantitative estimate of the benefit afforded by SAVI. The problems with integration of complex systems are well appreciated and experienced by a wide audience, hence the problem itself requires little motivation. However, the scope of the effort...
proposed by SAVI required an estimate that would assist members in prioritizing their resources. Concerns about exposing competition sensitive cost data constrained the approach taken to develop a Return on Investment (ROI) model based largely on publically available data. This model could then be adapted for each specific company by incorporating internal estimates of costs and error rates.

The model was limited to elimination of certain classes of software requirements errors due to the availability of data. It was reasoned that software costs represent a significant and growing cost in complex system development and that simple scaling arguments can be used to extended the analysis to the full aircraft (albeit with lower fidelity). While significant advantages of a model-based acquisition paradigm can be projected, the benefits were confined to elimination of associated rework costs. Conservative assumptions were made throughout the model to emphasize credibility of the model.

The model was comprised of sequential estimates:
1. New SLOC growth (ignoring reused code) for future programs.
2. Development life cycle cost escalation factor (see Fig. 2) for requirements errors.
3. Probability that requirements errors would be detected and corrected with SAVI as a function of development phase.
4. Scale factor for hardware systems.
5. Costs for new SLOC development (using COCOMO II).
6. Net Present Value for a new aircraft development using the costs and cost savings estimated above over a typical development time period.

Pessimistic, most likely, and optimistic estimates were computed and presented along with the rationale behind each estimate.

The results indicate that acceptable rates of return would be experienced for a full development program, even using the pessimistic estimates. This assessment agrees with the intuitive understanding prevalent in industry, which is based on ubiquitous experience with requirements errors and rework costs. Furthermore, one SAVI participant had their company’s cost estimation organization independently compare the results from the SAVI ROI model with those from existing cost estimation processes and found the NPV to agree within 2%.

It is important to note that the ROI model considered the full development program and did not consider allocation of benefit to the various tiers of the supply chain. However, it stands to reason that benefits of virtual integration will scale with the amount of system integration activity performed by a given organization.

The project participants also felt it important to develop a roadmap for the development of SAVI to a level of readiness suitable for deployment on a production program. This roadmap emphasized the immediate need to manage complexity by considering a spiral development built on incremental funding and benefits, and by aggressively engaging the industry to integrate with existing complementary efforts to develop related technologies.

It was recognized that the standard AVSI model of fully self-funded projects would need to be adjusted to accommodate the scope of the SAVI development. Thus, a follow-on effort was proposed to continue to progress the understanding of the technology gaps identified in the SAVI roadmap.

B. SAVI Expanded Proof-of-Concept Demonstration

The current effort by the SAVI team is the Expanded Proof-of-Concept Demonstration (EPoCD) project. The intent of this effort is to extend some of the developments begun under the PoC project and to extend models to incorporate mechanical, electrical, and hydraulic systems. The goals of the EPoCD project are to (1) begin development of the data exchange and model repository requirements, (2) address some questions raised during PoC concerning modeling languages, (3) improve the ROI to go beyond scale-factor consideration of hardware in embedded systems, (4) develop a program plan that implements the SAVI roadmap, and (5) extend the

Fig. 5. The SAVI “to-be” acquisition paradigm.
outreach effort to engage additional stakeholders and complementary technology development efforts.

Preliminary development of the requirements for the fundamental SAVI constructs, i.e., the data exchange mechanism and reference model repository, is being addressed in two ways. First, a set of early integration use cases is being collected and prioritized. Second, the Proof-of-Concept demonstration is being expanded to implement a specific subset of the prioritized use cases. This expansion specifically considers mechatronic subsystems.

Several questions were raised during the PoC effort that need to be resolved during the EPoCD effort. AADL was selected for use in the PoC demonstration due to the results of the initial AVSI ADL language survey indicating a reasonable subset of the envisioned SAVI capability, the availability of open source tools, and the availability of a supporting organization interested in participating in SAVI. However, AADL was developed primarily to model embedded electronic systems and several questions were raised as to its suitability to model an entire aircraft including mechanical systems. Other technologies such as SysML have matured or emerged since the original language survey and these need to be considered in light of the early integration use cases being developed under the EPoCD project. Furthermore, the original SAVI concept stressed a strong, centralized reference model, which suggested a common modeling language. It was recognized through the PoC outreach efforts that a multi-model implementation is more likely to emerge. The impact of this change in thinking on the requirements for the model repository is being investigated in the current EPoCD effort.

The ROI model developed during the PoC was corroborated by at least one participating company; however, the range in values from pessimistic to optimistic estimates indicated that a refinement of estimates was warranted. Furthermore, a simple scale factor was used to estimate the effect SAVI would have on mechanical system development. This estimate is of course critical for suppliers that produce these systems. The EPoCD ROI effort is addressing these shortcomings of the PoC model. SAVI was created to address immediate concerns regarding the affordability of the next generation of commercial aircraft. Thus SAVI must reach a level of maturity that will allow it to be deployed in the time frame consistent with the start of the next new program. To support these aggressive schedule constraints in the reality of incremental funding, competing resources, and collaborative development, the EPoCD effort includes the development of a program plan that implements a revised SAVI roadmap. This plan must be rigorous enough to allow stakeholders to commit resources, yet retain enough flexibility to handle the program risks listed above.

Finally, the SAVI team has always maintained that there is no room for “not invented here” to impede progress toward a workable solution. As stated above, the need is ubiquitous and thus a common solution needs to be developed, rather than multiple point solutions. The EPoCD outreach effort is designed to (1) raise awareness of the SAVI objectives and approach, (2) engage additional stakeholders, (3) identify complementary technology development programs that may contribute to the SAVI approach and potentially offset the estimated SAVI development costs, and (4) promote a concerted international effort to rapidly develop a solution to current system integration problems that minimizes duplication of effort.

Additionally, producers of system engineering software tools are being encouraged to become involved with the SAVI program. It is recognized that commercial and open-source tools necessary for a deployable implementation of SAVI will need to mature alongside the SAVI technologies to minimize risk to the first SAVI-based development program.
C. Future Plans

The current EPoCD effort is scheduled to conclude in February 2011. The intent is to launch a second phase of the EPoCD in early 2011 that will broaden the number and types of use cases demonstrated and that will include “shadow” pilot projects to add credibility to the SAVI virtual integration process. The development may be ramped up to the original plan at the end of 2011, subject to the availability of resources. Alternatively, a third phase of the PoC demonstration will be pursued in 2012, which will include extended “shadow” pilot projects and implementation of an expanded array of prioritized use cases. It is likely that development of SAVI will be at a TRL of 4.5 to 5.0 at the completion of the two-phase EPoCD, after which the full development schedule for SAVI can be initiated to arrive at a TRL of 9 approximately 2.5 years later.

There are several technology development programs in the U.S. and in Europe with which the SAVI team is engaged that are scheduled to continue through 2011. The incremental development spelled out above and depicted in Fig. 7 should be able to draw from the results of those efforts to mature SAVI for industrial applications.

V. CONCLUSION

The growth of costs associated with integration of complex systems is inhibiting the introduction of these systems. Virtual integration offers a means to mitigate these costs. SAVI is an industry-based collaborative effort to develop the technologies necessary to implement a reference model approach to systems development. The benefits of a single-truth system architecture model and early and continuous verification of system integratability are needed to support the development of the next generation of complex systems.

REFERENCES

Virtual Integration of Cyber-Physical Systems by Verification

Panagiotis Manolios  
Northeastern University  
pete@ccs.neu.edu

Vasilis Papavasileiou  
Northeastern University  
vpap@ccs.neu.edu

Abstract—In this position paper, we advocate the use of verification technology to tackle the virtual integration problem for cyber-physical systems. In particular, we advocate the use of high-level modeling languages that allow designers to declaratively specify what properties their architectural models should have, not how to achieve them. We further advocate the use of verification technology to analyze such models, in particular to synthesize concrete architectural models that certifiably satisfy all properties. We discuss our work in this direction and outline some of the challenges. Specifically, we show that virtual integration in the presence of real-time scheduling constraints leads to problems that cannot be dealt with in a compositional manner. They can, however, be handled in a semi-compositional fashion, as we outline in this paper.

I. INTRODUCTION

The design of cyber-physical systems is an increasingly important problem. Cyber-physical systems use software to interact with and exert control over the physical world. Examples abound in the aerospace, automotive, and medical fields. Cyber-physical systems tend to consist of many interacting, complex components that share resources and operate under real-time constraints. Their design and development often involves large teams working for many years. Design flaws routinely remain undiscovered until integration testing, at which point they are prohibitively expensive to deal with. The idea of virtual integration testing is to analyze models of cyber-physical systems before they are built, in order to find and correct design flaws as soon as possible.

In this position paper, we propose tackling the virtual integration problem for cyber-physical systems by using verification technology. More specifically, we propose using declarative high-level models, from which more traditional, architectural models can be synthesized, using verification technology. The synthesized models will come with a certificate showing that they satisfy their architectural-level properties.

Currently, the highest level models used are architectural models. Such models describe the high-level structure of a system, which includes a set of components interacting through their connectors. Several architecture description languages (ADLs) have been developed to describe and reason about architectural models. Examples include AADL [3] and ACME [4]. Medvidovic and Taylor have written a survey on architecture description languages [7]. Notice that in an architectural model, one has to explicitly specify component connections.

A point of departure from current methods, is that we insist on the use of a declarative high-level modeling language. The reason is that it is crucially important to have analyzable, formal models as early as possible in the design cycle. The gap between declarative high-level models and architectural models can be significant, representing multiple person-years of effort. We use verification technology to enable designers to automatically synthesize architectural models that certifiably satisfy their design specifications. Not only does our approach relieve the designers from the multi-year effort to define an architectural model, but, more importantly, it allows designers to get immediate feedback on their designs, allowing them to more fully explore the design space, to find design flaws early, and, thereby, to design superior cyber-physical systems.

The rest of the paper is organized as follows. In Section II, we describe the CoBaSA language, a high-level declarative language along the lines we have been proposing. We have used CoBaSA to solve what we call the system assembly problem: from a sea of available components, which should be selected and how should they be connected, integrated, and assembled so that the overall system requirements are satisfied? [6] Section II includes a simple, representative example designed to give a sense of what is involved in using CoBaSA to synthesize architectural models. CoBaSA works by transforming the system assembly problem into a satisfiability problem, that is then handled by a verification tool. Why is this not the end of the story? Well, one of CoBaSA’s major limitations is that it cannot handle real-time scheduling constraints. In Section III, we describe static cyclic scheduling, which is non-preemptive and quite difficult to satisfy. In Section IV, we show that the combination of the system assembly problem and the scheduling problem cannot be solved in a compositional way, and outline a semi-compositional approach to solving the problem. Conclusions and directions for future work are given in Section V.

II. SYSTEM ASSEMBLY

The CoBaSA system, as described in [6] allow us to express architectural constraints between components and automatically synthesize architectural models that satisfy these constraints. CoBaSA provides an object-oriented language
for describing the structural properties of the components, and mechanisms for imposing constraints on how these components will inter-operate. In this section we provide an introduction to the CoBaSA language. To this end, we use as an example an architectural model from the aerospace domain and terms consistent with the ARINC standards 651-1 and 664-7 [1].

The basic datatypes in CoBaSA are integers (bounded or arbitrary precision), Booleans, and strings. CoBaSA entities correspond to classes in an object-oriented programming language. Entity inheritance is supported. Data can be organized in single- or multi-dimensional arrays.

Resource allocation is the main objective of the CoBaSA system. We think of the resource allocation problem in terms of mapping resource consumers to resource providers. CoBaSA objects can consume resources, or provide resources, or both. In our model, resource providers are processors that provide CPU time, RAM and other resources. Our consumers are jobs (or hosted functions) and global memory spaces with certain resource requirements. The corresponding CoBaSA entities are:

```plaintext
entity processor {
    ; id STRING
    ; location STRING
    ; cpu-time-available 1000000
    ; nvm-memory-available 16384
    ; ram-memory-available 131072
    ; rom-memory-available 65536
    ; no-of-rx-vls-available 384
    ; no-of-tx-vls-available 128
    ; rx-vl-bandwidth-available 100000
    ; tx-vl-bandwidth-available 100000
}

text job {
    ; id STRING
    ; location STRING
    ; rate INT
    ; cost INT
    ; cpu-time-req INT
    ; nvm-memory-req INT
    ; ram-memory-req INT
    ; rom-memory-req INT
}

text gms {
    ; id STRING
    ; nvm-memory-req INT
    ; ram-memory-req INT
    ; rom-memory-req INT
}
```

The processor entity has an identifier, a string corresponding to the location where the processor resides (a set of locations "Loc1", ... "Locn"), and fields for the resources provided: CPU time, different kinds of memory (non-volatile, RAM, and ROM), the number of virtual links the processor can receive and transmit, and the bandwidth available for the incoming and outgoing virtual links. Since all processors in this model are homogeneous, all the fields corresponding to resources have default values; thus, we do not have to define them when instantiating the entity.

The job entity has fields for the ID, the location of the job (a set of locations, as above), the rate and cost (how many times per second the job is executed, and how long each such execution lasts), the total CPU time requirement (the product of rate and cost), and the different memory requirements.

The entity gms has fields for the ID and the memory requirements of each global memory space.

We define instances of the above entities as follows:

```plaintext
var
    ; processor P_1 = {
        ; "P_1" ; "Loc1" ; ; ; ; ; ; ; ;
    };
    ; job J_1 = {
        ; "J_1" ; "Loc1" ; 20 ; 8750
        ; 175000 ; 64; 8192 ; 4096)
    };
    ; processor[4] processors-Loc1 = [P_1, P_2, P_3, P_4]
    ; job[64] jobs-Loc1 = [J_1, J_2, ..., J_64]
```

In this example, we defined a processor with id P_1 that is located at location "Loc1" and has default values for the remaining fields, and a job with id J_1 that can be mapped to processors at location "Loc1" and has the resource requirements shown. We also defined an array of 4 processors whose location is "Loc1", and an array for the 64 jobs that can be mapped to them. Separate arrays are used for the processors at different locations and the jobs that can be mapped to them. For jobs can be mapped to multiple locations, we have yet another array.

CoBaSA maps are functions from resource providers to resource consumers. A map constraint relates two arrays, which we call the domain and the range of the map. Each component in the domain is mapped to exactly one component in the range. The following definition maps jobs to processors:

```plaintext
map jobs-to-procs-Loc1 jobs processors-Loc1
    constraint jobs-to-procs-Loc1
    (cpu-time-req,
    nvm-memory-req,
    ram-memory-req,
    rom-memory-req)
    (cpu-time-available,
    nvm-memory-available,
    ram-memory-available,
    rom-memory-available,
    rx-vl-bandwidth-available,
    tx-vl-bandwidth-available,
    no-of-rx-vls-available,
    no-of-tx-vls-available)
```

66
ram-memory-available, rom-memory-available)

A map definition is usually accompanied by field constraints, which relate resource requirements to resource availability. In our model, we need the above four field constraints for the CPU time and the different kinds of memory. According to the first field constraint, the sum of the CPU time requirements of the jobs mapped to a specific processor cannot exceed the available CPU time in the processor.

The mapping of jobs to processors in the model is subject to requirements like job separation (a pair of jobs have to be mapped to different processors). For example:

\[
\text{for}_\text{all } p \text{ in processors-Loc1}\\
\text{((jobs-to-procs-Loc1}(J_7, p) \implies \neg \text{jobs-to-procs-Loc1}(J_8, p)))
\]

The constraint above states that jobs J_7 and J_8 cannot be co-located. The for_all statement was used above to apply a constraint to a whole array. Map references indicate whether an element in the domain of a map is mapped to an element in the range: jobs-to-procs-Loc1(J_7, p) is true if and only if job J_7 is mapped to processor p. The usual logical connectives are provided. A separation constraint as in the example above can be used to deal with job replication requirements. For system reliability reasons, we need multiple copies of a job running on different processors. We can deal with such constraints by defining multiple jobs, one per instance, with the same values for the resource requirements, and with associated separation constraints.

We similarly encode job co-location constraints: such constraints state that a pair of jobs have to be mapped to the same processor.

The designer of a safety-critical system has to consider the possibility of failures. Spare processors allow us to operate safely in the presence of a limited number of processor failures. In our architectural model, a set of processors per location are considered spare processors. We are given sets of jobs at most one of which can be mapped to a spare processor. The jobs that do not appear in any of these sets cannot be mapped to a spare processor. In our model, P_4 is the only spare processor for location Loc1. The constraints below state that at most one of the jobs J_36, J_48 and J_52 can be mapped to a spare processor and that J_53 cannot be mapped to a spare processor.

\[
(+ \text{jobs-to-procs-Loc1}(J_36, P_4) \\text{jobs-to-procs-Loc1}(J_48, P_4) \\text{jobs-to-procs-Loc1}(J_52, P_4)) \\
\leq 1 \\
\neg \text{jobs-to-procs-Loc1}(J_53, P_4)
\]

Notice that in the example we used an arithmetic operation on map references: in CoBaSA, Booleans are treated as 0 or 1 when used in an arithmetic context.

In some cases, a resource consumer has to be mapped to more than one resource provider. In the architectural model multiple copies of global memory spaces are allowed. This can happen if multiple jobs that reside on different processors need read-only access to the same memory space. In this case, each copy of the memory space consumes an amount of RAM belonging to the processor it is mapped to. We define a generalized map from memory spaces to processors:

\[
gmap \geq 1 \text{gmss-copies-to-procs} \\
gmss-copies \text{procs} \\
\text{constraint gmss-copies-to-procs} \\
\text{(nvm-memory-req, ram-memory-req, rom-memory-req)} \\
\text{(nvm-memory-available, ram-memory-available, rom-memory-available))}
\]

The map above states that each memory space in the gmss-copies_array is mapped to at least one processor. We have a separate array gmss and an associated map for the global memory spaces that have to be mapped to exactly one processor. In addition to the generalized map, we need constraints that force a copy of the global memory region to be co-located with a job that needs read-only access to it. The map above also illustrates a case in which resource consumers of different kinds (namely, jobs and global memory spaces) share the same resources. CoBaSA handles this correctly.

Other families of constraints that a safety-critical system has to satisfy can be encoded with arbitrary Boolean variables and constraints on them. The jobs in our architectural model communicate through virtual links. Every link has a exactly one publisher, but potentially multiple subscribers. The sum of the bandwidth required for the virtual links that the jobs located on a processor publish or subscribe to cannot exceed the available outgoing or incoming bandwidth of the processor, respectively. In case two or more subscribers of the same virtual link are mapped to a specific processor, we count the incoming bandwidth of the link only once.

We encode the incoming virtual link constraints as follows. We use a Boolean variable to denote the fact that a processor receives a specific virtual link. For example, jobs J_12, J_25 and J_38 subscribe to the virtual link VL_1. We define P_3-sub-VL_1 to be true if and only if at least one of J_12, J_25 and J_38 is mapped to processor P_3:

\[
(P_3-sub-VL_1 \iff \text{jobs-to-procs-Loc1}(J_12, P_3) \text{jobs-to-procs-Loc1}(J_25, P_3) \text{or jobs-to-procs-Loc1}(J_38, P_3))
\]
jobs-to-procs-Loc1(J_38, P_3))

The model involves 110 virtual links, the incoming bandwidth provided by \( P_3 \) is 100,000 Kbps, and the incoming bandwidth requirements of the virtual links \( VL_1 \), \( VL_2 \) and \( VL_{110} \) are 8,000 12,000 and 16,000 Kbps respectively.

The incoming bandwidth constraint for processor \( P_3 \) can be expressed as follows:

\[
(+ (* P_3-sub-VL_1 8000) (* P_3-sub-VL_2 12000) \ldots (* P_3-sub-VL_{110} 16000)) <= 100000
\]

In our model, virtual links are comprised of messages. Jobs really need access to messages, not virtual links. Therefore, the messages they need determine what virtual links they subscribe to and they read only the subset of messages they care about from the virtual links they subscribe to. Jobs buffer a given number of bytes for each message, a number that can differ among subscribers of the same message. Each processor provides a single buffer for both message transmission and reception. The sum of the buffering requirements for the messages that the jobs mapped to a processor publish or subscribe to cannot exceed the capacity of the buffer. When multiple jobs on the same processor subscribe to the same message with different buffer requirements, the maximum buffer requirement is taken into account. Finally, each hosted function that subscribes to a message uses either a queue buffer, or a sampling buffer for it and hosted functions that subscribe to the same message but use different buffer types cannot reside on the same processor. The CoBaSA framework is expressive enough to accommodate message the constraints described.

The CoBaSA framework is fully automated. The system receives as input a program in the language we have described and returns an allocation of resource consumers to resource providers. The way the framework works is by translating the constraints to a propositional Satisfiability (SAT), pseudo-Boolean Satisfiability or Integer Linear Programming (ILP) problem and calling an off-the-shelf SAT or ILP solver. We do use some novel ideas in the generation of constraints \([2],[5]\). The satisfying assignment that the solver returns corresponds to a solution of the system assembly problem. The available solvers for the three different formalisms perform better with different system assembly instances. Since we strive to offer a general framework, it is beneficial to support all three.

CoBaSA can also be used in optimization mode, in which we look for a solution that is optimal with regards to an objective function. For example, we can use this feature for load balancing: we can minimize the difference between the minimum load among the processors and the maximum load.

III. Scheduling

In this section we describe static cyclic scheduling, which is non-preemptive and periodic. It is used in industry and it happens to be a very demanding type of scheduling, which has its advantages and disadvantages. The disadvantage is that it is often very hard to determine if even relatively small sets of jobs are static cyclic schedulable on a single processor. The advantage is that static cyclic scheduling provides very strong guarantees, which can dramatically simplify the kinds of safety analyses often required for safety-critical systems.

We now define the static cyclic scheduling problem. Time will be divided into an infinite number of cycles, each of \( s \) slots. A slot is the smallest interval of time we will consider. For example, we might divide time into cycles, each of which lasts for a second, consisting of \( s = 10^6 \) slots. In this case a slot corresponds to a microsecond. The parameter \( s \) is really processor-dependent. If we have a collection of heterogeneous processors, then we can account for their different rates of speed by setting their \( s \) parameters appropriately.

Jobs will be denoted by a pair \((r, c)\), where \( r \) is the rate of the job and \( c \) is the cost. We require that for any job \((r, c)\) that is to be scheduled on a processor with \( s \) slots per cycle, that \( r \) divides \( s \). For example, if a cycle corresponds to a second and \( s = 10^6 \), as above, then \( r = 100 \) is allowed, whereas \( r = 7 \) is not. Given a multiset of jobs, a schedule is simply a starting time (slot) \( t \) for each job such that \( t < s \) and no two jobs occupy the same slot. The slots occupied by a job are slots of the form \( t + k\frac{r}{s} + i \), for \( k \) a natural number and \( 0 \leq i < c \). That is, if a job is scheduled to start during slot \( t \), then it occupies \( c \) consecutive slots starting with slot \( t \) and this process repeats at slot \( t + \frac{r}{s} \), \( t + 2\frac{r}{s} \), etc. For example, if a cycle corresponds to a second and \( s = 10^6 \), as above, and \((r, c) = (100, 5)\), then once the job starts, it has to be scheduled every \( \frac{1}{10^5} \)th of second for \( \frac{1}{5} \) consecutive seconds each time.

Given a multiset of jobs, the uniprocessor static cyclic scheduling problem is to determine whether there exists a schedule. This problem is NP-complete, as is the multiprocessor version of the problem.

IV. Semi-Compositionality

The question we address in this section is: what if we have a model that includes both the kind of system assembly constraints we saw in Section II and the kinds of scheduling constraints we saw in Section III? Do we need a completely new way of dealing with the combination of these constraints, or can we compose the verification algorithms we use for solving the system assembly problem with the scheduling algorithms used for static-cyclic scheduling?

Unfortunately, we cannot simply run one algorithm after the other. To see this, consider the following simple multiset
of jobs:
\[
\{ j_1 = (2,1), j_2 = (3,1), j_3 = (3,1), j_4 = (6,1) \}
\]
where \( j_1 \) and \( j_2 \) must be co-located and \( j_2 \) and \( j_3 \) must be separated. In addition we have two processors each with 12 slots. If we solve the system assembly problem, a possible solution is to map \( j_1, j_2, \) and \( j_4 \) to processor 1 and \( j_3 \) to processor 2. Unfortunately, this allocation is not schedulable.

If, instead we try to schedule the jobs first, then we might wind up with \( j_1 \) and \( j_4 \) on processor 1 and \( j_2 \) and \( j_3 \) on processor 2. Unfortunately this does not satisfy the system assembly constraints.

What this example shows is that we cannot solve the problem in a compositional way.

The best we can hope for is a semi-compositional approach where we use the same algorithms as before for solving the system assembly problem and the scheduling problem. However, the algorithms are now part of a framework that enables them to interact in a way that allows us to solve the combined problem. We are experimenting with various approaches that are motivated by work from the verification community on combining decision procedures.

V. CONCLUSIONS AND FUTURE WORK

We advocated the use of verification technology to tackle the virtual integration problem for cyber-physical systems. The idea is to use high-level modeling languages that allow designers to declaratively specify what properties their architectural models should have, not how to achieve them. From such high-level models, we proposed to use verification technology to automatically synthesize architectural models that certifiably satisfy the high-level properties. One of the challenges is dealing with virtual integration in the presence of real-time scheduling constraints. While this problem is inherently not compositional, we advocated the use of semi-compositional methods.

ACKNOWLEDGMENT

We are indebted to John Chilenski of Boeing Commercial Airplanes for providing extensive guidance, support, feedback, and collaboration. This research is funded in part by NASA Cooperative Agreement NNX08AE37A.

REFERENCES


