Proceedings of DETC'98
1998 ASME 18th Computers in Engineering Conference
September 13-16, 1998, Atlanta, Georgia, USA

DETC98/CIE-5704

A MODULAR COMPOSABLE SOFTWARE ARCHITECTURE FOR THE SIMULATION OF MECHATRONIC SYSTEMS

Antonio Díaz-Calderón[1, 2]
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
Email: adiaz@cs.cmu.edu

Christiaan J. J. Paredis
Pradeep K. Khosla
Institute for Complex Engineered Systems
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
Email: cjp@cs.cmu.edu, pkk@cs.cmu.edu

ABSTRACT

This paper presents a software architecture for composing complete system-level simulations of mechatronic systems. The proposed architecture will provide the designer with the infrastructure to rapidly create simulations of alternative designs. The architecture promotes modularity and composability through the use of the *design entity*. Moreover, the architecture supports hierarchical modeling and provides the infrastructure to seamlessly integrate mechanics models with electronics and information technology models. Finally, the architecture facilitates distributed computing to take full advantage of the power of networked computers. This paper introduces the individual concepts of our architecture, and illustrates them in the design of a missile seeker.

INTRODUCTION

In early stages of design, modeling and simulation play an important role in supporting the evaluation of candidate designs (18). During conceptual design, the designer needs to evaluate candidate designs to find the ones that best meet the initial requirements. To do so, the designer typically recurs to modeling and simulation. When modeling a candidate design, the designer needs to use general behavior descriptions for all components of the design and define the interactions between these components. The designer must then decide which parts of the interaction rules can be considered second-order effects; these effects may initially be neglected leading to a simplified model. If the evaluation criteria change, the model needs to be modified. Moreover, to make a reasonable choice of candidate designs, the designer needs to repeat the whole process for as many candidate designs as he/she is able to handle in a reasonable amount of time.

The focus of this research is to develop a computational tool for supporting the modeling process of mechatronic systems. A mechatronic system is a complex system[3] which combines electronics and information technology to form both functional interaction and spatial integration in components, modules, products, and systems (5). Typical examples of mechatronic systems include automatic cameras, miniature disk drives, missile seeker heads, and consumer products like CD players, camcorders, and VCRs.

A large amount of work has been devoted to the improvement of modeling and simulation environments for dynamic systems. Researchers have focused on improving the modeling process such that it becomes easier for the engineer to write and maintain models. Different approaches have been proposed including the use of graph languages (i.e., Bond Graphs), and object oriented model description languages. Computational tools based on bond graphs (12) include CAMBAS (13) and the work reported in (16). CAMBAS is an automated modeling tool that supports the modeling task by automatically making improvements to the model based on some input requirements. The work reported in (16) also makes use of bond graphs to provide a computational tool that supports evolution of models of physical systems. In both systems, the authors provide the

[1]Affiliated with The Institute for Complex Engineered Systems.
[2]Address all correspondence to this author.
[3]Complex systems can be characterized as highly coupled, multi-system, multi-phenomenon entities.

means to store models in *model libraries* where the designer can search for components and use them to build the model of the physical system at hand.

Model description languages (MDL) for continuous/discrete time systems have received much attention in the last decade. These languages provide the semantic constructs to represent dynamic system entities in a way that it is easy to code and maintain. Furthermore, by providing an object oriented approach, models can be easily reused and developed incrementally (7; 6; 3; 4).

The basic property that these different approaches share is that they all are geared towards component-level modeling. Using component-level MDL, one can describe the complete system's model in terms of component elements. The resulting model is then translated into an executable representation. If there are changes in the model, a new translation is required to update the executable representation. This modeling approach is viable when there are only analytic representations of the processes involved. However, information modules (procedural information) also plays an important role in the behavior of a mechatronic system. Including procedural information in the current modeling environments is not possible: MDL does not provide control structures and support for complex data structures which are often used in information modules. Furthermore, if the MDL indeed supports procedural information (as is the case in SIDOPS+ (4)) the level of complexity of the algorithms that can be implemented in the MDL are often limited by the semantics of the MDL.

As an example of a system for which we need to consider information models, consider the following design problem:

Design a tracking device to track an object over a $100m \times 100m$ *area moving at* $v_{max} = 100m/sec, a_{max} = 7.25m/sec^2$. *The minimum distance to the target is 100m; the maximum distance to the target is 500m.*

We can envision a design, which will be comprised of a mass, actuators, sensors, and control and tracking algorithms that implement the tracking of the object (Fig. 1). Most of the components can be described in any of the modeling languages mentioned above (e.g., actuators and the mechanical system.) However this is not true for the algorithms that are also part of the final product (e.g., control algorithms). It is impossible to capture such procedural information in simple MDLs.

To alleviate this problem, we propose a computational tool that facilitates the synthesis of system-level models that include mathematical as well as information modules. Providing an automated mechanism for model synthesis, will improve the quality of the designed artifacts because the designer can explore many more design alternatives than with current manual technology. Ideally one would like to have a CAD framework that allows the designer to build custom simulators that include information modules as well as mathematical models.

In order to create the proposed software framework for the composition of simulation software, two more areas of research must be considered. These include software architectures and software reuse. Software architectures (10; 1) provide the tools to define an architectural plan. That plan describes how the components of a software system are put together. A closely related concept to a software architecture is that of software reuse. Based on software synthesis, interface adaptation, and object-oriented design, software reuse promotes (as its name implies) the reusability of software components. A very successful attempt at providing reusable software is implemented in the **Chimera** real-time operating system (15; 14). The basic building blocks in Chimera are port-based objects. A port-based object combines object-oriented programming with port automata. By using port-based objects, the user is able to reconfigure real-time software through assembly of software components.

Another example of an environment for rapid prototyping of analysis tools is the Engineering Methodology Application Tool (EMAT) framework (17). EMAT provides a Methodology Description Language to describe the interactions between different software components, and provides an environment for process execution and scheduling.

HIGH LEVEL ARCHITECTURE

Typically, the design of an artifact starts with an idea of what the final product will be in terms of functional components and the relationship among them. As the design evolves, the designer builds mathematical/information models that are then used in the analysis of the design. Further along in the design process, more information is included and the components are selected based on the information available. Similarly, a simulation model for a system will evolve along with the design. By capturing the *concept* of the design and translating it into a simulation program, critical information can be derived even before the concept is verified in virtual prototypes.

Component models experience a morphism which takes a model from one level of abstraction to another. Abstraction levels can be divided in three classes: conceptual, component, and process level (Fig. 2):

Conceptual level: The system is represented as a graph of interconnected functional components providing a high-level overview of the system. The graph can be hierarchically grouped such that each node in the graph can in turn be a sub-graph representing a high-level descrip-

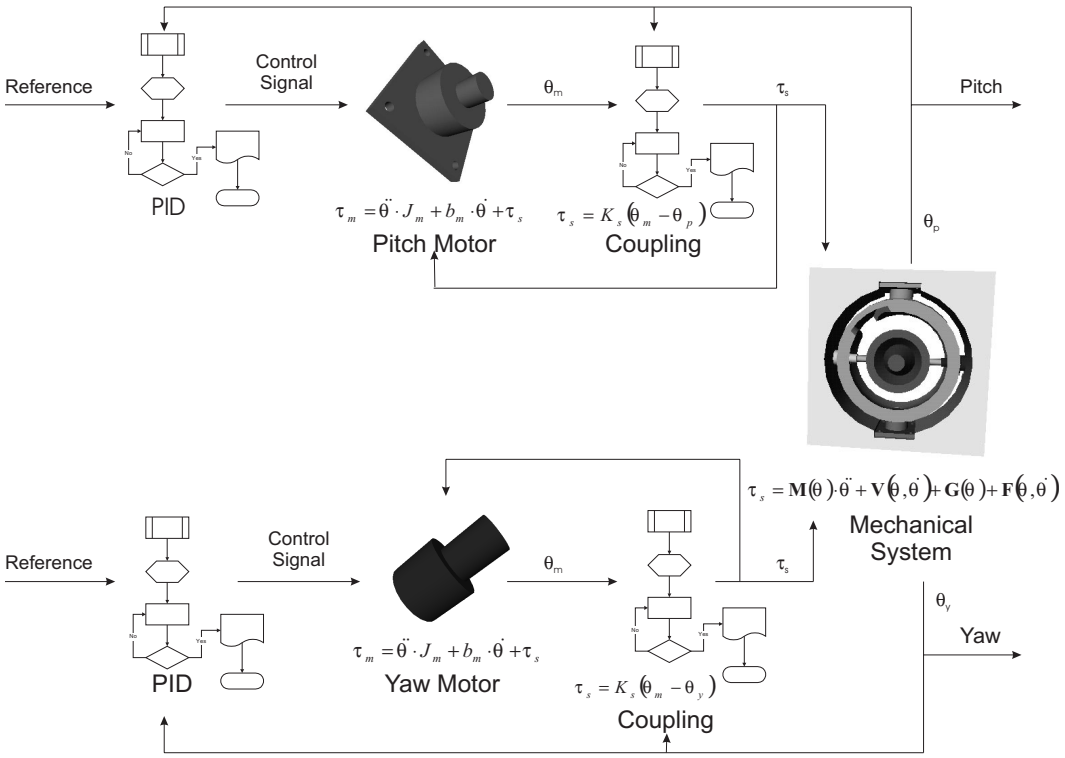

Figure 1. CONCEPTUAL DESCRIPTION OF THE TRACKING SYSTEM.

tion of a subsystem.

Component level: This level provides an abstract representation of the behavior of the component, and describes the *interface* of the component, i.e., inputs and outputs from and to the environment.

Process level: represents the algorithmic entities needed to provide the functionality that is publicized by the component at the component level.

As the design evolves, the system-level simulation model experiences a number of changes (improvements) in its basic composition. These changes can be viewed as movements in a two dimensional space. One dimension of this space represents hierarchical improvements of the model; the other dimension represents localized improvements in a particular sub-model.

Hierarchical refinement provides a vertical refinement mechanism: models can be assembled from existing software components or they can be substituted by an expansion of a model into a collection of sub-models. This dimension is motivated by the fact that the design of any product is hierarchical in nature. We start with basic building blocks. Obeying their physical and functional constraints, we arrange them into more complex components. We proceed in this way assembling subcomponents into larger components until we achieve the desired goal.

Consider the gimbal mechanism for a missile seeker. We may use a model that describes the dynamic behavior of the mechanism when some external forces are applied. If we also want to know the internal stresses on the gimbal mechanism, we need to refine the model. The new model will consist of the aggregation of two sub-models: the dynamic model and the stress model. This operation must take care of the interaction that may exist between the two sub-models. Likewise, the basic functionality of the model which is exported by the interface has to be maintained. This ensures that the resulting model can be used in the current context. The process of hierarchically aggregating models into more complex models is called hierarchical refinement.

Same-level refinement provides a horizontal refinement mechanism: refinement is achieved by replacing models with other models that have the same functionality but that provide more accurate results. Consider for instance the analysis of a control system. The designer can choose from a collection of control algorithms and test them with the complete system until he finds one

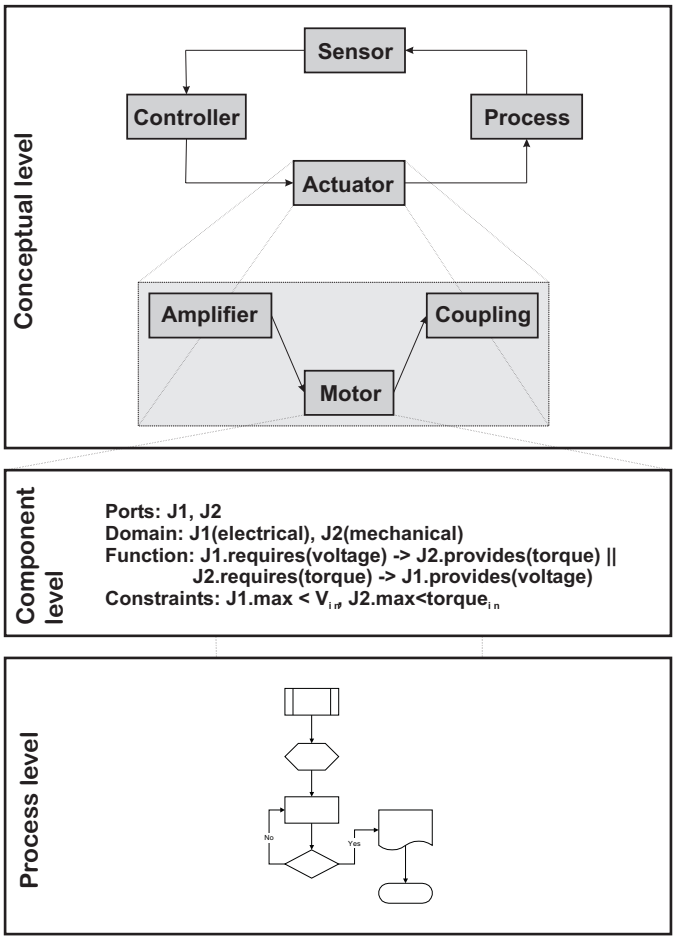

Figure 2. MODEL ABSTRACTION LEVELS.

that matches the requirements. Exchanging control algorithms at the process level allows the designer to refine the design through same-level refinement.

The mechanism by which models are given the freedom to move around in the model space is *composability*. Composable simulation is the ability to assemble software modules into a structure that is semantically correct with respect to the original design. The proposed architecture provides the ability to combine currently available simulation software modules into an executable configuration. The configuration must resemble the physical design in the sense that it captures all the interactions among different elements. As a final product, we obtain a software simulation tailored to the problem at hand.

To support composability, we propose a software architecture based on the three abstraction levels described above and shown in Fig. 2. The highest level in the architecture (conceptual level) defines the system using a *conceptual graph*. A conceptual graph is a directed graph in which nodes represent components of the system and edges represent their interactions. Leaving edges represent output ports of the subsystem, incoming edges represent input ports. At the *component level*, the conceptual graph is mapped into a simulation software architecture. The resulting architecture is the main representation of the semantics of the system: a network of interfaces and interactions among components. The third level in the architecture is the *process level* in which the software architecture is instantiated and executed. Instantiating an architecture requires finding processes that match the features listed in the interface of each component. Execution of the architecture may require calls to external libraries such as Matlab, ACIS or Dymola (Fig. 3.) This level of the architecture deals with the communication and synchronization of processes included in the architecture and ensures data consistency.

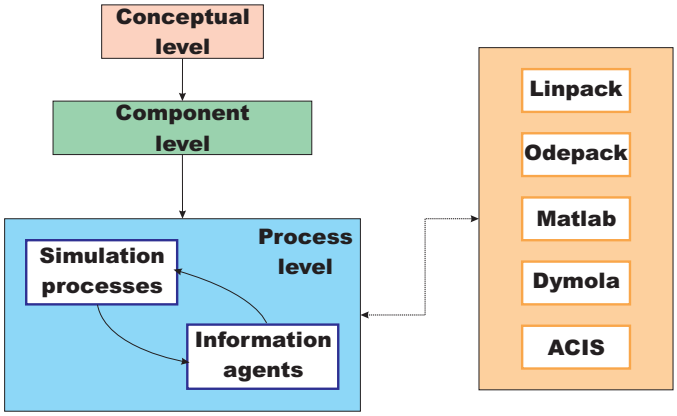

Figure 3. OVERVIEW OF THE CAD FRAMEWORK ARCHITECTURE

CONFIGURATIONS AND DESIGN ENTITIES

In our framework for the analysis of the design of mechatronic systems, the primary abstraction is the *design entity*. It represents a portion of a design that has well-defined interaction points and performs a well-defined function. The structural arrangement of design entities defines the configuration of the system. A configuration describes how design entities are combined to form a complete design. The user interacts with the environment through a graphical user interface (GUI) (Fig. 4); at the same time, the configuration representing the design is constructed in the background (Fig. 5.) A system is a network of configurations which in turn are composed of a network of subconfigurations and design entities. In Fig. 5, system W happens to be a linear sequence of configurations; however, the topology of the configuration may be any acyclic directed

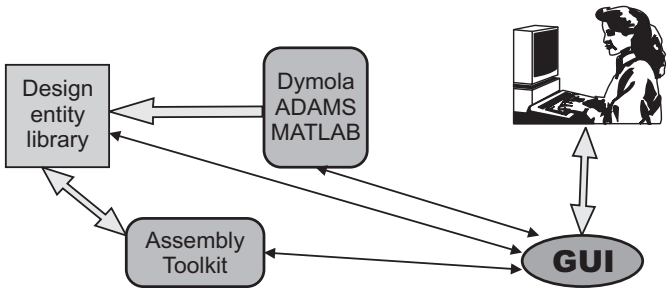

Figure 4. INTELLIGENT CAD FRAMEWORK: FRONT END.

graph and it is only restricted by the causality of the component's connection ports.

To achieve composability, the design entities must exhibit two fundamental properties: reconfigurability and assemblability. Reconfigurability is the property by which a design entity can adapt itself to changing requirements. Assemblability is the property that allows a design entity to be embedded in a configuration. A very elegant approach that complies with these two properties is based on the theory of software systems architecture (9; 10; 11; 2; 1).

In the next section, we will review some elementary concepts in software architectures, which we will then use to define the architecture of our simulation framework.

Software architectures

A software architecture is a specification of a class of systems. It consists of a set of specifications called *interfaces*, a set of *connection rules* that define valid communication channels between the interfaces, and a set of *formal constraints* that define legal or illegal patterns of communication (10). Each element in the architecture is divided into two major parts:

Interface — a description of the component's features including its input/output relationship with the environment.

Implementation — a procedural description of the component's interface.

The implementation of a component is hidden from the rest of the world. This separation promotes re-usability and assemblability of architectural components. The set of connection rules defines the topology of the *connection graph* of the architecture. The connection graph is a graph in which the nodes represent the interfaces to the elements in the configuration, and the edges represent the communication channels defined by the communication rules. The interfaces specify the components of the system, and the connections and constraints define how the components may interact.

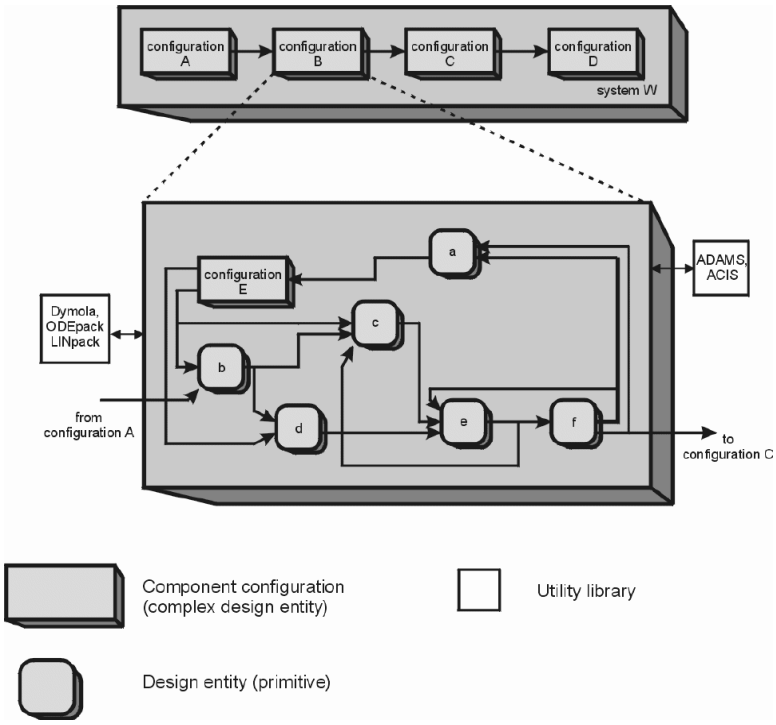

Figure 5. INTELLIGENT CAD FRAMEWORK: BACKGROUND PROCESS.

Interfaces An interface defines a class of objects (10). It defines the kind of events an implementation can observe or generate, the information exchanged with other modules, and constraints on its external behavior. The instances of an interface are the implementations that match the interface. An interface may declare types and actions; these are the exported constituents of the interface. There may be many different implementations that have the same interface. For instance, different control algorithms for the missile seeker may share the same input/output specifications.

Implementations An implementation defines the computational processes that perform the actions publicized by the interface. Implementations are algorithmic representations usually coded in some high-level programming language.

Connectors Connectors (or connection rules) define direct communications between interfaces. They specify valid communication channels between interfaces.

Figure 6. DESIGN ENTITY.

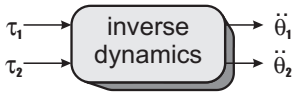

Figure 7. INVERSE DYNAMICS DESIGN ENTITY.

Formal constraints Formal constraints define legal or illegal patterns of communication between elements of two interfaces, e.g. bounds on the values that are exchanged through a communication channel.

We will now use the software architecture concepts described above to define the basic object in our CAD framework: the *design entity*.

The design entity

A design entity consists of two separate parts: an interface through which it interacts with other design entities, and an implementation that either encapsulates an executable prototype of the design entity, or hierarchically defines it as a configuration[4] of other design entities. Interfaces communicate through ports (Fig. 6.) An interface defines:

1. Input/Output ports.
2. Energy domains associated to each I/O port.
3. The events a design entity can generate or to which it can respond.
4. The functions it provides to other design entities or requires from other design entities.
5. Constraints on its external behavior.

For instance, the interface definition for a DC motor/generator design entity might look like:

```
Ports       : J1, J2
Domain      : J1(electric), J2(rotational)
Functions   : J1.requires(current) -
> J2.provides(torque) ||
              J2.requires(torque) -
> J1.provides(current)
Constraints : J1.max < Im, J2.max < Torquem
```

A design entity is the generalization of a port-based object. A design entity's interface may be satisfied by more than one implementation. For example, the interface definition for the mechanical system of a missile seeker may specify as input ports the torques applied to the system, and as output ports the angular accelerations of the system produced by the given torques (Fig. 7.) There are two alternatives to choose from to select the implementation for this interface: one using the Newton-Euler iterations, the other using the Lagrange-Euler method. Both implementations conform with the interface but provide different mechanisms to compute the desired results. This feature can be used to reconfigure the simulation dynamically to test more accurate models, to produce finer simulation results, or simply to test different implementation approaches.

We are going to define a *task* as the thread of execution of a design entity implementation. Tasks are periodic elements that perform the functionality described by the interface. Also, we define the implementation repository, as an object-oriented database of design entity implementations that are available to instantiate a configuration. For example, implementations included in the mechatronics repository are actuators, sensors, integrators, etc. A connection between two interfaces is established by connecting an output port of one interface to a corresponding input port. A configuration is legal only if every input port is connected to one output port and all the constraints are satisfied. An output port may be connected to multiple input ports, but an input port (of an instance of a design entity) may only be connected to a single output port. This rule is based on the fact that connections represent the flow of information from an output port to an input port. From an output port information can flow to multiple input ports; however, multiple output ports feeding a single input port would cause contentions (an adder could be added to combine multiple outputs into a single output.)

An instance of a configuration of a system is created when all the interfaces in the configuration are assigned conforming implementations (10). An implementation conforms to an interface if it contains all features specified by the interface and exhibits the same behavior as that publicized by the interface. In (10) the authors define three conformance criteria that we adopt to define semantically correct systems:

Interface conformance — each implementation in the system must conform to its interface. This means that the implementation has to match the interface definition semantically; otherwise, the implementation cannot be used in the given context.

Decomposition — each particular instance of a configuration is decomposed in a number of implementations; these implementations must conform to the interfaces of the configuration. This means that for each interface

[4] A configuration is a software architecture in which the elements of the architecture are design entities.

there must be at least one conform implementation in the database.

Communication integrity — the system's components interact only as specified by the configuration.

A configuration is recursively defined to be composed of sub-configurations or interfaces of design entities. This definition supports the hierarchical nature of a mechatronic system. As for a design entity, a configuration is also separated in two parts one being its interface, the other being its implementation. The interface of a configuration will export only those features visible at the sub-system level. The implementation of the configuration will be defined by the network of sub-configurations and design entities described in the definition of the configuration. Design entities in the implementation of a configuration have a well defined scope. This means that messages sent locally in a configuration cannot be heard outside the boundaries of that configuration; only those features indicated in the interface are exported and therefore can be used by other configurations. Since an interface may have more than one conforming implementation, it is valid to replace the complete network attached to the implementation of a configuration by a different network or with a single design entity. The new implementation may be either more or less complex than the original as long as it maintains the basic functionality specified by its interface.

The mappings from modeling abstractions to design entities

The model presented page 2 defines three levels of modeling: conceptual, component, and process level. Each level has a clear mapping to a component of a design entity. At the conceptual level, design entities are structured hierarchically into configurations. Connections defined in the configuration specify the flow of information between its members. The data flow is used to schedule the execution order of the implementations in the configuration.

The component level is represented by the interface of the design entity, and the process level is represented by the implementation that conforms with the interface of the design entity.

In the next section, we will describe a prototype system that illustrates the concepts developed in this section.

PROTOTYPE SYSTEM

Some of the concepts presented in this paper have been tested in a prototype system that supports only one family of devices: a missile seeker (Fig. 8). The current version of the system implements the conceptual level and the process level. The component level is being implemented as part of the next generation of our system. A snapshot of a session

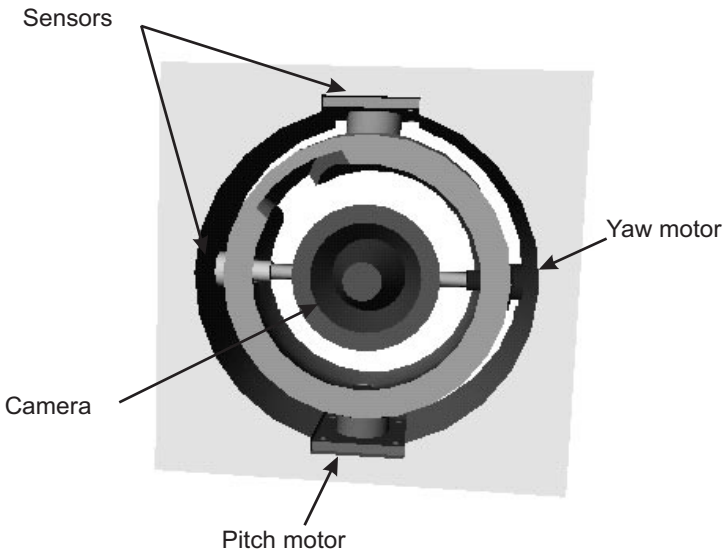

Figure 8. THE MISSILE SEEKER.

is shown in Fig. 9. A general purpose graph editor (8) has been modified to allow the user to define the conceptual graph as a network of interfaces. The implementations for all interfaces are manually assigned by the user after which the configuration is automatically instantiated. This means that all the tasks are spawned and the communication between them is established automatically. The user is free to change the implementation associated with an interface (same-level refinement); just re-instantiating the configuration, will generate the updated simulation software.

The configuration for the missile seeker is defined by four different interfaces: PID controller, DC motor, coupling, and mechanical system. Assemblability is achieved when the instantiation of components like the PID controller is realized. The interface describing the PID controller is instantiated twice. The interface specifies how the PID controller is used in the system and what the valid connections are. Since each instance of the interface is connected through different communication channels, the implementation of the PID controller, although the same in this case, receives different input values from the neighboring components. The same is true for the DC motor and the coupling device.

Enabling technologies like object-oriented design and distributed object systems are providing us with the tools to completely support the three abstraction levels.

SUMMARY

In this paper, we described the architecture of a computational tool to rapidly create simulations for mechatronic

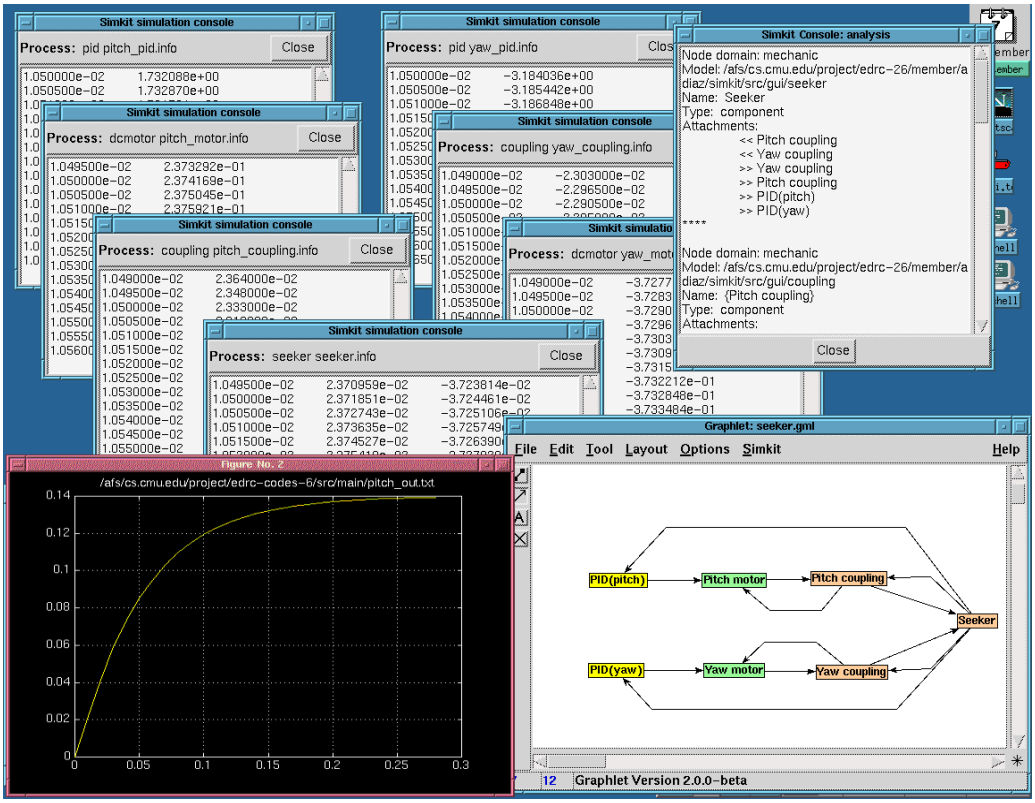

Figure 9. SCREEN DUMP OF THE PROTOTYPE SYSTEM.

systems. To create these simulations the designer combines software modules into a complete system-level simulator. The novel features include 1) the creation of simulation software by combining software modules, 2) the inclusion of information agents (e.g., vision and control algorithms) in the simulation process, and 3) the ability to reconfigure the simulation dynamically to achieve different levels of granularity.

The architecture is based on a three-level abstraction: *conceptual level, component level* and *process level*. Each abstraction level represents a different aspect of the model, and the three collectively support *composability* of software modules. The fundamental abstraction used in the architecture is the *design entity*. Composability of design entities is a powerful mechanism that provides hierarchical and same-level refinement of simulation models. Furthermore, it allows one to build the simulation of a complex system that integrates mechanics with electronics and information technology modules.

Testing of the prototype system confirmed the validity of our approach. In the future, we will develop a distributed implementation of the simulation on a network of computers, allowing us to take advantage of the computing power of networked computers.

ACKNOWLEDGMENT

Our thanks are due to Dr. Satyandra K. Gupta for suggesting the missile seeker system as a test case, and for his insightful comments on the definition of the conceptual graph, and on the implementation of the prototype system.

This research was funded in part by DARPA under contract ONR # N00014-96-1-0854, by the Institute for Complex Engineered Systems at Carnegie Mellon University, and by the National Council of Science and Technology of Mexico (CONACyT.)

REFERENCES

Robert J. Allen and David Garlan. Formal connectors. Technical Report CMU-CS-94-115, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, March 1994.

Robert J. Allen and David Garlan. Formalizing architectural connection. In *16th International Conference on Software Engineering*, Sorrento, Italy, May 1994.

Mats Andersson. *Object-oriented modeling and simulation of hybrid systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, dec 1994.

Arnold Pieter Jan Breunese. *Automated support in mechatronic systems modeling*. PhD thesis, Department of Electrical Engineering, University of Twente, Enschede, The Netherlands, dec 1996.

J. Buur. *A theoretical approach to mechatronics design*. PhD thesis, Institute for Engineering Design, Technical University of Dennmark, Lyngby, Dennmark, 1990.

Francois E. Cellier and Hilding Elmqvist. Automated formula manipulation supports object-oriented continuous-system modeling. *IEEE Control Systems*, 2(13):28–38, April 1993.

Francois E. Cellier, Hilding Elmqvist, and Martin Otter. Modeling from physical principles. http://www.ece.arizona.edu/~cellier, Department of Electrical and Computer Engineering The University of Arizona, 1996.

Michael Himsolt. *Graphlet*. Passau University, Passau, Germany. http://www.fmi.uni-passau.de/Graphlet.

David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, September 1995.

David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

David C. Luckham, James Vera, and Sigurd Medal. Three concepts of system architecture. Technical Report CSL-TR-95-674, Computer Systems Laboratory, Stanford University., Stanford, CA 94305, July 1995.

Ronald C. Rosenberg and Dean C. Karnopp. *Introduction to physical system dynamics*. McGraw-Hill, New York, NY, 1983.

Jeffrey L. Stein and Loucas S. Louca. A component-based modeling approach for system design: theory and implementation. In *Proceedings of the 1995 International Conference on Bond Graph Modeling and Simulation*, Las Vegas, NV, January 1995.

David Bernard Stewart and Pradeep K. Khosla. Rapid development of robotic applications using component-based real-time software. In *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 465–470. IEEE Computer Society, August 1995.

David Bernard Stewart and Pradeep K. Khosla. The chimera methodology: Designing dynamically reconfigurable and reusable real-time software using port-based objects. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):249–277, June 1996.

Jan Lubertus Top. *Conceptual modelling of physical systems*. PhD thesis, University of Twente, Enschede, Netherlands, september 1993.

N. Wang and Cheng J. Emat: an engineering methodology application tool. In *Proceedings of the Computers in Engineering Conference and the Engineering Database Symposium, ASME*, pages 21–26, 1995.

Peter M. Will. Simulation and modeling in early concept design: an induatrial perspective. *Research in Engineering Design*, 3(1):1–13, 1991.