

PACT: An Experiment in Integrating Concurrent Engineering Systems*

Mark R. Cutkosky, Robert S. Engelmore, Richard E. Fikes,
Michael R. Genesereth, Thomas R. Gruber, Stanford University

William Mark, Lockheed Palo Alto Research Labs
Jay M. Tenenbaum, Jay C. Weber, Enterprise Integration Technologies

Keywords: agents, distributed design, knowledge sharing

Abstract

The Palo Alto Collaborative Testbed (PACT) is a laboratory for joint experimentation in computer-aided concurrent engineering being pursued by research groups at Stanford University, Lockheed, Hewlett-Packard, and Enterprise Integration Technologies. The current prototype integrates four preexisting concurrent engineering systems into a common framework. Each individual system is used to model different aspects of a small robotic manipulator, and to reason about them from a different engineering perspective (dynamics, digital electronics, and software). The systems interact via knowledge-based communication languages and services. Initial PACT experiments have explored engineering knowledge exchange in the context of a distributed simulation and simple incremental redesign scenario. This paper describes the nature of these experiments, the technology underlying them, and the results produced.

*This work was partially supported by DARPA prime contract DAAA15-91-C-0104 through Lockheed subcontract SQ70A3030R (monitored by the U.S. Army Ballistic Research Laboratory), through the support of Hewlett-Packard Laboratories, and by the Office of Naval Research (ONR N00014-92-J-1833).

Introduction

Several research groups are jointly developing the Palo Alto Collaborative Testbed (PACT), an infrastructure for concurrent engineering that encompasses multiple sites, subsystems, and disciplines. Through PACT, we are examining the technological and sociological issues of building large-scale, distributed, concurrent engineering systems. The approach has been to integrate existing multi-tool systems that are themselves frameworks, each developed with no anticipation that they would subsequently be integrated. We take as a given that individual engineering groups prefer to use their own tool suites and integration environments; there is a significant investment in these self-contained systems. Nonetheless, for projects that involve large segments of the enterprise or multiple enterprises, the engineering activities that take place within individual frameworks must be coordinated. PACT experiments have explored issues in building an overarching framework along three dimensions: cooperative development of interfaces, protocols and architecture; sharing of knowledge among systems that maintain their own specialized knowledge bases and reasoning mechanisms; and computer-aided support for the negotiation and decision making that characterize concurrent engineering. PACT serves as a testbed for knowledge sharing research emerging from the artificial intelligence community, as well as for emerging data exchange standards such as PDES/STEP [3].

The PACT architecture is based on interacting agents (programs that encapsulate engineering tools). The agent interaction in turn relies on shared concepts and terminology for communicating knowledge across disciplines, an interlingua for transferring knowledge among agents, and a communication and control language that enables agents to request information and services. This technology allows agents working on different aspects of a design to interact at the *knowledge level*: sharing and exchanging information about the design independent of the format in which the information is encoded internally.

The PACT systems include Next-Cut, a mechanical design and process planning system from the Stanford Center for Design Research, Designworld, a digital electronics design, simulation, assembly, and testing system from the Stanford Computer Science Department and Hewlett-Packard, NVisage, a distributed knowledge-based integration environment for design tools developed by the Lockheed Information and Computing Sciences group, and DME (Device Modeling Environment), a model formulation and simulation environment from the Stanford Knowledge Systems Laboratory. In the initial experiments, each system modeled different aspects of a small robotic device and reasoned about them from the standpoint of a different engineering discipline (software, digital and analog electronics, and dynamics). The systems then cooperated to produce a distributed device simulation, and to synchronize on a subsequent design modification.

In this paper we discuss the motivations for PACT and the significance of the approach for concurrent engineering. We review our initial experiments in distributed simulation and incremental redesign, describe PACT's agent-based architecture, and discuss lessons learned and future directions. For details on the underlying language, protocols and knowledge-sharing research issues, we refer the reader to references [1, 2, 4].

The Distributed Design Problem

Although concurrent engineering is almost universally advocated today, it is hard to accomplish when large, multidisciplinary projects are involved. To illustrate some of the issues, consider the team depicted in Figure 1. At any instant, the team members may be working at different levels of detail, each employing his or her own representations of physical artifacts, engineering models, and knowledge. For example, the kinematician is primarily concerned with the geometry of the device

Figure 1: A multidisciplinary design team.

and its configuration space, while the dynamicist is constructing an accurate set of equations to predict device behavior. At the same time, the controls engineer needs a linearized dynamics model that captures the joint-space to task-space mapping and the primary inertial effects.¹ Despite their differences in perspective, the specialists share considerable information. For example, an appreciation of the dynamics is necessary when choosing the force sensors. Conversely, information about non-linearities in the amplifiers and motors will be of interest to the dynamicist in setting up the simulation. In applying information technology to support engineering projects such as this, we seek tools that will help the team members share knowledge and keep track of each others' needs, constraints, decisions and assumptions.

Ironically, contemporary computer design and manufacturing tools appear to exacerbate the problems that concurrent engineering is trying to solve. Most tools provide point solutions to particular modeling and analysis problems, based on idiosyncratic representations and algorithms. Designers are often frustrated because these tools are developed by and for experts. Effective use requires one to know the conventions (e.g., for representing spatial transformations and instantaneous velocities), the assumptions (e.g., that components are rigid bodies), and other characteristics or limitations of the approach (e.g., whether the method is strictly conservative and whether it always converges). Such information is rarely explicit in the models themselves, but must be absorbed by wading through extensive documentation and consulting with the developers and frequent users of the tools. More fundamentally, by preventing designers from sharing their design models, idiosyncratic design tools contribute to their isolation.

¹Indeed, as far as the controls engineer is concerned, the equations could just as well represent a gear train or an electronic circuit, with an analogous set of partial derivatives relating inputs and outputs.

Figure 2: Manipulator system and division of responsibilities for the PACT experiment

Integration via Shared Design Models

A design such as that depicted in Figure 1 embodies a large number of interacting constraints that must be met by the set of components that make up the design. To ensure that constraints are met, component descriptions must be organized into a representational framework that represents the evolving design. This framework is the design model. The descriptions in the model are composed from shared *ontologies*, i.e., sets of agreed-upon terms and formally described meanings, of the design domain. In contrast to the superficial representations (e.g., CAD data files) found in current design environments, the design model forms a basis for knowledge sharing among diverse systems.

One solution for providing a shared design model is shown in Figure 3a. However, this approach runs into trouble in large-scale distributed design environments. Individual design tools tend to be highly specialized, making use of particular representations and reasoning methods to do their work efficiently. Any sharing must take into account the fact that these tools do not share the same internal model. Furthermore, design is a process of negotiation: decisions are made and changed frequently as specifications change and new ideas are brought forward. A single shared database encompassing all of the data of participating tools would quickly become a bottleneck.

The PACT experiments, though preliminary, have shown the feasibility of design through the interaction of distributed knowledge-based reasoners. Each reasoner has its own local knowledge base and reasoning mechanisms. As shown in Figure 3b, agents interact through (currently very simple) *facilitators* that translate tool-specific knowledge into and out of a standard knowledge interchange language. Each agent can therefore reason in its own terms, asking other agents for information and providing other agents with information as needed through the facilitators.

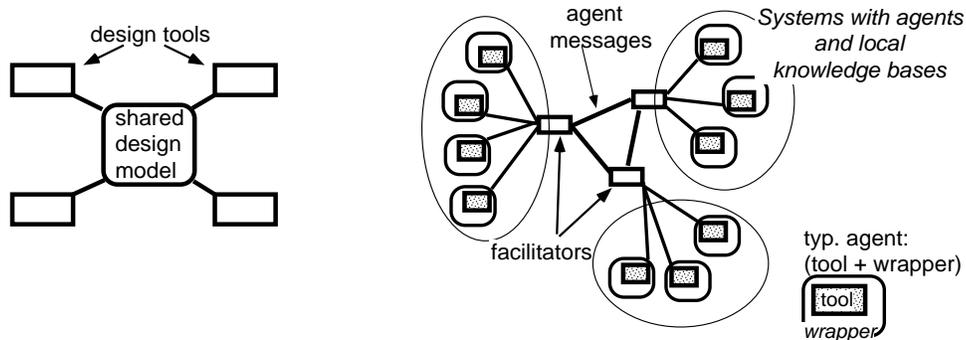


Figure 3: Interacting design tools require a shared design model, as shown at left. Scalability requires distribution of the model, with mechanisms for translating among different representations and coordinating interactions, as shown at right.

PACT Software Interoperation

The PACT architecture was designed to conveniently and flexibly integrate the diverse operating systems, control mechanisms, data/knowledge formats, and environmental assumptions of software not originally written for wide-area interoperation. It is an extension of Open Distributed Processing (ODP) architectures appearing on mainstream computing platforms. For example, the publish/subscribe mechanism of Macintosh System 7 and the Dynamic Data Exchange mechanism of Windows 3.0 provide common data formats and mechanisms for interapplication messaging. On a larger scale, the Object Management Group's Common Object Request Broker Architecture (OMG/CORBA) [7] defines sophisticated service naming and parameter type translation.

Although ODP architectures define the *form* of software interoperation, they say little about the *content* of shared information. Complex software interoperation, such as CAD data translation, recursive request decomposition, knowledge-based interaction (e.g., assert, retract, evaluate), and service location based on functionality, are all left to *ad hoc* agreements among individual programs (i.e. programmers). This situation limits the interoperation to collections of programs engineered to interact with one another. Scalability suffers because idiosyncratic agreements about message content must be reevaluated whenever a new participant joins the interaction.

Message content standards Our software interoperation architecture extends ODP architectures by further standardizing the style of program interactions. We do this by defining message content at three levels:

- Messages are expressions in a common *agent communication language* (ACL) that supports knowledge-base operations (assertions, queries, etc.)
- The message contents of the knowledge-base operations are expressions in a common *knowledge interchange format* that provides an implementation-independent encoding of information.
- The expressions stated in the interchange format use a standard vocabulary from *common ontologies* that are defined for the shared application domain.

Each of these levels is the subject of widespread discussion and development, primarily by DARPA-sponsored knowledge-representation standards committees [1]. One group has proposed an agent communication language called the Knowledge Query and Manipulation Language (KQML) [2].

KQML specifies a relatively small set of *performatives* that categorize the services that agents may request of one another. For example, one agent may request to **assert** a fact to another's local data/knowledge, **retract** a previous assertion, or obtain the answer to some **query**.

A second proposal is a specification of the knowledge interchange format called KIF [4]. KIF can be used as a format for KQML arguments (KQML allows multiple formats). KIF is a prefix version of the language of first order predicate calculus, with various extensions to enhance its expressiveness. KIF provides for the communication of constraints, negations, disjunctions, rules, quantified expressions, and so forth. With the application of AI technology to practical problems, more programs are able to manipulate information of this sort.

The third effort concerns the development of engineering ontologies. This effort is focusing on defining formal vocabularies for representing knowledge about engineering artifacts and processes, specifying the assumptions underlying the common views of this information. Given the application-specific nature of such ontologies, this activity is not producing a single specification, but is addressing various domains of importance to knowledge sharing such as device behavior modeling. This effort is complimentary to the work of PDES/STEP vocabulary committees, which have been most successful in specifying data models for domains such as solid modeling and finite-element geometry.

Facilitators Plugging into the PACT architecture requires a substantial commitment to supporting the above-described message content. To ease this burden, we introduced the use of *facilitators*. Each facilitator is responsible for providing an interface between a local collection of agents and remote agents, by serving four purposes: (1) providing a layer of reliable message-passing, (2) routing outgoing messages to the appropriate destination(s), (3) translating incoming messages for consumption by its agent, and (4) initializing and monitoring the execution of its agents.

Communication and coordination, therefore, occur between agents and facilitators and among facilitators, but not directly between agents. We call this arrangement of agents and facilitators a *federation architecture*. Messages from agents to facilitators are undirected, i.e., they have content but no address. It is the responsibility of the facilitators to route such messages to agents able to handle them. In performing this task, facilitators can go beyond simple pattern matching — they can translate messages, decompose problems into subproblems, and schedule the work on those subproblems. In some cases, this can be done interpretively (with messages going through the facilitator); in other cases, it can be done in one-shot fashion (with the facilitator setting up specialized links between individual agents and then stepping out of the picture).

PACT agent communication The PACT demonstration involved 31 agent-based programs executing on 15 workstations and microcomputers. When grouped by engineering discipline, these programs compose into the following six “top-level” agents: the digital circuitry agent, the control software agent, the power system agent, the physical plant agent, the sensor agent, and the parts catalog agent. All but the latter two existed before we built the PACT system (see sidebars).

Each top-level PACT agent worked through a facilitator to coordinate its interactions with other agents. Each PACT facilitator consisted of two parts: a *connection associate* and an *agent manager*. A connection associate implemented a layer of reliable message-passing above the widely available TCP/IP transport protocol, and was nearly identical in implementation across facilitators. The connection associate supplied message strings for the agent manager. If the message was a control message from another facilitator, the agent manager would interpret the message and react accordingly; if the message was from an agent (via a facilitator), the agent manager would perform any necessary translations before forwarding the message to its agent.

The PACT Experiments

To test our ideas about knowledge sharing, interoperability, and agent-based architectures for concurrent engineering, we built a demonstration that was first shown in October 1991. The demonstration scenario involves four geographically distributed teams, collaborating on the design, fabrication, and redesign of an electromechanical device. Each team has responsibility for different subsystems, and is supported by its own computational environment. The environments are linked through the federation architecture.

The subject of the PACT experiments was a robotic manipulator, chosen as a convenient example of a system that combined mechanics, electronics, and software, with extensive design documentation. The manipulator system is shown schematically in Figure 2. The two fingertips are positioned by five-bar linkages connected directly to the shafts of DC torque motors. The motors are powered by linear current amplifiers, and their joint angles are measured using shaft encoders. A digital circuit counts pulses from the encoders and multiplexes the resulting numbers so that high and low bytes are fed sequentially over an 8-bit parallel line to the controller. Depending on the application, various control laws are run at rates between 200 and 500Hz. Commanded torques from the controller are produced as voltages from a D/A converter, which drive the amplifiers.

Figure 2 also shows the division of responsibilities among the PACT teams. The control software was the responsibility of the Lockheed team, using their NVisage environment. The power system and sensors were the responsibility of the Stanford Knowledge Systems Laboratory (KSL) team, who used their Device Modeling Environment (DME). The manipulator mechanism was the responsibility of Stanford's Center for Design Research (CDR) and Enterprise Integration Technologies (EIT), using their Next-Cut system. Finally, the digital circuit was the responsibility of the team from Stanford's Logic Group and Hewlett-Packard, using their Designworld system. Designworld also maintained a catalog of components with shape and size information.

The participants each had their own hardware and software platforms, including two Macintoshes, a DEC 3100, a TI Explorer LISP machine, and HP and Sun Workstations — all linked through the Internet.

The demonstration scenario consisted of three parts. The first part was an example of cooperative design refinement. The second part was a simulation of the manipulator, requiring exchange of data among all four subsystems. The third part illustrated a typical interaction in which a design change initiated by one team necessitated changes by another.

Cooperative Design Refinement In cooperative design refinement, each team begins with an initial subsystem design and is required to produce a revised design that meets new specifications on accuracy, measurement methods, etc. Considerable communication is required to resolve design interactions.

Some design interactions for the manipulator are depicted in Figure 4. Next-Cut's analysis uses a detailed rigid-body dynamics model whereas NVisage needs a less detailed model for its controller. However, these models have to be consistent. There are also many interacting constraints. For instance, encoder resolution and maximum motor velocity constrain the choice of chips used for decoding and multiplexing the encoder signals in the digital circuit.

The PACT demonstration directly addressed only a few such interactions. The most interesting were those between NVisage and Next-Cut. For the experiment, the Lockheed design engineer selects a simple Cartesian position controller from an NVisage database of skeletal controller designs. To design the controller, NVisage needs a kinematic mapping from joint space to Cartesian work space. One of the modules within NVisage therefore sends expressions of the form `(interested-in nvisage '(ASSERT (closed-form '(pmx $q1 $q2) $f)))`. This state-

Figure 4: Interactions among aspects during manipulator design refinement

ment specifies that the NVisage agent requests a closed-form expression $\$f$ for determining the horizontal fingertip coordinate of the planar manipulator (\mathbf{pmx}) in terms of the joint variables q_1 and q_2 . The meanings of terms `closed-form` and `pmx` are specified in the common ontology for the PACT experiment. This request is picked up by Next-Cut, which is able to respond with equations (e.g., bindings for $\$f$). NVisage receives this information, compiles the equations into equivalent statements in executable code, and dynamically links these into its component functionality module.

The CDR team was not required to know anything about how the controls specialist was using the requested information to create a control law. Similarly, the Lockheed team was insulated from the extensive geometric and kinematic knowledge of the mechanism used by the CDR team. On the other hand, it was necessary for the two systems to agree on their definitions of space, time, coordinate frames, closed-form expressions, units of measure, and so forth, to effect the exchange of the coordinate transformations and inertial terms.

Distributed Simulation In the next part of the demonstration, the Lockheed team initiates a simulation of the entire device to test their controller design. The simulation is distributed over the PACT network, with each of the four systems simulating their respective components and communicating their results to the others. As Figure 2 shows, one can effect the simulation through a simple loop, where each system sends commands or data to the next. The simulation proceeds around the loop until DME, which models the motors and power system, signals that a motor has overheated, due to overloading. The simulation has therefore indicated that a redesign is needed.

The simulation was actually developed first, as a forcing function to get all the component systems communicating via message passing, and to test the adequacy of their models. The exchange of information was primarily achieved by passing simple assertions around the loop. For example, the assertion

```
(ASSERT (= (val pm-encoder-w-1 10000)4095))
```

states that the position encoder for wheel 1 at time 10000 is reading 4095. These messages trig-

gered reevaluations of the subsystems in response to the state changes. Although this part of the demonstration served to initiate the experiments in interoperation, extensive negotiations among system designers were required to decide on control sequence and data formats. In that sense the integration of systems for the distributed simulation was done in the traditional *ad hoc* fashion.

Distributed Redesign The objective of the last part of the demonstration was to explore interactions that occur during redesign, when the decisions of one team member have consequences for other parts of the design. In response to the motor burning out, the power subsystem designer uses DME to replace the motors originally selected for the manipulator with a larger model and to notify the other subsystems of the change. The change notice is broadcast on the network as an ASSERT message, stating the part number of the new motor. Next-Cut is interested in such a change because the motor forms part of the manipulator arm linkage assembly, and changing the motor may affect the mating features on the links. Next-Cut therefore posts a request for the dimensions of the motor with the new part number. That query is handled by Designworld, which maintains a components catalog. The Designworld agent sends back the needed dimensions, thereby allowing Next-Cut to adjust mating features on the links, along with the process plan for machining them.

Lessons Learned

The introduction characterized PACT as a testbed for cooperative research, knowledge sharing, and computer-aided engineering. This section summarizes what was learned in each area.

Cooperative research Designing an integrated system by committee is always hard, and PACT was no exception. The most difficult task, by far, was agreeing on the ontological commitments that enable knowledge-level communication among the systems. Designing a shared ontology is difficult because it must bridge differences in abstractions and views. Agreements must be reached about concepts in the natural world, such as position and time, shape and behavior, sensors and motors. For each concept, agreement is required on many levels, ranging from what it means to how it is represented. For instance, how should two agents exchange information about the voltage on a wire (what units, what granularity of time?); how should manipulator dynamics be modeled (as simultaneous equations or functions? in what coordinate frame?). The four systems comprising PACT used various coordinate systems and several distinct representations of time (e.g., discrete events, points in continuous time, intervals of continuous time, piecewise approximations). These representations were chosen for valid task- and context-dependent reasons. They cannot simply be replaced by one standard product model (e.g., with a single representation of time). However, the agents involved in any transaction must agree on a common ontology, which defines a standard vocabulary for describing time-varying behavior under each view of time that is needed.

What went on behind the scenes in PACT, and is not represented in computational form at all, was a careful negotiation among system developers to devise the specific pairwise ontologies that enabled their systems to cooperate. The developers met and emulated how their respective systems might discuss, say, the ramifications of increasing motor size. In this fashion, they ascertained and agreed upon what information had to be exchanged, and how it would be represented. Therefore, what PACT actually demonstrates is a mechanism for distributing reasoning, not a mechanism for automatically building and sharing a design model. The model sharing in PACT, as in other efforts, is still implicit — not given in a formal specification enforced in software. The ontology for PACT was documented informally in email messages among developers of the interacting tools. The tools in PACT are able to interact coherently because the commitments to common concepts and vocabulary are “wired” into their interactions (e.g., programs were built to use a fixed vocabulary

in messages). PACT could get away with this *ad hoc* approach to ontology building because its task was constrained, and only a handful of developers and tools were involved. Scaling up will require standard, reusable ontologies for generic concepts such as kinematics, dynamics, and the structure and behavior of electromechanical devices. It will also require a systematic process for developing and extending such ontologies, supported by CASE tools.

Knowledge Sharing Conventional approaches to integrating engineering tools depend on standardized data structures and a unified design model, both of which require substantial commitments from tool designers. PACT departs from such approaches in two fundamental ways. First, tool data and models are encapsulated rather than standardized and unified. Each tool is thus free to use the most appropriate internal representations and models for its task. Second, the encapsulating agents help tools communicate by translating their internal concepts into a shared *language* (grammar, vocabulary and meanings) of engineering. This shared language need only cover the intersection of tool interests, usually a small fraction of the contents of a full, shared design model. In principle, the language can evolve from a few core concepts[1, 6].

The agents communicate among themselves using KQML and KIF. Most KQML messages consist of queries and assertions in KIF (as well as basic control functions such as reset). Agents also use KIF expressions to inform each other of their interests and to request notification of informational changes that may affect them. Because queries and interests are expressed in a formal declarative language, the meanings of terms are not dependent on particular programs, and can therefore be shared among programs with different implementations and knowledge stores. Moreover, facilitators can use the content of messages to route them selectively to agents that have relevant knowledge or interests.

Several commercially available systems support distributed messaging across heterogeneous applications. These systems permit tools to subscribe to “interest” groups and then publish results which are communicated to all interested parties. However, such substrates do not enable arbitrary tools to exchange knowledge. PACT provides the framework for specifying shared content. It enables tools that should interoperate to do so without committing to common data formats. It also makes possible routing and notification based on message content, not just simple syntactic patterns.

Computer-Aided Engineering A key issue in concurrent engineering from a designer’s perspective is how to bridge the multitude of models required to support a complex design at various stages of the design process. Programs, like people, use many models. The challenge is to use the right model (i.e., abstraction, granularity) for each task and to communicate the results in an appropriate form to others with different needs and interests. PACT has yet to deal with such issues in a deep way. For example, the PACT distributed simulation demonstration employed a constant level of abstraction and granularity, each iteration around the loop corresponding to one sample of the digital servo system. Not only is this approach slow (the simulation ran over the network at something like 1/100 real time), but it is at an unnecessarily fine scale for purposes of verifying that the subsystems are behaving properly. A better approach would be to recognize that the participants in a simulation or verification exercise need to receive information from different sources and with different quanta of time, space, etc. depending on the circumstances. For example, during the early stages of controller design it might be best to bypass the encoder, amplifier, and digital circuit agents, and directly send sequences of commanded torques to the dynamics module and receive sequences of joint angles from it.

Summary

Through PACT, we are exploring a new methodology for cooperatively solving engineering problems based on knowledge sharing. Engineering tools and frameworks are encapsulated by agents that exchange information and services through an explicit shared model of the design. Conceptually, this shared model is centralized; in practice, it is distributed among the specialized internal models maintained by each tool or framework.

To create the illusion of a shared design model, all interactions between agents are mediated by facilitators. Each agent's facilitator is responsible for: (1) locating other agents on the network capable of providing requested information or services, (2) establishing a connection across (potentially) heterogeneous computing environments, and (3) managing the ensuing conversation. Information is passed among agents and facilitators in an interlingua (KIF) based on first order logic, with agents translating between the interlingua and their clients' internal representations. Knowledge sharing across disciplines is possible because of *a priori* ontological agreements among the agents about the meanings of terms. Agents and facilitators coordinate their activities using a communication and control language (currently, KQML).

In the PACT experiment, this agent-based interoperability architecture was implemented on a distributed messaging substrate that links platforms that support the TCP/IP transport protocol. Agents were used at two levels: first, to integrate individual tools in the Next-Cut, Designworld and NVisage frameworks; second to integrate the frameworks themselves. Although the initial implementation has limited robustness, functionality, and scale, its potential for concurrent engineering is clear.

More generally, we believe that agents communicating on a knowledge level are the right way to compose large, complex systems out of existing software modules. Instead of literally integrating code, modules can be encapsulated by agents and then invoked remotely as network services when needed. Such an approach is clearly advantageous in situations where installing the software locally would require expensive system reconfiguration or where experts are required to run and maintain the code. Many engineering software packages have these characteristics, discouraging occasional users and small organizations from exploiting them. With PACT, such problems can be overcome by creating a corporate-wide CAD framework that links software services run and maintained by specialists.

PACT is an ongoing collaboration. Current activities are aimed at alleviating shortcomings in the initial demonstration and expanding PACT into a broadly-based engineering infrastructure. First, the prototype software currently handling low-level message passing between agents must be upgraded to improve reliability, scalability and ease of use. The next version will likely build on a commercial substrate such as the OMG/CORBA [7] that can support multicast protocols in environments containing thousands of agents. Second, the simulation and analysis services currently provided in PACT will be transformed into generic engineering services (e.g., dynamics simulation, logic simulation) with published interfaces and ontologies, and made available on the Internet, 24 hours a day. Additional services provided by remote Internet sites will then be brought online. Initial experiments are already underway to make rapid prototyping facilities at Carnegie-Mellon University and the University of Utah available as PACT services. Transforming such existing resources into network services involves, among other things, installing agent wrappers that enable them to communicate via KQML and KIF. We are distributing PACT tool kits to facilitate such integration. Finally, the skeletal ontology and limited interactions that characterized the original planar manipulator scenario will be relaxed to support a broader range of electromechanical devices and concurrent engineering tasks. For example, any PACT agent should be able to modify its subsystem at any time, selectively triggering re-simulations by any agents whose subsystems are

affected by that change. These re-simulations should be performed at the highest abstraction and granularity sufficient to assess the consequences. Each agent should watch for violations of design constraints and notify other agents who have registered an interest in them.

Acknowledgements

The experiments would have never gotten off the ground without the talents and determination of the PACT programming core, consisting of Greg Olsen, Brian Livezey, Jim McGuire, Sampath Srinivas, Amr Assal, Narinder Singh, and Vishal Sikka. In addition, several others contributed significant ideas and code, including Pierre Huyn, Bruce Hitson, Rich Pelavin, Randy Stiles, and Reed Letsinger. The individual systems have a cast of inspired researchers that is too long to list here. Finally, PACT stands on the shoulders of ongoing knowledge representation standardization efforts, and the DARPA-sponsorship that makes our efforts possible.

References

- [1] R. Neches, R. E. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. R. Swartout, "Enabling Technology for Knowledge Sharing," *AI Magazine* **12**(3), 16–36, 1991.
- [2] T. Finin, D. McKay, and R. Fritzson, "An Overview of KQML: A Knowledge Query and Manipulation Language," *Technical Report*, Computer Science Department, University of Maryland, 1992.
- [3] J. A. Fulton, "The Semantic Unification meta-model: technical approach," *Standards Working Document ISO TC184/SC4/* WG3 N 81 (P 0)*, IGES/PDES Organization, Dictionary/Methodology Committee, 1991. Contact James Fulton, Boeing Computer Services, P. O. Box 24346, MS 7L-64, Seattle, WA 98124-0346.
- [4] M. R. Genesereth, R. E. Fikes, et al., "Knowledge Interchange Format, Version 3.0 Reference Manual," *Technical Report Logic-92-1*, Computer Science Department, Stanford University, 1992.
- [5] D. B. Lenat, R. V. Guha, K. Pittman, D. Pratt, and M. Shepherd, "Cyc: Toward Programs with Common Sense," *Communications of the ACM* **33**(8), 30–49, 1990.
- [6] T. R. Gruber, J. M. Tenenbaum, and J. C. Weber, "Toward a Knowledge Medium for Collaborative Product Development," *AI in Design '92*, J. Gero, Ed., Kluwer Academic Publishers, pp. 413–432, 1992.
- [7] "The Common Object Request Broker: Architecture and Specification," OMG Document# 91.12.1 Object Management Group, Framingham, MA., December 1991.

Glossary (sidebar)

agent — A computer program that communicates with external programs exclusively via a predefined protocol. An agent is capable of responding to all messages defined by the protocol, and uses the protocol to invoke the services of other agents. The PACT agents use the KQML protocol to send and receive messages represented in the KIF interchange format using a shared, domain-specific vocabulary.

facilitator — A program that coordinates the communication among agents. Facilitators provide a reliable network communication layer, route messages among agents based on the contents of the messages, and coordinate the control of multi-agent activities.

KIF — Knowledge Interchange Format: A standard notation and semantics for an extended form of first-order predicate calculus. KIF allows programs to make assertions and ask queries in a neutral format, independent of internal data structures.

KQML — Knowledge Query and Manipulation Language: A protocol for agents that specifies a set of domain-independent communication operations. For example, **assert** and **evaluate** allow agents to exchange information using some notation (e.g., KIF) and a specified vocabulary (i.e., an ontology).

ontology — A specification of a domain of discourse among agents, in the form of definitions of shared vocabulary (classes, relations, functions, and object constants). Together with a standard notation such as KIF, an ontology specifies a domain-specific language for agent interaction. The PACT ontology includes vocabulary for describing behavior in terms of time-varying parameters.

NVisage (sidebar)

NVisage is an engineering tool integration framework that supports “spreadsheet-style” design and development. When designers modify one aspect of the design, they immediately see the change reflected in other aspects. As with numerical spreadsheets, this facilitates what-if experimentation: designers interactively try various options, receiving immediate feedback on their decisions.

The NVisage framework is knowledge sharing technology that enables each engineering tool to encode and maintain its own separate model of a design, while inter-tool communication mechanisms maintain consistency among the models. Knowledge representation techniques have driven the development of shared languages and ontologies, and distributed knowledge-based systems techniques have driven the development of knowledge exchange protocols. The separate models and interchange mechanisms are specifically designed to include information about functionality (corresponding to plans for PDES level 4 [3]) as well as the information currently representable in CAD interchange formats.

Reference: J. C. Weber, B. K. Livezey, J. G. McGuire, R. N. Pelavin, “Spreadsheet-Like Design Through Knowledge-based Tool Integration,” *International Journal Of Expert Systems: Research and Applications*, JAI Press, 1992.

Device Modeling Environment (DME) (sidebar)

The Stanford University Knowledge Systems Laboratory (KSL) is developing an evolving prototype “designer’s associate” system called the Device Modeling Environment (DME). DME is focused on helping designers of electromechanical devices to experiment with alternative designs at all stages of the design process by providing rapid feedback about the implications of design decisions, and to document a design for use in other engineering steps such as diagnosis and redesign.

DME operates on multiple engineering knowledge bases that include libraries of process and component models, representations of the principles of physics, and knowledge about modeling itself. Multiple models of devices at various levels of abstraction can be represented, and the relationships among device models are explicitly represented. Given a model, which can be quite abstract, a simulation module can make predictions about the device behavior under specified conditions. Explanation capabilities generate natural language text in an interactive medium to

communicate the results to human engineers. Model formulation tools permit engineers to create useful models appropriate for various information requirements.

Reference: Y. Iwasaki and C. Low, "Model Generation and Simulation of Device Behavior with Continuous and Discrete Changes," *Technical Report KSL-91-69*, Knowledge Systems Laboratory, Stanford University, 1991.

Designworld (sidebar)

In its current form, Designworld is an automated prototyping system for small scale electronic circuits built from standard parts (TTL chips and connectors on prototyping boards). The design for a product is entered into the system via a multi-media design workstation; the product is built by a dedicated robotic cell — in effect, a microfactory. If necessary, the product, once built, can be returned to the system for diagnosis and repair.

The Designworld system consists of 18 processes on 6 different machines (2 Macintoshes and 4 HP workstations). These processes perform various tasks including design solicitation, simulation, verification, diagnosis, test and measurement, layout, assembly planning, and assembly execution. Each of the 18 processes is implemented as a distinct agent that communicates with its peers via messages in ACL. Any one of these programs can be replaced by an ACL-equivalent program without changing the functionality of the system as a whole. Any agent can be moved to a different machine (with equivalent capabilities). Any agent can be deleted and the system will continue to run correctly, albeit with reduced functionality.

References: M. R. Genesereth, "Designworld," *Proceedings of IEEE Conference on Robotics and Automation*, 1991. Also, see "Software Interoperation," this issue.

Next-Cut (sidebar)

Next-Cut is a prototype system for concurrent product and process design of mechanical assemblies. Next-Cut consists of several modules that surround representations of design artifacts, process plans and tooling. Next-Cut presently includes modules for feature-based design of components and assemblies, tolerance analysis, kinematic analysis and synthesis, geometric analysis and CNC process and fixture planning. A version of the Next-Cut framework has been applied to an industrial project on cable-harness design and fabrication.

The modules in Next-Cut communicate largely through the central representations, modifying them and extracting information from them. A notification mechanism alerts modules about changes that affect them. The representations of designs and plans in Next-Cut are hierarchical and include explicit dependencies both among different levels of detail and between artifacts and process steps. Individual components are described in terms of features, which are composed of geometric elements, location and form tolerances, reference frames, etc. To provide fast response during interactive design sessions, Next-Cut employs incremental planning and analysis methods in which previous results are reused, where possible. The primary mechanism behind these methods is the maintenance of dependency structures.

Reference: M. R. Cutkosky and J. M. Tenenbaum, "Toward a Framework for Concurrent Design," in *the International Journal of Systems Automation: Research and Applications*, Vol. 1, No. 3, pp. 239-261, 1992.