PACT: An Experiment in Integrating Concurrent Engineering Systems

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

William S. Mark, Lockheed Palo Alto Research Labs

Jay M. Tenenbaum and Jay C. Weber, Enterprise Integration Technologies

Large design projects can involve multiple sites, subsystems, and knowledge-representation schemes. Using this testbed system, four teams produced a distributed robotic-device simulation and synchronized on a design modification.

Several research groups are jointly developing the Palo Alto Collaborative Testbed (PACT), a concurrent engineering infrastructure 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.

Our approach has been to integrate existing multitool systems. These systems are themselves frameworks that were developed with no anticipation of subsequent integration. We take as a given that individual engineering groups prefer to use their own tool suites and integration environments. Certainly, they have significant investments in these self-contained systems. Nonetheless, projects that involve large segments of an enterprise or multiple enterprises must coordinate engineering activities that take place within individual frameworks. 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 (Product Data Exchange Using Step/Standard for the Exchange of Product Model Data).[1]

0018-9162/93/0100-0028$03.00 © 1993 IEEE

The PACT architecture is based on interacting *agents* (that is, programs that encapsulate engineering tools). The agent interaction in turn relies on three things: 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

- NVisage, a distributed knowledge-based integration environment for design tools developed by the Lockheed Information and Computing Sciences Group;
- DME (Device Modeling Environment), a model formulation and simulation environment from the Stanford Knowledge Systems Laboratory;
- Next-Cut, a mechanical design and process planning system from the Stanford Center for Design Research; and
- Designworld, a digital electronics design, simulation, assembly, and testing system from the Stanford Computer Science Department and Hewlett-Packard.

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 synchronize on a subsequent design modification.

In this article 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 Neches et al.,[2] Finin et al.,[3] and Genesereth et al.[4]

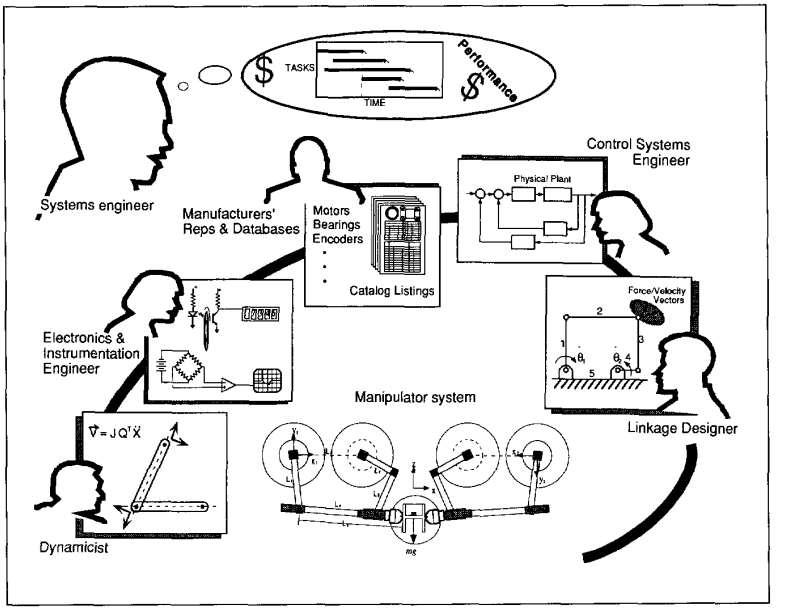

Figure 1. A multidisciplinary design team.

The distributed design problem

Although concurrent engineering is almost universally advocated today, it is hard to execute 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 linkage designer is primarily concerned with the geometry of the device 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. (Indeed, 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.)

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 nonlinearities in the amplifiers and motors will interest 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 expert users. Effective use requires knowing the conventions (for example, for representing spatial transformations and instantaneous velocities), the assumptions (for example, that components are rigid bodies), and other characteristics or limitations of the approach (for example, 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 tools' developers and frequent users. More fundamentally, idiosyncratic design tools contribute to the isolation of

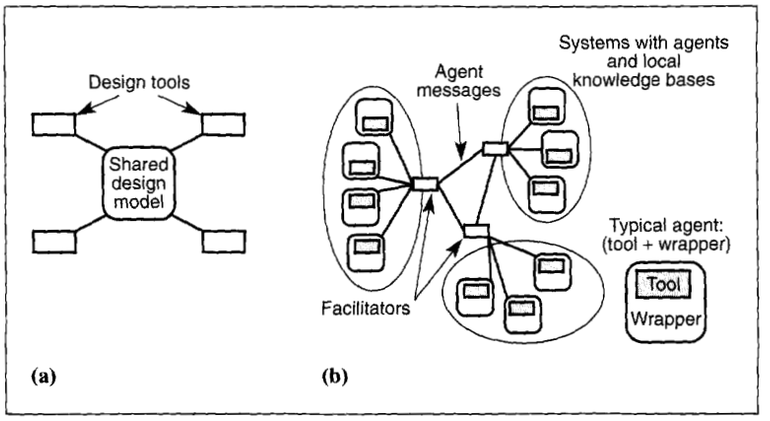

Figure 2. Interacting design tools require a shared design model (a). Scalability requires distribution of the model, with mechanisms for translating among different representations and coordinating interactions (b).

designers by preventing them from sharing their design models.

Integration via shared design models

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

Figure 2a shows one solution for providing a shared design model. 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 consider 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 come forward. A single shared database encompassing all 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 2b, 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.

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)[5] 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 (assert, retract, evaluate, etc.), and service location based on functionality are all left to ad hoc agreements among individual programs (that is, programmers). This situation limits 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:

(1) Messages are expressions in a common *agent communication language* that supports knowledge-base operations (for example, assertions and queries).

(2) Message contents of the knowledge-base operations are expressions in a common *knowledge-interchange format* that provides an implementation-independent encoding of information.

(3) 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.[2] One group has proposed an agent communication language called the Knowledge Query and Manipulation Language.[3] KQML specifies a relatively small set of *performatives* that categorize the services 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 can manipulate information of this sort.

The third effort concerns the development of engineering ontologies. This effort focuses on defining formal vocabularies for representing knowledge about engineering artifacts and processes. These vocabularies specify the assumptions underlying the common views of such knowledge. Given the application-specific nature of such ontologies, this activity is not intended to produce a single specification. Instead, it addresses various domains of importance to knowledge sharing, such as device behavior modeling. This effort complements 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 the message-content effort. We use facilitators to ease this burden. Each facilitator provides an interface between a local collection of agents and remote agents. The facilitator (1) provides a layer of reliable message passing, (2) routes outgoing messages to the appropriate destination(s), (3) translates incoming messages for consumption by its agent, and (4) initializes and monitors the execution of its agents.

Communication and coordination, therefore, occur between agents and facilitators and between facilitators, but not directly between agents. We call this arrangement of agents and facilitators a *federation architecture*. Messages from agents to facilitators are undirected (that is, they have content but no address). The facilitators route such messages to agents that can 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, they can do this interpretively (with messages going through the facilitator); in other cases, they can do it in one-shot fashion (with the facilitator setting up specialized links between individual agents and then stepping out of the picture).

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

In the demonstration system, each top-level PACT agent works through a facilitator to coordinate its interactions with other agents. Each PACT facilitator consists of two parts: a *connection associate* and an *agent manager*. A connection associate implements a layer of reliable message passing above the widely available TCP/IP transport protocol; it is nearly identical in implementation across facilitators. The connection associate supplies message strings for the agent manager. If the message is a control message from another facilitator, the agent manager interprets the message and reacts accordingly. If the message was from an agent (via a facilitator), the agent manager performs any necessary translations before forwarding the message to its agent.

PACT experiments

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

The subject of the PACT experiments was a robotic manipulator, a system

Glossary

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 it uses the protocol to invoke the services of other agents. 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 on the basis of message contents, and coordinate the control of multiagent 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 (for example, KIF) and a specified vocabulary (that is, 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.

that combined mechanics, electronics, and software, with extensive design documentation. Figure 3 shows the manipulator system schematically. The two fingertips of the manipulator mechanism are positioned by five-bar linkages connected directly to the shafts of four 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 control system. Depending on the application, various control laws are run at rates between 200 and 500 hertz. A digital-to-analog converter produces the controller-commanded torques as voltages that drive the amplifiers.

Figure 3 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 Stanford's Knowledge Systems Laboratory team, using their Device Modeling Environment (DME). The manipulator mechanism was the responsibility of Stanford's Center for Design Research and Enterprise Integration Technologies, 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.

All participants 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: first, an example of cooperative design refinement; second, a simulation of the manipulator, requiring exchange of data among all four subsystems; and third, a typical interaction in which a design change initiated by one team necessitated changes by another.

NVisage

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 intertool 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[1]) as well as information currently representable in CAD interchange formats.

Reference

1. J.A. Fulton, *The Semantic Unification Metamodel: Technical Approach*, Standards Working Document ISO TC184/SC4/* WG3 N 81 (P0), IGES/PDES Organization, Dictionary/Methodology Committee, 1991. Contact James Fulton, Boeing Computer Services, PO Box 24346, MS 7L-64, Seattle, WA 98124-0346.

Further reading

J.C. Weber et al., "Spreadsheet-Like Design Through Knowledge-Based Tool Integration," *Int'l J. Expert Systems: Research and Applications*, Vol. 5, No. 1, 1992.

Cooperative design refinement

In cooperative design refinement, each team begins with an initial subsystem design and produces a revised design that meets new specifications for accuracy, measurement methods, etc. Resolving design interactions requires considerable communication.

Figure 4 depicts some design interactions for the manipulator. Next-Cut's analysis uses a detailed rigid-body dynamics model, whereas NVisage needs a less detailed model for its controller. However, these models must be consistent. The interactions also include many 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

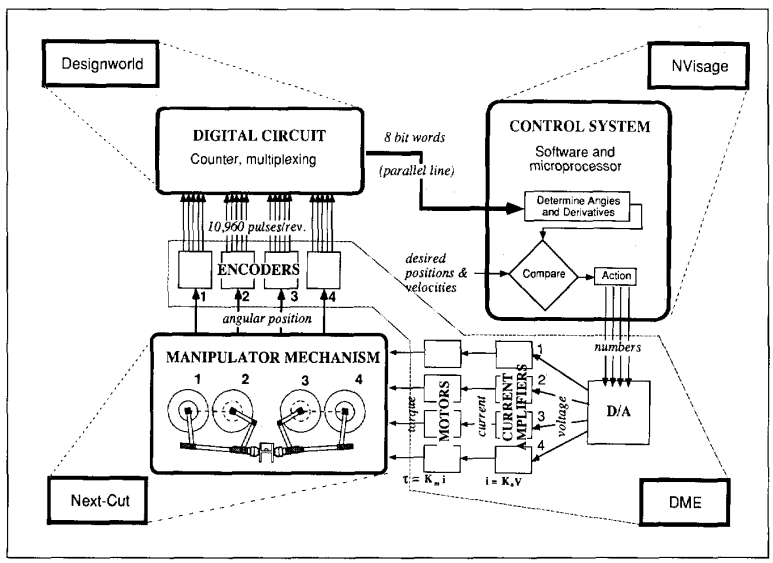

Figure 3. Manipulator system and division of responsibilities for the PACT experiment.

addressed only a few such interactions. The most interesting were those between NVisage and Next-Cut. For the experiment, the Lockheed design engineer selected a simple Cartesian position controller from an NVisage database of skeletal controller designs. To design the controller, NVisage needed a kinematic mapping from joint space to Cartesian workspace. One of the modules within NVisage therefore sent expressions of the form

```
(interested-in nvisage '(ASSERT
  (closed-form '(pmx $q1 $q2) $f))).
```

This statement specifies that the NVisage agent requests a closed-form expression $f for determining the horizontal fingertip coordinate of the planar manipulator (pmx) in terms of the joint variables q1 and q2. The meanings of terms *closed-form* and *pmx* were specified in the common ontology for the PACT experiment.

Next-Cut picked up this request and responded with equations (for example, bindings for $f). NVisage received this information, compiled the equations into equivalent statements in executable code, and dynamically linked these into its component functionality module.

Device Modeling Environment

The Stanford University Knowledge Systems Laboratory is developing an evolving prototype "designer's associate" system called the Device Modeling Environment. DME focuses on helping electromechanical-device designers experiment with alternative designs by providing rapid feedback about the implications of design decisions. It also helps 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. It can represent multiple models of devices at various levels of abstraction. It also explicitly represents the relationships among device models. 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.

Further reading

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

Stanford's Center for Design Research (CDR) team did not have 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, the two systems had to agree on their definitions of

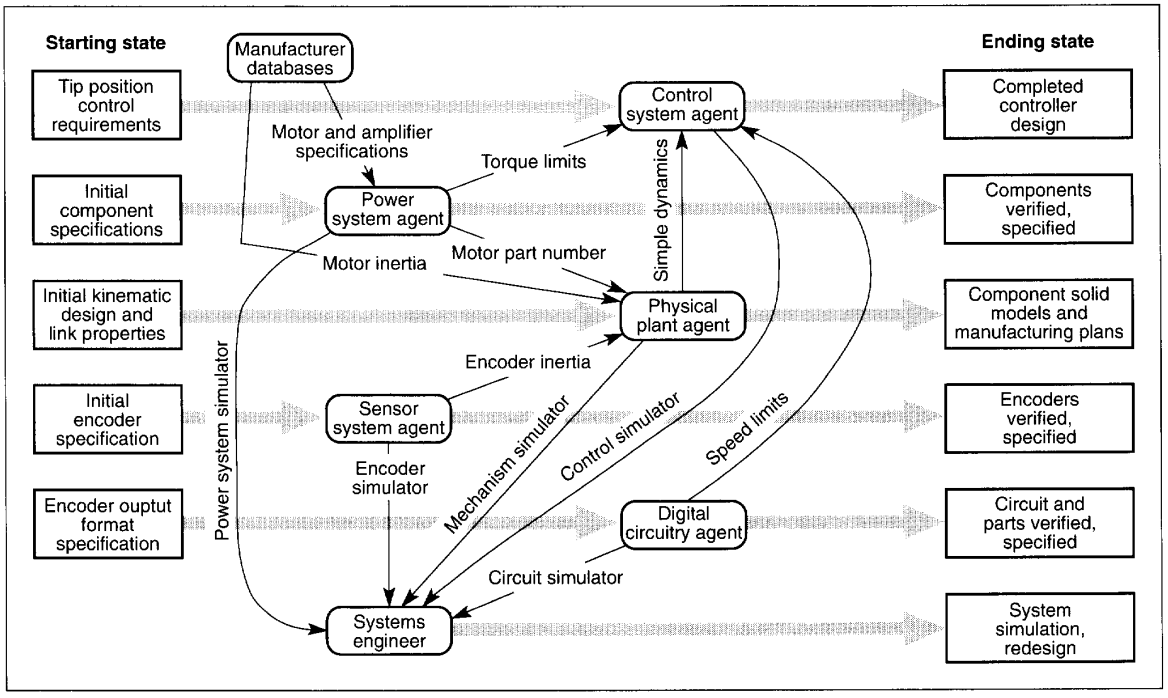

Figure 4. Interactions among aspects during manipulator design refinement.

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 initiated a simulation of the entire device to test their controller design. The simulation was distributed over the PACT network, with the four systems simulating their respective components and communicating their results to the others. As Figure 2 shows, the simulation can be effected through a simple loop, where each system sends commands or data to the next. The simulation proceeded around the loop until DME, which modeled the motors and power system, signaled that a motor had overheated due to overloading. The simulation thus indicated the need for a redesign.

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 10,000 is reading 4,095. These messages triggered reevaluations of the subsystems in response to the state changes.

Although this part of the demonstration initiated the experiments in interoperation, the system designers negotiated extensively to decide on the control sequence and data formats. In that sense, systems for the distributed simulation were integrated 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 used DME to replace the originally selected motors with a larger model and to notify the other subsystems of the change. The change notice was broadcast on the network as an ASSERT message, stating the part number of the new motor.

Next-Cut was interested in this change because the motor forms part of the manipulator-arm linkage assembly, and changing the motor can affect the mating features on the links. Next-Cut therefore posted a request for the dimensions of the motor with the new part number. Designworld, which maintains a components catalog, handled that query. The Designworld agent sent back the needed dimensions, thereby allowing Next-Cut to adjust mating features on the links, along with the process plan for machining them.

Designworld

In its current form, Designworld is an automated prototyping system for small-scale electronic circuits built from standard parts (such as TTL chips and connectors on prototyping boards). The design for a product enters the system via a multimedia design workstation. A dedicated robotic cell — essentially a microfactory — builds the product. If necessary, the product can be returned to the system for diagnosis and repair.

The Designworld system consists of 18 processes on six different machines (two Macintoshes and four HP workstations). These processes perform various tasks including design solicitation, simulation, verification, dignosis, 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 agent communication language (ACL) messages. 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.

Further reading

M.R. Genesereth, "Designworld," *Proc. IEEE Conf. Robotics and Automation*, IEEE CS Press, Los Alamitos, Calif., Order No. 2163-02, 1991, pp. 2,785-2,788. Also, see "Automated Concurrent Engineering in Designworld," this issue, pp. 74-76.

Lessons learned

The introduction characterized PACT as a testbed for cooperative research, knowledge sharing, and computer-aided engineering. We summarize here what we 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. Participants must agree about natural-world concepts such as position and time, shape and behavior, sensors and motors. Each concept requires agreement 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 composing PACT used various coordinate systems and several distinct representations of time (for example, discrete events, points in continuous time, intervals of continuous time, and piecewise approximations). These representations were chosen for valid task- and context-dependent reasons. We cannot simply replace them by one standard product model that has a single representation of time. Instead, we must get the agents involved in any transaction to 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 e-mail messages among developers of the interacting tools. The PACT tools can interact coherently because the commitments to common concepts and vocabulary are "wired" into their interactions (for example, 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 it involved only a handful of developers and tools. 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 engineering *language* (that is, a system of grammar, vocabulary, and meanings). 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.[2,6]

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.

What PACT actually demonstrates is a mechanism for distributing reasoning.

Several commercially available systems support distributed messaging across heterogeneous applications. These systems permit tools to subscribe to "interest" groups and receive published results relevant to that group. However, such systems 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 for each task (that is, the right abstraction and granularity) 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

Next-Cut

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 that reuse previous results where possible. The primary mechanism behind these methods is the maintenance of dependency structures.

Further reading

M.R. Cutkosky and J.M. Tenenbaum, "Toward a Framework for Concurrent Design," *Int'l J. Systems Automation: Research and Applications*, Vol. 1, No. 3, pp. 239-261, 1992.

only is this approach slow (the simulation ran over the network at something like 1/100 real time), but it also is at an unnecessarily fine scale for purposes of verifying that the subsystems behave 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 to send sequences of commanded torques directly to the dynamics module and receive sequences of joint angles directly from it.

Future directions

Through PACT, we are exploring a new methodology for cooperatively solving engineering problems on the basis of knowledge sharing. PACT encapsulates engineering tools and frameworks by using 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 that each tool or framework maintains.

To create the illusion of a shared design model, facilitators mediate all interactions between agents. 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 by 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.

Instead of literally integrating code, users can encapsulate modules.

Agents were used at two levels: first, to integrate individual tools in the NextCut, 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 using agents to communicate on a knowledge level is the right way to compose large, complex systems out of existing software modules. Instead of literally integrating code, users can encapsulate modules in agents and then invoke them 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, we must upgrade the prototype software currently handling low-level message passing between agents to improve reliability, scalability, and ease of use. The next version will likely build on a commercial substrate such as the OMG/CORBA[5] 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 (for example, dynamics simulation and 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 on line. Initial experiments are already under way 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 toolkits 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 resimulations by any agents whose subsystems are affected by that change. These resimulations 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. ■

Acknowledgments

The experiments would never have 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. Each individual system has a cast of inspired researchers that is too long to list here.

Finally, PACT stands on the shoulders of ongoing knowledge-representation standarization efforts and the DARPA sponsorship that makes our efforts possible. The work described here was partially supported by DARPA Prime Contract DAAA15-91-C-0104 (monitored by the US Army Ballistic Research Laboratory), by the support of Hewlett-Packard Laboratories, by the Office of Naval Research Contract ONR N00014-92-J-1833, and by NASA Grants NAG -2-581 and NCC -2-537.

References

1. J.A. Fulton, "The Semantic Unification Metamodel: Technical Approach," Standards Working Document ISO TC184/

SC4/* WG3 N 81 (P0), IGES/PDES Organization, Dictionary/Methodology Committee, 1991. Contact James Fulton, Boeing Computer Services, PO Box 24346, MS 7L-64, Seattle, WA 98124-0346.

2. R. Neches et al., "Enabling Technology for Knowledge Sharing," *AI Magazine*, Vol. 12, No. 3, 1991, pp. 16-36.

3. T. Finin et al., "Specification of the KQML Agent-Communication Language," Tech. Report EIT TR 92-04, Enterprise Integration Technologies, Palo Alto, Calif., 1992.

4. "Knowledge Interchange Format, Version 3.0 Reference Manual," M.R. Genesereth and R.E. Fikes, eds., Tech. Report Logic-92-1, Computer Science Dept., Stanford Univ., Palo Alto, Calif., 1992.

5. "The Common Object Request Broker: Architecture and Specification," OMG Document #91.12.1, Object Management Group, Framingham, Mass., Dec. 1991.

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, Norwell, Mass., 1992, pp. 413-432.

Mark R. Cutkosky is an associate professor in the design division of mechanical engineering at Stanford University. He is the principal investigator of a project on concurrent product and process design at Stanford's Center for Design Research. In addition to concurrent engineering, his research interests include dextrous manipulation and tactile sensing with robotic hands.

Cutkosky has a PhD in mechnical engineering. He is a member of ASME, SME, and Sigma Xi.

Robert S. Engelmore is a senior research scientist and executive director of the Heuristic Programming Project in the Computer Science Department at Stanford University. His primary professional interest is the application of artificial intelligence to scientific and industrial domains.

Engelmore received his BS, MS, and PhD degrees in physics from Carnegie Mellon University. He is editor emeritus and past editor-in-chief of *AI Magazine*. He is a fellow of AAAI and a member of the IEEE Computer Society.

Richard E. Fikes is a research professor in the Computer Science Department at Stanford University. He is also codirector of the Heuristic Programming Project in Stanford's Knowledge Systems Laboratory. He currently leads projects to develop multiuse representations of engineering knowledge, reasoning methods for performing core engineering tasks, and knowledge interchange technology.

Fikes received his PhD in computer science from Carnegie Mellon University. He is a fellow of AAAI.

Michael R. Genesereth is an associate professor of computer science at Stanford University. His current research includes topics in logic, automated reasoning, the theory of individual rationality, and the theory of communication and cooperation, with applications in engineering and robotics.

Genesereth received his ScB from MIT in physics and his PhD from Harvard University in applied mathematics.

Thomas R. Gruber is a research scientist at the Knowledge Systems Laboratory, Stanford University. His current research interests include design-knowledge capture and reuse, machine-generated explanation, computer-mediated human communication, and the development of shared ontologies for representing engineering knowledge.

Gruber received his PhD in computer science from the University of Massachusetts. He serves on the editorial boards of *Knowledge Acquisition* and *IEEE Expert*.

William S. Mark is the chief scientist of information and computing sciences at Lockheed Palo Alto Research Laboratories. He has been involved in artificial intelligence and advanced software research and applications for the past 15 years and is the author of numerous publications in these areas.

Mark received his PhD from MIT in computer science, specializing in artificial intelligence.

Jay M. Tenenbaum is president and founder of Enterprise Integration Technologies. He is also a consulting professor of computer science at Stanford University.

Tenenbaum holds BS and MS degrees in electrical engineering from MIT and a PhD in electrical engineering and computer science from Stanford University.

Jay C. Weber is a project leader at Enterprise Integration Technologies. His current projects are in distributed-agent technology and multimedia work-flow software. He is also cochair of the DARPA Knowledge-Sharing Effort's KQML working group.

Weber holds a BS degree from the University of Maryland and MS and PhD degrees from the University of Rochester, all in computer science. He is a member of AAAI.

Readers can contact Mark Cutkosky at the Dept. of Mechanical Engineering, Stanford University, Stanford, CA 94305.