Automated Concurrent Engineering in Designworld

Pierre Nam Huyn *Stanford Science Center / Hewlett-Packard Labs*

Michael Genesereth *Logic Group / Computer Science Department / Stanford University*

Reed Letsinger *Stanford Science Center / Hewlett-Packard Labs*

1.0 Designworld: an automated engineering system

Designworld [1] is an automated engineering system for digital circuits built from standard parts (TTL chips and prototyping boards). The distinguishing feature of the system is that it provides integrated support for various phases in the life cycle of a product – from design, through manufacture, to maintenance. The design for a product is entered via a multi-media design workstation; the product is built automatically by a dedicated robotic cell; and, if necessary, the product, once built, can be returned to the system for automatic diagnosis and repair.

The concept of Designworld is similar to that of desktop publishing. The user of a desktop publishing system typically interacts with a "what-you-see-is-what-you-get" editor to create the desired document on the computer screen and then uses a laser printer to produce the hardcopy. In Designworld, the robot assembly cell is the printer, and the cad workstation is the editor. The user creates his design on the screen, and the robotic cell "prints" the hardware.

2.0 The integration problem

Designworld was created from of a set of pre-existing tools that were not designed to operate together. The first problem we encountered in integrating these tools was *software interoperability* – not only did the tools use different communication formats and protocols, they also represented information in idiosyncratic ways. Another problem was the achievement of *automated concurrent engineering* – the exchange of incomplete design information and the timely propagation of design constraints. Note that whereas the phrase "concurrent engineering" usually applies to human work, here we are discussing concurrent engineering among programs.

3.0 Agent-based framework

These requirements led to the development of a novel software interoperability framework [2], termed "agent-based", that addresses many important issues critical to effective concurrent engineering. The idea in this approach is for programmers to write their programs as individual *software agents*. Communication is assured through the use of a standard *agent communication language*. Agent interaction is assisted through the services of a coordinated set of operating system programs, called *facilitator agents* or, more simply, *facilitators*.

A software agent is a program capable of communicating with other programs in the *agent communication language* (ACL) described below. Moreover, an agent is self-documenting, i.e., it should be capable of providing its own specification – definition of concepts used, interests, capabilities and I/O behavior – in machine processable form (i.e. ACL). These properties are used by agents to achieve collaboration.

The key feature of ACL is its expressiveness. It provides for the communication of constraints, negations, disjunctions, rules and quantified expressions in a variety of communication modes such as assertions, questions and commands (see Figure 1). With the application of AI technology to practical problems, there is an increasing number of programs able to manipulate information of this sort and the existence of a standard communication language with this degree of expressiveness makes it possible for these programs to interoperate. The design of ACL follows the recommendations of the various committees involved in the Darpa Knowledge Sharing Initiative [3].

Agents and facilitators are linked together in what is often called a *federation architecture*. Figure 2 illustrates this architecture for the simple case in which there are just three machines, one with three agents and two with two agents each. As suggested by the diagram, agents do not communicate directly with each other. Instead, they communicate only with their local facilitators, and facilitators communicate with each other. In effect, the agents form a "federation" in which they surrender their communication autonomy to the facilitators; hence, the name of the architecture.

Figure 1.A message in ACL

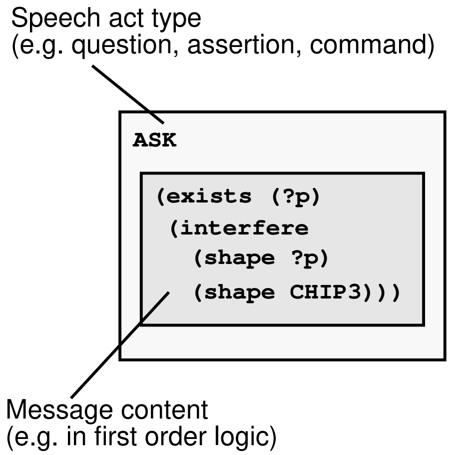
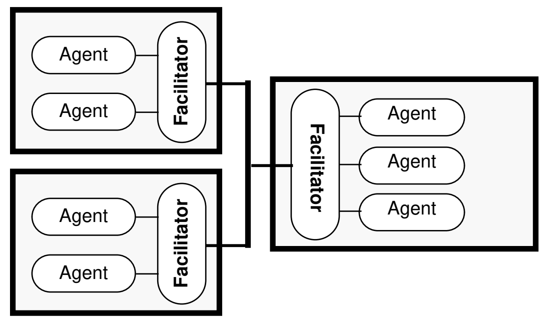


Figure 2.Federation architecture



Messages from agents to facilitators are undirected, i.e. they have content but no addresses. 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, they can decompose problems into subproblems, and they can 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.)

The agent-based approach to software interoperation rests on the key ideas of communication standards and federation architecture. Neither of these ideas is new. Communication standards have certainly been around for a while, and they are routinely used to achieve software interoperation within specific application areas. For example, electronic mail programs from different vendors manage to interoperate through the use of mail standards like SMTP. Disparate graphics programs interoperate using standard formats like GIF. The idea of a facilitator is not novel either. The event and directory services in extended DCE (Open Software Foundation's Distributed Computing Environment), the *Publish* and *Subscribe* mechanism on the Apple Macintosh, the Object Request Broker in OMG (Object Management Group) are just a few instantiations of the idea.

What is new here is the application of agent technology to help in pushing these ideas to their limits. The novelty stems from (a) the use of ACL to encode general knowledge about application areas (e.g. definitions of concepts) and about programs (i.e. their specifications) and (b) the application of automated reasoning and automatic programming technology in the implementation of facilitators that can use this knowledge to match up clients and servers in much more interesting ways than in the past.

4.0 Implementation of Designworld

The implementation of Designworld typifies the agent-based software engineering approach (see Figure 3). Pre-existing tools undergo an "agentification" process to make them comply with the interface and behavior requirements of an agent: a vocabulary is defined if not already so; the agent's interests and message handlers are specified; message handlers are obtained by wrapping existing code with the agent communication code that intertranslates the tool's native representation and the agent communication language. For instance, the Designworld geometry agent's interests include questions about shape interferences expressed in ACL. The associated message handler translates the questions into procedures and data structures acceptable to the geometry engine and retranslates the results into replies in ACL. Once all agents have been agentified, one final piece of integration may be needed: the augmentation of the facilitator dictionary with translation rules when differences in representation arise between agents. A typical example is the representation of location: a more concise single variable representation vs. a clearer and more expanded multiple variable representation; relative vs. absolute positioning; different choices of origin and axis orientation; idiosyncratic representation such as "the third column on the left half of this board"; different units of measure used; 2D vs. 3D.

Designworld in its current implementation consists of sixteen agents running on four machines (see Figure 3). Using the agent-based framework, the Designworld system was completed within a few months.

5.0 Concurrent Engineering in Designworld

Unlike many engineering systems, information flowing between Designworld agents may be incomplete: instead of having to wait until a design is completely specified before sending it to the "next stage", partial design decisions may be disseminated soon after being made. Two kinds of incompleteness can be distinguished here. First, a complete description of some part of the whole design may be specified. Examples of this arise when a design can be broken down into independent subparts that can be worked on simultaneously.

Second, a complete description may not be available. Instead, constraints are used to restrict the space of possible solutions. This situation can arise because the information is actually not available or because one does not want to make premature design decisions that might lead to dead ends.

Figure 3.Designworld architecture

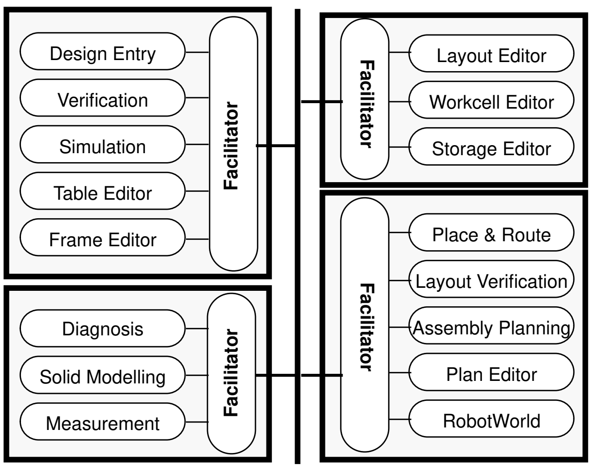
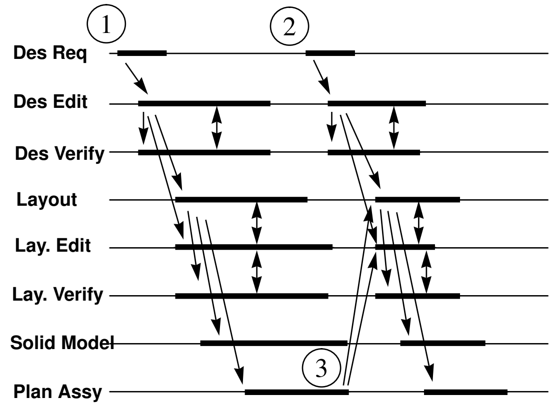


Figure 4.Agents activity chart



As an example, consider the simple digital circuit design scenario depicted in Figure 4. This figure shows parallel activities among some of the Designworld agents during the early phases of a hypothetical product life cycle. Each line, labelled on its left with an agent's name, represents the agent's activities through time. Thick segments represent periods of heavy activities. Lines with single arrows represent causal relationships, while those with double arrows represent tight interactions.

(1) A new digital circuit design is started. Design requirements are extracted from the end user and translated into design constraints. Soon after this, manual design entry is started; and the design verification agent, running in the background, compares the design with those requirements. Any discrepancy detected is promptly reported to the design editor agent. Without waiting until the design is complete, the automatic layout agent starts working on placement and routing for those design fragments that are sufficiently specified. The layout engineer may choose to participate in the layout design in a concurrent fashion. Like the design verification agent, the layout verification agent ensures that all design decisions are consistent with the netlist specification. While the layout is taking shape, the geometry agent incrementally forms a 3D model of the lay-

out. With the assistance of the geometry agent, the assembly planning agent checks for spatial interferences, makes choices of insertion devices and derives assembly ordering constraints. It is possible that some of those decisions have to be undone later, as the density of the layout design increases.

(2) The end user modifies the design requirements. This translates into some design constraints being retracted and others being added. In order to bring the design back to consistency with the new requirements, design changes are made and translated into changes in layout constraints. Note that a local change might trigger a chain reaction in which constraints are propagated across the system.

(3) In the meantime, the assembly planning agent has detected a fragment of the layout design which cannot be assembled. New layout constraints are generated and fed back to the layout agent. Note that constraints can be propagated "forward" as well as "backward" and through arbitrary layers of agents. Indeed it is possible for constraints to be propagated all the way back to the end user, which means that as specified, the artifact cannot be built.

This scenario, despite its simplicity, already illustrates a few points. In Designworld, a high degree of concurrency between different engineering activities is possible. Tools do not have to be explicitly invoked, as they react to information being disseminated. Engineering constraints are propagated in order to maintain consistency between difference phases of the lifecycle.

References

[1] M. R. Genesereth, "Designworld," *Proceedings of IEEE Conference on Robotics and Automation,* 1991.

[2] M. R. Genesereth, "An Agent-Based Framework for Software Interoperability," Technical Report, *Logic-92-02*, Dept. of Computer Science, Stanford University, 1992.

[3] 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.