Testing of Interaction-Driven Manufacturing Systems

KC Morris
David Flater
Don Libes
Al Jones

Manufacturing Systems Integration Division
National Institute of Standards and Technology
Department of Commerce

NISTIR 6260

December 1998

Testing of Interaction-Driven Manufacturing Systems

Table of Contents

Abstract

As automation of manufacturing processes becomes more widespread, manufacturing systems are becoming more and more software dependent. At the same time, the software used is becoming more and more modularized allowing for the creation of customized systems consisting in large part of pre-existing components. This combination of factors leads to considerable flexibility for manufacturing systems, but not without a cost. The reliability of those systems is uncertain due to the lack of experience in how to test such systems. To address this need the National Institute of Standards and Technology's Manufacturing Systems Integration Division has undertaken a study of how to test "interaction-driven manufacturing systems." This paper contains an analysis of both the problem space and the solution space for testing these systems and is the first outcome of that study.

Introduction

Manufacturing systems are inherently distributed and heterogeneous. Products are designed and manufactured by a range of people with different skills using a variety of systems specialized for different functions. Not long ago, the product development process (spanning the product life-cycle from design to distribution) was characterized by islands of automation as automation was applied to the various computation-intensive tasks in a product's development. More recently, significant effort has gone into bridging the islands of automation and streamlining the development process. In many industries this streamlining has become highly tuned so as to minimize the actual time from raw material to point-of-sale. Other industries are still working to reach this minimum.

In such an environment, system reliability is of utmost importance. One weak link can interrupt the entire chain of events and delay the delivery of the product. Manufacturers have traditionally planned down time for preventive maintenance of machines and had back up machines for those in need of repair, but in this new environment, the physical machines are a lesser risk than the software that guides the processes. Techniques for maintaining and repairing software are themselves not as reliable as for their hardware counterparts. Furthermore, software is not as

interchangeable as hardware in that it may maintain knowledge about the state of a system which is in constant change, as well as historical knowledge about the system.

Maintenance and repair are one aspect of a bigger problem which involves the usability and reliability of an individual software component in a larger system. How does one diagnose failure of a software component? How is a component impacted by the system of which it is a part and how does it impact the system? How much tolerance is there for component performance? What is the impact of system failure? Can another component be swapped in when one fails? Can the interaction between the components result in an unreliable system state? These questions are all essential to implementing a reliable product development process. To answer these questions, one needs methods and a framework in which to test software components and the systems in which they operate.

The TIMS Project

To this end, NIST has initiated an effort to develop competence in the testing of software systems. The Manufacturing Systems Integration Division (MSID) will focus more specifically on the testing of *interaction-driven manufacturing systems* (TIMS). Interaction-driven manufacturing systems are those composed of multiple software components in which the interactions between those components are automated. Automation is typically achieved through the definition of program interfaces, both standard and proprietary, which allow the components to directly interact without the need for human intervention at every step along the way.[1]

Today's manufacturing software, and software systems in general, are built differently than they were a decade ago. While NIST has established competence in the traditional sequence of unit, integration, and system testing,[2] what we once called "systems" are now single components in larger, integrated, distributed systems.

1. Such interfaces are often referred to as API's, or Application Program Interfaces, where the "application" is anything that happens to make use of the software component. API's include the specification of function calls or commands as well as context-specific rules for the use of those commands. Together, these form the specifications governing the interactions of the system.
2. *NBS Special Publication 500-98, Planning for Software Validation, Verification, and Testing*, November 1982.

The best of practice techniques[3,4] have become dated, and the best minds in the business are scrambling to come to grips with the reality of systems that, using traditional terminology, would have to be called "large and complex distributed systems of systems."[5] We, too, must make this step and revise our techniques.

Unlike manufacturing systems of the past, which were only designed to optimize throughput, today's systems are being designed for flexibility while not compromising throughput. Instead of building systems from scratch, manufacturers are seeking to be able to integrate off-the-shelf software components into a coherent system with minimum expense on custom programming. To do this, they need to be able to upgrade or replace individual components without breaking the system. In reaction to this need, manufacturers are now paying more attention to *open systems* and standards.[6,7]

Open systems are based on the premise that if interfaces are clearly and publicly defined (i.e., standard) then systems can be modified incrementally to include newer, better functionality as it emerges in the form of improved system components. A host of software standards, both formal and *de facto*, have evolved to support these types of architectures.

Manufacturing Interface Specifications[8]

At the initiation of the TIMS project, two specifications had been identified as suitable case studies for understanding the needs of testing interaction-driven manufacturing systems:

---

3. Boris Beizer, *Software Testing Techniques (second edition)*. Van Nostrand Reinhold, 1990.
4. Boris Beizer, *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, 1984.
5. Genevieve Houston-Ludlam, in "Call for Papers, Testing Computer Software Conference '99," <URL:news:comp.software.testing>, September 1, 1998.
6. Richard Kuhn, William Majurski, Wayne McCoy, Fritz Schulz, "Open Systems Software Standards in Concurrent Engineering," *Advances in Control and Dynamic Systems*, vol. 62, Academic Press, Inc., 1994.
7. Jeanine Katzel, "Moving Down the Path to Open Systems," *Plant Engineering Online*, May, 1998, <URL:http://www.manufacturing.net/magazine/planteng/archive/1997/ple0901.97/098513.htm>.
8. Names of companies and products are used in order to adequately specify the information contained herein. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology.

- STEP's[9] Data Access Interface (SDAI)[10] and
- the emerging Process Specification Language (PSL).[11]

Subsequently a number of other manufacturing specifications have been identified as potential candidates for study:

- Manufacturing Message Specification (MMS),[12,13]
- the CIM (Computer-Integrated Manufacturing) Framework,[14]
- the Advanced Process Control (APC) Framework,[15]
- Object Management Group's[16] Product Data Management (OMG's PDM) Enablers,[17]
- STEP's Application Protocols (AP) using SDAI,
- CAM-I's Application Interface Specification (AIS),[18] and
- OLE (Object Linking and Embedding) for Design and Modeling.[19]

9. The Standard for The Exchange of Product Model Data (STEP) is a project of the International Organization for Standardization (ISO) Technical Committee on Industrial Automation Systems and Integration (TC184) Subcommittee on Industrial Data (SC4).
10. *ISO 10303 Industrial automation systems and integration—Product data representation and exchange—Part 22: Implementation methods: Standard data access interface*, International Organization for Standardization, Draft International Standard, 1998.
11. Process Specification Language home page. <URL:http://www.mel.nist.gov/psl/>.
12. MMS Information Server. <URL:http://litwww.epfl.ch/MMS/mms_main.htm>.
13. *ISO/IEC 9506-1 Industrial automation systems - Manufacturing Message Specification, Part 1: Service definition* and *ISO/IEC 9506-2 Industrial automation systems - Manufacturing Message Specification, Part 2: Protocol specification*, International Standard, International Organization for Standardization, 1990.
14. SEMATECH's CIM Framework Home Page, <URL:http://www.sematech.org/public/division/fi/cim/cim-home.htm>.
15. Advanced Process Control (APC) Framework Initiative 1.0 Specifications, 1997. <URL:http://www.sematech.org/public/docubase/abstract/3300aeng.htm>.
16. Object Management Group Home Page. <URL:http://www.omg.org/>.
17. *Revised Submission (including errata changes) — PDM Enablers — Joint Proposal to the OMG in Response to OMG Manufacturing Domain Task Force RFP 1*. <URL:http://www.omg.org/arch2/mfg/98-02-02.pdf>, 1998.

These specifications are intended to serve as standard definitions governing the software supporting a manufacturing process. The specifications fall into three categories:

- infrastructures used to connect the system,
- applications used in the system, and
- the operation of the system itself.

Manufacturing systems rely on software specifications of many sorts. Many of these specifications are generic in that they support many types of systems and are not restricted to manufacturing. A 1995 report from MSID surveys standards relevant to manufacturing systems integration.[20] The report identifies three broad classes of standards: computing technology, industrial practices, and manufacturing equipment. The category of industrial practices contains the specifications which are unique to interaction-driven manufacturing systems and which will be the focus for the TIMS project. While there are many infrastructural software standards (the computing technology category) relevant to manufacturing, the manufacturing specific specifications in this category are MMS, potentially PSL, and SDAI.

A specialized subsystem, such as a database, geometry engine, or machine tool, provides a particular function needed by one or more applications in the system. The subsystems typically provide an application program with interfaces to enable their integration into the larger system. While many, if not most, manufacturing systems provide application program interfaces, only a few of these are formally recognized as standards through an open consensus-based process. The standard interfaces include CAM-I's AIS, OMG's PDM Enablers, and OLE for Design and Modeling. Other public specifications in this category are expected to emerge in the near future. OMG is working on interfaces in the areas of Manufacturing Exe-

18. *Application Interface Specification (AIS), Version 2.1: Volume I, 'Functional Specification'* and *Volume II, 'C Language Binding'*, Report R-94-PM-01, Consortium for Advanced Manufacturing International (CAM-I), Inc., Bedford, TX, USA (April 1994).

19. "What is OLE for Design and Modeling?" Design and Modeling Applications Council (DMAC), <URL:http://www.dmac.org/whatis/whatis.htm>. 1998.

20. Ed Barkmeyer, *et al*., *SIMA Background Study*, NISTIR 5662, September 1995.

cution Systems, Enterprise Resource Planning, and Machine Control. ISO has just recently approved SDAI as a Draft International Standard, which is expected to result in implementations of Application Protocols for STEP using an interaction-driven interface.

Finally, specifications of the systems themselves guide how components, whether they be infrastructural or specialized, are supposed to interact to support a final system. System specifications provide a framework for the appropriate sequencing of events in the system. Specifications of manufacturing systems include SEMATECH's CIM Framework and the APC Framework.

Scope of Testing for TIMS

Software testing has different meanings depending on its purpose. Its broadest purpose is to find problems in order to improve the robustness and reliability of a software product. It can be used for the sole purpose of improving products or it can be used as a means of evaluating products by distinguishing their differences.[21] We distinguish three purposes of testing and the roles of the participants:

- Testing for product release: this type of testing is focused on finding problems for the purpose of debugging a product which is to be distributed to outside users. Both white box and black box testing can and should be used for this purpose.
- Testing for product evaluation: this type of testing is done by an organization in the process of designing and implementing a system with specific functional and performance requirements.[22] In this case the testing of the software is performed under the auspice of the organization that will be using the software but is not responsible for the development of the software.

21. Rick Hower published a rather exhaustive listing of the various types of testing at <URL:http://www.charm.net/~dmg/qatest/qatfaq1.html#FAQ1_10>.

22. Kurt C. Wallnau, David Carney, Bill Pollak, "How COTS Software Affects the Design of COTS-Intensive Systems," *SEInteractive*, June 1998. <URL:http://interactive.sei.cmu.edu/Features/1998/June/COTS_Software/Cots_Software.htm>

- Testing for product certification: this type of testing supports the marketing of products. It is based on the premise that standards can be used to define software such that multiple vendors can provide software that can be used interchangeably by end-user systems. Testing against such standards, whether interface specifications, performance parameters, security protocols, or others, provides a measure of the software's quality against an independent benchmark. The verification that the software does support the standards gives the users confidence in the software.

As an independent organization, NIST is in a unique position to provide methods and tools which support all categories of testing and to address the latter category of testing specifically. The TIMS project will investigate what types of methods and tools will be needed to support testing of interaction-driven manufacturing systems and whether there is a role for NIST in certifying these types of systems. The anticipated results of the project are methods and tools which will benefit all those involved with software to support interaction-driven manufacturing systems.

The TIMS project examines testing from the perspective of systems integration. In this context the goal of testing is focused on

- whether components reliably support the interface specifications,
- whether the specifications are complete and consistent, and
- whether the system can be constructed using multiple instantiations of the specifications — in other words, using different vendors' products.

Much of the testing process can by executed by simulation of the system or parts of the system in concert with actual components.

The remainder of this paper discusses some of the issues surrounding the questions raised. It provides background material for the initiation of the project. We begin with a description of characteristics of software supporting manufacturing which is followed by an analysis of how to test interaction driven manufacturing systems. We discuss three categories of testing — component, system, and interoperability—that are understood within the new context of "large and complex distributed systems of systems." Each of these categories is described with respect to the manufacturing standards that fall within the category, the state of the practice in testing these types of interfaces (often by analogy to similar interfaces in the broader soft-

ware world), and finally a summary of the state of the art in testing technology as it applies to the category, to the extent that we know it.

Manufacturing Software Environment

"The first line of defense against these bugs is the design. The first bastion of that defense is that there be *a design for the overall software architecture. Failure to create an explicit software architecture is an unfortunate but common disease."*

—Boris Beizer

The first step in testing an interaction-driven manufacturing system is to identify the components of the system. These should be clear from the system architecture. The components in a typical manufacturing system will include several commercial off-the-shelf (COTS) products which may support open or proprietary interfaces. Components may be both infrastructural or specialized and range from machine control devices through scheduling, inventory, and planning systems. Other components in the system will be custom built applications that interface with and bridge the gaps between these vendor-supplied products. Finally, the human or end-user can be considered a component in the system as well since much of the system control is under user guidance. Simple component diagrams are often used to illustrate the system architecture. More formal Architecture Description Languages (ADL)[23,24,25] are emerging. More detailed and formalized system models can also be constructed using various aspects of the Unified Modeling Language (UML).[26]

The components in an architecture are connected by interfaces of various sorts. The TIMS project is concerned with the interfaces that can be automated in one

23. The Rapide™ Language, Stanford University, <URL:http://pavg.stanford.edu/rapide/>.

24. The *Acme* Architecture Description Language, Carnegie Mellon University, <URL:http://www.cs.cmu.edu/~acme/>.

25. SADL: A Structural Architecture Description Language, SRI Computer Science Laboratory, <URL:http://www.csl.sri.com/dsa/sadl-main.html>.

26. *Unified Modeling Language, version 1.1.* <URL:http://www.rational.com/uml/documentation.html>. September 1, 1997.

way or another. However, it is worth noting that a system architecture may contain other more static interfaces such as the exchange of data based on file formats. This is a very common type of interface in today's manufacturing systems, and a system architecture can be divided into subsystems where these static interfaces serve as borders. Components whose interfaces are all implemented with static data exchange can be tested as stand-alone systems.[27]

Interaction-driven Systems

To facilitate the discussion of appropriate test methods for manufacturing systems, we present an analysis of the style of interactions seen in such systems, to be followed by some background on the modeling of system behavior.

Just as data flow diagrams can be used to identify data sources and sinks, a new kind of diagram can be used to identify sources and sinks for *interactivity. Interactivity* is the activity of interacting, or the business of interaction. *Interaction* is more than just *reaction*; hence, interactivity is more than just the sending of messages. It represents a possible source of non-determinism in the system, which is a critical factor in the system's testability.

Flows of interactivity are used to show the sources of non-determinism and the ways that this non-determinism ripples through the system. By drawing an arrow representing *interactivity flow* in one of the following diagrams, we are saying that the component at the start of the arrow is perturbing the component at the end of the arrow in a partially or completely non-deterministic way. The specific "way" could be a data flow, a control flow, a message, an event, or something else -- it does not matter. These *interactivity flow diagrams* therefore differ from data flow

27. *ISO 10303 Industrial automation systems and integration—Product data representation and exchange—Part 31:Conformance testing methodology and framework: General concepts*, International Standard, International Organization for Standardization, 1994.

diagrams and similar diagrams because they contain information about the sources of non-determinism and the parts of the system that are affected by it.

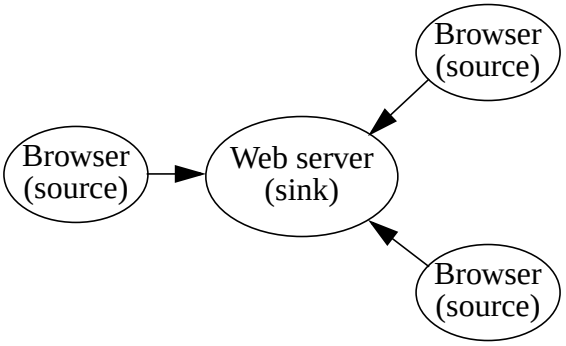

FIGURE 1. Simple interactivity diagram

In a classic client-server architecture such as the World Wide Web (modeled above), all components are either *sources* or *sinks* of interactivity. Because they do not have complex dependencies on other components, sources and sinks can be tested as stand-alone systems by replacing the components on the other side of the interactions with a simple test harness.

The non-determinism in this system is that the arrival times of requests from browsers and the contents of those requests are random as far as the web server is concerned. Because web servers are designed to be stateless, this would seem to be an insignificant observation. However, if we instead have a manufacturing system where orders for products and machine control commands are being entered through web interfaces, the interactivity coming from the web browsers may easily be sufficient to trigger timing-related failures in the system.

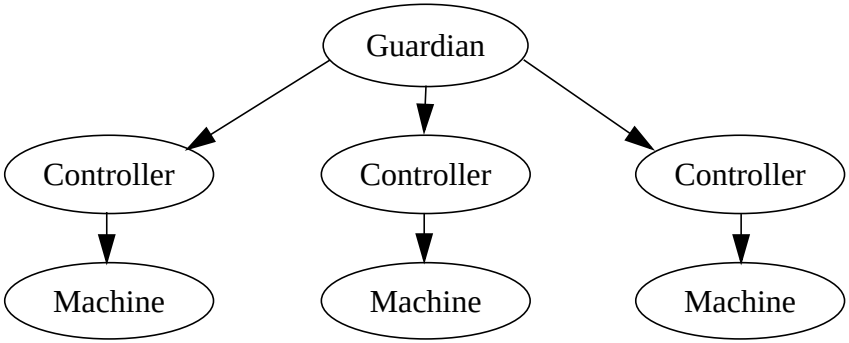

FIGURE 2. Hierarchical interactivity diagram

Components that act as both client and server create more difficulty for testing the system as a whole. However, some multi-tier systems are designed so that they can be decomposed and tested as independent subsystems. The open loop control hierarchy shown above is an example of such a system. The interactivity is acyclic, flowing downwards from the Guardian (a user interface component like a web browser), so the behavior of each subsystem is determined by the layer above. The test methods can therefore remain typical of those used by the software industry for large software projects.

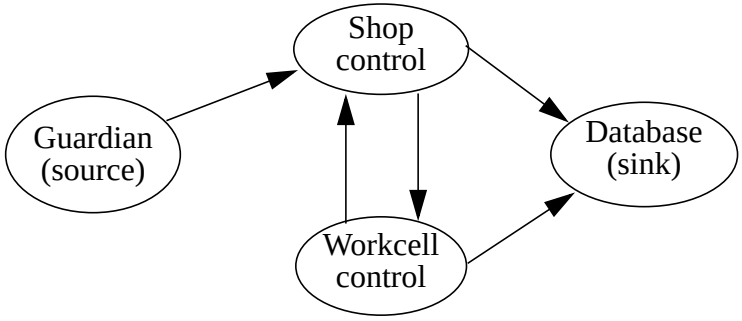

FIGURE 3. Cyclic interactivity diagram

In this example, the shop and workcell controllers each have their own thread of control and may initiate interactions at any time. When interactivity becomes cyclic as it has here, a new category of problems such as deadlocks, race conditions, and inconsistent world views is introduced. These problems have been explored extensively in network testing; special considerations for the context of interaction-driven systems will be described later in this document.

Systems having cyclic interactivity are inherently more difficult to test than those with acyclic interactivity because merely controlling the top-level sources of interactivity is no longer sufficient to remove non-determinism from the system — there may still be uncontrolled interactions at lower levels.

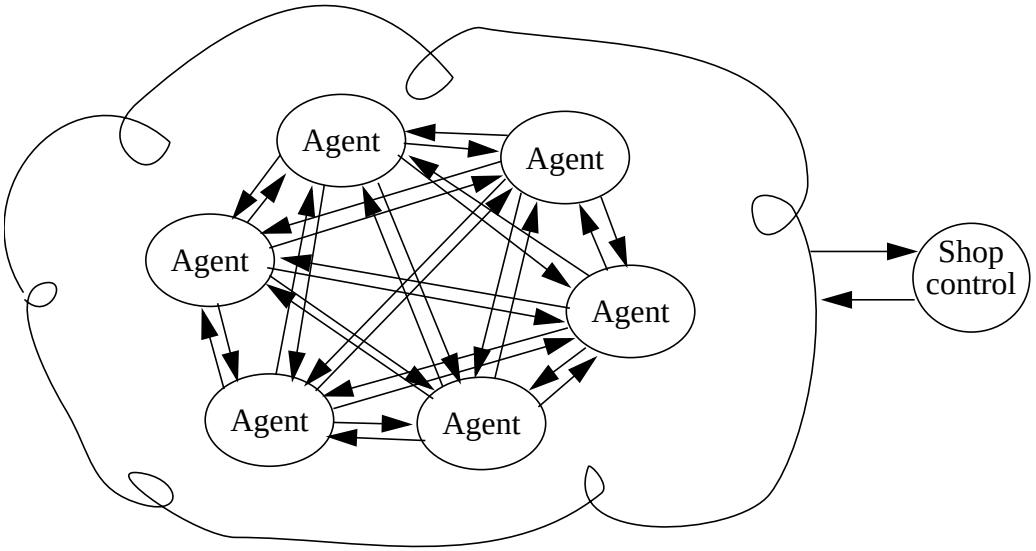

FIGURE 4. Chaotic interactivity diagram

Finally, there exist non-traditional architectures that use large numbers of competing agents interacting chaotically to produce an emergent behavior. These are very difficult to test because they are highly non-deterministic and the permissible state spaces of the systems are not well defined.

Modeling System Behavior

Formal techniques for modeling the behavior of manufacturing systems have been in use for many years. These techniques are equally applicable for hardware and software systems when the behavior of the software is fairly simple. However, as software components are integrated into ever larger, interaction-driven systems, modeling becomes increasingly problematic. This section summarizes a general approach for formal modeling of behaviors and common approaches for testing such systems and explains the roadblocks to using it on complex interaction-driven systems.

Representation of Manufacturing Systems

• State Space Representation for Continuous Time Systems

Typically, we describe continuous time systems at any time t with a state function, **x**(t). **x** can be a single variable or a vector of variables. To model the system's evolution, we define a state transition function, f (•). This function uses both the current state at time t, **x**(t), and the current control law, **u**(t), to predict the state at time t+δ. That is

$$\mathbf{x}(t+\delta) = f(\mathbf{x}(t), \mathbf{u}(t)) \qquad t \in [t_0, T]. \qquad (1)$$

The simplest and most widely used form for this function is linear. That is,

$$\mathbf{x}(t+\delta) = A\mathbf{x}(t) + B\mathbf{u}(t)) \qquad t \in [t_0, T].$$

The actual output from the system is called **y**(t), which is generated from an output function, **g**(t). This output function uses both **x**(t) and **u**(t),

$$\mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t)) \qquad t \in [t_0, T]. \qquad (2)$$

Optimal control theory addresses the formulation of the "optimal" control policy for given problem and performance criteria. For example, a robot end effector is located at a given position and orientation at time t_0. The problem is to have the

end effector at another location and orientation by time T. The performance criteria may be to minimize total energy consumption, minimize total time, etc. The optimal control policy would define the trajectory that is to be followed by the end effector. Subsequently, specifications are determined for managing the motions of the various linkages in order to realize the desired trajectory.

As previously described two types of control policies exist: open loop and closed loop. If the policy is open loop, the trajectory is initiated at time t_0, terminated at time T, and does not change in between:

$$\mathbf{u}(t+\delta) = \mathbf{u}(t) \qquad t \in [t_0, T]. \qquad (3)$$

If the policy is closed loop, then feedback information, as denoted by **y**(t), is provided at a series of points, $t=t_i$, i=1,...k, between t_0 and T. This feedback can be used to evaluate, and possibly change, the trajectory at any of these times.

$$\mathbf{u}(t+\delta) = \mathbf{u}\,(\mathbf{y}(t), \mathbf{u}(t)) \qquad t \in [t_0, T]. \qquad (4)$$

At this point, we have said nothing about the parameters of the functions described above. They may or may not have the time variable “t” as a parameter. If they do not, the system is said to be time invariant. Most real-world systems, however, are time variant. If a system is time variant, both the state transition and output functions will be dependent on time:[28]

$$\mathbf{x}(t+\delta) = f\,(\mathbf{x}(t), \mathbf{u}(t), t) \text{ and } \mathbf{y}(t) = g\,(\mathbf{x}(t), \mathbf{u}(t), t). \quad t \in [t_0, T]. \qquad (5)$$

• State Space Representation for Discrete Time Systems

Many manufacturing systems have state variables that are not continuous. That is, the state variables change only at discrete points in time. Two important examples are queuing systems and inventory systems. The state variables for these systems

28. J. Reid, *Linear System Fundamentals: Continuous and Discrete, Classic and Modern*, McGraw-Hill Book Company, New York, New York, 1983.

change only when there is a new arrival or a new departure. The discrete analogs to equations (1) - (4) are given below.

$$\mathbf{x}(t_k) = f\,(\mathbf{x}(t_{k-1}), \mathbf{u}(t_{k-1})) \qquad k=0,1,...,n \qquad (6)$$

$$\mathbf{y}(t_k) = \mathbf{g}\,(\mathbf{x}(t_{k-1}), \mathbf{u}(t_{k-1})) \qquad k=0,1,...,n \qquad (7)$$

$$\mathbf{u}(t_k) = \mathbf{u}\,(\mathbf{y}(t_{k-1}), \mathbf{u}(t_{k-1})) \qquad k=0,1,...,n \qquad (8)$$

Equation (8) represents the control policy, which can be either open or closed loop. Open loop is a special case, since $\mathbf{y}(t_{k-1})=0$, and $\mathbf{u}(t_k)= \mathbf{u}(t_{k-1})$ for all k.

Models for Evaluating Manufacturing Systems

• Simulation Models for Manufacturing Systems

Simulation software has been used to model different processes and systems within the manufacturing enterprise for many years. The most important concept in manufacturing simulation is *state variable*. State variables describe what is happening in the process or system at any point in time. Continuous simulation models are used for state variables that change continuously over time. The models are mainly mathematical, differential, or difference equations that represent the evolution of some physical phenomena, which changes continuously over time. They are used primarily during product design and process selection. Examples include fluid and structural dynamics, stress analysis, heat transfer, and machine tool program verification.

Event and process modeling methodologies can be used only for state variables that change at discrete points in time. Examples of such variables include the number of jobs waiting in the queue in front of a machine, the status of each machine on the shop floor, and the location of each job in the factory. The simulation models that are built using these methodologies are mainly flow models which track the flow of entities through the factory. In the case of discrete-event simulation, the tracking is done using times at which the various events occur. In process simula-

tions, the tracking is done based on the resources utilized at the various steps in the process.[29]

• Finite State Machine Models for Manufacturing Systems

Another common way of modeling discrete time systems, and discrete approximations to continuous time systems, is with a state machine. A state machine can be defined using a 6-tuple,[30]

$$\{I, O, S, Y, F, s_0\}$$

where I is the set of Input conditions, O is the set of Output events, S is the set of all possible States, Y is a mapping from $S \times I \rightarrow O$, F is a mapping from $S \times I \rightarrow S$, and s_0, $s_0 \in S$, is the initial state. The set of possible states, S, can be finite or countably infinite. If the system is under a closed loop policy, then I typically contains feedback information, F contains rules for the determining the next state, Y represents the new control policy, and O the actions required to implement that policy.

State machines are often visualized as a state table or state transition graph. An example is provided in Figures 5 and 6. Given the current state, and the current inputs, the next state and the output events can be determined completely from the table or graph. Figure 7 shows how simply this can be implemented.[31]

Despite their simplistic nature, state machines can become unmanageable when applied to real world systems. The cardinality of $S \times I$ can grow very quickly, and the mappings Y and F can be quite complex. Frequently, the principles of hierarchy and abstraction are used to overcome these problems. We can aggregate states

29. *Ibid*.

30. B. Selic, G. Gullekson, and P. Ward, *Real-time Object Oriented Modeling*, John Wiley & Sons, Inc., New York, New York, 1994.

31. The figures are reproduced from J. Albus, C. McLean, A. Barbera, and M. Fitzgerald, "An Architecture for Real-time, Sensory-interactive Control of Robots in a Manufacturing Facility," in *Proceedings of the Conference on Information Control Problems in Manufacturing Technology*, National Bureau of Standards, Gaithersburg, MD, pp. 81-89, 1982.

into a single composite state using an abstraction process. This process hides the details of the original states and makes it possible to decompose the original state machine into a hierarchy of simpler machines. This means that a state machine description of a real-world system can be developed in stages. The state machine description can be initiated as a single, abstract machine; decomposed into a hierarchy of abstract machines; then gradually refined into the required detail at each level.

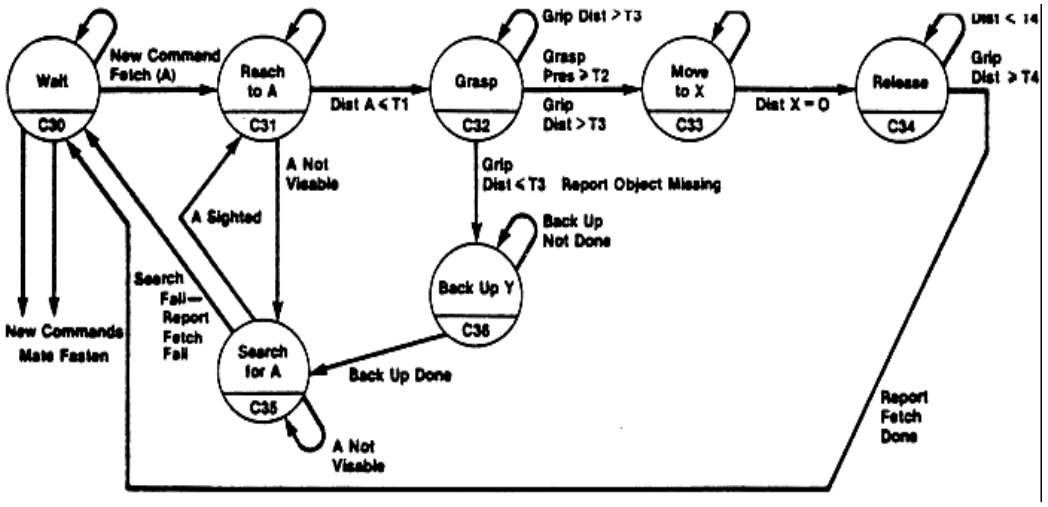

FIGURE 5. Example state transition graph

Command	State	Feedback	Next State	Output	Report
—	C30	No New Command	C30	Wait	—
Fetch (A)	C30	New Command	C31	Reach to (A)	—
"	C31	Distance to A>T1	C31	Reach to (A)	—
"	C31	Distance to A≤T1	C32	Grasp (A)	—
"	C31	A Not Visable	C35	Search for (A)	—
"	C32	Grasp Pressure <T2 Grip Dist >T3	C32	Grasp (A)	—
"	C32	Grasp Pressure ≥ T2 Grip Dist > T3	C33	Move to (X)	—
"	C32	Grip Dist ≤ T3	C36	Back Up (Y)	Object Missing
"	C33	Distance to X>0	C33	Move to (X)	—
"	C33	Distance to X=0	C34	Release	—
"	C34	Grip Dist <T4	C34	Release	—
"	C34	Grip Dist ≥T4	C30	Wait	Report Fetch Done
"	C35	A Not Visable	C35	Search for (A)	—
"	C35	A In Sight	C31	Reach to (A)	—
"	C35	Search Fail	C30	Wait	Report Fetch Fail
"	C36	Back Up Not Done	C36	Back Up (Y)	—
"	C36	Back Up Done	C35	Search for (A)	—

FIGURE 6. Corresponding state-transition table

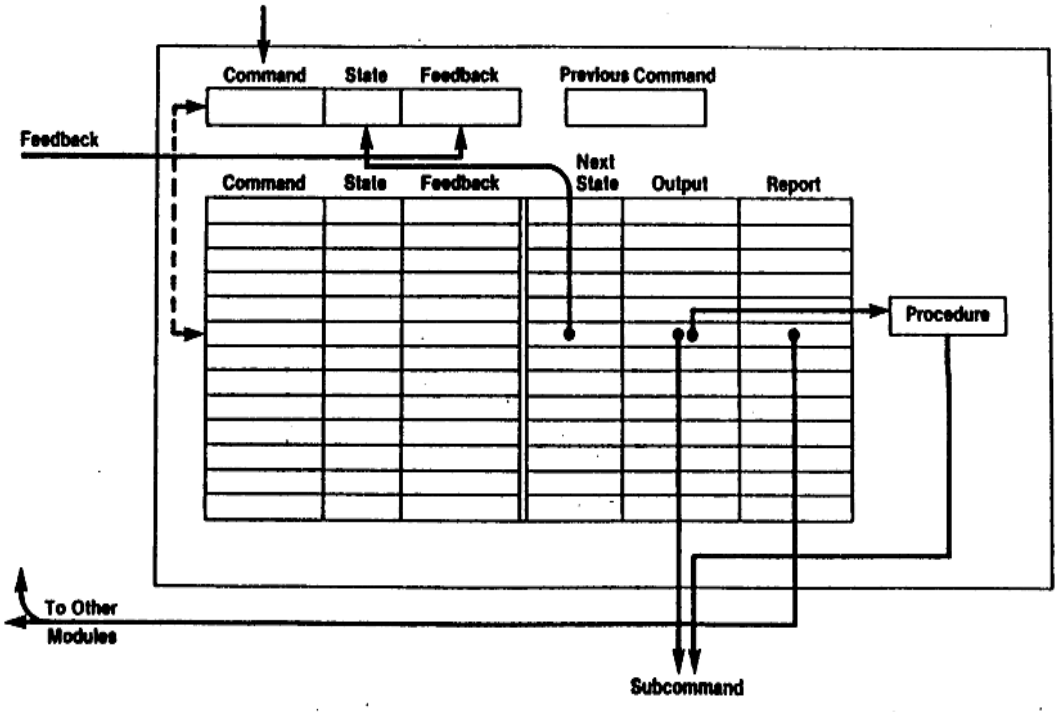

FIGURE 7. Computing structure to execute state-transition tables

Manufacturing Software Testing

• Software Programs as State Machines

State machines can be used to model the behavior of software programs in a number of ways. In its simplest form, a state represents the valid values for a single program variable. While this may be a valid, formal description of the program, it is of little practical use. This is true because even for elementary programs with a few variables, there can be an exponential explosion in the size of the state space. Therefore, it is necessary to have another, more abstract, definition of state in which a change in the value of any variable does not automatically imply a change

in state. Such a definition must capture the qualitative behavior of the program, without direct reference to the quantitative values of specific variables.[32]

Hierarchical state machines can also be used to model certain aspects of software programs. The resulting nesting of lower-level state machines within more abstract states is similar to the nesting in block-structured procedural programming languages. Moreover, the scoping rules are the same:

1. each nesting has a distinct lexical scope with its own variable declarations,
2. higher levels cannot access lower level scopes, and
3. lower levels can access all of their containing scopes.

There is one important difference between nested states and nested blocks: nested states are static, but nested blocks are dynamically created and destroyed as the program executes.

This notion of a software program as a state machine has been used, with some success, as the basis for unit testing.[33] A fixed set of inputs is generated, either manually or automatically, for which the desired state transitions and output values are known. Determining an appropriate set of inputs, based on some specification of program behavior, is itself a complex problem. Nevertheless, given such a set, program errors are said to occur whenever the actual state transitions or output values differ from the desired ones. Whenever errors occur, the source is found and corrected (another difficult problem), and further testing is done. This scenario continues until no further errors are detected, or until some predetermined, stopping condition is met.

32. Selic, 1994.

33. David Banks, William Dashiell, Leonard Gallagher, Charles Hagwood, Raghu Kacker, Lynne Rosenthal, *Software Testing by Statistical Methods - Preliminary Success Estimates for Approaches Based on Binomial Models, Coverage Designs, Mutation Testing and Usage Models*. NISTIR 6129, <URL:http://sdct-sunsrv1.ncsl.nist.gov/~ftp/stsm/mar98ir.pdf>, March 12, 1998.

• Systems of Systems

Problems arise when one attempts to use this approach to test certain kinds manufacturing software applications — namely, those that rely directly on other, "source" applications for their inputs and those with inputs that are based on events that happen in the real world. These are interaction-driven applications. Consequently, either we need the source application/real-world system to carry out the testing, or we need substitutes that behave exactly like them. In the latter case, the substitutes themselves must be tested to ensure correct behavior. In the former case, two scenarios arise.

If a source application provides the inputs, that application can be modeled as a finite state machine with its own set of transition points. If the inputs come from a real-world system, that system can be continuous time or discrete time. If the system is continuous, its evolution is governed by equations (1-4); if it is discrete, then equations (6-8) apply. This causes a problem known as racing. That is, the system being tested must race to complete its execution before the next event. The impact of racing on testing is not known.

In many cases, outputs from the test system can be inputs to the source application or cause additional changes in the real world. This frequently occurs when the test system and its source have a hierarchical relationship. When this happens, an output from the supervisor becomes input to the subordinate and feedback from the subordinate becomes input to the supervisor. In manufacturing, a single application can have many subordinates and one or more supervisors. This provides an added complication to testing.

In addition to these interaction-driven systems, there is an increasing emphasis on the use of software agents in manufacturing. These agents are neither typical procedural programs nor typical object-oriented programs. They are intended to have emergent behavior, which means that their response to various inputs can change as the agent becomes more "intelligent." From a state-machine modeling perspective, this means that the state space changes over time, the state transition function changes over time, and the outputs change over time. Furthermore, the relationships among these agents can change over time. This, again, has an impact on testing.[34]

Summary

If composable/component-based software delivers on its promises, it ought to be more amenable to modeling with hierarchical state machines than the average interaction-driven software system. We should investigate whether we can in this way alter our software architectures to make them tractable with the existing modeling techniques, and hence more testable. However, at this point in time, composable software is yet to be demonstrated on a large scale system, and even if all this comes to fruition, the modeling of racing and nondeterminism remains a problem.

Testing of Interaction-driven Manufacturing Software

As described in the introduction, three categories for the testing of interaction-driven manufacturing systems have been identified:

- component
- system
- interoperability

Based on a system architecture separable components can be identified for component testing, which addresses the question of whether a single component of the larger, distributed manufacturing system supports the interface and behavior described in its specification. When a component is based on a proprietary interface, testing will focus on performance and stress testing of the component to determine if it meets the needs of the system in those respects. Functional testing, while important for system integration, is typically left to the component vendor in the case of proprietary specifications; with public specifications a conformance testing program may be established to provide functional testing services and certification that a vendor's products comply with the open specifications. Alternatively, or in conjunction with conformance testing, interoperability testing may

34. Selic, 1994.

also be used to certify that a vendor's product conforms well enough to be plug-compatible with other products.

A component based on an open specification is much more desirable for integrated manufacturing systems for several reasons:

- The same performance and stress tests can be performed against multiple vendor products.
- Third party organizations can be used to perform functional tests, thereby providing an objective measure of a systems conformance.
- The system architecture can be designed without predetermining a particular vendor's solution.[35]

System testing concentrates on the interactions between the components of the system. System testing answers the question: does the integrated system behave as designed? System testing may include a combination of simulation in which the system design is analyzed, integration testing which is used to bring the system on line, and system monitoring which detects errors in the behavior of the system.

Finally, to support open systems interoperability testing is performed on components supporting an open interface. Interoperability testing answers the question: could a component be replaced by another component without compromising the system? Where component testing is a systematic approach to fully exercise a component interface, interoperability testing is testing by trial. Theoretically an exhaustive functional test of a component would answer the same question as interoperability testing. In practice, however, exhaustive testing is tedious, difficult, expensive, and frequently intractable given the wide range of inputs that systems will allow. Interoperability testing is most often performed by users in the market place as they try multiple vendors products in working systems,[36] but it can be performed in a more controlled setting, as it often is through vendor consortia.[37]

---

35. Wallnau *et al*., June 1998.

36. A case in point is programming language compilers: when software is ported between compilers, bugs both in the program source code and the compiler may be found. Compiler bugs are reported to the vendor and thus interoperability testing is taking place.

37. STEPnet, <URL:http://www.stepnet.org/>.

Interoperability testing uses natural selection to limit the number of tests to those that are relevant to the products that need to be interchanged. Often a combination of functional component tests and interoperability testing yields the most cost effective results.[38]

The remainder of this section discusses each of these categories of testing in more detail in terms of the open manufacturing interfaces that fall into the categories, the existing test efforts that are underway, test techniques suitable for addressing the category, and directions for future work.

Component Testing

As was discussed earlier, a system architecture is described in terms of the specialized subsystems it contains and the connections between those subsystems. Upon closer examination one finds a number of different methods for designing and implementing the connections.[39] For the purpose of this discussion, the specifications supporting the connections in a system are referred to as infrastructural components; the specialized subsystems are referred to as specialized components. The connections can be described both in terms of communication channels and transaction flow. The channels are based on specifications, whether standard or proprietary, formal or informal. The transaction flow is the series of events that happen during the operation of the system. While it may be repeated, the flow is unique to the operation of a given system at a given time.

A specialized component supports a specific functional requirement of the overall system. In a manufacturing system, specialized components will include software such as databases, product data managers, schedulers, machine controllers, inspection systems, and inventory systems to name a few. Testing of static interfaces, such as file exchanges, can be considered a sub-category of specialized component testing. The interfaces to specialized components are usually defined either within the context of a particular infrastructure or as a combination of an abstract inter-

38. James Kindrick, John Sauter, Robert Matthews, "Improving Conformance and Interoperability Testing," *StandardView*, May 1996.

39. David Garlan and Mary Shaw, *An Introduction to Software Architecture*, CMU Software Engineering Institute Technical Report CMU/SEI-94-TR-21, ESC-TR-94-21, January 1994.

face and a binding of that interface to a particular infrastructural approach or implementation mechanism.

Infrastructural specifications provide a framework in which specialized component interfaces can be defined and operate. Infrastructural components support flexibility with respect to integrating new specialized components into a working system.

Traditional software architectures use a tight integration of subsystems based on shared libraries which are linked into run-time execution modules. In these systems the common infrastructure is the programming language. Dynamic linking (linking at run-time) is one step towards creating more flexibility in component-based systems. Even so, connections between components traditionally have been reduced to file exchanges rather than direct interactions. Advances in system engineering and software standards to support distributed processing are leading to more flexible integration of systems which are capable of direct interactions. Software standards which support distribution (including NFS/RPC,[40,41,42] CORBA/IDL,[43] and Java's Remote Method Invocation and JavaBeans[44]) provide even more flexibility.

Infrastructural specifications address the coupling between specialized components and provide a protocol (in the general sense) for the connections. Aspects of the connections can be tested for consistency and completeness in much the same way that a compiler tests a program for consistent use of a programming language. Infrastructural standards include programming languages, scripting languages, distribution protocols such as NFS/RPC and CORBA/IDL, messaging languages such as MMS, UNIX-style pipes, and sockets.

---

40. Network File System/Remote Procedure Call.

41. B. Callaghan, B. Pawlowski, P. Staubach, *NFS Version 3 Protocol Specification, Request for Comments: 1813*, Sun Microsystems, Inc., June 1995, <URL:http://www.cis.ohio-state.edu/htbin/rfc/rfc1813.html>.

42. *The NFS Distributed File Service, NFS White Paper - March 1995*, <URL:http://www.sun.com/software/white-papers/wp-nfs/>.

43. OMG, *The Common Object Request Broker: Architecture and Specification*. (CORBA) Includes a definition for the Interface Description Language (IDL). <URL:http://www.omg.org/corba/c2indx.htm>.

44. Java Technology Home Page, <URL:http://java.sun.com/>.

Manufacturing Interfaces

This section describes manufacturing specific specifications for both the infrastructural and specialized components of a system and the implications both of these have with respect to testing of interaction-driven manufacturing systems.

• STEP

ISO 10303,[45] a.k.a. STEP, is a family of specifications for product data exchange covering the complete product life-cycle. STEP defines models of information to be shared between systems and implementation methods for sharing the data. The information models (STEP's Integrated Resources and Application Protocols) describe the content of the data; the implementation methods describe the computing mechanisms to be used in sharing data. Together they define a variety of interfaces for components in a software system.

From a testing perspective, STEP is interesting because of the wealth of component interfaces defined by combining the standard's specifications and because of the rigor with which these standards are specified. Testing of the implementation methods can be defined either independent from or in conjunction with the content specifications. Two types of implementation methods are defined or emerging within the STEP: an exchange file format[46] (Part 21) and an application program interface (SDAI). Each is discussed independently.

• STEP/Part 21

An interface based on a STEP AP and the STEP exchange file format is an example of a specialized component interface. In this case the interactivity quotient for the components involved in the data exchange is very low and methods for testing such interfaces are well established.[47] This style of interface is mentioned here as one end of the spectrum of interactions a component may have with a system.

---

45. *Industrial automation systems and integration—Product data representation and exchange—Part 1: Overview and fundamental principles*, International Standard, ISO, 1994.

46. *Industrial automation systems and integration—Product data representation and exchange—Part 21: Implementation methods: Clear text encoding of the exchange structure*, International Standard, ISO, 1994.

47. The STEP Conformance Testing Project Home Page. <URL:http://www.erim.org/cec/steptest/>.

• STEP/SDAI

An interface based on SDAI supports a greater amount of interactivity. Using SDAI, the specialized components can be probed for data values interactively and data values can be altered in the underlying database system. SDAI is defined as a functional specification and a set of language bindings which include the programming languages C and C++ and the interface definition language IDL. The SDAI specification defines six implementation classes based on five characteristics. The lowest implementation class provides minimal support for data exchange and is essentially an API to an exchange file. The higher classes provide progressively more sophisticated features culminating in rich support for expression evaluation which enables complete constraint checking and calculation of derived attributes from the information models. The definition of implementation classes allows systems supporting the standard to evolve better and better support.

While supporting system evolution, SDAI's implementation classes also provide guidance for software componentization. As applications adapt to newer SDAI implementations, more and more of the functionality that they may require will be resident in the SDAI implementation. With this in mind, applications based on SDAI should be designed such that they will not be adversely impacted by the evolution.

Test methods for SDAI implementations will need to address the multiple implementation classes of SDAI's functional specification as well as multiple implementation classes defined for each of the language bindings. Test methods for APs using SDAI will need to further address the functional requirements of a particular AP as well as the implementation classes defined that AP. Test methods for STEP implementations will need a strong framework in which to accommodate all of the variability imposed by the different specifications and the implementation classes involved.

• OMG/PDM Enablers

The OMG's PDM Enablers specification[48] defines twelve IDL modules to support eight conceptually separate PDM functions, or "enablers" of product data management. A conforming implementation need not support all of the modules or all of the enablers; dependencies between the modules are spelled out in a dependency

graph,[49] and the modules required by each enabler are spelled out in another table.[50] There is also a mapping from eleven sub-processes of the product development process to the enablers that they, in turn, require.[51]

The interfaces defined in the PDM Enablers specification provide a combination of data model and data access mechanisms. However, the behaviors exhibited and constraints enforced by conforming PDM implementations are not formally defined. Although there are UML diagrams and much explanatory text, only the IDL interfaces are considered normative.[52] Since the PDM Enablers specification is heavy on data modeling and light on interactivity, conflicting interpretations may not be as big a problem as they could be.

• CAM-I AIS

The CAM-I AIS interface specifies an API to a solid modeling system. The system can then be used as an engine to other software that needs solid modeling capabilities. The interface is specified as both a file format and as a package of C library routines. While no testing activities specific to this interface have been identified, methods for testing both file formats and software libraries are established.

• OLE for Design and Modeling

Object Linking and Embedding (OLE), a proprietary Microsoft infrastructure, has been used to define an interface for Design and Modeling.[53] OLE for Design and Modeling is an industry-led effort and it is an open interface. The interface allows 3-D graphical objects from one application to be included in (or embedded in) another application. If the original application is changed, the referencing applica-

48. Revised Submission (including errata changes) — PDM Enablers — Joint Proposal to the OMG in Response to OMG Manufacturing Domain Task Force RFP 1. <URL:http://www.omg.org/arch2/mfg/98-02-02.pdf>, 1998.

49. *Ibid*., section 1.16.2.5 (“Module Interdepencies”).

50. *Ibid*., section 1.9 (“PDM Enablers”).

51. *Ibid.*, section 1.12 (“Mapping the Product Development Process to the PDM Enablers”).

52. *Ibid.*, section 1.16.2.1 (“IDL Specifications”).

53. OLE for Design and Modeling. <URL:http://www.intergraph.com/iss/technologies/jupiter/ole.htm>, 1998.

tion will reflect those changes. Using this interface, interactivity is a one-way process. The client is able to read information from and create references to the server, and even override some of the server's data on the client side, but the client is not able to affect any changes in the server.

• PSL

Not yet a mature standard, the Process Specification Language (PSL) defines a mechanism for applications of process information to exchange that information. It includes an ontology to support the exchange. It is envisioned that this specification will evolve to support interactivity between systems representing and controlling manufacturing processes. A goal for the TIMS project is to establish guidelines for testability which will be used in the development of PSL.

• Summary

Table 1 summarizes the component interfaces for manufacturing systems and associates them with the infrastructure on which they are based and with an analogous form of software. Testing techniques have been developed or are under development in association with the various kinds of software infrastructures. Some of these are discussed further below.

TABLE 1. Infrastructures supporting Manufacturing Interfaces

Component Interface	Software Infrastructure(s)	Type of Software
AP xxx[a] + STEP Part 21	none	static file checker
AP xxx + SDAI C++ binding	SDAI and C++	class library/ database
AP xxx + SDAI C binding	SDAI and C	software library/ database
AP xxx + SDAI IDL binding	SDAI and CORBA	distributed invocation method/ database
AP xxx + SDAI Java binding	SDAI and Java	web/ Java/ database
PDM Enabler	CORBA	distributed invocation method/ database
CAM-I AIS	C	software library
OLE for Design	Microsoft COM/OLE	distributed invocation
PSL	tbd	tbd

a. The notation xxx is used to indicate any AP.

Testing based on SDAI uses a multi-tiered infrastructure: one tier being the language binding and the other tier being SDAI. In Table 1 the column *Type of Software* refers to general categories of software for which testing activities and tools exist. When testing of the identified component interface is pursued, a more in-depth investigation of these types of testing and associated tools will be conducted.

Existing Testing Efforts

The only identified effort that directly addresses testing of specialized component interfaces for manufacturing is that for STEP's AP 203[54] exchange file. Other activities are relevant in their analogy to manufacturing interfaces. Efforts for testing generic components based on the infrastructural components used in manufacturing include the 100% Pure Java testing program[55] and NIST's VRML (Virtual Reality Modeling Language) testing program. NIST's SQL (Structured Query Language) testing program addressed some of the challenges faced in testing manufacturing specifications.

• STEP/Part 21/AP 203

This testing effort generates test cases from the specification for AP 203.[56] The methodology used is rigorously defined within the context of STEP and is being used for other APs within STEP. One unique characteristic of the STEP methodology is that the standard includes the specification of abstract test suites for each AP. The standard abstract test suite for AP 203 formed the conceptual basis for the executable AP 203 tests. While this method is extensively developed for testing the functional requirements of an application, there has not been work in the area of testing an interactive interface such as would be available with SDAI. Parts of the methodology may be adaptable to testing implementations of STEP AP 203 using SDAI, as well as other APs with an SDAI interface.

---

54. *ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Part 203: Configuration Controlled Design*, International Standard, ISO TC184/SC4, 1994.

55. 100% Pure Java Program Home Page, <URL:http://www.javasoft.com/100percent/cert.html>.

56. *ISO 10303 Industrial automation systems and integration—Product data representation and exchange—Part 303/TR: Abstract test suite: Configuration controlled 3D Designs of Mechanical Parts and Assemblies*, working draft, 1998.

Additionally, while providing a different level of abstraction, where there is overlap, the OMG PDM Enabler specification is consistent with AP 203's definitions for configuration control and thus provides an interactive interface to AP 203 which is not based on SDAI. The AP 203 test cases for file exchange are in part applicable to testing the PDM Enabler interface as well.

• 100% Pure Java

The 100% Pure Java testing program certifies that Java classes support their public interfaces.[57] In this program the interfaces are not necessarily openly defined. The developer submits the classes to be tested along with the test suites and the expected results of the tests. The testing program then verifies that the program runs with the Java Virtual Machines from multiple vendors and that it produces the expected results in each case. The testing program provides an independent evaluation of a program's portability and an evaluation of run-time correctness, as defined by the implementor, for Java applications.

Some aspects of this approach are of interest in the discussion of testing of manufacturing interfaces. The approach is generic to Java Class Libraries and thus should be applicable to any manufacturing interface defined in Java. The development of test suites by the vendors may be a practical approach to test suite development providing that someone is able to verify the reliability of the test suite. A NIST competence project in the Information Technology Laboratory, Software Testing by Statistical Methods, is studying methods for assigning a level of certainty to test suites.[58] If successful, these methods could be applied to evaluating vendor developed test suites.

57. In this context *public interface* is the technical term defined for the Java programming language to indicate the availability of the interface for use by someone other than the class developer. This is somewhat different than the less technical use of the term to mean that the interfaces that are published in an open forum for use by anyone.

58. Banks *et al*., NISTIR 6129, 1998.

• **VRML**[59]

In this testing activity, NIST devised a way to evaluate the state of a VRML browser at run-time using features of the VRML language. Those states are then compared to the correct state as defined by a reference implementation to determine the system's correctness. The approach is to probe and monitor a system relative to a reference implementation.

• **SQL**

The NIST SQL Test Suite[60] addressed several of the challenges faced for testing manufacturing component interfaces, including multiple implementation options in a specification, bindings to multiple programming languages, and access by more than one process concurrently.

Both the SDAI and PDM Enablers specify interfaces that may be configured in more than one way. In the SQL Test Suite, conformance tests were catalogued according to the "profiles" that they tested. Some tests were designed to test profiles singly while others tested them in useful combinations. A script to run the correct sequence of tests was then generated by a program according to which profiles the vendor claimed to support. This same technique also helped to address the complexity of bindings to multiple programming languages.

SDAI and the PDM Enabler are both interfaces to systems which will be used by more than one process. The SQL experience with issues of concurrent use is also applicable here.

Test Techniques

Component testing is the most studied of the types of testing identified. While there is some argument over whether interoperability or conformance testing is a better way to approach component testing, there is no doubt that the components in a complex distributed system of systems need to be tested in order to have any

59. NIST VRML Project Home Page <URL:http://www.itl.nist.gov/div897/ctg/vrml/vrml.htm>.
60. NIST SQL Test Suite. <URL:http://www.itl.nist.gov/div897/ctg/sql_form.htm>, 1995.

chance at success in complete system testing. Here we present an overview of some of the most pertinent aspects of component testing.

• Domain testing

A premise for theories of component testing is that exhaustive testing is only possible in a very limited number of cases. In most cases, testing is approached using a strategy designed to produce optimal results; in other words, to find the most bugs and the most serious bugs with the least number of tests. Much of the study of software testing revolves around what is the best strategy. Theories address both white box (e.g. debugging) and black box testing. The scope of the TIMS project is limited to black box testing.

Test coverage and software usage patterns are critical factors in testing strategies. Test coverage is an indication of how much of the software was actually tested. Test coverage can be measured in two ways — in terms of how much of a specification was exercised as a percentage of the complete specification, or in terms of how completely the tests covered the domain of input values defined for the specification. Tests which cover the complete range of input values can be particularly difficult to define and execute. For example, consider a program which simulates a continuous function.

Since exhaustive testing is typically not practical, two non-exclusive approaches are often used to limit the scope of the tests. One strategy is to analyze usage patterns of the component and use that analysis to prioritize the tests. Note that this is very similar to interoperability testing which achieves essentially the same thing in a less formal way. Another strategy referred to as domain testing is to identify a set of input values and test the component with those values rather than attempting the complete range of values. Beizer[61] provides an in depth discussion of domain testing which describes systematic approaches to identifying an optimal set of input values.

61. Boris Beizer, *Software Testing Techniques*, Chapter 6,Van Nostrand Reinhold, NY, NY, 1990.

• The role of infrastructures

Testing of infrastructural components differs from testing of specialized components in that, for the most part, infrastructural components are not manufacturing specific and can be exercised by generic types of test suites. The one exception identified as a manufacturing specific infrastructure is MMS.

Generally speaking, infrastructures are most vigorously, thoroughly, and quickly tested for conformance to a specification through use rather than by any formal testing activity. When there is a formal testing activity it generally comes after the fact and is designed to differentiate multiple vendor products supporting the same specification and highlight those areas of inconsistent support so that they may be addressed.

On the other hand, infrastructural components are integral to the testing of specialized component interfaces since the specialized components are based on the infrastructures. Many testing tools, and hence techniques, revolve around the use of a particular infrastructure. Additionally, as is illustrated in Table 1, layered infrastructures are emerging (e.g., SDAI/a programming language, IDL/a programming language, SDAI/IDL). Test methods based on infrastructures are essential to managing the layers.

• Formal methods

The application of mathematical methods to software development has resulted in a discipline known as *formal methods*. The original emphasis for formal methods was on software design and development. Formal methods are used to rigorously describe software specifications before any code is implemented. Some success has been shown in applying formal methods in this manner. More recently, attention has been given to applying formal methods to the specification and testing of standards[62,63,64] and several standards for formal description techniques have been defined within ISO.[65,66,67,68] The application of formal methods to standards dif-

62. Richard Botting, Anthony Godwin, "Analysis of the STEP Standard Data Access Interface using Formal Methods," *Computer Standards and Interfaces*, vol. 17, 1995.

63. Richard Sinnott, Kenneth Turner, "Applying Formal Methods to Standard Development: The Open Distributed Processing Experience," *Computer Standards and Interfaces*, vol. 17, 1995.

fers from the traditional use in software development since the internals of the software implementations are not defined. The result of applying formal methods to software standards is improvement in the standard itself, since it is more rigorously defined, and improvement in the ability to generate test software from the specification.

Future Directions

SDAI and PDM Enablers are both good examples of specifications for interaction-driven manufacturing components which are intended to operate in a distributed environment.

The SDAI specification provides the infrastructure for a series of interfaces to interaction-driven manufacturing systems. The SDAI approach is based on a rigorous methodology which has not yet been fully exercised. To date, test techniques for these types of interfaces have only been applied on individual components and we have found no activities to systematically reuse the methods on a variety of interfaces as would be possible by combining SDAI with STEP APs in a number of different application areas.

The PDM Enablers use an infrastructure, namely CORBA, for which testing activities to date have been *ad hoc*. Currently, several efforts are underway to formalize testing methods using this infrastructure.

---

64. Kathy Liburdy, Martha M. Gray, and Lynne S. Rosenthal, "Formal Specification Languages in Conformance Testing." Presented at the Eleventh International Software Quality Week 1998, May 26-29, 1998.

65. *ISO/IEC 8807, Information Processing Systems — Open Systems Interconnection — LOTOS —A Formal Description Technique based on the Temporal Ordering of Observational Behavior*, International Organization for Standardization, Geneva, 1989.

66. *ISO/IEC 8807, Information Processing Systems — Open Systems Interconnection — ESTELLA —A Formal Description Technique based on an Extended State Transition Model*, International Organization for Standardization, Geneva, 1989.

67. IUT-T, *Specification and Description Language*, CCITT Z.100, International Consultative Committee on Telegraphy and Telephony, Geneva, 1992.

68. C. Ruggles (Ed.), *Formal Methods in Standards: A Report from the BCS Working Group*, Springer-Verlag, British Computer Society, 1990.

Integration and System Testing

While many standards concentrate on individual software interfaces within a manufacturing system, the concern of end users of manufacturing software is whether the final integrated system performs as required. From both practical and formal perspectives, verifying correct behavior of the entire system is quite different from verifying the behavior of individual components. As Brooks observed, the system test is "unexpectedly hard," and is often made harder by a lack of conceptual integrity between independently specified and/or developed components.[69] The risk and cost of system level problems only becomes greater as industry moves away from monolithic, proprietary solutions and instead assembles systems from open standard software components bought from completely different sources. The individual "components" that we are integrating today would have been considered "systems" at one time, so "integration testing" and "system testing" need to be understood in this new, larger context.

Interaction-driven systems pose a special challenge because dynamic interaction between disparate components is much more sensitive to errors than is static file exchange. If two components disagree on a syntax issue for file exchange, we might be lucky enough to lose only a small part of the input; but if they disagree about an interactive interface, it is almost certain to stop the show. Integration and system testing are therefore crucial to the reliability of interaction-driven systems.

A publication from Rational Software Corporation defines integration and system testing as follows:

> Testing a specific feature together with other newly developed features is known as integration testing. Testing the interface of two components explores how components interact with each other. Integration testing inspects the variables passed not only between two components, but also the global variables. This test phase assumes the components and the objects they manipulate have all passed their local unit tests.

69. Frederick P. Brooks, Jr., *The Mythical Man-Month*, 1995 edition. Addison-Wesley.

> [...]
>
> System testing is designed to reveal bugs that cannot be attributed to individual components, or to the interaction among components and other objects. System tests studies all the implementation aspects of the design similar to the customer's environment. System testing concerns issues and behaviors that can only be exposed by testing the entire integrated system or a major part of it. System testing includes testing for performance, stress, security, accountability, configuration sensitivity, usability, data integrity, start-up and recovery.[70]

By specifically excluding interaction-related faults, this definition of system testing suggests that serious problems such as deadlocks and race conditions are expected to have been found during integration testing, much as functional faults are expected to have been found during unit testing. This is consistent with a great deal of our own precedent which steadfastly separates integration testing from system testing.[71] However, for our larger distributed systems, the only real differences between these tasks are the invasiveness of the testing and the types of faults that we hope to find in each stage. For the purposes of this discussion, then, we will not struggle to keep them separated, but assume that Brooks' advice to "Add one component at a time" will nonetheless be followed in the software development process.

System-Level Manufacturing Specifications

• SEMATECH CIM Framework

The SEMATECH Computer Integrated Manufacturing (CIM) Framework Specification[72] is a specification for the software infrastructure within semiconductor fac-

70. Laura Lee Rose, *Getting the Most Out of an Automated Test Tool*. Rational Software Corporation, <URL:http://www.rational.com/support/techpapers/pacorpwp/>, 1998.

71. NBS Special Publication 500-98, *Planning for Software Validation, Verification, and Testing*, November 1982.

tories. It provides interface definitions in OMG's IDL, Harel Statecharts for component states,[73] Sequence Diagrams to show how components are meant to interact,[74] and prose explanations to explain the definitions and intent. SEMATECH is no longer developing the CIM Framework *per se*, but the specification is being used as an input to the Manufacturing Execution Systems / Machine Control (MES/MC) Work Group of the OMG's Manufacturing Domain Task Force,[75] where a widely accepted standard is expected to emerge. Concurrently, related work is being pursued in the Semiconductor Equipment and Materials International (SEMI)[76] CIM Framework Task Force. SEMI is a global trade organization that publishes standards made by and for the semiconductor industry.

In their submission to the MES/MC Work Group,[77] SEMATECH commented that the formal representations that they had used in the CIM Framework were "incapable of capturing the level of semantics needed to enable higher levels of interoperability" and "weak in their representation of the behavioral semantics." They now believe that they may have found a better way:

> These problems led SEMATECH to contribute to the development of the BOCA [Business Object Component Architecture[78]] specification in response to the OMG's Business Object Facility RFP [Request for Proposals[79]]. The BOCA meta-model and the rigorous

72. CIM Framework Home Page, <URL:http://www.sematech.org/public/division/fi/cim/cimhome.htm>.

73. David Harel, "Statecharts: a visual formalism for complex systems." *Science of Computer Programming*, pp. 231-274, July 1987.

74. *UML Notation Guide, Version 1.1*, "Section 7: Sequence Diagrams." Rational Software Corporation, <URL:http://www.rational.com/uml/html/notation/notation7.html>, September 1997.

75. Manufacturing Execution Systems / Machine Control Work Group Home Page, <URL:http://www.omg.org/mfg/mfgmesmc.htm>.

76. *SEMI OnLine*, <URL:http://www.semi.org/>.

77. *Response to the Manufacturing Domain Task Force RFI-3: Manufacturing Execution Systems (MES)*. <URL:ftp://ftp.omg.org/pub/docs/mfg/98-05-03.pdf>, International SEMATECH and Fraunhofer IPA, May 1998.

78. Data Access Technologies, Inc., *et al.*, *Business Object Component Architecture (BOCA) Revised Proposal: revision 1.2*, July 1998. <URL:ftp://ftp.omg.org/pub/docs/bom/98-07-01.pdf>.

use of a Component Definition Language (CDL)[80] offer a way to capture a richer, more complete and more consistent specification of the CIM Framework semantics.

• APC Framework

The Advanced Process Control (APC) Framework[81] was once an independent system-level specification; it is now considered to be a subsystem of the CIM Framework, specifically the Advanced Process Control Group of components. An implementation of this subsystem is now being commercialized.

The CIM Framework itself may be viewed as the manufacturing execution subsystem of the manufacturing enterprise, which would include other subsystems such as enterprise resource planning. This is noteworthy because system testing issues arise at every level, and rigorous testing becomes less and less feasible as the scope of the system is enlarged.

• MSI Control Entity Interface Specification

The Manufacturing Systems Integration (MSI) project at NIST[82] produced a specification for control entities in a distributed, hierarchical manufacturing system.[83] Despite its "interface" designation, the specification in fact provides interface definitions, message definitions, state diagrams, and prose descriptions to specify the behavior of components implementing planning and job control interfaces that compose hierarchically to form an interaction-driven system.

---

79. *Common Facilities RFP-4: Common Business Objects and Business Object Facility.* <URL:http://www.omg.org/arch2/cf/96-01-04.pdf>, 1996.

80. Data Access *et al.*, section 2.3: *Component Definition Language Specification*.

81. *Advanced Process Control Framework Initiative (APCFI) 1.0 Specifications*, 1997. <URL:http://www.sematech.org/public/docubase/abstract/3300aeng.htm>.

82. M. K. Senehi, S. Wallace, and M. E. Luce, "An Architecture for Manufacturing Systems Integration." *Proceedings of the ASME Manufacturing International Conference*, Dallas, TX, 1992.

83. Sarah Wallace, M. K. Senehi, Ed Barkmeyer, Steven Ray, and Evan K. Wallace, *Manufacturing Systems Integration Control Entity Interface Specification.* NISTIR 5272, September 1993.

A primary objective of the architecture was to incorporate provisions for dynamic recovery from anomalous situations at every level, so cyclic interactivity pervades the system. The protocols and interfaces for distributed planning and control would provide fertile ground for system level testing. While this specification *per se* is clearly not the focus of industrial attention, it serves as an example of a well-defined hierarchical architecture for manufacturing, and its concepts and lessons learned have contributed to several OMG specifications.

Existing Testing Efforts

Industry-driven efforts towards developing an open methodology for interaction-driven distributed system testing are in their infancy. OMG's Test Special Interest Group[84] has as part of its mission to "offer guidelines to applications developers on how to perform appropriate testing of object-oriented and distributed applications" and to "encourage test tool vendors to create products that support and automate the testing process." At this time the group has not finished producing its initial white paper. Similarly, the Information Technology Laboratory at NIST has identified "testing methods for object-oriented software, software components, and component interactions in an integrated software system" as one focus of the Software Testing by Statistical Methods project,[85] but has not yet produced an initial paper on the subject.

Although testing of these systems is done every day by companies using common sense approaches (see "General technique," below), these efforts are seldom publicized. Conversely, a great deal of published work provides methods for the testing of various distributed phenomena at a high level of abstraction, but it does not meet the need for complete system testing.

Consider, for example, the common problem of different components giving subtly different interpretations to data that they exchange. Let us assume, as is usually the case, that the components were separately specified and do not formally share a common ontology. Then this fault will probably not be detected in component test-

84. Test Special Interest Group Home Page, <URL:http://www.omg.org/testsig/>.

85. Software Testing by Statistical Methods home page, <URL:http://www.itl.nist.gov/div897/ctg/stat/stsm.htm>. April 1, 1998.

ing because each component appears to satisfy its own specification. It also will not be detected by simulation and analysis of the component interactions because the correct protocol is being followed. In order to detect the fault, it is necessary to verify the *contents* of messages exchanged between components, not merely the message exchange itself or the components' behaviors by themselves. This is easily done using *ad hoc* techniques, but not so easily using most formal methods, which either abstract away the message contents in order to make the interaction protocol tractable, or abstract away the interactions in order to make the interface specification tractable. This disconnect carries through into available software toolsets (e.g., ObjectGeode[86] and the ObjectTime toolset[87]), which include tools that use different languages (e.g., finite state machines versus class diagrams) to perform these interrelated tasks. Rapide[88] tried to bridge this gap by adding computationally complete functions to an Architecture Description Language; we should investigate to see to what extent they were successful, and whether the resulting language is usable and understandable. (Of course, this is only useful for simulation, not for testing of implementations.)

Integration / System Test Techniques

• General technique for locating faults and testing conformance

In system testing, a piece of bad data may propagate through several components before a problem ever appears. Locating the fault in such cases can be difficult. A good strategy to narrow the possibilities is to replace one or more of the components with *dummy components*[89] to see whether the problem goes away. A dummy component anywhere on the path of the bad data will break the chain and cause correct operation, so the fault is eventually located by walking backwards to the source.

86. ObjectGEODE. <URL:http://www.verilogusa.com/solution/geode.htm>. 1998.
87. ObjectTime Developer toolset. <URL:http://www.objectime.com/>. 1998.
88. The Stanford Rapide™ Project. <URL:http://pavg.stanford.edu/rapide/rapide.html>. 1998.
89. Frederick P. Brooks, Jr., *The Mythical Man-Month*, 1995 edition, p. 148. Addison-Wesley.

Along the way, swapping dummy components for real components may detect hidden deviations from the specification, where a system is working only because all off the components using a given interface use the same incorrect interpretation, and are hence "bug-compatible" by accident. Incorrect but functional usage of an interface can propagate through a software project like a virus because developers will copy or re-use working code. Once this problem is detected, the code can be changed to match the documentation — or vice-versa, as is more often done in practice if there is no established standard. Clearly this could be a powerful technique for conformance testing as well.

• Network monitoring and capture/replay techniques

If some semblance of all of the necessary components for a networked distributed system already exists, a capture/replay tool can be used to examine system behavior and to emulate components. These commercially available tools begin by recording all network traffic during an actual run of the system. This record can be examined manually to insure that the messages exchanged between components are what was specified. The replay tool can then replay segments of the network traffic to emulate a component. With the use of "parameterization," the replay tool can change key fields of generated messages in order to act out various testing scenarios.

Capture/replay tools are popular due to their simplicity, flexibility, and robustness. Because their interaction with the system is on an entirely syntactic level, they will work to some degree with any networked system, and there is little or no application-specific scaffolding to build. Even opaque, COTS components with no publicly available specifications can be emulated without trouble. However, the purely syntactic treatment of system interactions is also their greatest disadvantage. The tester is obliged to operate at the level of raw data and machine code to construct meaningful tests using snippets of captured traffic, which must be reverse-engineered to map the raw data fields to their counterparts in high-level languages. There is no way to "get inside" of the emulated component to add test scaffolding and assertions. There is also no easy way to emulate components for which a reasonable facsimile does not already exist, and integration with semantics-based formal methods is unlikely to happen in the near future.

• Scenario based testing

While rigorously specifying the behavior of a distributed system in general is very difficult, specifying this behavior for a specific scenario is more tractable, as is demonstrated by the Component Interaction Specification (CIS) based method supported by the Manufacturer's CORBA Interface Testing Toolkit (MCITT),[90] UML Sequence Diagrams,[91] and Message Sequence Charts.[92] CIS has the advantage of being directly translatable into test scaffolding for CORBA systems, but it has disadvantages that will be discussed below.

CIS is a derivative of the integration testing method that was being used by Advanced Micro Devices to test components of the APC Framework. This method, in turn, made use of ideas that are also used in UML Collaboration Diagrams.[93]

A CIS interaction scenario consists of a tree of requests having specified inputs, outputs, and/or return values. The tree is rooted at a test client that initiates the entire chain of events. In order to capture the tree structure of the interactions in a flat ASCII script, an outline numbering convention similar to that of UML Collaboration Diagrams is used:

1 ... first request by testing client on server A ...

2 ... second request by testing client on server A ...

2.1 ... request by server A on server B ...

2.2 ... request by server A on server C ...

3 ... third request by testing client ...

90. MCITT home page. <URL:http://www.mel.nist.gov/msidstaff/flater/mcitt/>. 1998.

91. *UML Notation Guide, Version 1.1*, "Section 7: Sequence Diagrams." Rational Software Corporation, <URL:http://www.rational.com/uml/html/notation/notation7.html>, September 1997.

92. *ITU-TS Recommendation Z.120*, "Message Sequence Charts (MSC)." ITU-TS, Geneva, 1996.

93. *UML Notation Guide, Version 1.1*, "Section 8: Collaboration Diagrams." Rational Software Corporation, <URL:http://www.rational.com/uml/html/notation/notation8a.html>, September 1997.

In an actual CIS, the text comments shown above are replaced by machine-readable syntax specifying the remote operations that are invoked and the inputs, outputs, and/or return values that are expected.

Although the full extent of possible functionality is not supported by MCITT at this time, this approach enables code generation for dummy components and automatic generation of run-time assertions to verify that the inputs and returns for each interaction are as specified. The dummy components are useful in system and integration testing when some components are not available or not trusted, and in conformance testing to provide a more controlled testing environment for subsystems. One may also use dummy components to stress test a system; for example, if a shop controller is theoretically able to control up to N workcells, one may test the system with that many emulations.

Unfortunately, although the CIS syntax is expressive enough to describe an entire tree of interactions through a distributed system, it assumes a single source of interactivity. All interactivity is assumed to originate with the testing client. There is thus the question of how this approach can be extended to handle more complex interactivity. While it is trivial to extend the CIS syntax to *specify* cyclic or chaotic interactivity, it is much more difficult to *emulate* or *verify* that behavior because we now require the capability to monitor and control the sequencing of events at the system level. For the currently supported hierarchical interactivity, the ordering of events is inherently deterministic, and it suffices to embed monitoring and control into the components of the system. But with cyclic or chaotic interactivity, an emulated component must somehow manage to generate requests in the order that is specified without the benefit of inherent determinism, and a total ordering of system events cannot be derived without the aid of accurate synchronized clocks or a separate network level monitoring tool.

In any case, the total ordering of events imposed by the CIS approach is often not what we want in systems having cyclic or (especially) chaotic interactivity. Many formal analysis techniques proposed by academia for use on distributed systems explicitly address the issues of concurrency and non-determinism and allow the valid and invalid sequences of events to be identified; but there remains a disconnect between those formal analysis techniques and the available technology for testing actual systems. Similarly, UML Sequence Diagrams and Message Sequence Charts are more powerful in being able to model cyclic interactivity and

are extensively used for simulation, but they have not (to our knowledge) been adapted for concrete system implementation testing and run-time verification of message contents as CIS has been. This may be due to the problem of monitoring and controlling global state, of specifying message contents with sufficient rigor to be able to interoperate with actual live components, or both.

Future Directions

A possible direction for future work would be to investigate how to bridge this gap between abstract analysis and practical testing in the case of non-deterministic, interaction-driven manufacturing systems. Historically, industry has designed manufacturing systems to be as deterministic as possible in deference to the KISS principle ("Keep it simple, stupid"). If non-deterministic systems do not provide superior return on investment, then there is no motivation to pursue them. But we must also consider that the lack of good testing techniques is one of the risk factors that we might reduce if non-deterministic systems can provide superior performance, such as has been observed for some agent-based approaches in limited deployments.

On a more concrete level, we should acquire hands-on experience with the CDL that SEMATECH believes will solve their problems. At present, with no such experience, it is not obvious to us how it would help with the coherency of the system-level specification. We must gain competence with this emerging technique to learn its relevance to the problems of specifying and testing interaction-driven systems, and experiment with automated CDL tools as they become available.

ADL tools, such as Rapide, might also prove helpful. Most ADL work is still academic, showing wide variation even in the definition of what an ADL is and what it is intended to accomplish. Nevertheless, we should investigate them as resources permit, particularly as a possible solution to the protocol versus interface dichotomy that was discussed earlier.

There is, as of yet, no formal method to determine the best combinations of components to test or to emulate in order to achieve a given conformance testing or fault detection goal at the system level. By this we mean a formal protocol for conducting the tests and a formal protocol for when to stop testing, such as have been discussed in reports from Information Technology Laboratory.[94,95] To create an

analogous method that would serve our purposes would be extremely complicated because it would need to combine aspects of existing protocol specification and test methods with aspects of existing interface specification and test methods. These detailed formalisms would be needed to establish a rigorous connection between specified requirements for the system and the testing that is performed. Unfortunately, due to their complexity, combinatorics, and lack of cost effective tools, these formalisms have not exactly "taken industry by storm."[96] A grand unification is even less likely to be well received because the need to integrate disparate methods is likely to require more simplifying assumptions that will further damage the applicability of the resulting unified method. Nevertheless, this remains a possible topic for future work.

Interoperability Testing

One flight test is worth a thousand simulations. —Henry Spencer

Overview

Interoperability is a term that generally refers to the ability of existing products to work together in practice as opposed to theory. The focus of interoperability is the recognition that formal testing and analysis does not guarantee that conforming products will actually work together. Indeed, practical operability sometimes requires explicitly ignoring specifications when they would otherwise prevent interoperation, for example, with other non-conforming components.[97] Such realities make interoperability a very difficult field.

Testing of interoperability involves bringing together existing components and exercising their interoperation. Ideally, as many functionally similar components

94. James Yen, David Banks, P. Black, L. J. Gallagher, C. R. Hagwood, R. N. Kacker, and L. S. Rosenthal, *Software Testing: Protocol Comparison.* <URL:http://sdct-sunsrv1.ncsl.nist.gov/~ftp/stsm/simulationmar98.pdf>, March 28, 1998.

95. Banks *et al*., NISTIR 6129, 1998.

96. Dan Craigen, Susan Gerhart, and Ted Ralston, "Formal Methods Reality Check: Industrial Usage." *IEEE Transactions on Software Engineering*, v. 21, n. 2, February 1995.

97. *Networking Standards: A Guide to OSI, ISDN, LAN, and MAN Standards*, William Stallings, p. 545, 1993.

as possible are tested near simultaneously. Since the focus is on practical operability, the use of test scaffolding is discouraged. All of the components should be someone's actual product. The greater the number of components, the higher the expectation of the interoperability test being meaningful. On the other hand, interoperation between a large number of components guarantees nothing about future component interoperation. It does, however, raise reasonable belief and expectation.

Interoperability testing is very significant for interaction-driven systems because the science of specification-based interaction-driven testing is so meager. Thus, testing of such systems has relied heavily upon interoperability testing to augment more formal testing such as component interface testing.

The Reasons for Interoperability Testing

Interoperability testing is not required in all situations, but any sufficiently complex standard or interface brings with it the opportunity for errors, ambiguities, or incompleteness in the specification itself. Interoperability testing can catch these kinds of problems. Even with a formally verified specification, the rendering into code may introduce problems such that the implementation is no longer valid. (This can be avoided by reference implementations, but these invariably have other problems.)

Another need for operability testing is that conformance testing is rarely exhaustive due to complexity and/or cost constraints. All but the simplest state machines take an intractable amount of resources to be tested with all possible combinations of real-world data.

Interoperability testing also provides the opportunity for additional measuring of performance and reliability without the expense and overhead of full system testing.

Public Relations and Other Political Implications

Interoperability testing presents interesting semi- or non-technical opportunities:

- Implementations designed for interoperability testing tend to be much more robust with the expectation of having to work with other implementations that have entirely different assumptions or interpretations of a specification.
- Interoperability testing gives participants a good feel for the true state of the art at a single instant in time.
- Nondisclosure agreements are easier to swallow “all around” in a neutral forum as compared to having to sign such an agreement when purchasing or simply taking possession of a competitor's component in one's own workplace.

Examples

Computer networking has used interoperability testing very successfully for thirty years. Indeed, many of the interoperability testing practices stem from experiences with network interoperability testing. In the early days of the Arpanet (later to become the Internet), components were built by different contractors. The implementors gathered together on a regular basis for interoperability testing. Participants quickly realized the value of interoperability as a more important goal than adherence to the specifications — to the point that for many years after, the Internet was driven by implementations that resulted in specifications rather than the other way around. The interoperability tests became ritualized in a regular “Interop” conference, now officially known as Networld+Interop.

There are now interoperability forums (i.e., vendor consortia that do interoperability testing) in many areas. These forums provide vendors the opportunity to meet and test their components for interoperation without having to purchase (or simply install) other vendors' products. For instance, forums exist for interoperability testing of video, modems, encryption algorithms, security systems, file servers, as well as a wide variety of networking and communications such as internet commerce protocols and network management protocols. Of special note to the manufacturing domain is STEPnet,[98] where CAD package vendors exchange files in STEP Part 21 format to test the STEP-enabled interoperability between their products.

Many organizations have taken advantage of Internet connectivity itself to allow interoperability on a more *ad hoc* basis. As an example, Bell Laboratories provides HTTP (Hypertext Transfer Protocol)[99] interoperability testing at <URL:http://portal.research.bell-labs.com:8000>. This URL provides access to an HTTP 1.1 server that provides extensive logging to assist testing. This service many be used by anyone at any time and without any advance request.

Frameworks

Despite the use of interoperability testing as a complement to traditional formal component interface testing, interoperability testing can range from informal to formal.

At its most informal, interoperability testing is little more than stepping through functional tests between interacting components. However, highly formalized interoperability testing is possible as well. For instance, the "Interoperability Abstract Test Suite for PNNI (Private Network-Network Interface Specification)" is a rigorous specification that describes the requirements, objectives, test cases, and test set-up for carrying out such interoperability testing.[100] The PNNI test suite is a good example of a dynamic interaction-driven specification.

There is no *de facto* standard framework for interoperability testing. Generally, each area specifies its own. For instance, in the field closely related to PNNI, the ATM Forum has issued a series of its own specifications for interoperability testing.[101,102]

However, there are many references that provide general conformance testing background. In particular, ISO/IEC 9646: The OSI Conformance Testing Method-

98. STEPnet home page. <URL:http://www.stepnet.org/>. 1998.

99. Hypertext Transfer Protocol. <URL:http://www.w3.org/Protocols/>. 1998.

100. Interoperability Test for PNNI v1.0. <URL:ftp://isdn.ncsl.nist.gov/pubs/ATM_Forum_Contributions/PNNI/atm97-0089.ps.Z>, 97-0089, February 1997.

101. ATM Forum Technical Committee, *Interoperability Test Suite for the ATM Layer (UNI 3.0)*, <URL:ftp://ftp.atmforum.com/pub/approved-specs/af-test-0035.000.pdf>, April 1995.

102. ATM Forum Technical Committee, *Interoperability Test Suites for Physical Layer: DS-3, STS-3c, 100 Mbps MMF (TAXI)*, <URL:ftp://ftp.atmforum.com/pub/approved-specs/af-test-0036.000.pdf>, April 1995.

ology and Framework describes terminology and specifications for general conformance assessment.[103] This certainly represents a an appropriate starting point for interoperability testing as well.

Special Noteworthy Problems

Timing-related problems are among the most common errors exposed by interoperability testing and deserve specific mention. Such problems include deadlocks and race conditions. These are generally understood only by computer scientists trained in operating system principles. Alas, communications protocols and interfaces are often designed by personnel who are experienced primarily in their application field and to a much lesser degree in operating system principles if at all.

Specific tools for modeling and simulating timing related issues (e.g., Petri nets) are available but little used. In reality, many of these errors are detected only during interoperability testing.

Future Directions

By its nature, interoperability testing is not something to which NIST, with no product to sell, stands to contribute, except possibly as an impartial third party to conduct the testing. However, we have identified the fact that implementations designed for interoperability testing tend to be much more robust with the expectation of having to work with other implementations that have entirely different assumptions or interpretations of a specification. Interoperability testing might therefore be used as a means for testing the robustness of manufacturing software, whether or not interoperability as such was a design goal. In systems not designed to be interoperable, failures are inevitable, but it is nonetheless desirable for such failures to be detected and resolved in a safe fashion by the system.

103.*ISO/IEC 9646: The OSI Conformance Testing Methodology and Framework*, International Organization for Standardization, 1992-95.

Conclusions and Future Work

In our background study, we found that the art of component testing is better established than the art of system testing, and that interoperability testing is a pragmatic and well-established discipline. We will therefore take different approaches to the different classes of problems in component and system testing, and defer work on interoperability testing until such work appears to be necessary.

For manufacturing *components*, adapting and applying existing black box / infrastructure testing techniques in the new context of the interaction-driven system should be a workable solution. We expect to show this using the SDAI and PDM Enablers specifications. A challenge with these specifications is to define methods which can be re-used as new specifications using the same infrastructures come along. The specification of standards to support manufacturing interfaces is growing at an astronomical rate. Standards are being produced at high levels of abstraction and are intended to be used in combination with other specifications. This is resulting in the problem of a combinatorial explosion of interfaces with no existing methods developed to systematically handle testing of such interfaces.

The growth of modern manufacturing systems into what are effectively “systems of systems” has out-paced the available methods for specifying and testing them. While established rigorous techniques can be used in the context of a single coherent design and development effort (i.e. a single system), the testing of systems that are constructed by “gluing together” generic COTS software, specialized machine control software, and legacy systems is still an evolving art. For these manufacturing *systems*, we do not anticipate finding a complete solution very soon, but there is promising new work to investigate:

- Composable / component-based approaches may simplify the architecture of manufacturing systems to be more intuitive to specify and more tractable with existing specification and test methods. We must see what portions of the com-

ponent-based software work will extend to distributed, interaction-driven systems.

- New specification languages like CDL and various ADLs may help to avoid certain system-level problems and assist with simulation and testing of the systems.
- Formal methods for testing of components and protocols suggest the possibility of a system-level method to decide what combinations of components should be tested together, how they should be exercised, and when to stop testing. However, first we would need to have sufficiently formal system-level specifications to enable such an analysis. With luck, these may become available through the above mentioned work with new specification languages.

In addition, we make the following general observations:

- Testability of many systems is hindered by the low quality or lack of specifications for them. This problem has existed for many years, and we already know that preaching at industry and/or standards bodies to use formal methods does not help. There are some techniques now that are attracting voluntary attention from industry, possibly because they have found an acceptable compromise between rigor and usability. We should concentrate our efforts on these while also gaining proficiency with general-purpose testing tools which can operate in the absence of a formal specification.
- Just as Design-For-Manufacture considerations affect product design, Design-For-Test considerations should influence system design. If a formal method based approach is to be employed for testing, then its usage should start during initial requirements analysis. Furthermore, since utilities which provide automatic generation of tests and/or test scaffolding are often technology specific, the availability of suitable utilities should be considered before system implementation language and environment are chosen. In addition, the testability of the system may be enhanced through the choice of infrastructure technologies, such as the Java language or CORBA, that natively provide introspection capa-

bilities. Early consideration of issues such as these is paramount for enduring system testability.

- The problems caused by integrating components whose specifications do not share a common ontology are not easily revealed or diagnosed by current methods. Therefore, the testability of a system can be enhanced by choosing a common ontology before the system is implemented. This can be as simple as adopting a set of definitions from a pre-existing standard.
- Existing tools for monitoring the communication between components in a distributed system either operate at a very low syntactic level, which makes testing by inspection a thankless chore, or rely on application-specific test scaffolding. We should look into the possibility of defining a standardized, general-purpose inspection interface that components can support in order to permit a test driver to monitor their interactions at a higher semantic level, but still generically.
- The rigorous methods that we have for validating protocols are only usable when the state space is small. Therefore, increased testability can be counted among the many benefits of following the KISS principle in the design of manufacturing systems.

Our continuing work in the area of testability will focus on finding *usable* tools and techniques for constructing high-integrity specifications, defining general-purpose facilities to support the monitoring of interactions at a higher semantic level, and gaining a deeper understanding of the design-for-testability issues affecting manufacturing specifications. Complementing the testability work will be an ongoing effort to develop competence in the testing itself through proof-of-concept testing of dynamic, interaction-driven components and systems. We will develop new tools and techniques where necessary to take advantage of the testability enhancements that we make, or where we may have a solution to an open problem. Otherwise, we will gain competence in using the existing tools and techniques on previously untested systems.

By attacking the problem at both ends, building up our testing capability while simultaneously working to make systems more testable, we hope to find the best compromise for improving the reliability of systems. Neither the testing nor the development of systems should need to go to extremes if complementary improvements are made to each. This more moderate approach may then meet with a

higher level of acceptance and adoption than extremely invasive testing or extremely formal development processes have achieved