NISTIR 6484

Specification of Interactions in Integrated manufacturing Systems

David Flater

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

NIST

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and
Technology

NISTIR 6484

Specification of Interactions in Integrated manufacturing Systems

David Flater

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

March 2000

U.S. DEPARTMENT OF COMMERCE
William M. Daley, Secretary

TECHNOLOGY ADMINISTRATION
Dr. Cheryl L. Shavers, Under Secretary
of Commerce for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Raymond G. Kammer, Director

Specification of Interactions in Integrated Manufacturing Systems*

NISTIR 6484
David Flater

2000-03-10

Abstract

Systems of manufacturing software are often constructed by integrating pre-existing software components. Accurate specification of the component interactions in these systems is needed to ensure testability and maintainability. Moreover, standards for manufacturing systems must specify the interactions to achieve interoperability and substitutability of components. In this report we discuss approaches for specifying component interactions and examine a number of potentially useful specification techniques.

Introduction

Systems of manufacturing software are often constructed by integrating pre-existing software components. The interactions between the components are central to the operation of the system, yet without a specification that unambiguously explains the expected behavior, we have only intuition to tell us whether the observed behavior is correct. STEP, known informally as the Standard for the Exchange of Product model data, has demonstrated the value of rigorously specifying data exchange interactions (for the full story, read *STEP: The Grand Experience*[1]). But data exchange only scratches the surface of the interactions that are possible. Even if we have complete and consistent specifications for the functions provided by two components in a system, these do not necessarily define how the components would work together to achieve a specific goal. If interoperability and substitutability of components is a goal of the specification, then the interactions must be specified completely.

The Testability of Interaction-driven Manufacturing Systems (TIMS) project[2] in the Manufacturing Engineering Laboratory of the National Institute of Standards and Technology has an interest in tools and techniques that are useful for specifying system-level interactions because a specification of correct behavior is a prerequisite for formal testing. In the following sections we discuss different models of interaction and summarize the features of potentially relevant tools and techniques.

Interaction models

Nested control-flow models[3]

Interaction with nested flow of control is a special case of synchronous communication.[4] In these models, communication behaves like a procedure call. Borrowing terminology from the Common Object Request Broker Architecture (CORBA),[5] we would say that the component making a request remains in a blocked state until the response is complete.

Various communication infrastructures following in the footsteps of Remote Procedure Call (RPC)[6] are predisposed to the production of systems having a nested flow of control. Nested control-flow models, hereinafter referred to as nested models, enable us to analyze such systems without the complexity that

---

* Commercial equipment and materials are identified in order to describe certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

would be introduced by more flexible models. The amount of nondeterminism in a nested model is limited by the fact that a component will not receive extraneous events while it is awaiting the response to a previous request.

Flat control-flow models

In systems having a flat flow of control, components are designed around a dispatch loop that never blocks. Because new requests can be processed while previous operations are still pending, the ordering of events in the system becomes more random than it would be with a nested flow of control. Whether you are using "asynchronous messaging" or "event handling" as your model, the architectural ramifications are the same. This is independent of attributes such as payload, indirection mechanisms (subscribe/notify), or support of multicasting that distinguish the various approaches to communication in a "flat" system.

The need for manufacturing systems to be responsive and deadlock-free encourages the use of a flat model. However, flat models are more difficult than nested models from a testing perspective because the actual state of any executing component no longer has a direct relationship to any distributed process that may be in progress. It is also more difficult to specify the intended behavior of event-driven systems because what we see as independent processes may run concurrently in the same component and interact in unintended ways.

The challenge for semantic modeling of flat systems is to identify the behaviors that are intended to be created by the interactions of components, rather than the behaviors of components as seen from their own limited points of view.

Mixed models

Practical considerations sometimes get in the way of taking a purely flat-model approach to system design. Unintended interactions between separate processes, such as competition for a shared resource, force the designer to place restrictions on the sequences of events that the system may process. Resource locking and transactions are two features that are commonly used for this purpose. The introduction of locking and transactions into a flat system can produce a system that sometimes exhibits a nested flow of control.

In the other direction, the introduction of multi-threading into a system having nested control-flow can also produce mixed behavior. A designer might use multi-threading if non-blocking transactions are the exception rather than the rule; this would be simpler than defining all of the explicit locking behavior that would need to go with a flat approach.

Finally, there is the possibility that different components in the system are designed by different people, and some use nested flow of control while others use a flat model. Although the chief architect of a newly built system would be justified in enforcing a standard model, the need to integrate legacy components can ruin the most elegant of plans.

Specification languages and techniques

Architecture Description Languages (ADLs)

Background

ADLs appeared in the 1990s as a promising new formalism in the software domain. However, as the cited reference describes, "There is... little consensus in the research community on what an ADL is, what aspects of an architecture should be modeled by an ADL, and what should be interchanged in an interchange language."[7] The point of commonality in all ADLs is support for modeling the architectural features of a software system at a high level of abstraction.

Features

The cited reference compares ten ADLs and finds that the feature sets vary significantly. Some are designed to enable particular forms of automated analysis while others are primarily intended for abstract modeling. ADLs that enable much automated analysis necessarily have more of the flavor of a programming language or algebra than those designed only for human consumption. The kinds of automated analysis that are available with various ADLs range from deadlock detection to schedulability analysis.

Some ADLs support the modeling of system-level interactions, though it is not their primary focus. Rapide includes built-in support for modeling both synchronous and asynchronous interactions; other ADLs make use of process algebras for modeling components. Process algebras are discussed later in this document.

Like many formalisms, particular ADLs have been used to great benefit in large, isolated projects, but no specific ADL has yet achieved a "critical mass" of industrial usage.

Component Definition Language (CDL)[8]

Background

In the first year of the TIMS project, we identified CDL as having potential use for system-level specification and planned to experiment with CDL tools as they became available. Those plans were jeopardized by events in July, 1998, when the Business Object Component Architecture (BOCA) specification that defined CDL was withdrawn. It was in the process of failing to achieve the 2/3 majority needed for adoption, largely due to the widespread perception that it competed with the emerging OMG Components specification and the Unified Modeling Language (UML).[9]

To rescue the effort already expended, members of the Business Objects Domain Task Force (BODTF) drafted Requests for Proposals (RFPs) for a successor to the BOCA that would be harmonized with Components and UML. Ownership of the new RFPs was transferred to the Analysis and Design Task Force (ADTF), and they were issued in March, 1999.[10] At this time, the refurbished BOCA is still under construction.

Before its "demise," CDL was productively used in several OMG specifications, including Workflow. A CDL-to-Interface Definition Language (IDL) compiler is included in the BOCA IDL Development Kit that Data Access Technologies continues to develop and distribute freely from their web site.[11] The kit also includes a plugin for Rose '98™ (the popular UML modeling package from Rational Software[12]) that translates a UML model having appropriate "adornments" into CDL. (The UML model must be "adorned" with specific attributes to represent the CDL concepts that are not defined in baseline UML.) The UML-to-CDL-to-IDL mapping realized by the toolkit might be submitted as a response to the new RFPs.

Features

CDL provides liberal amounts of "syntactic sugar"[13] for frequently used services like relationships, state, and events. When processed, this syntax expands into IDL that could work with a purchased software library to eliminate some of the repetitive programming that is normally required to use those services.

CDL's approach to system-level issues is best explained by the following quote:

> "While we would like every object implementation to fully encapsulate all behavior related to that object, the reality is that business systems rely on complex interactions between objects and systems. This 'system behavior' is difficult to encapsulate in any one object because the actions of one object frequently depend on the state and actions of other objects. The event model is intended to allow this integrated system behavior without tightly binding component implementations in ways that would make the system brittle and difficult to change."[14]

CDL supports explicit declaration of *dependencies* between objects, with dependency defined as "the requirement an event client has to receive notifications."[15] In the BOCA, all changes to all object features

are tracked as potentially triggering events, so in many cases dependencies can be implemented without any changes to the event producer. Explicit events, called *signals*, are also supported.

Since most *business objects* are understood to be implicitly transactional and persistent,[16] it is not surprising that the events/triggers paradigm supported by the BOCA is similar to that of modern databases.

Component Interaction Specifications (CIS)[17]

Background

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),[18] and by UML Sequence Diagrams[19] and Message Sequence Charts.[20] Unlike the other notations, CIS was designed to be translatable directly into test scaffolding for CORBA systems, but this produced disadvantages that will be discussed below.

CIS is a derivative of the integration testing method that was being used by NIST's industrial partners in the Advanced Process Control Framework Initiative (APCFI).[21] This method, in turn, made use of ideas that are also used in UML Collaboration Diagrams.[22]

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 ...

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.

Features

CIS assumes a nested flow of control for interactions. For CORBA-based systems having a nested flow of control, CIS is a simple and powerful tool. It enables automatic generation of run-time assertions to verify that the inputs and returns for each interaction are as specified in an actual running system. 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 activity. To remove this limitation from the CIS syntax would be easy, but removing it from MCITT's test code generation would require a switch to a more intrusive testing approach.

Finite state machines

Background

Finite state machines (FSMs) are simple abstractions of component behavior comprised of a set of states and transitions between them, with defined criteria for triggering the transitions.

FSMs are the cornerstone of many different specification languages and techniques. For example, Specification and Description Language (SDL),[23] a standard of the International Telecommunication Union (ITU), is now used by a popular software product for realtime modeling, simulation, and analysis. Some impressive software products for FSM analysis are also available free from universities and research laboratories.

Features

Like process algebras (below), finite state machines are a high-level abstraction. They provide a straightforward way to model and analyze networking protocols, embedded control, and other algorithms that lend themselves to finite state analysis without unnecessary implementation detail. The tool support for finite state analysis is very good, enabling properties of systems to be proven or refuted automatically.

With FSM approaches, the focus is on specifying the behavior of individual components, rather than on specifying interactions directly. However, given a complete set of FSMs, all possible interactions and emergent system behaviors can be deduced. They are therefore an efficient tool for modeling systems with flat control-flow, where the interactions of components having simple state spaces give rise to countless possible behaviors depending on the interleaving of events.

FSMs are less useful for applications having a complicated flow of control because the task of identifying finite state spaces for the components of the system becomes complex and error-prone. The most frequently cited reason for failure of finite state analysis is an explosion in the size of the state space resulting from an attempt to model a complex system.

Process algebras

Background

A process algebra is a formal language for specifying and reasoning about the behavior of interacting processes. Many process algebras exist; here we will discuss only a few prominent examples.

Communicating Sequential Processes (CSP)[24] was published in 1978 in Communications of the ACM by C.A.R. Hoare. Since then it has been used as the basis for several parallel programming and specification languages. Among the most notable results are the programming language Occam,[25] which was the language of choice for programming a popular brand of parallel computer, and assorted software for simulating and/or checking CSP specifications for properties such as deadlock freeness.

Milner's Calculus of Communicating Systems (CCS)[26] was emerging around the same time and also became the inspiration for much later work. Both CSP and CCS, in their original forms, assume that processes synchronize on communication, so the domains of systems that they can model are basically the same. However, they have subtle semantic differences.[27]

Language of Temporal Ordering Specification (LOTOS)[28] is an International Organization for Standardization (ISO) standard which was used in the Open Systems Interconnection (OSI) project.[29] It inherits some ideas from both CSP and CCS and is generally preferred for formal specification and verification of networking protocols.

Features

A process algebra can help to separate the essence of an interaction-driven system from the details of any *given* interaction. Using process algebras, it is sometimes possible to construct formal proofs of properties of distributed systems. For reasons that are familiar to the formal methods community,[30] process algebras are not routinely used outside of those domains where formalism is a requirement. However, in cases where the pattern of interactions between software components becomes complex, process algebras can be employed to analyze or even define the specification at a high level of abstraction.

Different process algebras vary in their capacity for modeling time and synchronization, and the level of software support for using different algebras varies as well. Given an algebra with the correct feature set, it is possible to model nested, flat, or mixed control flows.

In general, process algebras permit more direct modeling of communication and synchronization than is possible with FSM-based techniques.

Unified Modeling Language (UML)

Background

UML is really a collection of modeling languages and techniques that have been harmonized with one another and bundled under a common name. It began as a unification of object models, but quickly expanded to enable modeling of many different aspects of a software system. With the support of the Object Management Group and leading software firms, UML has become established as the canonical language for software models.

UML encompasses Static Structure Diagrams, Use Case Diagrams, Sequence Diagrams, Collaboration Diagrams, Statechart Diagrams, Activity Diagrams, and Implementation Diagrams, as well as the recently canonized Object Constraint Language (OCL). OCL first appeared under that name in 1997 as part of a joint IBM[31] and ObjecTime Ltd.[32] response to the first Analysis and Design RFP.[33,34] OCL appears to have evolved at IBM from the Integrated Business Engineering Language (IBEL) and/or the Syntropy method.[35] Other parts of UML are easily recognizable as evolved and assimilated versions of popular, pre-existing computer science abstractions, including Entity-Relationship Diagrams,[36] Harel Statecharts,[37] Petri Nets,[38] and classical flowcharts.

Features

Such an assemblage of modeling techniques clearly aspires to provide every feature that could reasonably be needed. Nevertheless, UML is constantly being extended. Because the UML core has achieved a "critical mass" of industrial usage, it is generally easier to communicate using an extension to UML than using a completely different language.

For specifying interactions, the most applicable part of UML is the Sequence Diagram, which provides sufficient syntax and semantics to model flat, nested, or mixed control flows. There is even a software product that can verify whether the interactions in a simulated system conform to a Sequence Diagram. The syntax is easily extended to include special cases of control flow, such as balking and time-outs, that are not handled by most modeling techniques. These extensions are not understood by automated tools, but they are valuable for specifications that could not be expressed using a more formal, more restrictive notation.

Summary

Having reviewed the state of the art in techniques for specifying interaction-driven systems, we find that there are many ways to do it, but always with a tradeoff between rigor and flexibility. For a system having difficult kinds of interactions, one can obtain a precise characterization of a simplified view of the system that may not be accurate, or a less formal characterization of the system in all of its complexity that may prove to be insufficiently precise or incomplete. We must continue to watch developments as the state of the art matures to enable more complete, integrated modeling of interaction-driven systems.

References

"By selecting these links, you will be leaving NIST webspace. We have provided these links to other web sites because they may have information that would be of interest to you. No inferences should be drawn on account of other sites being referenced, or not. There may be other web sites that are more appropriate for your purpose. NIST does not necessarily endorse the views expressed, or concur with the facts presented on these sites. Further, NIST does not endorse any commercial products that may be mentioned on these sites."

[1] Sharon J. Kemmerer, editor, *STEP: The Grand Experience*, NIST Special Publication #939, U.S. Government Printing Office, Washington, D.C., 1999.

[2] KC Morris, David Flater, Don Libes, and Al Jones, *Testing of Interaction-driven Manufacturing Systems*. NISTIR 6260, http://www.mel.nist.gov/msidlibrary/summary/9827.html, 1998.

[3] The concepts "nested flow of control" and "flat flow of control" as applied to interactions are canonized in OMG Unified Modeling Language Specification version 1.3, http://www.omg.org/cgi-bin/doc?ad/99-06-08, 1999, Section 3.62, "Message and Stimulus."

[4] Michel Raynal and Jean-Michel Hélary, *Synchronization and Control of Distributed Systems and Programs* (English Language edition), John Wiley & Sons, 1990, Section 1.2.2.2, "Synchronization on communication." Originally *Synchronisation et Contrôle des Systèmes et des Programmes Répartis*, Editions EYROLLES, Paris.

[5] CORBA/IIOP 2.3.1 Specification, http://www.omg.org/cgi-bin/doc?formal/99-10-07, 1999.

[6] RFC 1831, "RPC: Remote Procedure Call Protocol Specification Version 2," http://www.faqs.org/rfcs/rfc1831.html, 1995.

[7] Nenad Medvidovic and Richard N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," Technical Report UCI-ICS-97-02, Dept. of Information and Computer Science, U. of CA Irvine, February 1997. Available at http://sunset.usc.edu/~neno/papers/ADL-TSE.pdf. A revision of this report is to appear in *IEEE Transactions in Software Engineering* in 2000.

[8] Data Access Technologies, Inc., *et al.*, *Business Object Component Architecture (BOCA) Revised Proposal: revision 1.2*, http://www.omg.org/cgi-bin/doc?bom/98-07-01, 1998, Section 2.3: Component Definition Language Specification.

[9] UML 1.3, ad/99-06-08, 1999.

[10] PTC Philadelphia Meeting Minutes from March 26, 1999, http://www.omg.org/cgi-bin/doc?ptc/99-03-13.

[11] Data Access Technologies home page, http://www.d-a-t.com/, 1999.

[12] Rational Software home page, http://www.rational.com/, 1999.

[13] "Syntactic sugar," in Eric S. Raymond, *The Jargon File*, version 4.1.4, http://www.tuxedo.org/~esr/jargon/, 1999.

[14] BOCA Revised Proposal, bom/98-07-01, Section 2.2.1.1, "General Concepts," p. 26.

[15] BOCA Revised Proposal, bom/98-07-01, p. 27.

[16] Fred Cummins, *Business Object Concepts*, http://www.omg.org/cgi-bin/doc?bom/99-01-01, 1999.

[17] Component Interaction Specifications, http://www.mel.nist.gov/msidstaff/flater/mcitt/cis.html, 1998.

[18] MCITT home page, http://www.mel.nist.gov/msidstaff/flater/mcitt/, 1999.

[19] UML 1.3, ad/99-06-08, Notation Guide, Part 7, Section 3.59: Sequence Diagram.

[20] Recommendation Z.120 (10/96) – Message Sequence Charts (MSC). Available from ITU, http://www.itu.int/.

[21] Project Brief: Advanced Process Control Framework Initiative, http://jazz.nist.gov/atpcf/prjbriefs/prjbrief.cfm?ProjectNumber=95-12-0027, National Institute of Standards and Technology, 1996.

[22] UML 1.3, ad/99-06-08, Notation Guide, Part 8, Section 3.65: Collaboration Diagram.

[23] Recommendation Z.100 (03/93) – CCITT specification and description language (SDL). Available from ITU, http://www.itu.int/.

[24] C.A.R. Hoare, "Communicating Sequential Processes," Communications of the ACM, Volume 21, Number 8, August 1978, pp. 666-677.

[25] Alan Burns, *Programming in Occam 2*, Addison-Wesley, 1988.

[26] Robin Milner, *Lecture Notes in Computer Science #92: A Calculus of Communicating Systems*, Springer-Verlag, 1980.

[27] Stephen D. Brookes, "On the relationship of CCS and CSP," in *Lecture Notes in Computer Science #154: Automata, Languages, and Programming, 10th Colloquium*, Springer-Verlag, 1983, pp. 83-96.

[28] ISO 8807:1989, *Information processing systems — Open System Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour.* Available from ISO, http://www.iso.ch/.

[29] ISO/IEC 7498, *Information technology — Open Systems Interconnection — Basic Reference Model.* Available from ISO, http://www.iso.ch/.

[30] Dan Craigen, Susan Gerhart, and Ted Ralston, "Formal Methods Reality Check: Industrial Usage," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, February 1995, pp. 90-98.

[31] International Business Machines home page, http://www.ibm.com/, 1999.

[32] ObjecTime Ltd. home page, http://www.objectime.com/, 1999.

[33] IBM/ObjecTime Ltd. joint submission on AD RFP1, http://www.omg.org/cgi-bin/doc?ad/97-01-18, 1997.

[34] Object Analysis and Design PTF — RFP1, http://www.omg.org/cgi-bin/doc?ad/96-05-01, 1996.

[35] Steve Cook and John Daniels, *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Prentice Hall, 1994.

[36] Peter P. Chen, "The Entity-Relationship Model – Toward a Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, no. 1, 1976.

[37] David Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, 1987, pp. 231-274.

[38] W. Reisig, *Petri Nets: An Introduction*, Springer-Verlag, 1985.