14 An Architecture for Integrating Enterprise Automation

Jeffrey A. Barnett and Stephen P. Smith
Northrop Research & Technology Center[†]

Abstract

Justin is a software toolkit used to build functioning prototypes of integrated automated enterprises. It provides several languages to describe the life-cycle progression of design objects and the policies and procedures that are to be in effect during their development. Interpreters for these languages assure that object evolution is consistent with the relevant constraints. Similar mechanisms can provide the necessary flexibility and robustness to build the integrated automated enterprises of the future. The architecture of Justin and the architecture it induces on the systems and prototypes constructed with its toolkit are discussed.

14.1. The Problem

National programs like the concurrent engineering initiative are changing and will continue to change normative business procedures and practices. Thus, the capability to easily modify enterprise systems is important as is the ability to use prototypes to understand the potential impact of these changes.

The conventional wisdom is that the automation belonging to the hierarchy of customers and suppliers that comprise an enterprise will be interconnected in future systems. It is also assumed that multiple enterprises in the same corporate entity will be interconnected. However, there are variations between different enterprises and different corporations because of customer differences and because different sorts of products have different life-cycle progressions. Therefore, it is necessary to develop technological approaches that cater to the variety of needs exhibited by multi-enterprise corporations, their customers, and their suppliers.

[†]The Research Center is located at One Research Park, Palos Verdes Peninsula, CA 90274. Author internet addresses are jbarnett@nrtc.northrop.com and ssmith@nrtc.northrop.com.

Companies are currently automating functional areas in a patchwork fashion in an attempt to improve overall productivity and quality. However, this leads to an ever increasing effort to provide coordination and to develop supporting paperwork standards. Hence, there is a need to intelligently manage the flow of technical information and its associated documentation within an enterprise.

Joint DoD and industry efforts are under way to resolve pieces of this problem. The DoD is promoting the Computer-aided Acquisition & Logistic Support Initiative (CALS), in an attempt to replace paper document transfers with digital file exchanges by the mid 1990s. The efforts of the ISO TC184/SC4/WG1 Committee to evolve the Standard for the Exchange of Product Model Data (STEP) is an example of an industrial initiative. STEP promotes a normalized descriptive format for all product-related data. The proceedings of CALS Expo (1990) discusses the current status of, and future directions for, CALS and STEP development.

However, the major problems are not simply matters of data format conversions; rather, the substantive issues are how best to share data and control communications to integrate enterprise automation. Data becomes valuable only when it is at the right place at the right time and there are mechanisms to properly interpret and use it. CALS and STEP address the first class of these problems but not the others.

Some research efforts, e.g., Finger et al (1990) and Klein and Lu (1989), model interactions among technical organizations during the design process. The goal of these efforts is to infuse some of the technical knowledge used by the various engineering disciplines into future design-support systems. However, the models incorporate little or no understanding of the policies and procedures necessary to control the design process or its connections to other facets of a product's life cycle.

Lee and Malone (1990) describe methods to reconcile the different ways that different disciplines represent similar concepts. This work is an "AI" version of STEP because it assumes use of the knowledge necessary to automate the exchange of informal information among engineering organizations during the design process.

Katz (1990) describes a methodology to automate design version control. That technology addresses the relations among designs, versions, and releases but little knowledge of the policies and procedures that govern product development is included in the resulting systems. Knowledge of organizational structures and how they may vary from enterprise to enterprise is missing as well. Roboam, Fox, and Sycara (1990) tackle some of these issues by carefully modeling the decision-making process, as they have observed it, in large organizations. Both efforts extend IDEF (1981) concepts to better model parts of manufacturing enterprises.

The ability to use the policy and procedures of an enterprise to automate the movement of information through the enterprise is a key missing component of previous work. That is, policies and procedures for both automated and unautomated functions must govern what, where, when, and by whom product data is manipulated. Unless this occurs, current patchwork integration will continue to result in higher product cost, lower product quality, and increased lead times.

14.2. Summary

Justin is a set of tools and languages to construct functioning prototypes of complicated enterprises. The languages are used to represent the policies and procedures of an enterprise along with the specification of the coordination strategies necessary to ensure orderly progress of products through their life-cycle states.

The next section introduces Tinker Inc., a mythical corporation, and some of the problems it faces automating functions and implementing new business procedures. Subsequent sections describe the Justin toolkit, the languages it provides, and how the combination provides an approach that addresses Tinker's problems. A functioning Tinker prototype using this technology is described as well. Finally, the architecture induced upon systems constructed with the Justin toolkit is discussed.

This article is a snapshot of the ideas and progress made, through late 1990, by an ongoing research project at the Northrop Research and Technology Center. It is an update of an earlier version, Barnett (1990b), presented at the SIGMAN Workshop held in conjunction with the AAAI conference in Boston that year.

14.3. Tinker Inc.

The organization and business practices of Tinker Inc., a mythical company, are discussed in this section. A prototype of the company, described later in this article, was created using the Justin tools and languages. The resulting implementation integrates several pieces of application software and enables concurrent engineering practices during a portion of the product life cycle.

Tinker Inc. designs and prepares manufacturing plans for a simple class of products: planar trusses manufactured from rods and end connectors. Figure 1 is a CAD drawing of a typical design. Customers specify general characteristics of a desired product to a Tinker project manager. The project manager writes the Design Requirements Document and determines whether the new design should be realized as a version or modification of an old one. The project management team retains review and product

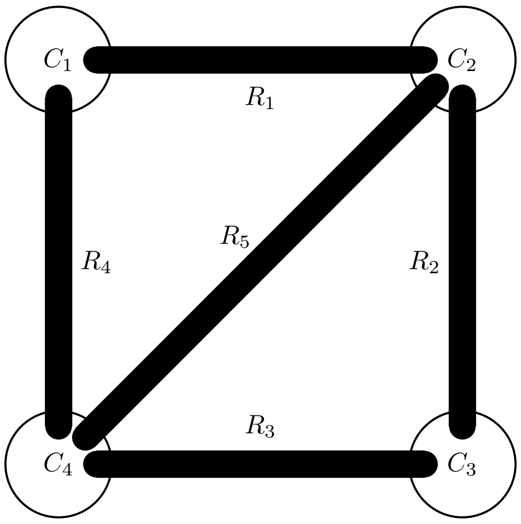

Figure 1: TCAD drawing of a planar truss.

sign-off authority.

Several engineering organizations provide support to project managers. In addition to design engineering, there are structural and producibility analysis groups, and an organization that produces process and assembly plans. Another group prepares parts lists. Each organization uses specialized automated tools suited to its particular function and most use a common CAD system, TCAD, developed by the centralized Tinker Information Services Center (TISC).

In spite of universally high regard for Tinker engineers, the corporation imposes constraints on their activities. These constraints are contained in the Policy and Procedure Documents (PPDs). One document provides the details of the review and sign-off authority mentioned above. Another PPD provides the guidelines for design analyses: When design drawings are completed, copies are sent, simultaneously, to both the Structures and the Producibility Organizations for advice and consent. This is a relatively new procedure. In the past, the completed design was sent to Structures and changes were iterated with the Design Organization. Producibility was consulted only after this step was finished.

Tinker is a very modern, high technology company so the initial experiments with concurrent engineering were done early on. The experience was positive and they are in the process of updating the PPDs to permit more concurrent procedures. Dr. Tinker, PhD., the company's founder and CEO has always said "*Rules are rules!*" and, therefore, the company can only change its internal processes when new PPDs are in place. Thus, this is one corporation that believes good technology and good practice are compatible.

When Tinker decided to install new business procedures, TISC noted that the corporation's ability to perform would be enhanced if the automatic systems used by the various functional organizations could interact. However, it was unclear how the new business practices and the integrated automation should coexist. A typical problem is routing job folders to the proper organizations at the proper times. Approximately 70% of Tinker's PPDs deal with this problem and the entire function of the Document Control Organization (DCO) is to ensure that these PPDs are obeyed. How does the DCO operate when large portions of the data are in computer systems and these systems share some of that data, as a common resource, among several organizations?

At this point, TISC employed an outside consultant who recommended that the Justin system be used to solve some of Tinker's problems. He claimed that the company would benefit in many areas because the policies and procedures of Tinker become part of any system built with the Justin toolkit. Therefore, the PPDs can control many aspects of the behavior of the integrated system.

The consultant mentioned two other benefits: The first was that Justin provides mechanisms to assist in the integration of multiple disparate software systems and in the implementation of user interfaces that can interact with these multiple systems simultaneously. The second was that functioning prototypes can be implemented from the same pieces used to build functioning systems. Such prototypes permit low-cost experiments to quickly determine the impact of different policies and procedures. Since all of these advantages are important to Tinker management, they decided to use Justin. They believe this will maximize the benefits and reduce the cost of their planned automation efforts.

The following sections describe the Justin tools and their use to build working prototypes of Tinker Inc.

14.4. Approach: Enterprise Modeling + Tools

An enterprise produces products. Products, whether they are physical, such as aircraft components, or abstract, such as requirements documents and designs, are formed by their orderly movement through functional organizations that add value to them. En-

- The essence of enterprise integration is captured: Technical information flow is coordinated using the policies and procedures of the enterprise to assure process fidelity.
- The STEP objectives are met: The technical information required for product development can be captured.
- Exceptions to normal functional flow are explicitly represented: The basic building blocks to implement change management, version control, and many other project management functions are provided.
- The data stewardship of each functional role is explicitly represented: The management policy, with regard to data integrity and authority, is obeyed because the system is aware of and can enforce the relevant constraints at the proper time.

Figure 2: Advantages of enterprise modeling.

terprise integration takes place by properly providing for product movement strategies that follow relevant policies and procedures of the enterprise and its customers.

Our theoretical approach is to model an enterprise by specifying both its functional entities and the abstract objects that flow between these entities. (Physical objects are represented by abstract objects in the system.) Each such object has an associated set of life-cycle states that constitute all of its potential behavioral modes and encodes all of its relevant policies and procedures. Each state specifies valid interactions of an object with other objects as well as its interactions with individuals, organizations, and their specialized tools. Figure 2 summarizes several advantages of implementations that include enterprise models in their knowledge base.

Justin is a set of computer tools to build functioning prototypes of manufacturing concerns. Its capabilities provide technology to integrate automation and to deal with many of the attendant issues. These capabilities represent an approach useful in real systems as well as prototypes. Similar theoretical approaches are described by Karbe and Ramsperger (1990) and Lochovsky, Woo, and Williams (1990). The difference is that Justin includes mechanisms to **integrate software** as well as incorporate policy and procedure knowledge.

Justin provides several formal languages to define enterprise models. Interpreters generate and/or constrain behavior as described by the models written in these languages. The Control Regime and Critique Regime languages are used to describe object life-cycle progress as well as policy and procedures: The *Control Regime* language is used to describe the proper disposition of design objects, e.g., requirements, drawings, or change requests, as functions of documentation associated with the objects and the state transitions of other related objects. The *Critique Regime* language is used to prescribe which classes of individuals can make what kinds of modifications and/or additions to the documentation associated with each type of object.

In addition to these two languages, the system provides descriptive mechanisms to specify software interfaces to other significant subsystems. For example, the interfaces between control regimes and particular CAD systems are introduced this way.

Together, the Control and Critique Regime languages, along with the software interface specification mechanism, provide a reasonable technology to specialize Justin-built prototypes to new products and processes. Models written in these languages provide a description of the proper behavior and interactions of enterprise components. Similar languages will be useful to future builders of integrated automated manufacturing enterprises to gain desired flexibility and robustness.

14.5. The Justin Toolkit

Figure 3 shows the tools provided by Justin and the data flow among them. The main component is the **object controller**. Its functionality is similar to that of a configuration manager and data controller in one of today's manual systems. Thus, it provides services that catalogue documentation, and distribute and disseminate that documentation as determined by corporate and/or customer policies.

The **activity builder** integrates activity-specific software with the run-time components of Justin and provides the combination through an interactive interface. An *activity* is an individual or organization in an enterprise that performs a specific, known function in a product's life cycle. An activity receives a copy of a data record from the object controller. The information in the message is treated as a work assignment by the recipient. Its task is to determine updates to the record that appropriately reflect the results of the activity's function. For example, a structural analysis organization appends critiques describing the structural properties of proposed designs and a summary of the data calculated to support their conclusions.

The other construction tools—preprocessors for the Justin languages—are not discussed at length. Their function is to perform static syntax and semantic analyses, typical of compilers, and translate language forms into representations that are efficient at run time. Next, the object controller and the activity builder are described.

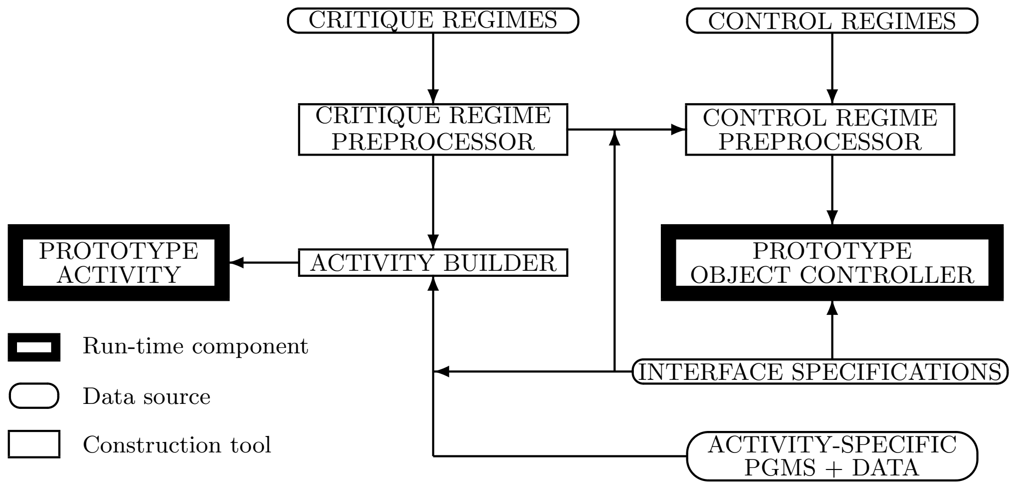

Figure 3: Data flow among Justin prototype-building tools.

14.5.1. The Object Controller

The object controller is the run-time coordinator of any system or prototype built with Justin. Internally, it is a combination of database methods and interpreters for control regime instances. The controller can run entirely automatically. However, it is possible to use it to interactively monitor the status and state transitions of design objects. In this mode, it is possible to add annotations to the critiques and influence control decisions. The relevant critique and control regimes place limits on these capabilities. The activity builder, described below, was used to implement much of the object controller.

The database organization is simple: the data record for each design object is a property list, i.e., a list of property name/property value pairs. Each record represents a design object and the control regime instance for that object. The object controller stores the interpreter's state for the object instance in the data record associated with it. Therefore, the controller can consult the state of a control regime instance to determine whether a proposed update is legitimate and the actions to take when the update arrives. This approach provides some measure of database security and integrity as well as a primitive daemon mechanism—control regimes can specify selective actions to perform in response to particular sorts of data modification requests. These actions include sending messages to activities and other objects and, hence, help to synchronize and coordinate the state of the entire system.

The object controller and the activities communicate through a byte stream protocol, named CADCAM, built on IP and TCP. This protocol is provided as part of the Justin toolkit. The object controller transmits a copy of a data record to an enterprise activity when directed to do so by the control regime. The activity performs its function on that data, then sends a message to the controller when it has finished. The message describes updates to the original record stored by the object controller and specifies: 1) the data record to update, 2) the property set to update in that record, 3) the data to use for the update, and 4) the update operations, e.g., replace or append, for each property in the set. Update sets are processed atomically to avoid race conditions.

The object controller is the repository of the stable version of all design objects and can mediate all changes to them before the modifications become visible to other parts of the system. Since the controller is the first to observe these changes, it can determine the proper continuation of an object's life cycle and prevent incorrect dissemination.

14.5.2. The Activity Builder

The activity builder is the primary Justin tool to assist in the task of software integration. It is used to construct interactive interfaces as well. It is implemented as an extension to the Dynamic Windows (DW) Presentation Manager, Symbolics (1990). Figure 4 is an abstraction of an interactive interface made by the activity builder. Individual software modules contribute panes (subwindows) to the interface. In the figure, the software associated with activity and design object identifications (essentially title information), critiques, interactions, and menus panes are part of the Justin run-time library. The CAD pane is normally associated with software that is used throughout an enterprise; however, it must be integrated with most individual activities. The other panes are associated with activity-specific applications.

The activity builder interprets forms that specify how window panes for one application are geometrically integrated with those of others. Basic presentation types are defined as well. These types are used to provide mouse sensitivity among the panes that comprise an application's interface. Thus, simple semantic coupling between programs is available because applications can partially share interaction cues without necessarily being aware of the low-level details of each other's implementation.

In addition to facilities to integrate interactive interfaces, the activity builder provides communication facilities and local database services. Figure 5 depicts the layering of this software with the activity-specific applications. The bottom layer receives data records from and sends updates to the object controller. It provides buffering and syn-

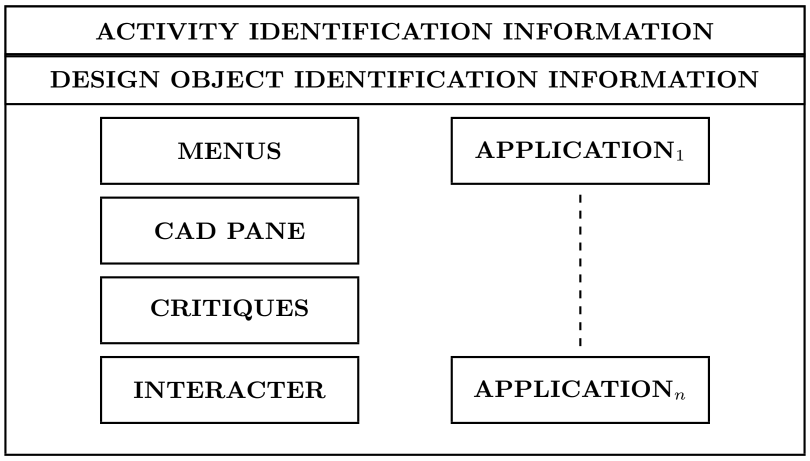
ACTIVITY IDENTIFICATION INFORMATION
DESIGN OBJECT IDENTIFICATION INFORMATION
MENUS
CAD PANE
CRITIQUES
INTERACTER
APPLICATION$_1$
APPLICATION$_n$

Figure 4: Sketch of an interactive interface window.

chronization services as well. If multiple design objects are sent to the activity, this software informs the interface and allows the user to switch focus of attention among the resident set. In case unrecoverable errors occur, the software in this layer captures state information and sends it to the object controller for safekeeping. The activity can be restarted later when the problem is repaired.

The local control layer is built on top of communications. It provides window management, process control, and data coordination services. The window manager integrates keyboard, mouse, and presentations among the window panes shown in Figure 4. Process control coordinates use of the CPU resource for the activity. User input, e.g., a moused menu item, and explicit scheduling by the software itself jointly determine the proper component to execute. Since the pieces of the interactive interface share input/output resources, the window manager must cooperate with the process controller to make these decisions.

The data coordinator provides access to data object copies resident at the activity. This software is responsible for observing local modifications so that the object controller can be informed when activity processing is complete. Some limited type and format conversion are done as well.

The local data controller is being extensively redesigned and reimplemented at the time of this writing. Two major changes are planned: The first modification is to in-

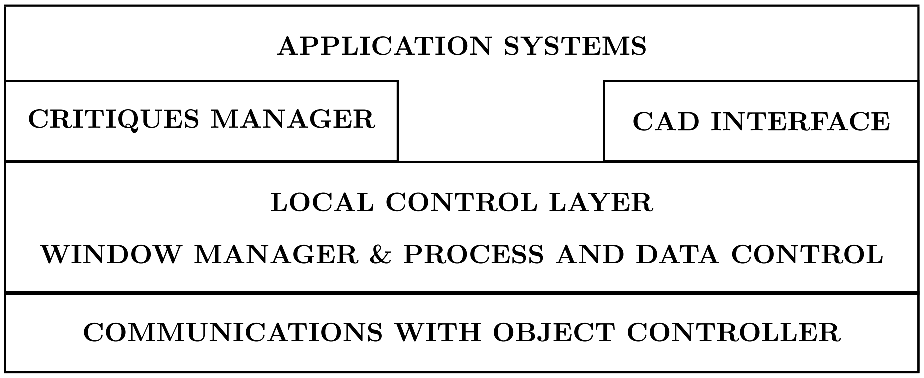
APPLICATION SYSTEMS
CRITIQUES MANAGER
CAD INTERFACE
LOCAL CONTROL LAYER
WINDOW MANAGER & PROCESS AND DATA CONTROL
COMMUNICATIONS WITH OBJECT CONTROLLER

Figure 5: Software layering of a Justin-built activity.

crease coupling with the control regime so that more uniform enforcement of policies—those limiting data access—is possible; The second change is to use the Software Interface Specifications language described in the following section.

The third layer of activity software consists of the critiques manager and the CAD interface. Application programs can append critiques to describe the results of their processing and the user can interactively augment and edit the contents of the critiques pane. The critiquing capabilities of application software, like that of interactive users, are circumscribed by the critique regime. The critiques manager interprets the critique regime to enforce these policies.

The CAD interface integrated with the activity is responsible for hiding choices of low-level implementation details. For example, the prototypes implemented with the current Justin system store CAD data in the record associated with each design object. Since the amount of this data can be enormous, this is not practical in many cases. The approach in future releases of Justin is to only store a *path* to the CAD data. The CAD software in the activity accesses a CAD server on the network to retrieve whatever is necessary for this particular activity. In some cases, e.g., when the CAD system uses X-windows (Scheifler and Gettys 1990), a pane can be directly controlled by the remote server. Thus, the CAD interface is singled out from among all of the non-Justin software shown in Figure 5 because it must cooperate with the local data manager to maintain smooth data access abstractions.

The top layer of activity software is comprised of the application-specific programs used by the activity. The other layers provide relatively straightforward methods to integrate these routines with each other, the general software packages of the enterprise, and those that are part of Justin.

14.6. Justin Languages

Justin provides Control and Critique Regime languages to express enterprise models. The models constrain and coordinate the run-time components (the prototyped activities and the object controller) shown in Figure 3. In addition, linguistic mechanisms are provided to describe software interfaces between these components and other existing systems. The Justin languages are discussed in this section.

A system that integrates enterprise automation, like any intelligent system, must reason about the flow of control among its components. Davis (1980a and 1980b) argues the case for rule- and knowledge-based systems: control knowledge makes system behavior more efficient, rational, explainable, and, hence, more acceptable to its users. These criteria are just as important to the users of enterprise automation. The specialized languages provided by Justin permit much of the relevant control knowledge to be encoded for the types of applications discussed here.

A problem not addressed below is that the policy and procedure guides for large corporations are typically multi-volume collections produced over a period of years. As such, they represent a major investment in time and thought. It is not trivial or inexpensive to recast them in a form consumable by a Justin-like system. Not only is the representation media different but the fact that so much is automated, means that these policies and procedures must be reconsidered.

The cost of rethinking basic approaches to coordinate and control enterprise activities is necessary no matter which integration approach is chosen. We believe that the types of linguistic mechanisms described below can actually lessen that cost. However, policy makers must be educated to think in terms of more formal specifications and intermediaries must be trained to help them make the transition. Policy formulation, review, and maintenance is, and will continue to be, a full-time activity.

14.6.1. Control Regime Language

It is natural to view the life cycle of an object as a progression of phases. For example, the DoD typically names the first few phases of major systems as *System Acquisition*, *Program Initiation*, *Concept Exploration*, *Demonstration Validation*, and *Program Go Ahead*. (See the Systems Engineering Management Guide (1986) and MIL–STD–483A

```
(analyzing (SEND structures-organization :CLASS analyst)
           (SEND producibility-organization :CLASS analyst)
           (WAIT (AND structures-token producibility-token))
           (CRITIQUE-CASE analysis-signoff
             (veto (ENTER review))
             ((okay warn) (ENTER analyzed))))
```

Figure 6: Example state in the Control Regime language.

for more information.) Major life-cycle phases are divided into states, e.g., Demonstration Validation includes the *design* and *analysis* states among many others. The progression through these states and phases is not linear except as an idealization; Feedback and iteration are typical.

The policy and procedures that govern major developments vary from one life-cycle state to another. In fact, the capabilities of an individual are different at different times. Consider an engineer who is responsible for the development of a particular component. During the design state, he has the authority to approve design revisions. Following analysis, he can sign-off and release the product but he can no longer approve revisions—the mechanism to do so now involves many other organizations. Thus, the current authorities that an individual has are intimately tied to, and determined by, the current life-cycle state. Therefore, knowledge of that state is essential to any agent who tries to ensure that proper policies and procedures are obeyed.

A control regime is represented as a state machine—a collection of states where each state is a named sequence of statements. Figure 6 is an example of a state named `analyzing` written in the Justin Control Regime language. A state name is typically a label for an epoch in the object's life cycle. Individual statements can: 1) Transfer control to a specified state—generally indicates the beginning of a new life-cycle phase; 2) Send a database record to a specified enterprise activity—initiates an appropriate action and response by the receiving organization; 3) Conditionalize the execution of other statements—selects among possible futures; and 4) Wait for a specific condition to occur—synchronizes actions and events.

Symbolic tokens as well as database updates are sent when activities communicate with the object controller. The Control Regime language provides a type of conditional statement that is sensitive to these tokens. Wait statements delay the control regime interpreter, for a particular design object, until tokens satisfying specified conditions arrive. Wait conditions are expressed as combinations of token names and `AND` and `OR` operators. The operators are logical connectives and token names denote reception of a token by that name. Thus, the statement `(WAIT (AND x y))` causes the interpreter to pause until two tokens, one named `x` and the other named `y`, are received. Another kind of conditional, a `CRITIQUE-CASE` described later, is provided to choose subsequent behavior depending on the commentary added to the object by the activities.

Figure 6 shows a fragment of the `analyzing` state for the Tinker prototypes. Copies of the data record are sent to the Structures and the Producibility Organizations when the state is entered. The interpreter pauses until both activities respond—each sends the object controller a token, indicating that it has completed its processing, along with updates, to the analysis critique collection, and other data. Depending on the contents of the critiques, the control regime will either forward the information to project management for review, or initiate the next state (`analyzed`) in the design's life cycle.

The example does not show control regime declarations. They describe the default portions of data records that `SEND` statements transfer to activities. Declarations also indicate how particular instances of destinations (e.g., the Structures and Producibility Organizations) should be determined as functions of other object attributes.

The nature of state languages and state machines make control regimes **sequential** because an object only can be in a single state at a given moment. However, a state machine can initiate **parallel** external activities by sending work assignments to multiple destinations as shown by the example. A sublanguage, an augmentation of the Control Regime language, is being developed to coordinate many low-level details of concurrent implementations.

A specialized representation for concurrency control can reduce the number of states necessary to coordinate the behavior of multiple objects. If n objects can each be in any one of s states, then there are s^n possible states for the combination. Another reason to use a specialized representation is that it can abstract some of the issues concerning organizational structure and personnel policy. For example, representing procedures to deal with reorganizations, vacations, and employee terminations, can exponentially increase the complexity—number of states—necessary to describe concurrency. Additional motivation and details of concurrency control for Justin are described by Barnett, Cass, and Betts (1990).

14.6.2. Critique Regime Language

The Critique Regime language is used to code both forms and authority control. As a forms controller, a critique regime determines what collections of commentary are associated with which types of objects. As an authority controller—the primary function of a critique regime—the regime specifies who can do what to that commentary.

The critiques added to design objects by the activities are generally written in natural language for human consumption. Unfortunately, today's technology is not sufficiently mature to automatically understand critiques written this way. Therefore, the critique creator is required to attach a label, such as `warn` or `okay`, to each critique. The `CRITIQUE-CASE` statement in Figure 6 conditionalizes behavior on the appearance of such descriptive summaries. These labels, called *forces*, are the only parts of critiques that are referenced by control regimes and, hence, by the object controller. Therefore, forces are the formal carriers of critique meanings in Justin. A similar approach is described in Lai et al (1988).

Critique regimes define critique collections and application classes: A *critique collection* is a coherent set of commentary governed by a uniform set of rules and an *application class* is a job type such as designer, reviewer, or analyst. Regimes also specify, for each application class, the capabilities individuals in that class possess to create and/or modify critiques in each of the defined collections. Force sets are defined as enumerated types and critiquing capabilities are parameterized by them. For example, the capability to add a critique specifies—explicitly or by default—which forces can be put on a critique, in a particular collection, by members of a particular application class.

The combination of forces with critique collection and application class definitions makes the description of many policy aspects straightforward. For example, the policy guide might say that analysts can either sign-off on a design, sign-off with a warning of a possible problem that does not appear to affect form, fit, or function, or refuse to sign-off at all. Thus, the critique regime specifies that 1) there is an `analytic-signoff` collection, 2) there is an `analyst` application class, and 3) `analysts` can add critiques to the `analytic-signoff` collection with a force of `okay`, `warn`, or `veto`. This capability is not allocated to the `design` application class because the policy and procedures, in the example, say that only `analyst` have it. Since most authorities are specified in terms of job class, job assignment, and a discrete set of meaningful actions, the Justin scheme provides a powerful and natural way to express them.

Control regimes are built on top of critique regimes—declarative information in the former incorporate one of the latter. When a send primitive transmits a copy of a design object to an activity, critiquing capability appropriate to that activity is transferred. The capability specification is done by naming a defined application class. For example, both `SEND` statements in Figure 6 designate the `analyst` class. Thus, both structural and producibility analysts are restricted to the critiquing capabilities prescribed for an `analyst` by the relevant critique regime.

The interplay between the capabilities and constraints expressed in the Critique Regime language and the information about object disposition expressed in the Control Regime language makes it relatively straightforward to encode many types of enterprise policies and procedures.

14.6.3. Software Interface Specifications

Functioning enterprise prototypes, like real systems, must interface with many existing pieces of complicated automation. Some, such as CAD systems, are used by multiple activities. Other pieces are the tools of a specific activity, e.g., the rigidity analysis program used only by the Structural Analysis Organization at Tinker Inc. Justin provides a simple method to specify interfaces to both. Essentially, the system programmer describes argument/value patterns for the interfaces to the individual software systems. The control regime uses this information to validate invocations of the software by the Justin system and to partially automate some simple data conversions.

The Justin languages are currently evolving from typeless semantics, typical of many Lisp dialects, to more strongly typed representations. This move allows the correctness of systems and prototypes to be statically checked by the compilers before regimes are actually used. The interface specifications are part of this evolution; they provide some of the necessary type information.

14.7. Current Status — A Tinker Inc. Prototype

A version of the Justin tools is operational and has been used to construct Tinker Inc. prototypes. The current implementation is restricted to Symbolics Lisp machines. The introduction of software interface specifications, as described above, will make it easier for future Justin prototypes to integrate existing code that executes on a variety of other platforms as well.

Figure 7 shows the architecture of one Tinker Inc. prototype built with the Justin toolkit. The company is composed of three divisions: Project Management, Manufacturing, and Design Engineering. Since concurrent engineering is in vogue, a design can be processed simultaneously by organizations in more than one division as described below. In spite of this, many functions have been automated **and** they operate according to reasonable policy and procedure guidelines set down by management.

The object controller in Figure 7 is a component of the Justin toolkit shown in Figure 3 and the organizations are implemented using the activity builder. The controller is responsible for many data control and dissemination functions, e.g., routing design

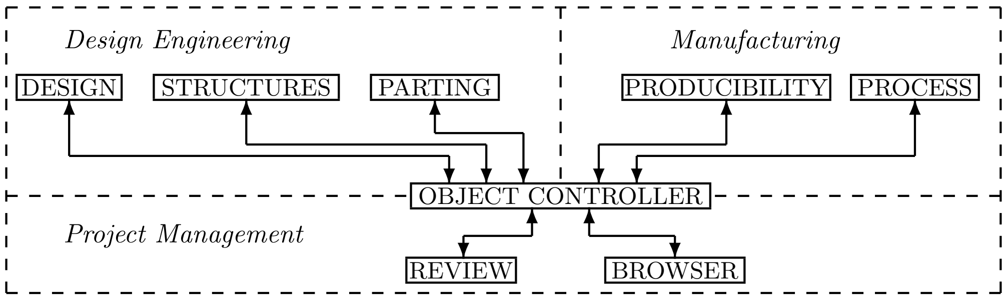

Figure 7: A Tinker Inc. prototype built with Justin tools.

documents to a proper review authority when a conflict between organizations occurs. These tasks are performed using a copy of the Tinker policy and procedures formally coded in the Control and Critique Regime languages provided, along with the object controller and the activity builder, as part of Justin. Thus, it performs most of the functions of the Document Control Organization (DCO) described earlier.

Each organization in the prototype enterprise uses software that is unique to its function and the choices of the individuals that it employs, e.g., the Structures Organization must select an analysis package that does the appropriate checks on planar truss designs and is easy to use given the background and training of its analysts. Other software found at the enterprise is common to many groups. For example, TCAD is used, either to create or review designs, by most functional organization.

The functional organizations of the three divisions are described below in the approximate time order in which they typically interact with a design during its life cycle. The actual order is determined by the policy and procedure of Tinker Inc. as coded in the languages understood by the object controller: The control regime is the primary determinate of this order and conditionality is a reaction to critiques whose variability are constrained by the critique regime. Figure 8 shows an idealization of the work-flow of a design through Tinker. It is an idealization because it does not show the feedback paths that occur when there is conflict among the organizations.

Each organization shown in Figure 7 is a separate process and each can be executed on a separate processor or several can share a host. Several designs can be processed simultaneously by the prototype and the same design can be simultaneously processed at several workstations. The activity builder was used to implement the organizations—each is an activity. The interactive interfaces coordinate enterprise software such as

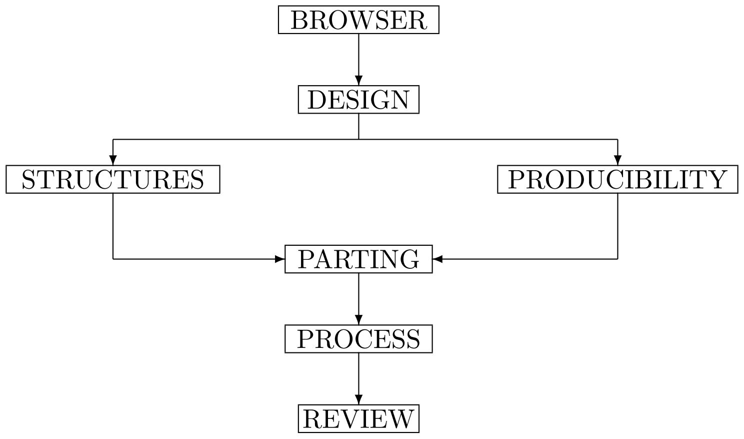

Figure 8: Idealized work-flow sequence at Tinker.

TCAD with special-purpose packages used by the individual organizations and enterprise policy, as encoded in control and critiques regimes, is enforced.

The **Browser** allows project managers to view the current state of any truss design in the system. It also displays the tree formed by using the "is-a-version-of" relation to form edges. In addition to viewing design states and the design tree, users can initiate new designs or design versions with the assistance of the Browser. The critiques added during initialization are the new entity's requirements documentation.

The **Design Organization** provides the interactive capabilities necessary to edit and create truss drawings. Part dimensions and materials usage are specified as well. Thus, this is the engineer's design station and the TCAD tools needed to specify new products are integrated here. The engineer prepares preliminary product designs which are routed to the analysis organizations as determined by the control regime. If problems are detected, the review organization is consulted and the engineer may be requested to improve his design in response to this feedback.

The **Analysis Organizations** are Structures Analysis and Producibility Analysis. Each organization simultaneously receives design documents, including CAD models, when the Design Organization is ready for comments. After analyses are completed, the design either passes to the next stage of its development, returns to the designer for improvement, or goes to the Review Organization for conflict resolution. The policy of Tinker Inc. dictates which action is appropriate based on the critiques added by the analysis organizations. The prototyped organizations can operate entirely automatically or in an interactive mode. The user interface for each organization integrates the common TCAD and critique software with the particular analysis packages needed by that organization.

When designs arrive at the Structures Analysis Organization, specialized programs check stability criteria, e.g., connectivity and rigidity. These programs add analysis critiques to the design. In automatic mode, the design is returned to the object controller immediately. In interactive mode, the analyst is given an opportunity to perform additional tests and to edit the critiques before the analysis is finished.

The Producibility Analysis Organization is similar to the Structures Organization. The difference is that the programs integrated here are concerned with manufacturing constraints. For example, checks are made to ensure that there are sufficient gripper positions for robots to manipulate the parts during assembly and that the product size does not exceed workbench area.

The **Parting Organization** canonicalizes parts representations and produces parts lists. This is so straightforward an activity that it is fully automatic in the prototype: a user is not given an opportunity to override its output. The software used by this organization is an "automated parts catalogue" and the integration task is to apply it to designs at the proper stage in their life cycle. The control regime defines that time to be after design and analysis are completed and ensures that this organization processes the design before it is sent to the process planners.

In the old days, Tinker employees manually generated parts lists with the aid of a simple automated support tool. They formatted the output using a standard template. Since this is such a straightforward operation when the systems are integrated, an entire organization has been eliminated. This is one of many benefits that Tinker got from its enterprise integration efforts.

The **Process Planning Organization** is fully automated but it allows the user to interact and suggest plan modifications. When a design with canonicalized parts representations is sent to the planner, an assembly plan is produced under the assumption that two one-armed agents (human or robot) will do the construction. The user can see the plan in either a procedural language format or as an animated display of the assembly steps. Barnett (1990a) describes some of the issues that such an automatic planner should address.

The automatic part of the planner is a set of planning specialist (knowledge sources) that determine assembly order. These specialists express their preferences as weighted votes on what to do next. The votes are represented and combined using the Dempster/Shafer belief calculus (Shafer 1976). In addition, these specialists can return text strings that describe motivations for their preferences. The planner provides a facility to combine these strings into paragraphs that explain the proposed ordering decisions. This application software is integrated in the planner with the systems necessary to interpret TCAD models and use the general formats produced by the Parting Organization. The output of the Planning Organization is the product delivered to the customer when and if it is approved by the project manager.

The **Review Organization** has the final authority to approve a design product and release it to the customer. In addition, it is invoked whenever a problem is discovered that is deemed to need management attention and, therefore, it does not have a single typical place in the life cycle. The control regime describes the conditions that necessitate the intervention of this management function. Only managers in the Review Organization have the authority to release a design to its next life-cycle stage when one of the functional organizations has objected to an aspect of that design. The limitation of this authority is described by the critique regime.

14.8. Discussion

The availability of both Critique and Control Regime languages in Justin makes it possible to easily implement prototypes of automated integrated enterprises. The addition of a relatively simple language to control some low-level details of simultaneous object access will allow experimentation with a richer variety of corporate and customer policies including those necessary for concurrent engineering.

The fact that Justin-built systems use the object controller to interpret and maintain the state of all design objects, makes the implicit connection topology a star: The object controller is the center and it is directly connected to each enterprise activity as exemplified by the Tinker Inc. prototype shown in Figure 7.

Since the controller must observe modifications and additions to objects in order to determine their state changes, inter-activity communications must **logically** pass through the object controller. This does not mean that the object controller must **physically** be centralized. Rather, it can be distributed to multiple hosts and portions of the controller can even live in the implementations of enterprise activities. The real requirement is that the total system function as if a centralized interpreter could see all events responsible for object state changes. The impediment to distributing interpreter instances with object instances is that multiple copies of an object can exist.

An important research topic is to determine, under what circumstances, portions of the system can be exported from this central core to improve performance. Another important problem is to find enterprise policies that are compatible with and can be used to exploit these theoretical opportunities to decentralize system control. An improved version of Justin will be used to investigate these and many other related issues. This research will help to develop the technology necessary to integrate automation in the enterprises of the future.

Acknowledgments

We wish to thank H. R. Keshavan and Dwight Cass for their many important contributions to the Justin project and for their help in reviewing and improving the presentation of this article.

References

Barnett, J. A. 1990a. A system engineering approach to automated assembly planning. In Proceedings of the IEEE International Conference on System Engineering, 387–390. Pittsburgh, PA.: Department of Electrical Engineering, Wright State University and the IEEE Aerospace and Electronic Systems Society.

Barnett, J. A. 1990b. An architecture for integrating enterprise automation. In Proceedings of the SIGMAN Workshop on Intelligent Manufacturing Architectures, 43–46. Boston, MA.: Special Interest Group in Manufacturing of the American Association of Artificial Intelligence.

Barnett, J. A., Cass, D. E., and Betts, W. E. 1990. An architecture for concurrency control. In the Proceedings of the Workshop on Concurrent Engineering Design. Boston, MA.: American Association of Artificial Intelligence.

CALS Expo. 1990. *Directory and Conference Notebook.* December 4–6, Dallas, TX. Sponsored by the CALS/CE Industry Steering Group.

Davis, R. 1980a. Meta-rules: reasoning about control. *Artificial Intelligence* 15(3): 179–222.

Davis, R. 1980b. Content referencing: reasoning about control. *Artificial Intelligence* 15(3): 223–239.

Finger, M., Fox, M., Prinz, F. B., and Rinderle, J. R. 1990. Concurrent Design. Technical Report EDRC 24–26–90, Engineering Design Research Center, Carnegie Mellon Univ.

IDEF. 1981. *Integrated Computer-Aided Manufacturing (ICAM) Architecture* Part II. DTIC: AFWAL–TR–81–4023.

Karbe, B. H., and Ramsperger, N. G. 1990. Influence of exception handling on the support of cooperative office work. In *Multi-User Interfaces and Applications*, eds. S. Gibbs and A. A. Verrijn-Stuart. Elsevier Science Publishers.

Katz, R. H. 1990. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys* 22(4): 375–408.

Klein, M., and Lu, S. C. Y. 1989. Conflict resolution in cooperative design. *The International Journal for Artificial Intelligence in Engineering* 4(4): 168–180.

Lai, K. Y., Malone, T. W., and Yu, K. C. 1988. Object Lens: a "spreadsheet" system for cooperative work. *ACM Transactions on Office Information Systems* 6(4): 332–353.

Lee, J. and Malone, T. W. 1990. Partially shared views: a scheme for communicating among groups that use different type hierarchies. *ACM Transactions on Information Systems* 8(1): 1–26.

Lochovsky, F. H., Woo, C. C., and Williams, L. J. 1990. A micro-organizational model for supporting knowledge migration. *ACM Special Interest Group on Office Information Systems Bulletin* 11(2&3): 194–204.

MIL–STD–483A. *Configuration Management Practices for Systems, Equipments, Munitions, and Computer Programs.*

Roboam, M., Fox, M., and Sycara, K. 1990. Enterprise Management Network Architecture: The Organization Layer, Technical Report, CMU–R1–TR–90–22, Robotics Institute, Carnegie Mellon Univ.

Scheifler, R. W. and Gettys, J. 1990. *X Window System: The Complete Reference to Xlib, X Protocol, ICCM, XLFD*, 2nd Edition. Digital Press.

Shafer, G. 1976. *A Mathematical Theory of Evidence.* Princeton NJ: Princeton University Press.

Symbolics. 1990. *Programming the User Interface: Book 10.* Burlington, MA: Symbolics, Inc.

Systems Engineering Management Guide, 2nd Edition. 1986. Defense Systems Management College. Fort Belvor, VA.