DISTRIBUTED INTELLIGENT CONTROL AND MANAGEMENT (DICAM) APPLICATIONS AND SUPPORT FOR SEMI-AUTOMATED DEVELOPMENT[1]

Frederick Hayes-Roth, Lee D. Erman, Allan Terry,
Teknowledge Federal Systems,
Cimflex Teknowledge Corp.

& Barbara Hayes-Roth
Knowledge Systems Laboratory
Stanford University

Introduction

We have recently begun a 4-year effort to develop a new technology foundation and associated methodology for the rapid development of high-performance intelligent controllers. Our objective in this work is to enable system developers to create effective real-time systems for control of multiple, coordinated entities in much less time than is currently required. Our technical strategy for achieving this objective is like that in other domain-specific software efforts: analyze the domain and task underlying effective performance, construct parametric or model-based generic components and overall solutions to the task, and provide excellent means for specifying, selecting, tailoring or automatically generating the solution elements particularly appropriate for the problem at hand.

For intelligent control tasks, we believe that the domain-specific software approach holds the promise of providing great leverage on the software development task whether software generation is manual, automated, or semi-automated. In our view, complex and mission critical systems generally must have a human analyst in the loop both to specify desired behavior and to validate tested designs. Until this process is made extremely regular and routine, the human will necessarily be involved in nearly every step of the software development process as well. Given the lack of regularity and proven automatic generation means, the human's ability to validate overall designs requires insight into and hands-on experience with the details of the software design and generation. Nevertheless, we believe that significant progress on the "time to market" for such systems requires much of the same supporting infrastructure, regardless of the extent to which productivity enhancements are achieved through automation or merely improved methodology. This position is similar to that held by experts in many fields who state that one should not automate poorly characterized, highly variable processes. First, we must attempt to regularize the process, support it with an effective and efficient methodology, and then automate those portions of the process which give the greatest return on investment.

In this paper, we first present our specific domain focus, briefly describe the methodology and environment we are developing to provide a more regular approach to software development, and then later describe the issues this raises for the research community and this specific workshop.

Project Objectives and General Approach

Our project aims to develop a new technology foundation and associated methodology for the rapid development of high-performance intelligent controllers. These controllers will be employed in distributed intelligent control and management (DICAM) applications. Examples of such applications include intelligent highway systems, military command and control systems, and factory floor control systems. Our near-term domain of application is vehicle management systems, where one or more controllers may be employed to control a single vehicle, and these composite controller/vehicles are further aggregated and organized into higher-levels of control and capability. In a military context, for example, a single controller may be used for each subsystem within a tank, each tank system may be controlled by collectively organizing its subsystems, the overall tank may be controlled by another controller that coordinates the tank system controllers, several tanks may combine to form a platoon with its own control level, one or more platoons may form a battalion, and so on.

Our research project is one of several sponsored by DARPA (the Defense Advanced Research Projects Agency of the US Defense Department) and the US Army to advance the technology for domain-specific software architecture (DSSA). Our project for the Army address the specific vehicle management task of a howitzer, a tank-like vehicle that aims at more distant targets. The project has four principal focus elements. First, we are formulating a *reference architecture* for intelligent control. Second, we are supporting the construction of applications in a *development workspace* in which system requirements are ultimately satisfied by choosing design components that specialize and particularize components of the generic reference architecture. Many of the specialized modules and particular data used to instantiate a design are taken from a *repository*. The entire development process is supported by a rich array of *development tools*, which incorporate numerous techniques from both software engineering (e.g., control law specifiers, code generators, protocols, compilers, debuggers) and knowledge engineering (e.g., domain modeler, requirements manager, and various knowledge-based design assistants).

[1] This work reported here has been supported in part by DARPA and the US Army ARDEC through contract number DAAA21-92-C-0028. The opinions expressed here are those of the authors, not the sponsors.

The DICAM Framework and Supporting Technology

We are developing DICAM simultaneously as a "model" or *framework* for understanding control problems and as an *architecture* and related *environment* for building controllers. There are many reasons why we seek to formulate such a unifying framework. Foremost among these is our belief that the difficult, time-consuming and often unsatisfactory process of controller development would benefit from a more "standard" but flexible approach. Our DICAM framework provides a generic but customizable model of controllers that seems to unify a variety of views and experiences in the control, software and knowledge engineering disciplines.

DICAM is closely related to the NASA/NBS reference model for telerobot control systems (NASREM) [Albus, McCain, & Lumia, 1989]. The reference architecture includes two principal components in any distributed intelligent control and management application. First, an *information base and world model* (IB/WM) is a "conceptually centralized" database/knowledge base that represents the state of the world. The second principal component of the DICAM reference architecture is a collection of semi-autonomous interconnected controllers. These controllers are differentiated in terms of the scope of behavior they address, the resources they control, and the time frame spanned by their decisions.

Each controller is actually divided into two separate but interrelated components called the *domain controller (DC)* and the *meta-controller (MC)*. The DC contains several modular functions and prescribes how they interact using dataflow conventions. The functions include sensing, input filtering, situation assessment, planning, plan assumption analysis, execution and effector activation. Each controller has its own local world module which is a cached view of the global state represented by the IB/WM. Several messages flow into and out of the DC. The inputs include messages received from a superior controller specifying goals for the controller, messages from sibling controllers at the same level (such as another vehicle in the same group), and messages from subordinates, typically reporting on the outcomes of their efforts. Outputs include subgoals assigned to subordinates for delegated execution as well as messages to siblings, for example, to report on current plan execution objectives or status or to request operating resources.

Although this general DC structure has proved effective in applications such as the Pilots Associate [Smith & Broadwell, 1989; Lark *et al.*, 1990] and robotics [Becker, 1989], dataflow programs in general exploit only weak knowledge about when to execute functions. The general rule is to compute any function when all of its inputs are available. However, there are often too many possible instantiations to execute all simultaneously, or even with a small delay. Thus, in situations where more knowledge is required to achieve excellent results with scarce resources, a metal-level of control is required [Garvey & Hayes-Roth, 1989; Hayes-Roth, 1985; Hayes-Roth, 1990]. Our meta-controller is based on the knowledge-based scheduler of the BB1 blackboard system. This controller utilizes three basic functions to determine on a cyclical basis which pending action is best to execute next: an *agenda manager* to store and evaluate pending actions; a *scheduler* to determine the next action based on the degree of fit between goals of a control plan and actions pending on the agenda; and, finally, an *executor* gives control to the selected actions.

Our basic methodology for development of DICAM applications consists of a blackboard-like environment where the "blackboard" is a *development workspace* and the "knowledge sources" are system developers augmented by a wide variety of computer-based tools, including some expert systems that are capable of autonomous development activity. We are assembling an Application Development Support Environment (ADSE) for DICAM applications (see Figure 1) to provide these capabilities.

The development workspace contains a representation of the emerging system being developed incrementally over time. Its elements represent decisions or specifications linked into a "web" of mutually supporting decisions that both specify the system design and justify it. We have combined three lines of research in formulating this development workspace. First, we have drawn on the blackboard model and opportunistic reasoning [Erman *et al.*, 1980; Hayes-Roth & Hayes-Roth, 1979; B. Hayes-Roth *et al.*, 1986] as an organizing methodology for incremental design and development processes. Second, we have adopted the emphasis of the domain analysis and domain-specific architecture approach to software specification, reuse and rapid development [Prieto-Díaz & Arango, 1991]. Lastly, we have adopted and generalized the approach of module-oriented programming from our previous research on ABE [Erman, Lark & Hayes-Roth, 1988; F. Hayes-Roth *et al.*, 1989; F. Hayes-Roth *et al.*, 1991]. This includes the ideas of recursive modular composition, distributed control through message passing using ADTs, system construction through module composition, and system realization by deferred binding of processors to modules.

Specifically, the workspace provides a multi-faceted, multi-level representation of DICAM software applications. It provides means for describing the domain model, i.e., the general characteristics of the task and environment in which the vehicle or machinery will operate. The general domain model is then augmented with specialized information about the specific application being built, such as how many vehicles, the distances to be traveled, the specific threats and so forth that the application will address.

At a lower (more concrete) level, the workspace provides means for representing the functional components and the physical resources that make up the controlled system, and it describes how the functional components are composed and how they are implemented using specific processors, communication capabilities or other machinery.

In addition, the workspace provides means for representing the status of the software development process, including the history of activities and characteristics of the current overall development.

As is typical of blackboard systems, the workspace provides means of representing decisions and using state change to trigger the invocation of appropriate tools. Decisions in this workspace range from abstract characterizations of components such as requirements or

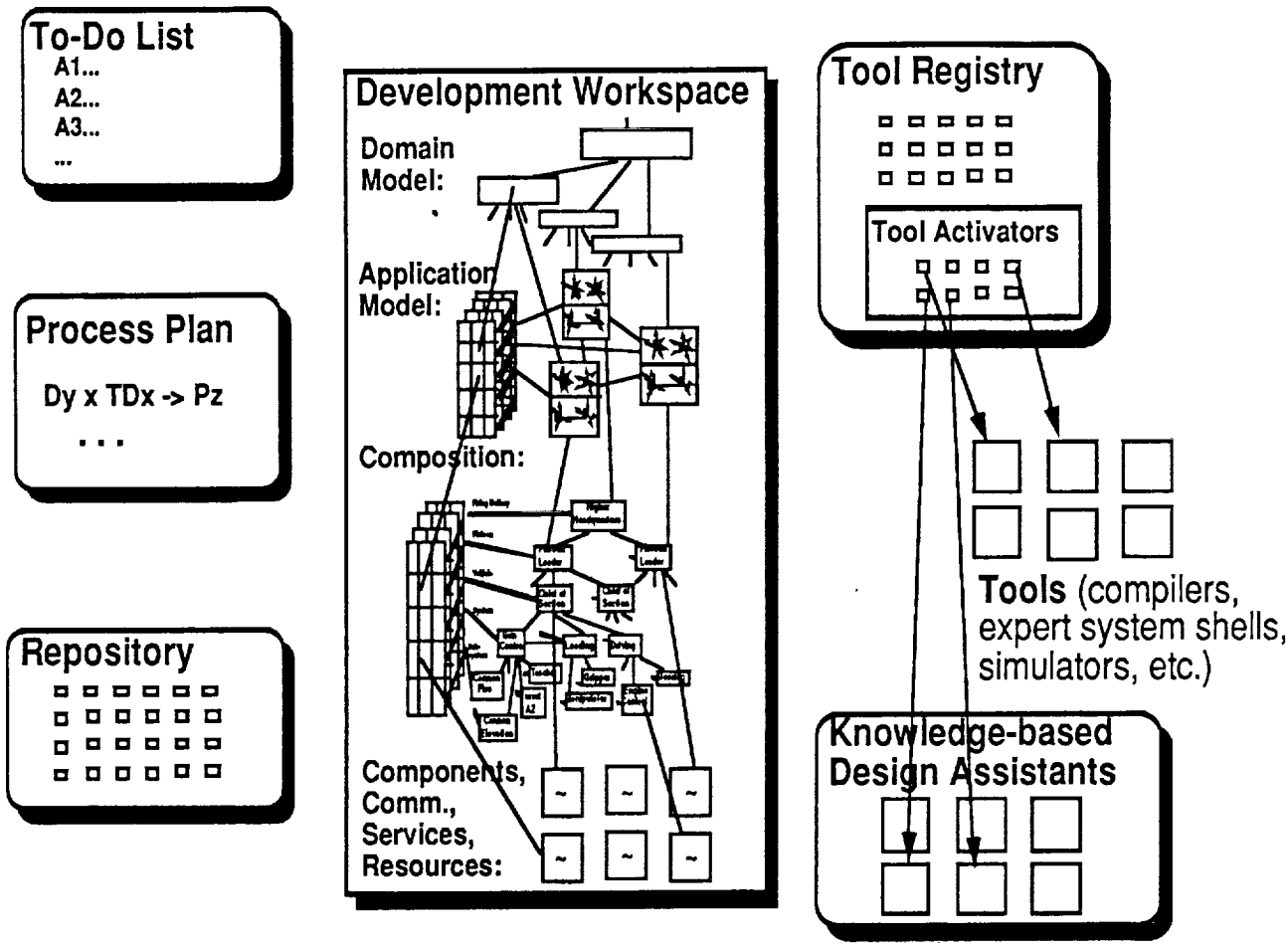

Figure 1. The Application Development Support Environment.

goals to particular specifications, including detailed functional characterization or specific software or hardware packages that realize the required capability. We have not yet settled upon final or formal representation sublanguages for each level, but are considering various alternatives that are being suggested in other groups' efforts to conceive potentially standardizable descriptions of modules and module interfaces (e.g., the DARPA module interconnect formalism, the DoD STARS repositories, etc.). Regardless of which specific formalism is used, the description of modules must include input/output datatypes, function characterization, implementation requirements, domain assumptions, and performance metrics. When making a design decision, the developer specifies some or all of these attributes along with his or her name and some rationale. As in all blackboard systems, decisions are changeable, and multiple competing decisions may coexist. Ultimately, those decisions that form the best coherent "web" win: these decisions constitute the overall system specification, from requirements to implementation, which particularizes the domain and application models.

Other features of the ADSE that we are developing and assembling include: A *To-Do List* keeps an agenda of pending tasks for the software developers. As with blackboard systems, actions are triggered when the state of the Workspace matches a pattern of interest. A *Process Plan* is supported that effectively maps patterns of interest found in the Development Workspace and the current To-Do List into proposed actions. The proposed actions (shown as Pz) in the figure might include any of the following: make a specific design decision; apply a particular tool to a particular design component with certain parameters; raise the priority on doing one pending task over others, etc. Our plan is to support a wide variety of SE methodologies by providing a general mechanism for representing and implementing corresponding process plans. A *Repository* of reusable components is provided that stores, classifies, and searches for previously used Development Workspace structures. Typically these include reusable domain models, application characteristics, generic function modules, specific implementation modules, and data to customize or particularize generic functions for specific application domains. A *Tool Registry* provides mechanisms for enrolling software development tools, describing their required inputs and associated outputs in terms of patterns that match characteristics found in the Workspace or Process Plan and, finally, providing *Tool Activators* that can automate or semi-automate invocation and application of tools. The tools consist of compilers, generators, simulators, expert system shells, etc. Lastly, the ADSE incorporates specialized tools called KBDAs that provide knowledge-based assistance in to the software development process. These tools can include, for example: requirements

Table 1. Aspects of the Development Methodology

Aspect	Elements
Opportunistic Design	Multiple levels and representations Abstract to particular characterizations Incremental decisions Linked decisions form design web Prescriptive process models permitted Humans and computer tools cooperate
Controller Architecture	Generic modules Flexible, tailorable controllers Schema of ADTs for IB/WM Message processing using ADTs and intermodule protocols Distribution Fractal control model
Information Base/ World Model	Shared data managed by IB/WM Conceptually centralized, single-copy, but allows physical partitioning Typically distributed Time response must satisfy requirements ranging from sub-millisecond to a few seconds Different levels of aggregation Different meta-types: data, propositions, rules, plans Temporally organized and continually renewing
KB Design Assistants	Mini-expert systems watch process state and advise user at key events Tool-use expert systems help humans apply development tools
Repository	Stores and uses partial matches to retrieve "components" at any level Components classified in taxonomy from generic to particular Domain-specific customizations available to particularize generics
Engineering Foci	Domain modeling Requirements engineering Knowledge engineering, about DICAM and DICAM development tools and methods Performance objectives, measurement and attainment
Component Characterization	Goals & Constraints Models: Behavior, timing, functionality Interfaces: Datatypes Module partners Conversation types Protocols Messages to other devices Resource/environment prerequisites

analyzers that suggest appropriate reusable components; redesign advisors that suggest ways to modify an existing design in light of a change in requirements or capabilities; and intelligent interfaces that set up and run complex tools to assist a developer in generating or analyzing some code.

To implement the ADSE we are using a number of "off-the-shelf" technologies. Chief among these are: ABE [F. Hayes-Roth *et al.*, 1991], as an integration environment for tools, a composition framework for modular, real-time applications, and a catalog and classification system for the reuse repository; BB1 [B. Hayes-Roth, 1985], as an incremental workspace, process model interpreter, and agenda manager; M.4, a commercial expert system shell, for building the KBDAs; and the Requirements Manager (RM), a DARPA-sponsored software product for collecting, managing, and evaluating application requirements and validating application designs against requirements [Fiksel & Hayes-Roth, 1990]. We are also evaluating many other commercial and research SE tools for use in the ADSE [cf. NIST ISEE Working Group, 1991].

Development Methodology

The overall approach we are taking to development is summarized in Table 1. The seven principal facets fall into three basic categories of methods. The controller architecture and information base/world model constitute the reference architecture for the domain of intelligent control. The repository, engineering foci, and component characterization concerns define our approach to domain-specific software engineering. The opportunistic design and

KB design assistants define our approach to defining a process of software development that can, at least, be semi-automated.

We are currently applying the methodology to demonstration problems chosen from defense applications. As an example, consider the task of achieving intelligent control of field artillery systems, such as mobile howitzers. Howitzers, like other military vehicles, are self-propelled, mobile vehicles with offensive guns. Their primary mission is ground-based artillery shelling of over-the-horizon targets. They are very similar to tanks, armored personal carriers, and helicopters in general information processing terms. Thus, all military vehicles of this sort share elements of the domain model, but differ increasingly as these models and the corresponding application model become detailed.

The general DICAM architecture is specialized for Army Vehicle Management Systems by the selection of levels: battalion, battery, platoon, vehicle (section), system, subsystem. Then it is further specialized for a particular howitzer, e.g. the "ABC Howitzer," by the selection of functional controllers and their relationships. Each group of ABC howitzers is headed by a Platoon Leader who reports to higher headquarters. The Chief of Section of each vehicle reports to the Platoon Leader and is responsible for the Gun Control, Loading, and Driving functions.

Following the domain-specific approach, after developing the generic domain model, the next task for system developers is to elaborate the application model. The task application model enumerates desired functionalities associated with each level of control. Several generic functions appear repeatedly across different control levels, such as tasking subordinates with subgoals or performing external and system status analyses as part of situation assessment. These functionalities are also common across the analogous components in other vehicles: tanks, missile launchers, infantry fighting vehicles, etc. Thus, there are two levels of functional similarity:

- across different components within a vehicle, and
- across the similar components in different vehicles.

To convert the informal task analysis into a more formal, explicit application model, the system developer selects from among generic functionalities in the reference architecture, specializing and customizing them for the particular needs of the target application. Then to construct an application system, the developer uses these refined specifications either to select components from the repository that can perform these functions or to drive automatic, semi-automated, or manual code generation.

Issues Raised

Our research raises many issues, some of which are highlighted here:

- Is our methodology (as described in Table 1) effective?
- Does our reference architecture provide enough structure to make specification practical and component software reusable?
- How can a critical mass of reusable components be created?
- How can modules be characterized?
- How can the languages used for characterizing modules be standardized?
- How can modules be designed so that they can be specialized or customized to new applications?
- Which tasks in the development process are most worthy of automated support?
- How can the space generated by a diversity of vehicles, environments and control objectives be structured to maximize the potential for reusability of specifications and solution components?

References

[1] Albus, J. S., McCain, H. G., and Lumia, R. "NASA/NBS standard reference model for telerobot control system architecture (NASREM)," National Bureau of Standards, Tech. Note 1235. 1989.

[2] Becker, J. M. "The generic control level: a unifying view," *Proc. ROBEXS '89*, Palo Alto, CA, 1989.

[3] Erman, L. D., Hayes-Roth, F., Lesser, V. R., and Reddy, R. "The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty," *Computing Surveys* 12(2), June, 1980, pp. 213-253.

[4] Erman, L. D., Lark, J. S., and Hayes-Roth, F. "ABE: An environment for engineering intelligent systems," *IEEE Transactions on Software Engineering*, 14(12), December, 1988.

[5] Fiksel, J. and Hayes-Roth, F. "A requirements manager for concurrent engineering in printed circuit board design and production," *Proc. of the Second National Symposium on Concurrent Engineering*, Morgantown, WV, February, 1990.

[6] Garvey, A. and Hayes-Roth, B. "An empirical analysis of explicit vs. implicit control architectures," in Jagannathan, V. and Dodhiawala, R. T. (eds.), *Current Trends in Blackboard Systems*, Academic Press, 1989.

[7] Hayes-Roth, B. "Blackboard architecture for control," *Artificial Intelligence*, vol. 26, pp. 251-321, 1985. Reprinted in: Bond, A. and Gasser, L. (eds.), *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann Publishers, Inc., 1988.

[8] Hayes-Roth, B, "Architectural foundations for real-time performance in intelligent agents," *Real-Time Systems: The International Journal of Time-Critical Computing*, 2(1/2), 1990, pp. 99-125.

[9] Hayes-Roth, B. and Hayes-Roth, F. "A cognitive model of planning," *Cognitive Science*, 1979, 3, 275-310. Reprinted in A. Collins and E. E. Smith (eds.), *Readings in Cognitive Science: A Psychological and Artificial Intelligence Perspective*. Morgan Kaufmann, 1988; and in J. Allen, and J. Hendler, and A. Tate (eds.), *Readings in Planning*, Morgan Kaufmann, 1990.

[10] Hayes-Roth, B., Johnson, M.V., Garvey, A., and Hewett, M. "Applications of BB1 to arrangement-assembly tasks," *Journal of Artificial Intelligence in Engineering*, 1986.

[11] Hayes-Roth, F., Davidson, J.E., Erman, L.D. and Lark, J.S. "Frameworks for developing intelligent systems: The ABE systems engineering environment," *IEEE Expert*, June, 1991.

[12] Hayes-Roth, F., Erman, L. D., Fouse, S., Lark, J. S., and Davidson, J. "ABE: A cooperative operating