In press, *IEEE Transactions on Software Engineering*, expected April, 1995.

A Domain-Specific Software Architecture for Adaptive Intelligent Systems

Barbara Hayes-Roth, Karl Pfleger,
Philippe Lalanda, Philippe Morignot, Marko Balabanovic

Knowledge Systems Laboratory
Computer Science Department
Stanford University
701 Welch Rd.
Stanford, CA 94304
`{bhr, kpfleger, lalanda, morignot, mxb}@hpp.stanford.edu`

The research reported in this paper was supported by ARPA contract DAAA21-92-C-0028, under subcontract 71715-1 with Teknowledge Corporation.

Abstract

A good software architecture facilitates application system development, promotes achievement of functional requirements, and supports system reconfiguration. We present a domain-specific software architecture (DSSA) that we have developed for a large application domain of adaptive intelligent systems (AISs). The DSSA provides: (a) an AIS reference architecture designed to meet the functional requirements shared by applications in this domain, (b) principles for decomposing expertise into highly reusable components, and (c) an application configuration method for selecting relevant components from a library and automatically configuring instances of those components in an instance of the architecture. The AIS reference architecture incorporates features of layered, pipe and filter, and blackboard style architectures. We describe three studies demonstrating the utility of our architecture in the sub-domain of mobile office robots and identify software engineering principles embodied in the architecture.

Index Terms: *Software architecture, Domain-Specific Software Architectures, Software reuse, Intelligent agents, Mobile robots.*

1. Introduction

The architecture of a complex software system is its “style and method of design and construction” [25]. When applied appropriately, a good software architecture facilitates application system development, promotes achievement of the system's functional requirements, and supports reconfiguration. A “bad” architecture—or the absence of a clearly defined architecture—can thwart all three objectives.

Designing and implementing a good software architecture (with associated development and debugging environments) is challenging and expensive. Replicating this activity across many projects for systems that have similar requirements unnecessarily inflates their expense. Conversely, competition for limited project resources may limit the quality of the architectural support that can be realized for a given system.

How can we achieve the benefits of good software architecture while containing the cost?

We have been studying a software engineering methodology based on *domain-specific software architectures (DSSAs)* [18, 36, 34, 35, 37, 24], where the term “domain” refers to a class of applications. A DSSA comprises: (a) a *reference architecture*, which describes a general computational framework for a significant domain of applications, (b) a *component library*, which contains reusable chunks of domain expertise, and (c) an *application configuration method* for selecting and configuring components within the architecture to meet particular application requirements.

We have been developing and experimenting with a particular DSSA for the domain of *adaptive intelligent systems (AISs)* that perceive, reason, and act to achieve multiple goals in dynamic, uncertain, complex environments. AIS applications share functional requirements such as: concurrent perception, reasoning, and action; sensitivity to externally determined priorities and deadlines; and dynamic global control of the system's own behavior. As illustrated by the taxonomy in Figure 1, the AIS domain may be partitioned into sub-domains defined by more specific shared task requirements. For example, all autonomous robots require capabilities for path planning and navigation, while all monitoring systems require capabilities for pattern classification and fault detection. Sub-classes may introduce additional task requirements or place constraints on inherited requirements.

Instantiating the general definition given above, our AIS DSSA comprises: (a) an implemented AIS reference architecture that supports the shared computational requirements of adaptive intelligent systems and also provides a congenial framework for both compile-time and run-time configuration of components, (b) a framework for decomposing application-specific expertise into reusable components, along with an evolving library of implemented components, and (c) an application configuration tool (not completely implemented) that takes as input a domain description, instantiated and extended with application-specific requirements, and automatically selects and configures the best available components from the library. Each of these elements is described in section 2 below.

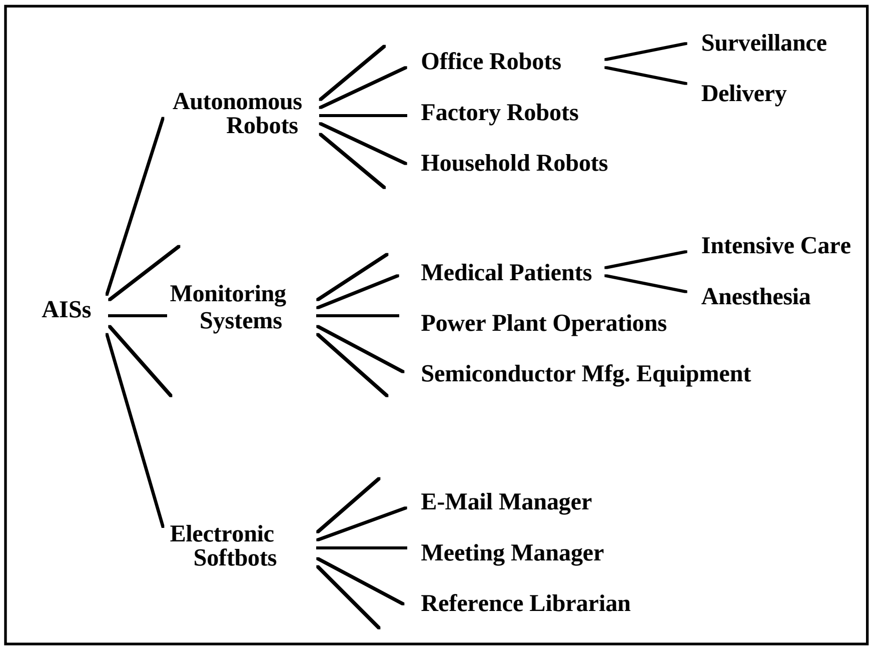

Figure 1. A partial taxonomy of adaptive intelligent systems.

We have experimented with our DSSA in three major sub-classes of AIS applications: (a) monitoring systems in four specific domains: intensive care [15, 20, 22], materials processing [29], semiconductor manufacturing [27], and power plant operations [31]; (b) layout design systems in two specific domains: protein structure modeling [17, 18] and construction site layout [33]; and (c) autonomous office robots in two specific domains: office surveillance and office delivery [19]. In section 3, we present three studies and their results from our work on office robots to demonstrate the utility of the AIS DSSA. Section 4 presents conclusions.

2. The AIS DSSA

2.1 The AIS Reference Architecture

The AIS reference architecture is a heterogeneous mixture of common architectural styles [9]. It is divided hierarchically into layers for different sets of computational tasks. The layers and the relations among them provide properties of pipe and filter style architectures. Each layer, itself, comprises a number of components, organized in a blackboard style, to allow for a range of potentially complex behavior.

2.1.1 Organization of Layers

The architecture currently has two layers, or levels, to control concurrent physical and cognitive behaviors. Behaviors at the *physical level* implement perception and action in the external environment. Behaviors at the *cognitive level* implement more abstract reasoning activities such as situation assessment, planning, problem-solving, etc. Information flow is bi-directional. The results of cognitive behaviors can influence physical behaviors and vice versa.

In addition to these categorical distinctions in behavior, levels differ in the following ways: (a) Information at the cognitive level tends to be represented symbolically, while information at the physical level tends to be metric. (b) Cognitive control plans (described in section 2.1.2.3) can be temporally extensive and relatively complex, while physical control plans are severely bounded on both dimensions. (c) At the cognitive level, the temporal horizon for state information is effectively unbounded in both directions, whereas at the physical level it is effectively immediate. (d) Reaction time is an order of magnitude faster at the physical level than at the cognitive level. In general, cognitive behaviors and representations are more abstract than those at the physical level.

While our current architecture contains only two levels, it could incorporate more levels, each with the same internal organization. The lowest level would interact with the external environment and the next higher level; each higher level would interact only with the levels immediately below and above it. The levels would exhibit graded differences along each of the dimensions of difference mentioned above and, more generally, higher levels would organize computations at higher levels of abstraction. Our AIS architecture has only two levels because our current applications do not require finer resolution along these dimensions. [11, 6] discuss similar architectures for robot agents. [1, 2, 21, 32] discuss extended multi-level architectures.

Our architectural organization provides the usual benefits of software layering. Restricting interactions to those between adjacent levels provides modularity, allowing easy replacement or enhancement of individual levels. (Note that this modularity is in addition to the modularity provided within each level, as described in the next section). Also, hierarchically increasing levels of abstraction facilitate construction of complex behaviors and manipulation of higher level concepts [9, 1, 2].

However, our architecture differs from the common architectural layering in terms of how adjacent levels interact. In conventional layered systems, interactions between levels occur through function or procedure calls from one level to the next lower level. The interface functions of each level serve as an abstract virtual machine to the next higher level [9]. In contrast, our levels communicate with a more flexible message-passing style. All levels operate concurrently, sending information to adjacent levels when appropriate. Thus, the architectural organization also can be viewed as a bi-directional pipe and filter model [9], in which each level reads from two input data streams and writes to two output data streams. (Of course the highest architectural level has only one pair of input/output streams with its single adjacent level and the lowest architectural level has one of its two pairs of input/output strems with the external environment.) As shown in figure 2, the two architectural levels asynchronously exchange information. The physical level sends processed perceptual information, including feedback from the execution of actions, to the cognitive level, while the cognitive level sends control plans to the physical level. Otherwise they share the normal pipe and filter properties: they do not share state or directly know about one another's computations. This hybrid of pipe and filter and layered architectural styles is quite useful. The two styles do not conflict and both provide modularity. In addition, the hybrid introduces the combined advantages of abstraction due to the layering style and concurrent execution due to the pipe and filter style [9].

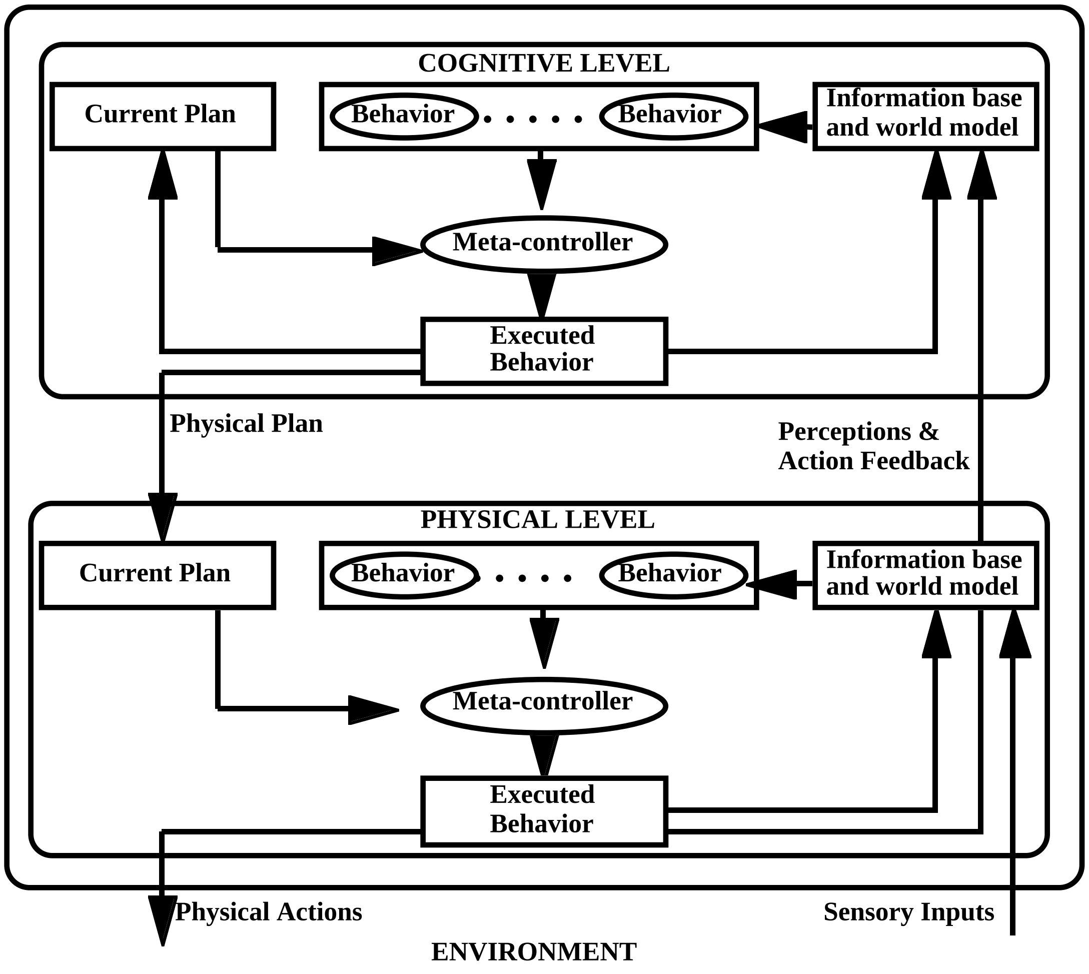

Figure 2. AIS reference architecture. Arrows show data flow, not control flow.

2.1.2 Internal Organization of Each Layer

Each level of the architecture has isomorphic internal structure. The shared abstract control model is embodied by the BB1 blackboard architecture [12, 13]. The BB1 system itself is used as the implementation for the cognitive level and a much simpler implementation of the same basic model is used for the physical level. As discussed in section 2.1.1, the simpler implementation at the physical level restricts computational power in order to achieve order-of-magnitude speed-up. The remainder of this section characterizes important aspects of this internal architecture. Section 2.1.3 discusses the BB1 blackboard style's distinctive control properties, which underlie the flexible run-time behavior that is crucial for AISs.

2.1.2.1 Behaviors

Behaviors embody the potential application of particular *methods* to particular *tasks.* For example, at the physical level, one behavior might apply a reactive feedback-control method to navigate along a path. At the cognitive level, one behavior might apply a specialized planning method to sequence travel destinations, taking into account constraints on the order in which the destinations must be visited.

Each behavior has a set of *triggering conditions* that can be satisfied by particular kinds of *events*—changes to the *information base/world model* (see below) resulting from perceptual inputs or previously executed behaviors. For example, a destination sequencing behavior might be triggered whenever a new set of places to visit appears. When an event satisfies a behavior's triggering conditions, the behavior is enabled and its parameters bound to variable values from the triggering situation. A given behavior will be enabled, and therefore executable, whenever events satisfying its triggering conditions occur, regardless of its relative utility in achieving the current goals. Conversely, at each point in time, many competing behaviors will be enabled and the system must choose among them to control its own goal-directed behavior.

To support these *control decisions*, each behavior has an *interface* that describes the kinds of events that enable it, the variables to be bound in its enabling context, the task it performs, the type of method it applies, its required resources (e.g., computation, perceptual data, effectors), its execution properties (e.g., speed, complexity, use of resources, completeness), and its result properties (e.g., accuracy, precision). Interface descriptions resemble the software "wrappings" of [23] and the model description language of [26], in providing information about how and under what circumstances to use available resources.

2.1.2.2 Information base / world model (IB/WM)

The disparate behaviors a system performs interact with one another via changes they make to the *information base/world model*—a declarative data base that houses a system's factual knowledge, descriptions of its potential behaviors, and a temporally organized representation of its run-time perception, reasoning, and action. In its capacity as knowledge base, the IB/WM provides a skeletal conceptual graph to which type hierarchies of tasks, methods, and domain concepts can be attached at compile time and accessed at run time. In its role as workspace, the IB/WM provides a data exchange medium for interacting behaviors. Each executed cognitive or physical behavior makes changes to the contents of the IB/WM, producing events that enable subsequent behaviors and information that may influence their execution. For example, the destination sequencing behavior mentioned above produces as part of its output a representation of the planned destination sequence in the cognitive IB/WM. The appearance of a new destination sequence will enable some of the available cognitive methods for path planning. Thus, the IB/WM provides a workspace in which a system can coordinate the interactions and products of groups of related tasks, each of which may be performed by any of a number of alternative methods, in order to achieve higher-order goals.

2.1.2.3 Control plans

Control plans describe the system's intended behavior as a temporal pattern of plan steps, each of which comprises a start condition, a stop condition, and an intended activity in the form of a 3-tuple: <task, parameters, constraints>. For example, the physical activity `<navigate, (origin, destination), {fast}>`, describes the task of moving quickly from `origin` to `destination`.

Control plans reside as data structures in the IB/WM, so the system can develop and modify them dynamically by means of whatever control planning methods are enabled in its run time situation. Note that control plans do not refer explicitly to any particular method in the system's repertoire. Unlike a simple list of machine instructions or program subroutines, they are not directly executable. Instead, control plans only describe intended behaviors in terms of the desired tasks, parameter values, and constraints. Thus, at each point in time, the system has a plan of intended action, which intensionally describes an equivalence class of desirable behaviors and in which currently enabled specific behaviors may have graded degrees of membership. (See [14, 19] for more detailed treatment of these ideas.)

2.1.2.4 Meta-controller

A *meta-controller* attempts to follow a system's current control plan by executing the most appropriate enabled behaviors. Specifically, at each point in time, the meta-controller executes the enabled behavior that: (a) is capable of performing the currently planned task with the specified parameterization; and (b) has a description that matches the specified constraints better than any other enabled behaviors that also satisfy (a). For example, a system might have two methods for navigation, a fast dead reckoning method that uses minimal or no sensory feedback, but requires an accurate metric map of the area, and a slower, reactive feedback-control method that is capable of avoiding collisions and requires less information about the area. Given the plan step `<navigate, (origin, destination), {fast}>`, this system will execute the dead reckoning method if it has detailed metric map information about the area from `origin` to `destination`, and will execute the reactive method otherwise. Conversely, given the plan step `<navigate, (origin, destination), {safe}>`, the system will execute the reactive method, to avoid collisions, regardless of the accuracy of its map. Thus, a system continuously improvises its specific course of behavior, following intended plans as well as possible, given the behaviors that happen to be enabled along the way.

2.1.3 Key Properties of the Reference Architecture

The AIS has two key architectural properties, the pipe and filter properties of its layered architecture and the adaptive properties of its control model, both of which are designed to support the distinctive interactions required between AISs and their environments [13, 14, 16].

2.1.3.1 Pipe and filter properties of the layered architecture

The AIS architecture's hybrid combination of layered and pipe and filter styles is designed to support resource-bounded, time-sensitive interaction with dynamic, complex, uncertain environments. Conventional application systems function in precisely structured, static environments for which function call interactions are appropriate. For example, an operating system's environment is the computer hardware, with its well defined behavior. Each successive layer of the operating system provides a higher-level abstraction of the computational machine it controls [9]. (One might argue that the user does not provide well-defined behavior; but the user of conventional layered systems typically interacts with the highest layer in the hierarchy, not the lowest.) By contrast, AISs function in environments where unpredictable external events occur asynchronously. The system has limited resources (computation, time, data, knowledge) for responding to events and must satisfy constraints on the timing as well as the quality of its responses. Successful performance in such environments demands opportunistic message passing, continuous filtering of input data streams, and continuous management of output data streams.

2.1.3.2 Adaptive properties of the control model

The AIS architecture's within-level dynamic control model is designed to support the considerable flexibility of behavior required in AIS environments. A system can have in its knowledge base many alternative behavioral methods for performing diverse tasks. It can coordinate different combinations and sequences of tasks and methods in order to achieve goals, without planning the exact sequence of tasks and methods in advance. Whatever events occur will enable associated behavioral methods. Depending on the current control plans, the meta-controllers will choose to execute whichever enabled behaviors best match those plans. Thus, a system can work toward high-level objectives by coordinating its performance of a variety of tasks under a variety of context-specific constraints. It can plan and pursue an intended course of action by incorporating the best available behaviors that are enabled in its immediate situation. And it can dynamically adapt its plans in response to changing environmental conditions. As a result, its plans and plan following are extremely robust over a range of situations. This run-time flexibility is important precisely because of the dynamic and uncertain natures of the external environment that are the defining characteristics of adaptive intelligent systems.

As demonstrated below, the dynamic control model also provides a framework in which appropriate sets of components can be configured at both design time and run time. *Application domain schemas* (see section 2.3) generalize the concept of descriptive control plans. They allow an application builder to describe the kinds of cognitive and physical tasks a system might have to perform and the kinds of constraints under which it might have to perform them. Moreover, applications can be configured at design time and reconfigured later, while the system is in operation. As shown in section 3 below, the meta-controller makes use of whatever plan-relevant components are available and enabled at run time.

2.2 Framework for a Component Library

The AIS reference architecture provides an extremely general computational framework in which to configure diverse software components and coordinate their activities. Amplifying this capacity for architecture reuse, we also provide a framework for developing software components that can be reused and reconfigured easily in a large domain of applications. Specifically, we decompose expertise along three orthogonal dimensions (Figure 3) so that each of the three components produced by a given decomposition may be reused in combination with multiple (not necessarily all) alternative components on the other two dimensions.

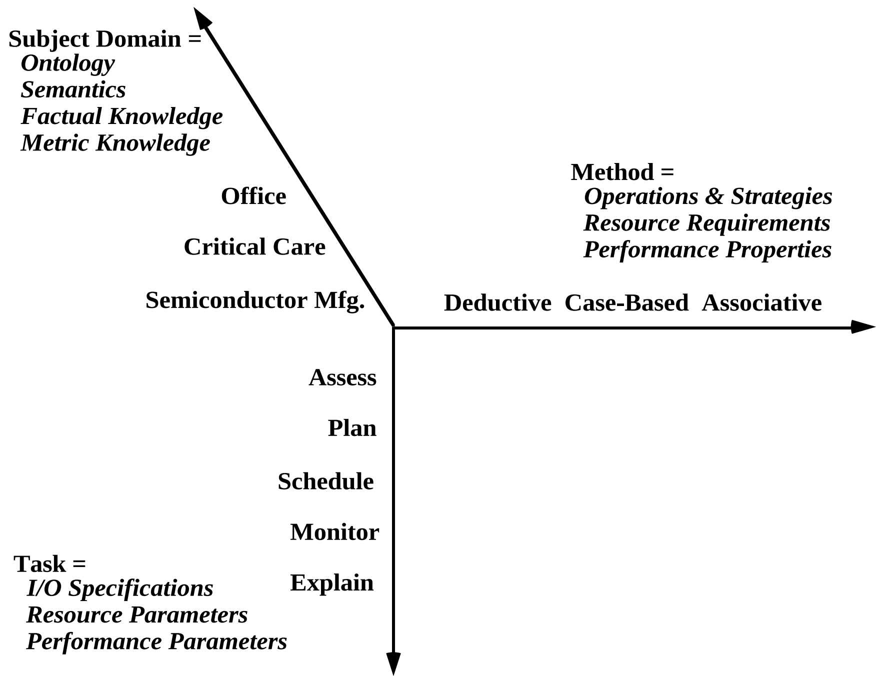

Figure 3. Orthogonal components of expertise: definitions and examples.

Tasks are classes of jobs a system might perform, defined by their abstract input/output specifications, independent of method and domain. For example, the task of path planning transforms an initial and final location into a representation for a path and can be performed by a number of different methods and in many different domains. Tasks may be specified further by imposing resource limitations (e.g., time limits) or performance requirements (e.g., precision, reliability). For example, an agent (used interchangeably with "system") may need to plan very quickly, but not necessarily guarantee optimal plans. As discussed in section 2.3, certain classes of applications are defined by characteristic configurations of required tasks along with potential constraints on the performance of those tasks. For example, a mobile office robot must perform navigation, but a bedside critical care monitoring agent need not.

Methods are classes of computational approaches a system exploits for a variety of tasks, independent of domain. They are defined in terms of sets of abstract component operations, each of which may be enabled by run-time events, along with abstract strategies for selecting and sequencing enabled operations at run time in order to achieve goals. For example, generative reasoning and case-based reasoning are two different cognitive methods a system might apply to a planning task. Case-based planning would comprise abstract operations such as "find a similar case" and abstract strategies such as "find the n most similar cases, then map the n cases onto the present situation, then ..." Methods for a given task may differ in their resource requirements (e.g., amount of real time, computation time, sensor utilization, domain knowledge), run-time properties (e.g., interruptability, intermediate results, incremental solution improvement), or their characteristic results (e.g., precision, reliability, qualitative contents of conclusions). Thus, different methods that are equivalent in their logical applicability to an abstract task may be more or less appropriate for different task instances or domains. For example, case-based methods may be more appropriate than generative methods for tasks that require fast real-time performance, but do not require guaranteed optimal solutions. More generally, case-based methods are appropriate only in domains for which large numbers of relevant cases exist.

Subject domains comprise the different kinds of knowledge (e.g., ontology, facts, relations) a system might have regarding its environment or subject matter, independent of the tasks it might perform or the methods with which it might perform them. For example, an office domain might include both cognitive knowledge (e.g., an ontology of office objects and services, a symbolic model of the relationships between certain objects and their locations, a topological model of the office layout) and physical knowledge (e.g., metric models of passageways, recognition templates for important objects). This knowledge could be used to support various methods for various tasks and jobs (e.g., office surveillance robot, office delivery robot, office design assistant).

Our three-way decomposition of expertise combines the complementary decompositions of software engineering and knowledge engineering practice. Software engineers typically decompose software into its interface (conflating our task and subject domain components) and its implementation (our method component). Knowledge engineers typically decompose software into its knowledge (our subject domain component) and its inference engine (conflating our task and method components) [5]. As demonstrated in the experimental results in section 3, our three-way decomposition expands opportunities for reuse. A component along one dimension might be reused in combination with alternative (not necessarily all) components from either or both of the other two dimensions to produce a large number of distinctive competencies. For example, at the cognitive level, either generative or case-based methods could be applied to either the destination sequencing or path planning tasks in several robot domains (e.g., office or factory surveillance, office delivery, household chores) or autonomous vehicle domains (e.g., errands, chauffeur). Similarly, at the physical level, either reactive feedback-control (closed loop) or blind dead reckoning (open-loop) navigation could be used to perform a path following task in a variety of domains. For all of these configuration and reuse purposes, a descriptive language of tasks, methods, and domains is critical. In our work we have identified a number of descriptive characteristics that are important in our experimental domains. However, developing a formal descriptive language for software components remains an important research problem. (See also [23, 26].)

2.3 Application Configuration

Even though the AIS architecture supports run-time enabling and selection of competing components, application system configuration remains important because inclusion of "extra" components in an application may not increase utility and may degrade performance. Inclusion of extra components will not increase utility if, for example, the components are irrelevant to the application, the necessary hardware (e.g., sensors, effectors) is not available on the application platform, or the required knowledge or data are not available in the application domain. Inclusion of extra components will degrade performance if, for example, they cause the knowledge base to exceed space limitations or meta-control decision time to exceed acceptable response latencies.

Together, the AIS reference architecture and the orthogonal decomposition of expertise support both design-time and run-time system configuration. At design time, one can select and configure an application-specific set of required tasks, an application-specific set of appropriate methods for performing those tasks, and the application-specific domain knowledge required to apply those methods to those tasks. In cases where a given task must be performed under variable circumstances, suitable alternative methods can be selected and configured at design time and then selectively enabled and executed at run time. At run time, if useful new application-relevant task, method, or domain components should become available, the new components can be substituted for old ones or added to the knowledge base alongside the old ones, without interrupting system operation. The architecture's event-based enabling of behavioral methods, its plan-based meta-control choices among competing methods, and its efforts to retrieve necessary knowledge from the IB/WM are not preprogrammed to require any particular tasks, methods, or domain facts; they operate on whatever task, method, and domain knowledge are available in the IB/WM at run time. The AIS architecture can accommodate new components acquired through machine learning in exactly the same fashion, as demonstrated for domain knowledge in the experiments below.

As demonstrated in our initial experiment below, we already can create diverse agents at design time and reconfigure agents at run time by manually selecting and automatically loading different combinations of components into the architecture. If the configuration of components is conceptually complete (e.g., it includes the required domain knowledge to apply at least one method to each of the specified tasks), the agent runs immediately and makes appropriate use of all available components. As illustrated in Figure 4, each agent instantiates a different, application-specific configuration of cognitive and physical task, method, and domain components. Application shemas will further automate this process. With them, we propose to use an application configuration tool to automatically configure diverse AIS agents at both design time and run time.

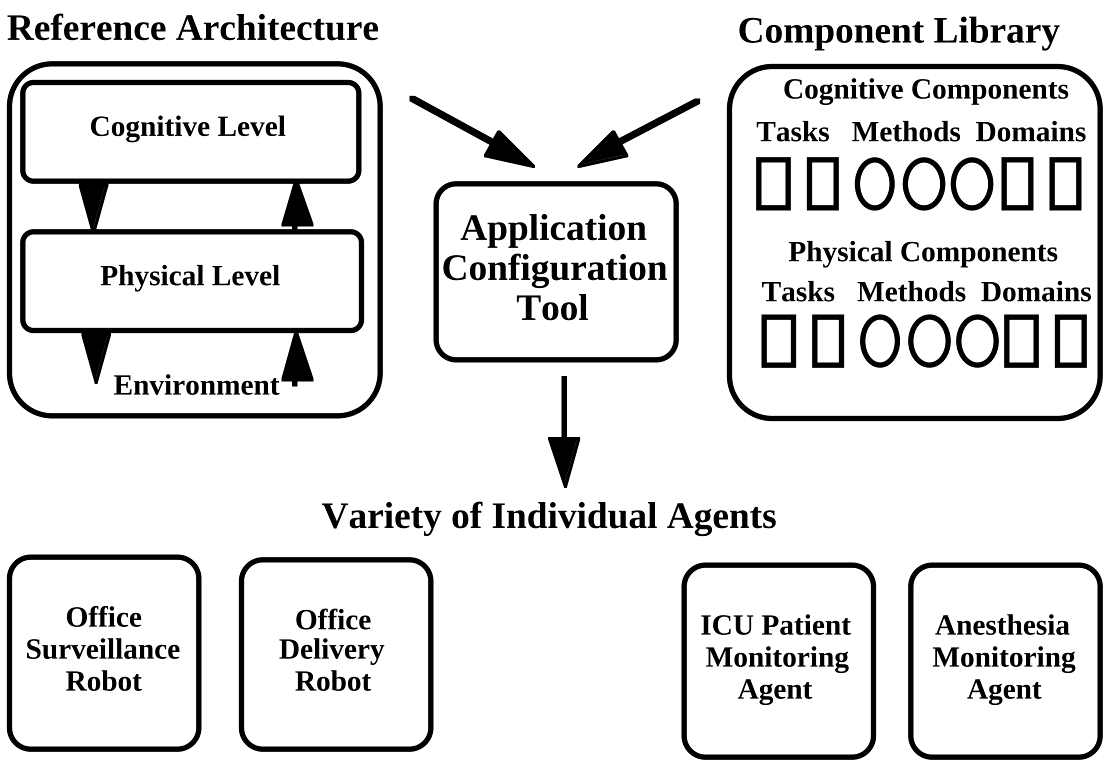

Figure 4. Building a variety of individual agents by configuring selected components within the reference architecture.

Application schemas generalize the abstract language of control plans (specified by tasks, parameters, and constraints), to describe the range of behavior capabilities an agent might need for its particular application. For every potentially useful task in an application, one or more schema entries specify the subject domain elements with which the task might be instantiated and the constraints that might be applied. Formally, an application schema is a 3-tuple: <task, domain, constraints>. For example, the schema entry `<plan-destination-sequence, KSL offices, {fast, optimal}>` specifies that the agent might have to perform a destination planning task, instantiated for any subset of the KSL offices, under either of two constraints: fast, real-time planning or guaranteed optimal plans. Given a schema entry, the application configuration tool would select the best available components for meeting each of the specified constraints, using the same component characterizations used by the meta-controllers for run-time selection among enabled components within an already configured agent.

Thus, the configuration tool and the meta-controller use the same language and semantics to describe and select task, method, and domain components. However the configuration tool operates at design time (or at run time in the case of system reconfiguration), assembling the repertoire of components an agent will need in order to function effectively over a period of time in its anticipated application environment. The meta-controller operates at run time, selecting the best available components in the agent's repertoire to apply in the present situation.

We envision a taxonomy of domain-specific skeletal schemas, corresponding to the AIS taxonomy sketched out in Figure 1. Each sub-domain schema would specialize and elaborate its parent schema. For example, an autonomous robot schema might describe an agent that can sequence a set of destinations using constraints, plan a path between two successive destinations, and control the robot's motion to follow a path. This schema would be quite general, specifying only a small number of components and a variety of constraints under which they might have to be performed. An office robot schema might describe a more specific kind of autonomous robot, specifying more tasks and a more restrictive set of constraints under which they may have to be performed. An application builder would begin by selecting the most specific schema that applied to the target application and then modify, specify, or elaborate it.

3. Empirical Studies

3.1 Overview of the Three Studies

We conducted three empirical studies of our approach using a Nomad 200 mobile office robot [39]. The Nomad 200 has hardware for sensing, moving, and communicating. For sensing, the robot has an orientable 2D laser beam and three sensor rings holding: 16 sonar sensors, 16 infra-red sensors, and 20 pressure-sensitive bumpers. For moving, it has effectors to control translational and rotational speed and rotation of the laser. For communicating, the robot has a voice synthesizer and can exchange electronic messages with other computers. The Nomad robot simulator provides the same interface and command language used by the actual Nomad robot for sampling sensor data and actuating effectors. It permits simulation of uncertainty for both sensing and moving. The three studies differ on several dimensions, summarized here (see Table 1) and described in detail in section 3.2.

In study 1, we automatically configured three different agents to control a simulated robot performing a surveillance job in office environment 1. Each agent instantiates the AIS architecture and a different subset of the components from component library 1. All of the components in library 1 were developed by our research group.

In study 2, we configured a single agent to control a simulated robot performing a delivery job in office 2. The agent instantiates the AIS architecture and all of the components in library 2, which includes a subset of library 1 components developed for the surveillance job, new components developed by our research group for the delivery job, and new components imported from another research group having no particular interest in either the surveillance or delivery job.

In study 3, we configured a single agent to control a physical robot performing the delivery job in our own laboratory, KSL Building C (office 3). The agent instantiates the AIS architecture and all of the components in library 3, which includes the same components as library 2, with three exceptions: the domain component is office 3 instead of office 2, the physical method components were modified to cope with real-world complexities and noise, and the learning component was excluded.

Table 1. Summary of differences between studies 1, 2, and 3.

	Study 1	Study 2	Study 3
Number of Agents	3 agents	1 agent	1 agent
Robot Embodiment	simulated	simulated	physical
Agent Job	surveillance	delivery	delivery
Office Environment	office 1	office 2	office 3 (= KSL Building C)
Components Used	library 1	library 2 (= subset of library 1 + new)	library 3 (= library 2 + modifications)

As discussed in section 3.3, each of the three studies demonstrates, in its own way, the advantages of the AIS DSSA: promotion of the functional requirements of adaptive intelligent systems, facilitation of new application development, and support for system reconfiguration. As discussed in section 3.4, the three studies together demonstrate the cumulative advantages of the DSSA in evolutionary development efforts.

3.2 Details of the Three Studies

3.2.1 Study 1: Surveillance with Three Different Simulated Robots

3.2.1.1 The surveillance job

The surveillance job requires an agent to respond to two kinds of electronic messages received asynchronously at run time, basic surveillance instructions and alarm signals. Basic surveillance instructions designate the regular destinations for the job, which are a subset of the potential destinations on the agent’s map, and constraints the agent must satisfy in visiting any regular or alarm destinations (e.g., pair-wise ordering constraints, relative frequencies, etc.). Alarm signals identify alarms occurring “now” at any subset of potential destinations on the map (possibly including regular destinations). To do its job well, a surveillance agent must perform three activities. It must repeatedly visit each of the regular destinations in accordance with the constraints among them. Whenever alarms occur, it must visit the alarm destinations as quickly as possible, in accordance with the constraints. When possible, it must acquire new knowledge (e.g., metric map information, cases of destination sequences) and exploit both new knowledge and new behavioral methods (e.g., new planning or navigation methods) to improve its performance.

3.2.1.2 Component library 1

For this study we developed an initial component library containing components for the tasks, methods, and subject domain knowledge useful for the surveillance job.

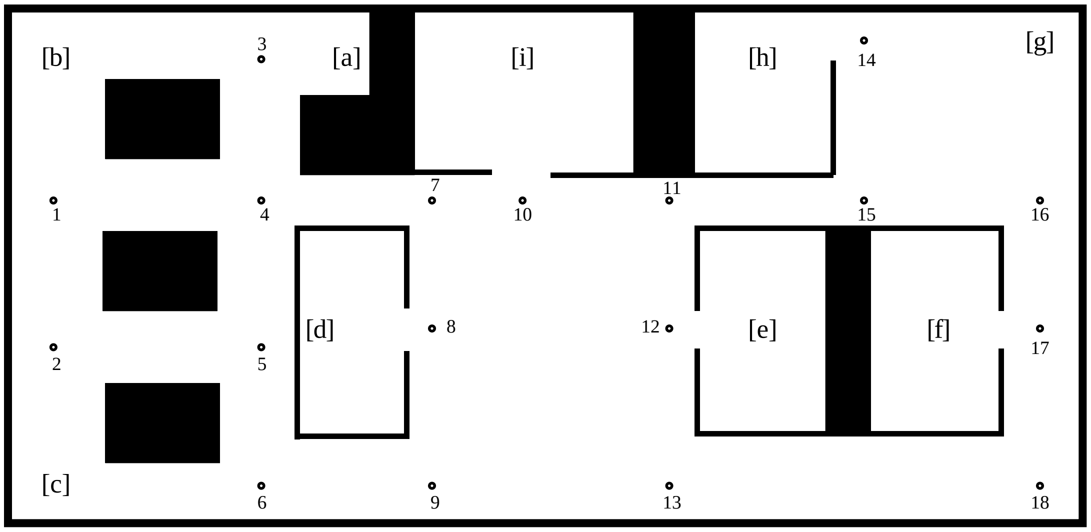

Figure 5. Office 1 subject domain used in study 1.

The domain of study 1 is office 1 (Figure 5). The cognitive domain component is a topological map that represents office 1 as a set of potential destinations (shown as alphabetic characters), intervening nodes (shown as numbers), and connecting paths. The physical domain component includes metric information about all objects and spaces, but no information about potential destinations.

Library 1 also includes components representing seven cognitive and physical tasks required for the surveillance job (Table 2) and 1-3 alternative methods for each of these tasks (Table 3). While some of these methods are powerful and robust, others are rudimentary or only simulate part of their ostensible functionality. Our purpose in creating and presenting them here is not to claim anything about the methods themselves, but to illustrate the kinds of methods that might exist in a component library and how they can be configured in our AIS architecture to produce a variety of surveillance agents.

Table 2. Cognitive and physical task components in library 1.

	Input	Output
Cognitive Tasks:	*(from physical level)*	*(to physical level)*
(1) Assess communication	New message received = Regular destinations or Alarm destinations	New goal = New set of destinations, Context-specific constraints
(2) Plan reasoning	Changed goal = New goal or Prior goal achieved	New reasoning plan = Sequence of reasoning tasks, Context-specific constraints
(3) Sequence destinations	New goal	New destination sequence = Sequence of destinations under [constraints]
(4) Plan routes	New destination plan	New route plan = Sequence of paths [a->b] under [constraints]
(5) Monitor execution	New perceived node	New physical command = Navigate [next path] under [constraints]
Physical Tasks:	*(from cognitive level or environment)*	*(to cognitive level)*
(6) Navigate	New node perceived *(from environment)* and Neighboring node *(from cognitive level)*	At new node, New node perceived, Other conditions perceived
(7) Interpret messages	New electronic message *(from environment)*	New perceived problem = Regular destinations or Alarm destinations

Table 3. Method components for cognitive and physical tasks in library 1.

	Methods	Requirements	Advantages
Cognitive Tasks:			
(1) Assess communication	Message driven	Messages	
(2) Plan reasoning	Skeletal planning	Skeletal plans	
(3) Plan destination sequence	Case-based planning Generative planning	Relevant cases Computation time	Fast planning Optimal, complete, case learned
(4) Plan routes	Graph search-D Graph search-U	Node-path graph Marked graph + Real time	Short distance & time Learns new paths
(5) Monitor execution	Step through	Sequential plan	
Physical Tasks:			
(6) Navigate	Dead reckoning Feedback-control Mapping nav.	path known Sonar, infra-red Sonar, infra-red, laser	Very fast Fast, safe Safe, path learned
(7) Interpret messages	Template based	Alarm template	

We summarize the seven tasks and their associated methods as follows. Note that, in all cases, the enabling and execution of methods are decoupled; enabling is automatic and inevitable, given the relevant events, whereas execution is discretionary and depends on a meta-control decision.

(1) Assess communication. When a newly perceived communication gives regular surveillance instructions, the agent sets a goal to visit the specified set of regular destinations, with the context-specific constraints “safe, learning, fast.” When a new problem signals an alarm, the agent sets a goal to visit the specified alarm destinations, with the context-specific constraints “fast.”

(2) Plan reasoning. Given a new regular goal, the agent instantiates skeletal control plan R:

```
(a) <plan-destination-sequence, (reg-dest-set, reg-dest-sequence),
   {safe, learning, fast}>
(b) <plan-routes, (reg-dest-sequence, reg-path-sequence),
   {safe, learning, fast}>
(c) <monitor-execution, (reg-path-sequence),
   {safe, learning, fast}>
(d) Go to (b) to begin the next surveillance round.
```

Given a new alarm goal, the agent temporarily deactivates plan R and instantiates skeletal control plan A:

```
(a) <plan-destination-sequence,
   (alarm-dest-set, alarm-dest-sequence), {fast}>
(b) <plan-routes, (alarm-dest-sequence, alarm-path-sequence),
   {fast}>
(c) <monitor-execution, (alarm-path-sequence), {fast}>
```

In addition to its goal-specific dynamic plans, static cognitive plans always influence the agent to plan reasoning and assess communication.

(3) Sequence destinations. Given a new goal, a case-based method retrieves a previously generated sequence for the specified destinations for reuse. It is fast but only applicable when a

relevant case is available. Alternatively, a generative method applies the relevant static constraints to construct the optimal sequence for the specified destinations and stores that sequence as a new case. This method is slower, but works for every goal.

(4) Plan routes. Given a new destination sequence, two alternative methods search the topological graph of traversable space (e.g., corridors). Method D searches for the most direct route to minimize travel distance and time. Method U searches for a route that includes previously untravelled paths, within some general constraints on distance, to create an opportunity to learn metric information in new regions.

(5) Monitor execution. Given a new route, a single method steps through a specified route plan, requesting the physical level to navigate along each successive leg of the journey while satisfying the specified constraints, and perceptually confirming its arrival at each expected next location.

(6) Navigate. Given a new leg to traverse, a dead reckoning method uses metric knowledge of the environment along a planned path to move quickly to the next node with little or no sensing. Alternatively, a reactive feedback-control method uses sonar and infra-red sensors to move more slowly, but more safely. An mapping method uses sonar and infra-red to move even more slowly, using the laser to gather metric map information. (We simulate this learning by prestoring all metric path information and making it accessible to the agent only after it traverses1 a given path using the information-gathering method.) All three methods asynchronously send whatever sensor data they acquire to the cognitive level.

(7) Interpret messages. Given a new electronic message, a single template-based method interprets the message.

3.2.1.3 Illustration of agent 1a's performance

We present excerpts from the performance of a particular agent, Agent 1a, which has all of the task, method, and subject domain components of library 1. Section 3.3.2 describes differences among the three agents configured in study 1.

In round 1 (Figure 6), Agent 1a is in office 1 for the first time, positioned at node 6. It receives an electronic message that specifies: (a) a set of regular surveillance destinations: d, e, and i in Figure 6; and (b) all constraints that apply to destinations in office 1. Agent 1a assesses its situation given this message, committing to the new goal and an appropriately ordered set of constraints: safe, learning, fast. Given the new goal, Agent 1a instantiates skeletal reasoning plan R (for regular destinations, rather than plan A for alarm destinations), which guides its subsequent cognitive behavior. To plan its destination sequence, Agent 1a uses generative planning (its only applicable method because it has no learned cases) and learns the resulting plan as a new case. For route planning, it chooses graph search U (rather than graph search R) to satisfy its learning constraint. Agent 1a then monitors its own plan following, sending each successive plan step to the physical level for implementation and waiting after each one to perceive that it has arrived at the intended destination. Since all paths are unlearned, it uses its mapping method for each successive instance of the navigate task.

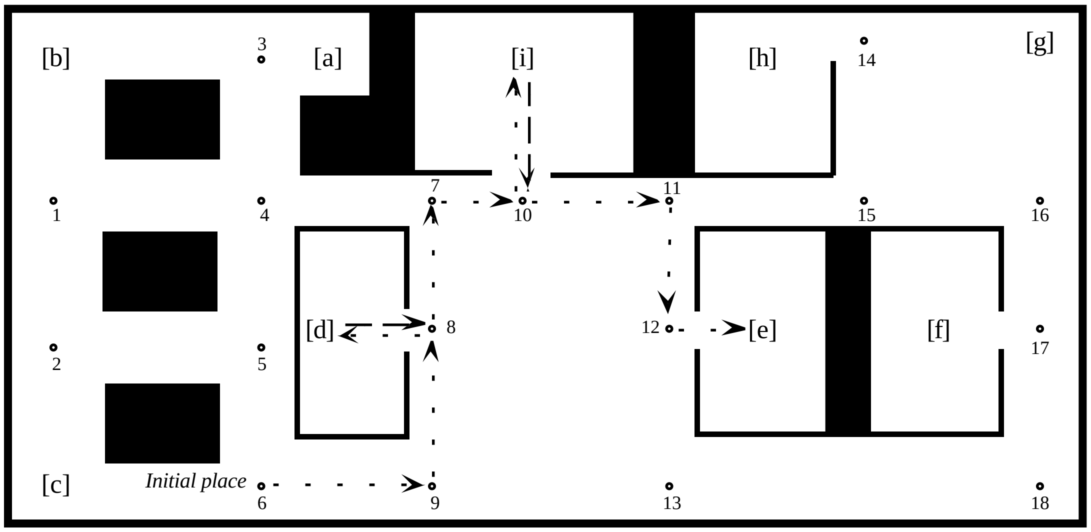

Figure 6. In round 1, agent 1a makes a regular surveillance round, visiting nodes in the sequence: 6, 9, 8, d, 8, 7, 10, i, 10, 11, 12, e. Dotted and dashed lines indicate use of mapping nav. and feedback control methods.

In round 2 (illustrated in Figure 7), Agent 1a returns to step (b) of its reasoning plan to replan its route for a second surveillance round. It plans a route from its current location, e, back to its first regular destination, d, and then chooses graph search U (under its learning constraint) to try to plan a different route among its destinations. However, because all potential new paths are too far out of the way, Agent 1a plans the same paths after all. It monitors plan following at the physical level, using its mapping method to learn the new paths from e to d, but using its faster feedback control method on already learned paths.

As Agent 1a approaches destination d, it receives a message reporting alarms at destinations a and b. Its static reasoning plans (plan reasoning and assess the situation), guide it to reason about the alarms, while monitoring its current plan. Agent 1a assesses its new situation, committing to the new alarm goal and a single constraint: fast. It replaces its current reasoning plan, instantiating skeletal reasoning plan A for the new alarms. It uses its generative method to plan the sequence of alarm destinations and learns the plan as a new case. Given its constraint to be fast, Agent 1a now chooses graph search D for path planning to minimize planning time and distance and maximize speed. It monitors plan following for the alarm destinations at the physical level. It uses feedback control to navigate along every path on its route to the alarm destinations. However, guided by the constraints of reasoning plan step d (safe, learning, fast), it uses mapping nav. to learn new paths on the return trip.

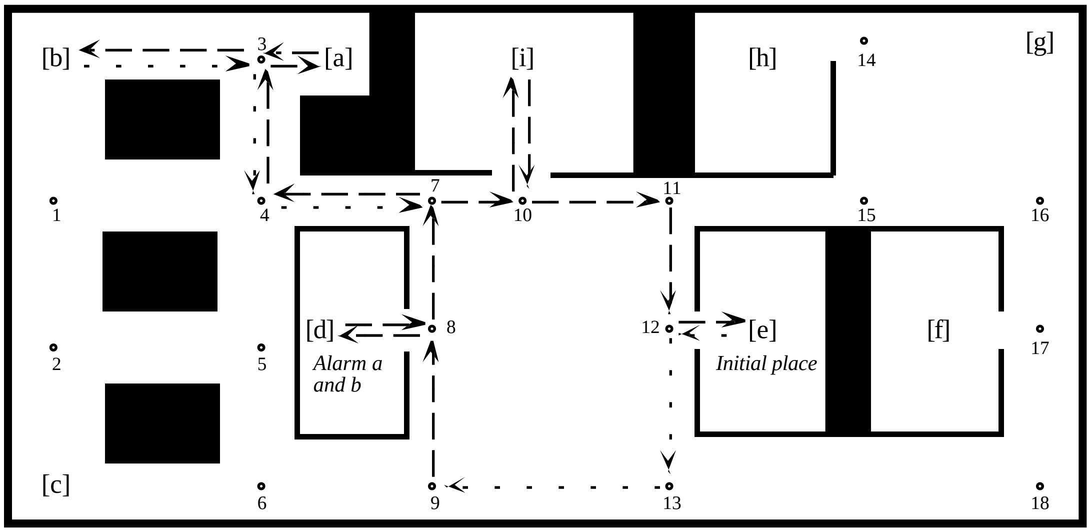

Figure 7. Agent 1a begins its second surveillance round, interrupting it to respond to an alarm, visiting nodes in the sequence: e, 12, 13, 9, 8, d, 8, 7, 4, 3, a, 3, b, 3, 4, 7, 10, i, 10, 11, 12, e. Dotted and dashed lines indicate use of mapping nav. and feedback control methods.

In round 3 (illustrated in Figure 8), Agent 1a encounters the same situation as in round 2: alarms at destinations a and b. But this time, it responds faster because it has learned the knowledge required by its faster case-based and dead reckoning methods. Again, while traversing known paths on its return from the alarms, Agent 1a uses its safer feedback-control method.

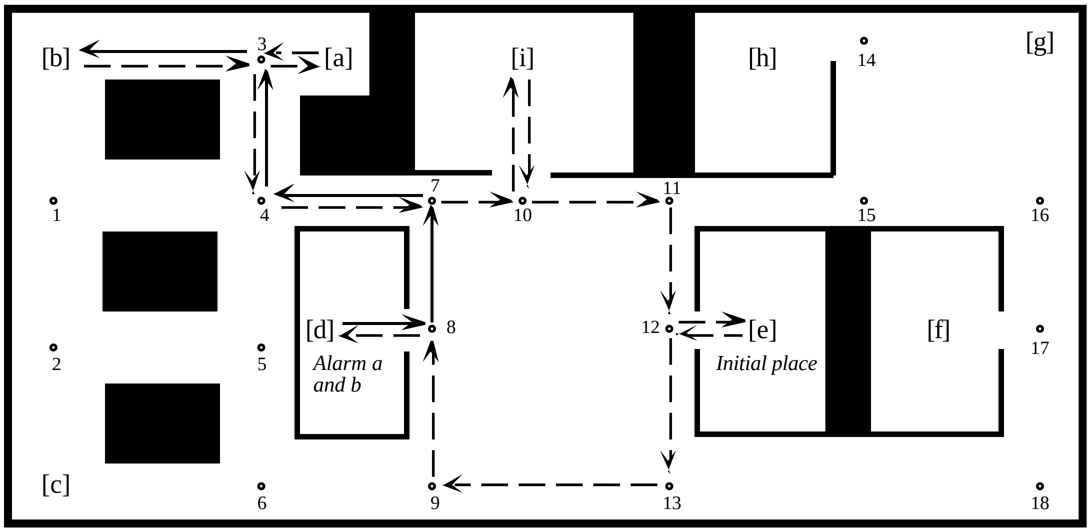

Figure 8. In round three, Agent 1a exploits recently learned subject domain knowledge to respond faster to a repeat alarm, visiting nodes in the sequence: e, 12, 13, 9, 8, d, 8, 7, 4, 3, a, 3, b, 3, 4, 7, 10, i, 10, 11, 12, e. Dotted, dashed, and solid lines indicate use of mapping nav., feedback control, and dead reckoning methods.

3.2.2 Study 2: Delivery with a Simulated Robot

3.2.2.1 The delivery job

The delivery job requires an agent to respond to electronic messages received asynchronously at run time. Messages may require the agent to: (a) deliver a verbal or e-mail message to a person in a known location; (b) deliver a document to a person in a known location; or (c) create and deliver copies or slides of a document to a person in a known location. The agent also can set goals for itself to learn the physical dimensions of physical objects newly detected in the course of its deliveries. Each goal may have an associated priority or a real-time deadline, which the agent tries to satisfy. The agent also tries to minimize time and distance traveled, to learn passively at all times, and to learn deliberately (pursue learning goals) when convenient.

3.2.2.2 Component library 2

For study 2, we developed library 2 for the new delivery job and the new environment. Some of the components are duplicates or extensions of components from library 1. Others are new components implemented by us or by members of another laboratory at Stanford.

The cognitive domain component for study 2 includes a map (see Figure 9) representing the office 2 environment. Unlike the map for study 1, it does not include any nodes. However, it is augmented with symbolic information about where particular people are located and in which offices particular functions (e.g., copying) can be performed. Again, the physical domain component includes metric information about static objects and spaces in the environment, but no semantic information. Dynamic or unanticipated objects do not appear in either component.

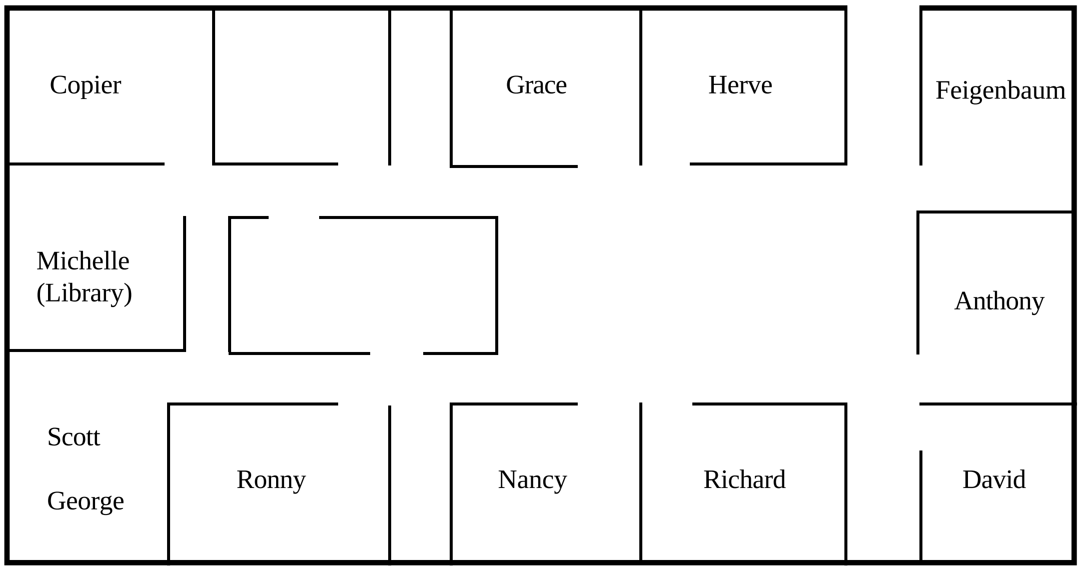

Figure 9. Office 2 subject domain of study 2.

Table 4 characterizes the ten task components used in the delivery job. Four of these correspond to or slightly elaborate on task components from library 1. Assess communication was extended to interpret a greater variety of requested tasks. Plan tasks is an extension of plan destination sequence to handle the increased variety of tasks and to create both linear and non-linear plans. Monitor plan execution was extended to monitor these more complex plans. Navigation was generalized to take coordinates, rather than map nodes, as arguments. Exactly one method was used for each task in this study, but multiple methods per task could easily be incorporated.

Table 4. Cognitive and physical tasks components used in Study 2.

	Input	Output
Cognitive Tasks:		
(1) Assess situation based on perception	New perceived sensor data	If unanticipated obstacle, New goal = Learn it
(2) Assess situation based on communication	New message received	New goal = Requested goal, priority, deadline
(3) Group goals	New goal	If compatible with pending goal or goal group, Make new goal group
(4) Plan tasks	New goal group	New plan = Sequence of physical tasks to achieve all goals in goal group
(5) Schedule real-time tasks	New plan includes RT tasks	Incorporate the RT tasks in the real time schedule
(6) Modify control plan	New goal or goal achieved	New reasoning plan = Add/remove reasoning tasks for goal
(7) Monitor plan execution	Perceived completion of task	New physical command = Next planned physical task
Physical Tasks:		
(8) Navigation	New navigation task	Go to and perceive commanded destination
(9) Interpret messages	New electronic message	Parsed instruction
(10) Learn map	New sensor data	Detect & record unanticipated obstacles

3.2.2.3 Illustration of agent 2's performance

We present excerpts from the performance of agent 2, starting at point a in Figure 10.

Agent 2 receives an electronic message: “Tell David: The book Human Information Processing is not available at this time, priority = 4.” It makes Plan 1, to go to David's office to deliver the message and begins monitoring execution.

Reaching point b, Agent 2 receives two new messages: “Bring 5 copies of Feigenbaum's computer industry paper to Anthony, priority = 3.” and “Tell Michelle: The meeting is canceled, priority = 3." Since the goals are independent, agent 2 makes separate plans for achieving them, Plans 2 and 3, while continuing execution of Plan 1.

Reaching point c, Agent 2 detects an unexpected obstacle (the box marked x). It adds a symbolic representation of the new obstacle to its cognitive map and metric representation of its perception (the heavy border) to its physical map. It creates a goal to learn the obstacle's complete dimensions.

At the same time, Agent 2 receives another message: "Bring Feigenbaum's expert systems paper to Michelle, priority = 5." This new goal is related to the two pending goals: it requires fetching something from Feigenbaum's office and delivering something to Michelle. Therefore, Agent 2 discards Plans 2 and 3 and creates Plan 4, which merges common sub-goals to achieve all three goals more efficiently. At point d in Figure 10, Agent 2 interrupts its execution of Plan 1 to begin executing Plan 4, which has higher priority.

Completing the first goal of Plan 4 (delivering Feigenbaum's expert systems paper to Michelle), Agent 2 goes to the copy room to make copies of Feigenbaum's computer industry paper. While it is making the copies, Agent 2 receives three new messages: "Tell Scott: The meeting has started, priority = 5, deadline = now;" "Tell Nancy: The workshop is full, priority = 5, deadline = soon;" and "Bring the book Building Expert Systems to Richard, priority = 2, deadline = today." Agent 4 begins scheduling the tasks it must perform for each of these goals to meet their deadlines. Since delivering the message to Scott has the deadline "now," it works on that goal first and interrupts its execution of Plan 4 (at point e) to begin executing the newly created Plan 5. At the same time, it constructs Plans 6 and 7 for its other new goals.

On completing Plan 5, Agent 2 executes Plan 6 in order to meet its deadline, "soon." Since Plan 7 has a longer deadline, "today," Agent 2 can handle its pending plans in priority order for the time being. It resumes the interrupted Plan 4, resumes the interrupted Plan 1, then executes Plan 7 (not shown in Figure 10). Agent 2 also will construct and eventually execute a plan to achieve its learning goal for box x.

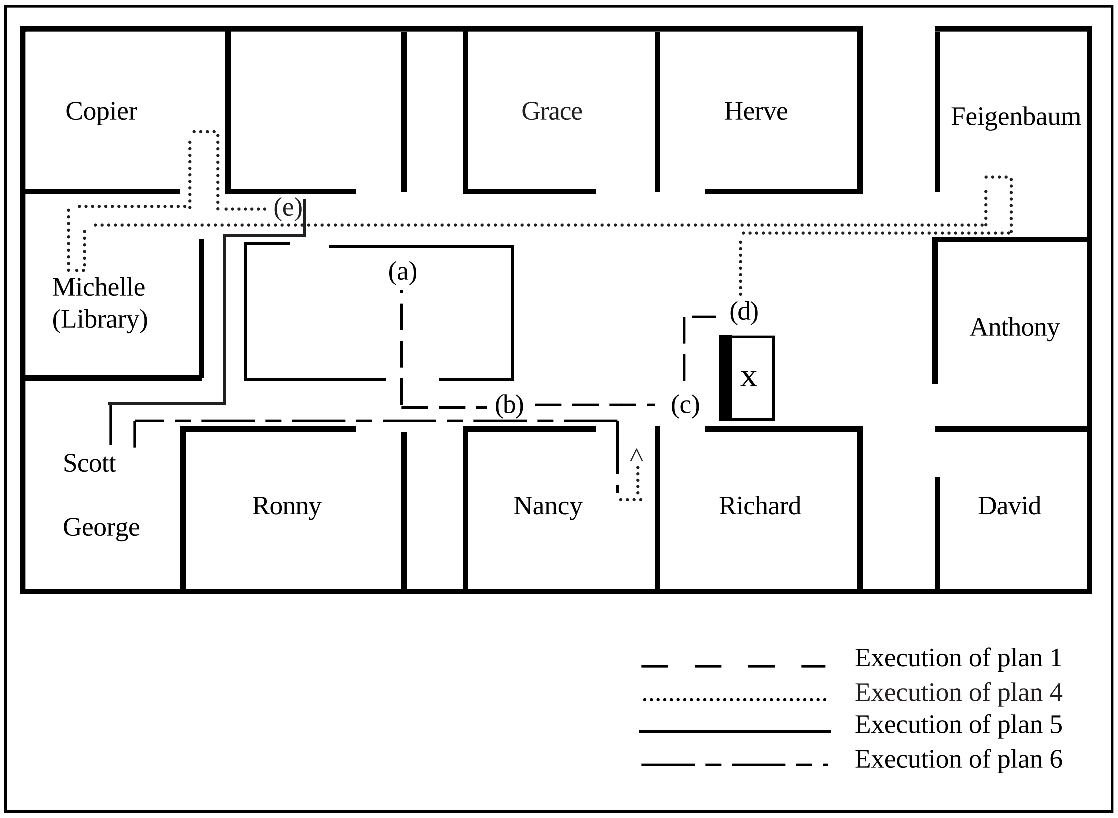

Figure 10. Excerpt of agent 2's performance of the delivery job in office 2.

3.2.3 Study 3: Delivery with a Physical Robot

In study 3, a real Nomad 200 robot performs the same delivery job in a real building, KSL Building C. Library 3 contains the same components used in study 2, except that the domain components describe the new office (Figure 11) and the navigation component was made more robust to handle real-world complexity—sensor noise and error in effector control. The map learning component needs similar improvements, which we have not made yet. Except the learning, Agent 3 replicates all of the delivery behavior of Agent 2.

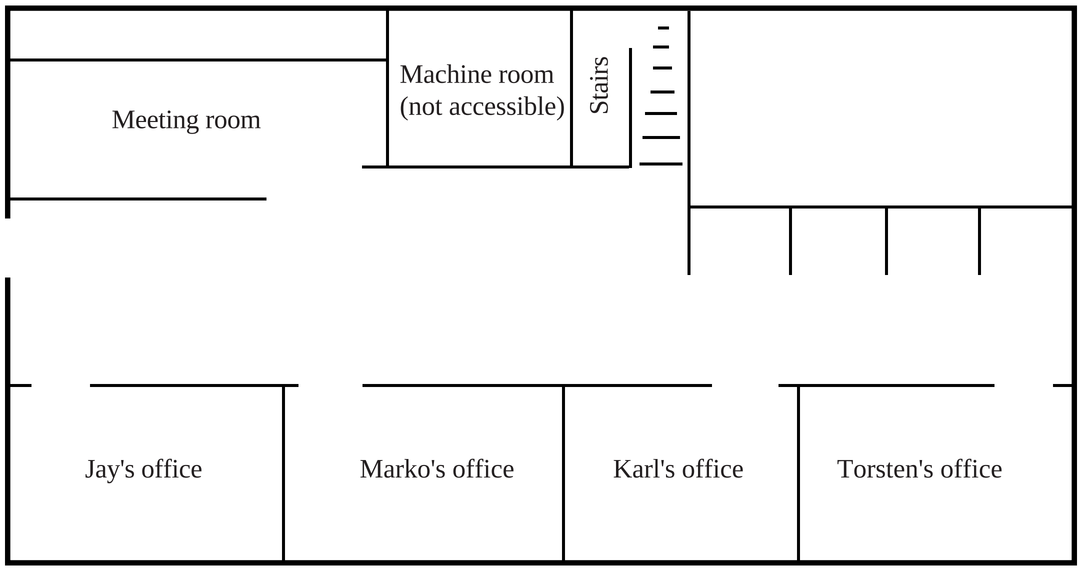

Figure 11. Office 3 (KSL Building C) domain used in study 3.

3.3 Results of the Three Studies

3.3.1 Meeting AIS Requirements

We briefly describe how the agents from these studies demonstrate the basic functionality required by the broad class of adaptive intelligent systems. (See [12, 13, 14, 16, 19] for further discussion of AIS functionality.)

In all three studies, agents perform concurrent perception, action, and cognition tasks, such as making new plans while navigating to a destination and being informed of a new goal. Based on their relative priorities and deadlines, agents “juggle” activities related to multiple, competing and complementary, asynchronously-arriving goals. Agents dynamically interrupt their execution of both cognitive and physical plans in favor of plans driven by other goals having higher priorities or shorter deadlines. Agents deliberately coordinate tasks relevant to related sets of goals and opportunistically choose the best available methods for tasks based on resource availability and performance constraints. Other things being equal, agents in all three studies focus attention on higher-priority and shorter-deadline activities. Agents automatically use new knowledge, such as new cases or new map information acquired at run-time. Finally, in all three studies, agents construct dynamic control plans to guide their cognitive behavior, which in turn influence their construction of plans to guide their physical behavior.

Although it is not the goal of this paper to comparatively evaluate alternative agent architectures, it is worth noting that other researchers have proposed and experimented with interesting agent architectures, for example [11, 6, 28]. Each of these architectures is designed to address a particular model of required agent functionality, overlapping but not identical with our model of AIS functionality. For present purposes, our AIS architecture can be distinguished as the only architecture that: (a) provides a uniform reasoning framework for constructing, modifying, and following abstract control plans at run time; (b) supports agents in a broad range of application domains (with the exception of Soar); and (c) directly supports design-time and run-time configuration of diverse agents out of reusable components.

3.3.2 Design-Time Configuration of Diverse Agents

We created a simple menu-based interface that allows an application builder to select library components for automatic configuration in an instance of the AIS architecture. If a user selects a sufficient set of task, method, and domain components for the job at hand, the resulting agent is ready to run immediately. For example, to perform the surveillance job of study 1, a sufficient set of components must include all seven task components in library 1, at least one method component for each task, and the relevant office 1 domain components.

Since multiple methods are available for three of the tasks in library 1, it is possible to configure different surveillance agents for different surveillance jobs. Table 5 describes three jobs that differ in their available resources and performance constraints, along with the three job-specific agents we configured. Job 1a provides ample computation, space, and power and imposes all three performance constraints. Therefore, we configured all available methods to give agent 1a maximum run-time flexibility. Job 1b limits space and computation, but imposes no deadlines. Given the space limitation, we configured only the most useful subset of method components in agent 1b: the case-based method because it is computationally efficient; Graph-

Search-D because new path exploration is unnecessary; Feedback-control because mapping navigation and dead reckoning are unnecessary. Job 1c limits space and power, but imposes no deadlines. Again we configured only the most useful subset of method components in agent 1c: the generative method because it produces optimal destination sequences; Graph-Search-D because new path exploration is unnecessary and consumes power; and Feedback-control because mapping navigation and dead reckoning are unnecessary and consume power. Each of the three agents performs its surveillance job well under the anticipated run-time conditions. Agents 1b and 1c fail in the obvious ways under unanticipated conditions (e.g., deadlines). Agent 1a flexibly adapts to the full range of run-time conditions.

Table 5. Configuring different agents for different surveillance jobs..

	Job 1a	*Job 1b*	*Job 1c*
Resources Available:			
Computation	Ample	Limited	Ample
Space	Ample	Limited	Limited
Power	Ample	Ample	Limited
Performance Constraints:			
Efficiency	Important	Important	Important
Safety	Important	Important	Important
Deadlines	Some deadlines	No deadlines	No deadlines
	Agent 1a	*Agent 1a*	*Agent 1a*
Components Configured:			
Sequence destinations	Case-based Generative	Case-based	Generative
Plan routes	Graph Search-D Graph Search-U	Graph Search-D	Graph Search-D
Navigate	Dead Reckoning Feedback-control Mapping navigation	Feedback-control	Feedback-control

Although we did not demonstrate the selection process for the full set of components from all three studies, it would be straightforward to set up the same menu-based interface to do so. However, even with a small number of components, the user needs more assistance than an unstructured menu. The application schemas discussed in section 2.3 would allow the user to begin with a basic job-appropriate agent configuration and to complete it by making a smaller number of more structured decisions. As Winograd [38] observes: “The fundamental use of a programming system is not in creating sequences of *instructions* for accomplishing tasks (or carrying out algorithms), but in expressing and manipulating *descriptions* of computational processes and the objects on which they are carried out.” Developing formal languages for describing application requirements, application schemas, and available components is an important research direction [23, 26].

3.3.3 Run-Time Reconfiguration

Our studies demonstrate two kinds of run-time reconfiguration of application systems. First, in all three studies, agents acquired new domain knowledge (e.g., destination sequence cases, metric map information) both incidentally and deliberately. Agents used newly acquired knowledge at the first subsequent opportunity to enable method components that required it. Second, in study 1, we loaded the case-based planning method and the dead reckoning navigation method into a running agent initially configured without them. The agent used the new methods at the first subsequent opportunity to perform tasks for which they were appropriate and for which it had the necessary knowledge. In both cases, agents exploited the new knowledge and method components with no modification of existing components. In general, we can add or remove components from the agent's IB/WM at any time. As long as the agent has at least a minimally complete set of components for the job at hand, it will continue performing to the best of its ability with the evolving set of available components.

3.3.4 Cooperative Evolutionary System Development

Together, our three studies provide a case-study of the efficacy of the AIS DSSA approach on a multi-person, evolutionary system development project. Study 1 presented an initial set of system requirements, and as frequently happens in successive stages of real system development projects, study 2 presented a new, but not disjoint, set of system requirements. We designed and implemented the study 2 agents by reusing the entire AIS architecture and a few of the components we had implemented for study 1. Study 3 dealt with the switch from simulation to a real, physical robot, requiring substantially more robust perception and navigation methods. We simply replaced the subject domain maps with those for the new environment and our collaborator modified the navigation component. Agent 3 immediately succeeded in controlling the Nomad robot's performance of the delivery job in the KSL offices.

Although our agents were experimental software systems, and some of their components were quite simplistic, they were not “toy” systems. The AIS architecture comprises the BB1 software system at the cognitive level (30K lines of Lisp code) and a newly implemented control loop at the physical level (400 lines of Lisp code). Components for agent 1 added another 4K lines of Lisp code. Components for agents 2 and 3 added another 7K lines of Lisp code and 6K lines of C code.

Our core team comprised five individuals who collaborated on the design and implementation of all agents in all three studies. Despite the small group size, group dynamics and the academic research environment amplified the problem of coordination and manifested many of the same issues that arise in larger, professional software teams. Members of the group possessed no shared model of computation, used different programming styles and even different languages, worked asynchronously, with no member of the group acting as manager or coordinator of the many separate sub-projects, and prior to this project had not worked together. The complete system comprises a heterogeneous set of hardware and software environments requiring complex interfaces among concurrently executing parts. Multiple Lisp processes communicate within a Lisp interpreter, and multiple Unix processes (C programs) communicate through Unix sockets, many across different workstations. Lisp and C routines call each other, and both interface with the robot simulator, and a Unix workstation communicates via radio-modem with the physical robot and its on-board 486 processor running MS-DOS. Furthermore, long periods of time, ranging up to a few months, especially between the three studies, separated intervals of work on the project over the course of a year. Group members worked less than half time on average directly on the systems and architectural issues described here, devoting the rest of their time to other aspects of the research or unrelated projects. In addition, as indicated below, we imported several software components: the BB1 system (designed and implemented by Barbara Hayes-Roth, Micheal Hewett, Lee Brownston, and others) and the goal grouper (designed and implemented by Scott Benson, a PhD student of Prof. Nils Nilsson) and the navigation components used in studies 2 and 3 (designed and implemented by David Zhu, a post-doctoral associate of Prof. Jean-Claude Latombe).

Thus, the present experiments provide a representative case study and demonstration of the AIS DSSA's support for cooperative evolutionary software development.

4. Conclusions

Reusable software components have been a persistent, but elusive goal of the software engineering community [4, 8, 30]. Despite progress on tools to organize, index, and select reusable software components, few existing components are amenable to such manipulations. Moreover re-engineering existing software components is expensive, while yielding only limited actual reusability [36, 34, 35]. In response, we argue that: *In order to realize the promise of reusable software, we need to design highly reusable software right from the start.*

The general DSSA approach aims to factor large classes of applications into reusable reference architectures and components. Our AIS DSSA elaborates on this prescribed factorization by imposing additional design constraints on the design of both the reference architecture and components, with attendant increases in reusability. As a result, it supports applications throughout the class of adaptive intelligent systems characterized in the taxonomy of Figure 1. Besides the office robots applications discussed in this paper, our research group has used the AIS architecture in two other sub-domains, intelligent monitoring systems and layout design systems (see section 1). As these examples illustrate, the domain of adaptive intelligent systems is larger than the typical domains targeted by DSSAs, which are closer in size to AIS subdomains such as mobile office robots or semiconductor manufacturing. The important question when evaluating the expected utility of designing a DSSA for a particular domain is to what extent the shared properties of instances of the domain demand common architectural properties and common components of expertise.

Our AIS reference architecture’s use of the basic blackboard organization specifically enables the integration and interoperation of diverse components, as it was originally designed to do [7]. The additional characteristics of the BB1 dynamic control model provide the necessary additional support for flexible run-time configuration and meta-control. It is particularly useful as a foundation for AIS applications that must integrate diverse methods for performing multiple loosely-coupled tasks in dynamic, uncertain environments and for complex systems whose components evolve over time. The architecture reflects many person-years of design, implementation, testing, debugging, and documentation. Thus, blackboards in general, and BB1 in particular represent cost-effective architectural foundations for large classes of applications.

Our three-way decomposition of expertise also extends opportunities for component reuse throughout larger application domains. In each of the three sub-domains we studied, we have been able to transfer task and method components among more or less diverse domains (e.g., office surveillance vs. delivery; protein structure vs. construction site layout; intensive care versus several manufacturing domains). Most of the components we developed for these applications are fully application-independent and reusable in other domains as well. The actual savings realized through component reuse is component-specific. For example, many of the components developed for the office robots domain are quite simple and easy to recreate; however, others (e.g., the planner and navigation components) are moderate-sized programs (2700 lines of Lisp code and 6000 lines of C code, respectively) that represent a substantial cost saving in each reuse application. Components for the monitoring and layout design applications are larger. However, since our interest has focused on architectural support for reuse and reconfiguration, we have not invested heavily in developing "industrial strength" components, which would provide a much higher savings on each reuse opportunity.

One of our architecture’s advantages, its support for run-time selection of components, also carries its most significant cost. Run-time selection requires time. Other things being equal, a system configured with only the optimal components and a hardwired control regime will outperform a system with multiple components from which to choose. To the extent that system designers can determine the exact set of situations in which a system will find itself, they can try to deduce the ideal configuration of system components and the optimal control regime. This is the familiar tradeoff between design-time (human) effort and run-time (system) effort and, to some degree, we can decide where to position a given development effort on the continuum. However, as the complexity, generality, and evolutionary life-span of target applications increase, so does the intrinsic requirement for run-time adaptation.

Of course, for both the reference architecture and the components, there is a marginal cost of developing software that, in addition to meeting its primary specifications, is highly reusable. This cost must be weighed against the benefits of subsequent reuse applications [3]. In our experience, making the AIS reference architecture application-independent contributed only a small part of its initial cost and may actually have simplified the tasks of designing and implementing it. Similarly, making components reusable through our three-way decomposition of expertise increases development cost only slightly, if at all. Most important, by offering a large domain of past and prospective reuse opportunities and substantial cost savings for each reuse application, the AIS DSSA offers a substantial expected return on reuse investment.

In conclusion, our AIS DSSA captures a powerful, but costly software architecture and a growing repertoire of components, making them available for reuse and reconfiguration throughout a large application domain.

References

1. Albus, J.S. Brains, behavior, and robotics. Peterborough, N.H. : BYTE Books, 1981.
2. Albus, J. S. Outline for a theory of intelligence. *IEEE Transactions on Systems, Man, and Cybernetics*, 21, 473-509, 1991.
3. Barnes, B.H., Bollinger, T.B. Making reuse cost-effective. *IEEE Software*, January, 1991, pp. 13-24.
4. Biggerstaff, T., and Richter, C. Reusability framework, assessment, and directions. *EEE Software*, March, 1987, pp. 41-49.
5. Chandrasekaran, B. Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert*, 3, 23-30, 1986.
6. Connell, J. H. SSS: A hybrid architecture applied to robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1992.
7. Erman, L., Hayes-Roth, R., Lesser, V., and Reddy, R. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12, 213-253, 1980.
8. Freeman, P. (ed.) Software Reusability. Los Alamitos, Ca.: IEEE Computer Society Press, 1991.
9. Garlan, D., and Shaw, M. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering,* Volume I, World Scientific Publishing Company, 1993.
10. Garvey, A., Hewett, M., Johnson, M.V., Schulman, R., and Hayes-Roth, B. BB1 User Manual. Stanford University, Knowledge Systems Laboratory Technical Report KSL-86-61, 1986.

11. Gat, E. Integrating planning and reacting in a heterogenous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the 10th National Conference on Artificial Intelligence,* 1992.

12. Hayes-Roth, B. A Blackboard for Control. *Artificial Intelligence*, 26, 251-321, 1985.

13. Hayes-Roth, B. Architectural foundations for real-time performance in intelligent agents. *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, 2, 99-125, 1990.

14. Hayes-Roth, B. Opportunistic control of action in intelligent agents. *IEEE Transactions on Systems, Man, and Cybernetics*, in press, 1993.

15. Hayes-Roth, B. A Domain-Specific Software Architecture for a class of intelligent patient monitoring agents. KSL Technical Report, 1994a.

16. Hayes-Roth, B. An architecture for adaptive intelligent systems. *Artificial Intelligence, Special Issue on Agents and Interactivity,* to appear, 1994b.

17. Hayes-Roth, B., Buchanan, B.G., Lichtarge, O., Hewett, M., Altman, R., Brinkley, J., Cornelius, C., Duncan, B., and Jardetzky, O. Protean: Deriving protein structure from constraints. *Proceedings of the National Conference on Artificial Intelligence*, 1986a.

18. Hayes-Roth, B., Garvey, A., Johnson, M.V., and Hewett, M. A modular and layered environment for reasoning about action. Stanford University: Technical Report No. KSL 86-38, 1986b.

19. Hayes-Roth, B., Lalanda, P., Morignot, P., Pfleger, K., and Balabanovic, M. Plans and behavior in intelligent agents. KSL Technical Report, 1993.

20. Hayes-Roth, B., Washington, R., Ash, D., Hewett, R., Collinot, A., Vina, A., and Seiver, A. Guardian: A prototype intelligent agent for intensive-care monitoring. *Journal of Artificial Intelligence in Medicine*, 4, 165-185, 1992a.

21. Hayes-Roth, F., Erman, L.D., and Hayes-Roth, B. Distributed intelligent control and communication. *Proc. of the IEEE Symposium on Computer-Aided Control System Design,* Napa, Ca., 1992b.

22. Hewett, R., and Hayes-Roth, B. Physical systems modeling for integrated reasoning, modularity, and reuse. *International Journal of Expert Systems*, 7, 1994.

23. Landauer, C., and Bellman, K. The role of self-referential logics in a software architecture using wrapping. *Proc. of the Irvine Software Symposium*, Irvine, CA: University of Irvine, 1993.

24. Mettala, E. Domain specific software architectures. Presentation at ISTO Software Technology Community Meeting, June, 1990.

25. Morris, W. (ed.) The American heritage Dictionary of the English Language. Boston: Houghton Mifflin Company, 1981.

26. Murdock, J. Matching and individuation for model-based diagnosis of manufacturing equipment. Stanford University Ph.D. Dissertation, 1994.

27. Murdock, J., and Hayes-Roth, B. Intelligent monitoring of semiconductor manufacturing. *IEEE Expert*, 6, 19-31, 1991.

28. Newell, A. *Unified Theories of Cognition,* Cambridge, Mass., Harvard University Press, 1990.

29. Pardee, W.J., Schaff, M.A., and Hayes-Roth, B. Intelligent control of complex materials processes. *Journal of Artificial Intelligence in Engineering, Design, Automation, and Manufacturing*, 4, 55-65, 1990.

30. Prieto-Diaz, R., and Arango, G. (eds.) Domain Analysis and Software Systems Modeling. Los Alamitos, Ca.: IEEE Computer Society Press, 1991.

31. Sipma, H., and Hayes-Roth, B. Integrating qualitative and quantitative methods for model-based diagnosis, prediction, and explanation. Stanford University Working Paper, 1993.

32. Terry, A., Hayes-Roth, F., Erman, L., Coleman, N., and Hayes-Roth, B. Overview of Teknowledge's domain-specific software architecture program. Palo Alto: Teknowledge, Inc. Technical Report, 1994.

33. Tommelein, I.D., Hayes-Roth, B., and Levitt, R.E. Altering the SightPlan knowledge-based systems. *Journal of Artificial Intelligence in Engineering, Automation, and Manufacturing*, 6, 19-37, 1992.

34. Tracz, W. (ed.) Software Reuse: Emerging Technology. Los Alamitos, Ca.: IEEE Computer Society Press, 1991.

35. Tracz, W. Software reuse technical opportunities. Paper prepared for DARPA Software Program PI Meeting, 1992.

36. Tracz, W. Software reuse maxims. *ACM Software Engineering Notices*, 13, 1988, pp. 28-31.

37. Tracz, W., Coglianese, L., Young, P. A domain-specific software architecture engineering process outline. SIGSOFT Software Engineering Notes, 18, 40-49, 1993.

38. Winograd, T. Beyond programming languages. In D. Partridge (ed.), Artificial Intelligence and Software Engineering. Norwood, N.J.: Ablex Publishing Corp., 1991.

39. Zhu, D. Nomadic host software development environment (Unix Version 1.1). Mountain View, Ca.: Nomadic Technologies, Inc., 1992.