ABLE: A toolkit for building multiagent autonomic systems

by J. P. Bigus
D. A. Schlosnagle
J. R. Pilgrim
W. N. Mills III
Y. Diao

This paper describes a toolkit for building multiagent autonomic systems. The IBM Agent Building and Learning Environment (ABLE) provides a lightweight Java™ agent framework, a comprehensive JavaBeans™ library of intelligent software components, a set of development and test tools, and an agent platform. We describe a series of agents built using ABLE components and present three case studies of applications using the ABLE toolkit. The Autotune agent is a closed-loop controller agent that supports hierarchical distributed control. The Subsumption agent defines specific behaviors or strategies and can be plugged into a multiagent subsumption infrastructure. The Autonomic agent architecture features sensors and effectors for interacting with the external environment, layers of reflexive, reactive, and adaptive subsumption agents, components that dynamically model the autonomic system itself and its environment, and components for emotions, planning, and executive-level decision-making. By using the ABLE component library to build agents running on the ABLE distributed agent platform, we discuss how we can incrementally add new behaviors and capabilities to intelligent, autonomic systems.

It has been over 50 years since Alan Turing described the prototypical test for intelligent machines. In Turing's view, a computer could be called intelligent if it could pass as a human while conversing via a computer terminal. For researchers delving into the mysteries of human and machine intelligence, the so-called Turing Test has been both an inspiration and a millstone around their necks since that time.

The field of artificial intelligence (AI) started out with great hopes and fanfare, beginning with the Dartmouth conference on AI in 1956. The next decade saw an exploration of both symbolic and neural network approaches to knowledge representation, reasoning, and machine learning. By 1970, all was going so well that Marvin Minsky declared in a *Life Magazine* article, "In from three to eight years we will have a machine with the general intelligence of an average human being. I mean a machine that will be able to read Shakespeare, grease a car, play office politics, tell a joke, have a fight. At that point the machine will begin to educate itself with fantastic speed. In a few months it will be at genius level and a few months after that its powers will be incalculable." Although his vision may still be valid, his timing was off by several decades. Even though substantial progress has been made on the pieces of intelligence—speech recognition, natural language understanding, knowledge representation, machine reasoning, machine learning, emotion, and speech generation—putting the pieces together to create human-level intelligence has proven to be difficult.

The artificial intelligence technology pendulum has swung from whole-hearted devotion to symbol-processing techniques, to reactionary forays into neural networks and other subsymbolic approaches, and on to biologically inspired genetic algorithms and fuzzy reasoning. Today, most researchers admit that a combination of technical approaches must be used to achieve human-level performance. In *The Society of Mind*, Minsky described a set of mechanisms that he called mental agents that operate in parallel and that compete and cooperate to yield human intelligence.[1] In his subsumption architecture, Brooks describes an architecture of behavioral layers that provides robust function and supports reactive behaviors in mechanical robots.[2] More recently, papers by Sloman and by Caulfield and Johnson have described architectures for consciousness or self-aware systems that rely on layered architectures with emotional components.[3,4]

Clearly, a monolithic software architecture using a single technology will not bring us closer to our goal. In our work, we are exploring an incremental approach for developing intelligent autonomic systems—systems that have self-awareness and can reason about their internal components and state.[5] Autonomic systems must adapt to environmental changes and strive to improve their performance over time. They must be robust and be able to routinely overcome internal component failures. Autonomic systems must interact and communicate with other systems in a heterogeneous computing infrastructure. Our approach to building autonomic systems is based on combining autonomous intelligent agents in a well-structured way. This approach mirrors the structure of the human brain wherein there are clearly defined, function-specific processing centers connected by forward and backward communication channels and adaptive feedback loops.

In this paper, we briefly describe an architecture that combines elements of these approaches and melds them into a coherent, scalable architecture that we believe will lead to robust deployed autonomic systems. These systems rely on sensors to obtain input from the world and effectors to take action and make changes to the world. There are memory components providing short-term, long-term, and associative memory functions. There are reflexive, reactive, and goal-oriented proactive components. There are components for reasoning, planning, and learning new behaviors from interactions with the world. There is an emotional component that associates feelings with internal states and influences decision-making and learning processes. This architecture reflects much of what is known about how people think and process information, including the role emotions play in our reasoning.[6,7]

This paper is organized as follows. First, we describe the Agent Building and Learning Environment (ABLE), a software architecture and framework, component library, development tooling, and agent platform for constructing autonomous intelligent agents and multiagent systems. We then present two application case studies, a system administration application using multiple agents, and a diagnostic application. Next, we describe several derivative ABLE agents including the Autotune control agent, a Subsumption agent, and an Autonomic agent that is an ABLE-based architecture for incrementally building autonomic systems. We then discuss the current status of ABLE and our plans for enhancing the toolkit and for implementing our autonomic system architecture.

Agent building and learning environment

The recent surge of interest in software agents has prompted a corresponding increase in toolkits for constructing them. Although many projects use a "roll your own" approach in which each agent is uniquely hand-coded, there are benefits to using a component-based approach. The Java** language has several characteristics that make it an ideal platform for implementing agents: code portability resulting from its use of a standard virtual machine, support for object-oriented programming techniques, native support for multithreading, and introspection of object properties and methods. In addition, the JavaBeans** component specification enables the creation of reusable Java components with well-defined interfaces and behaviors.

Quite a few agent toolkits and multiagent platforms are available for both educational and commercial use. The CIAgent framework developed by one of the authors is a lightweight agent framework written in the Java language and intended for educational use.[8] The Java Agent Template Lite (JatLite), developed at Stanford University, is focused on communications-related issues of agent systems. The IBM AGLETS* mobile agent framework, now an open source project, provides a Java platform for creating mobile agent applications. AgentBuilder** from Reticular Systems (a part of IntelliOne Technologies), is an integrated software development toolkit for constructing belief-desire-intention (BDI) agents in Java. The ZEUS agent-building toolkit developed by British Telecommunications plc features a Java component library with a planning and scheduling system, support for multiple interaction protocols, and a set of tools for building agents. The Foundation for Intelligent Physical Agents (FIPA), an international standards body working for interoperability between agents and agent platforms, has defined specifications for agents, agent management services, and agent communications languages.[9] Several projects implement FIPA compliant agent platforms. The FIPA Open Source (FIPA-OS), developed by Nortel Networks, is an open source implementation of the FIPA agent communication language and agent platform. The Java Agent DEvelopment (JADE) framework, developed at CSELT S.p.A. (now Telecom Italia Lab, or TILab) is another FIPA-compliant multiagent toolkit. For a more detailed overview of these agent environments, see Bigus and Bigus.[8]

The Agent Building and Learning Environment[10] is a Java-based toolkit for developing and deploying hybrid intelligent agent applications. Hybrid approaches synergistically draw on the strengths of each technology while compensating for any weaknesses. For example, rules have the advantage of explicitly defined knowledge, but they can be brittle and inflexible. Neural networks, in contrast, can adapt or learn from inputs, but the learned knowledge is often difficult to make explicit. The value of combining multiple techniques such as neural network learning with rule-based inferencing has been demonstrated by prior work.[11]

The ABLE toolkit was designed to provide a fast, reusable, and scalable architecture for the construction of intelligent software components and agents. A fundamental design philosophy of ABLE is that successful intelligent agents will require multiple reasoning and learning techniques. ABLE builds on the standard JavaBeans model by defining a lightweight framework for agent behavior. ABLE has a component library of data access, machine learning, machine reasoning, and optimization algorithms packaged as JavaBeans, known as AbleBeans. ABLE provides a Java Swing-based GUI (graphical user interface) for creating and configuring AbleBeans, and for constructing and testing the agents built from them. ABLE also provides an agent platform for deploying agents across a distributed computing system. By building a comprehensive suite of intelligent JavaBeans and tooling for easily combining and connecting those beans, ABLE permits developers to explore the applications of software agents and their behaviors in distributed multiagent systems. The ABLE toolkit has been available for downloading from the IBM alphaWorks* site since May 2000.[12] In the following sections, we describe the fundamental design and architectural attributes of ABLE.

ABLE agent framework

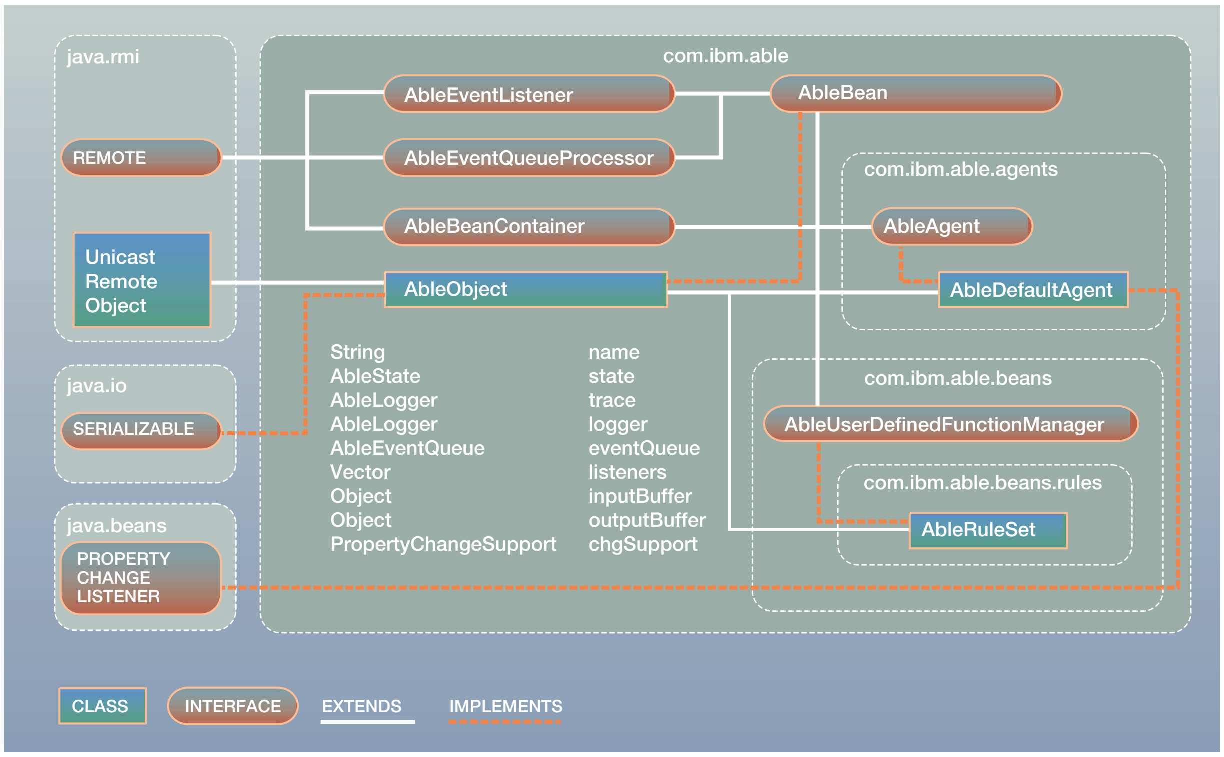
The ABLE agent framework is a lightweight software architecture that allows algorithms to be packaged as JavaBeans that can be deployed as standard Java components or as autonomous agents. Figure 1 shows the set of Java interfaces and base classes comprising the ABLE framework.

AbleBeans are standard JavaBeans components used in the ABLE framework. The AbleBean Java interface defines a set of common attributes (name, comment, state, etc.) and behavior (standard processing methods such as init( ), reset( ), process( ), quit( )), allowing AbleBeans to be connected to form AbleAgents. AbleBeans are connected using three fundamentally different methods: data flow, events, and properties.

Data-flow or buffer connections are used to wire together AbleBeans using a data-flow metaphor. Each AbleBean can have an input buffer and an output buffer that are implemented as Java Objects. A set of AbleBeans can be connected by buffer connections forming a directed, acyclic graph. The set of AbleBeans is then processed in sequence starting at the root of the tree. Each AbleBean takes the data from its input buffer, processes the data, and places the data in its output buffer. This data-flow mechanism is extremely fast and is very useful for applications such as neural networks that have a natural data-flow processing paradigm.

Event connections are used to register an object as a listener on an AbleBean. AbleBeans support synchronous and asynchronous event processing using AbleEvents, which extend the Java EventObject class. Each AbleBean has an event queue on which it receives AbleEvent notifications or action requests to be processed. Each AbleEvent contains a Boolean flag that indicates whether the event should be handled synchronously on the caller's thread or asynchronously by placing it on the receiver's event queue and processing it on a separate thread. Every AbleEvent can be used as either a data event with an associated eventId (event identifier) and data object, or as an action event, with an associated action string and data object. Data notification events hold

Figure 1 ABLE agent framework classes and interfaces



data and allow AbleBeans to inform one another of complex state changes. Action events allow method-invocation with arguments and contain an action field that maps to a method name on the notified Able-Bean. The AbleEvent argument object is also passed to the method on the receiving AbleBean. Thus, any method can be called on a listening AbleBean through this mechanism. Although event processing adds some overhead, it is more flexible than hard-coded method calls between AbleBeans.

Property connections are used to synchronize two different properties residing in two different Able-Beans. Whenever the first property value is changed via a setter method, the second property on the second bean is also changed via its setter method.

AbleBeans use Java serialization for persistence. All data-flow, event, and property connections are preserved during the passivation and activation cycles. The ABLE run-time environment has a well-defined set of properties that enable serialized AbleAgents to be portable.

AbleEvents. Figure 2 shows the data fields in the AbleEvent class. The AbleEvent class provides the means to send data between agents, to request actions to be performed by other agents, or to request transactions with results returned either to the original requesting agent or to some other agent. This design allows the ABLE event-processing infrastructure to be used to implement a variety of agent interaction models. For example, a dialog between two agents can be supported by exchanges of AbleEvents where the action holds the request and the replyTo and replyWith fields are used to correlate the responses. Alternatively, an intermediary agent could send a request to one agent and have that agent send its response to a third agent or back to the original requester. Another scenario is to have an agent broadcast its response to an event to multiple agents by specifying a list of agents on the replyTo field. Or, a single agent can send out requests to multiple agents, and use the transactionID field to correlate the responses to the original requests. To summarize, the base ABLE event-processing framework can

Figure 2 The AbleEvent class

```
AbleEvent(
   Object source,           // the source object or sender of the event
   int id,                  // type=ACTION, DATACHANGED, EOF, TRANSACTION
   String action,           // name of the action method
   Object arg,              // passed to action method
   boolean async,           // put on queue (true) or run on caller's thread (false)
   Object replyTo,          // null, single bean, list of beans
   String replyAction,      // used as action in reply event
   String transactionId     // used as work identifier (copied to reply)
)
```

be used to implement almost any agent communication design pattern.

AbleObject. The AbleObject class provides a base implementation of the AbleBean interface, defining the standard behavior for all AbleBeans provided with the ABLE toolkit. The AbleObject class extends Java UnicastRemoteObject and implements the AbleEventListener, AbleBean, and AbleEventQueueProcessor remote interfaces. It contains an instance of an AbleEventQueue that handles the optional autonomous timer facility as well as asynchronous event-processing functions for the bean. The timer allows an AbleBean to run autonomously by periodically going to sleep and then waking up to see whether anything needs processing.

The AbleEventListener interface defines two main processing methods: processAbleEvent( ) and handleAbleEvent( ). The first method takes an AbleEvent as an argument, examines its synchronous/asynchronous Boolean flag and processes it accordingly. The second method unconditionally processes the event in a synchronous manner. Default behavior is provided to interpret the action string as a method name and to invoke a method with that name on the bean, passing the argument object as a parameter.

Figure 3 shows the source code for an example AbleBean that extends the AbleObject base class. We import the *com.ibm.able* package and provide a no-argument constructor. The major methods that must be overridden include init( ), which performs one-time initialization; process( ), which is the method called to process the input buffers; and processTimerEvent( ), which is the method invoked when the bean is configured to run as an autonomous agent.

AbleAgent. One of the major decisions when creating an agent construction environment is the granularity of the agents. Many projects consider belief-desire-intention (BDI) agents to be the base line. In ABLE, we chose a model where the basic building blocks are functional software components but not complete agents. By making this choice, we can create customized agents with functionality and complexity suitable for their intended use.

The AbleDefaultAgent class provides a default implementation of the AbleAgent interface and defines the standard behavior for all AbleAgents provided with the ABLE framework. The AbleDefaultAgent class extends Java UnicastRemoteObject and AbleObject and implements both the AbleAgent and AbleBeanContainer remote interfaces.

AbleAgents are AbleBeans that are also containers for other AbleBeans. An AbleAgent has its own thread for processing events asynchronously. AbleAgents provide a useful abstraction for packaging a set of AbleBeans wired together to perform a specific function. This function is then available to other AbleBeans or AbleAgents through synchronous process( ) calls or through asynchronous event processing.

AbleAgents extend the AbleObject base class and implement the AbleBeanContainer and AbleUserDefinedFunctionManager interfaces. The AbleBeanContainer allows an AbleAgent to contain other AbleBeans and even other AbleAgents. This pow-

Figure 3 Sample AbleBean Java source code

```
import com.ibm.able.*;
public class SampleAbleBean extends AbleObject implements Serializable {

  public SampleAbleBean() throws RemoteException {
    // set processing options, data flow, timer, etc.
    this("SampleBean");  }

  public void init() throws RemoteException {
    // need to initialize state of this bean, algorithm vars, etc. -- do ONE TIME initializations
    // initialize asynchronous Timer (if used) and define Event processing behavior
  }

  public void process() throws RemoteException {
    // perform synchronous processing on caller's thread
  }

  public void processTimerEvent() throws RemoteException {
    // perform autonomous (asynchronous) processing on own thread
  }
}
```

erful design pattern allows extremely complex agents to be built from subagents and is exploited in our component library implementation. The AbleUserDefinedFunctionManager allows external software to be integrated with AbleAgents as sensors and effectors.

Note that alternate AbleAgent implementations could be developed with behaviors different from the AbleDefaultAgent. For example, we provide an AbleDefaultFIPAAgent that is an AbleAgent that provides all of the required FIPA agent behaviors, and the AutotuneAgent that enables hierarchical distributed control. We discuss the AutotuneAgent and additional AbleAgents in more detail later in this paper.

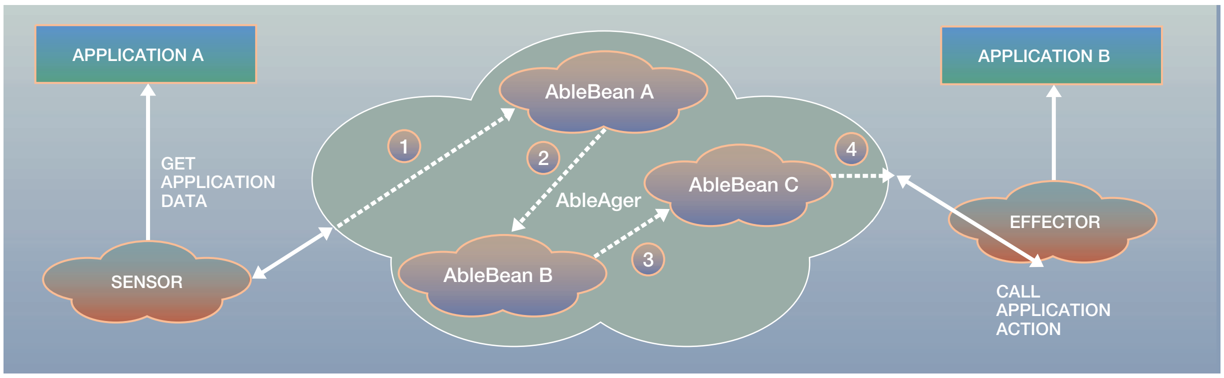
AbleAgents are situated in their environment through the use of sensors and effectors. In ABLE, sensors and effectors are AbleUserDefinedFunction objects that map to method calls on external Java objects. These methods usually call other application programming interfaces (APIs) to either obtain data (sensors) or take actions (effectors). AbleAgents are managers for sensors and effectors, and any contained AbleBeans can invoke those sensors and effectors. Sensors and effectors take arbitrary argument lists and return Java Objects to the caller.

A common scenario is for an AbleAgent to contain one or more beans that reference sensors and effectors. For example, in Figure 4, the AbleAgent contains three AbleBeans, a single sensor, and a single effector. AbleBean A first calls the sensor and obtains data from Application A. It processes the data and passes information to AbleBean B either through a direct method call, an event, or a property connection. AbleBean B processes these data and passes the data on to AbleBean C, which in turn invokes the effector, resulting in a method call on Application B.

ABLE component library

A fundamental piece of the ABLE system is the component library of AbleBeans. These include data access and filtering beans, machine learning algorithms, machine reasoning and inference engines, and higher-level data mining agents comprised of one or more core beans. In addition, ABLE contains a set of data type classes, defining Boolean, Categorical, Discrete, Numeric, and String literals, variables, and fields. This common data model is used by the beans in the component library. The set of core AbleBeans provided with the ABLE framework includes data beans, learning beans, and rule beans.

Figure 4 Example ABLE agent



Data beans. Data access and transformation beans are used to manipulate data for training and testing the learning and reasoning beans. They include:

- *Import*—reads space-, comma-, or tab-delimited data from flat text files
- *DBImport*—reads data from relational databases using JDBC** (JavaBeans Database Connectivity)
- *DataTable*—provides a view over an Import data set, with selected rows and columns
- *Filter*—filters, transforms, and scales data using translate template specifications
- *TimeSeriesFilter*—caches sequential data for use in time-series prediction
- *Export*—writes space-, comma-, or tab-delimited data to flat text files
- *DBExport*—writes data to relational databases using JDBC

Learning beans. The learning beans implement several different learning algorithms that can be combined with the data beans to provide lightweight data mining capabilities. They are:

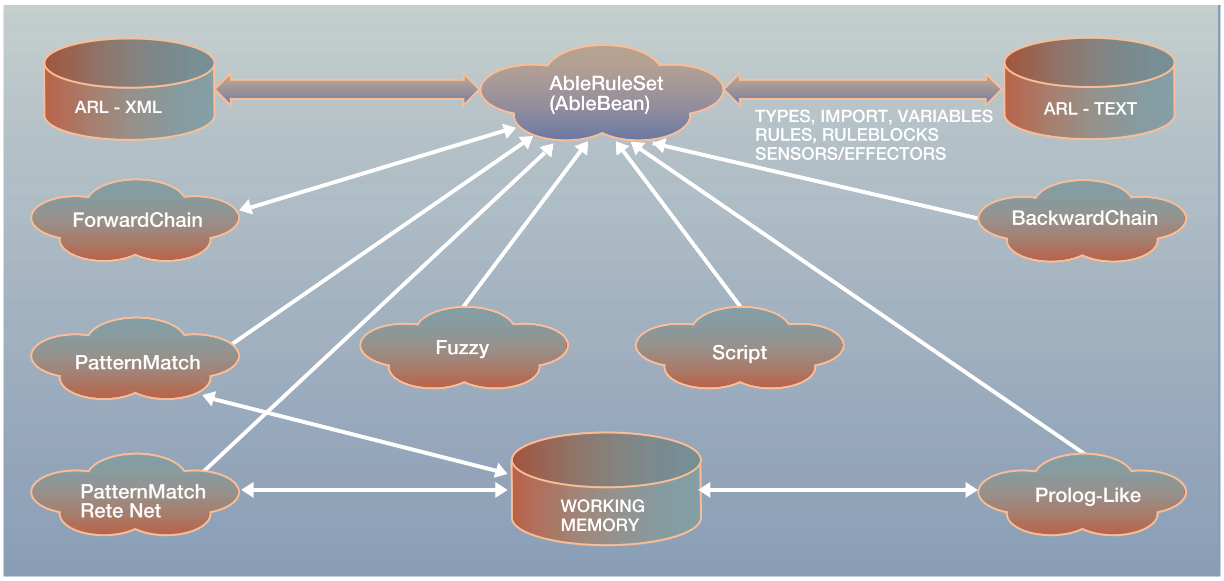
- *Back Propagation*—implements an enhanced back propagation algorithm with pattern and batch updates, hidden layer and output layer recurrence
- *Self-Organizing Map*—supports pattern and batch updates, and a Gaussian neighborhood function
- *Temporal Difference Learning*—supports sequence learning using a reinforcement learning algorithm
- *Radial Basis Function*—supports regression and classification using multiple basis functions with automatic Self-Organizing Map clustering of hidden layer weights
- *Naive Bayes Classifier*—supports incremental learning of discretized data using a Bayes statistics approach
- *Decision Tree*—supports tree-based classification of discretized data using the C4.5 algorithm

Rule beans. The ABLE Rule Language (ARL) defines a rich set of rule-based knowledge representation formats including scripting using simple assignments, if-then and if-then-else rules, when-do pattern match rules, and predicate style rules. ARL supports rule-blocks that are named groups of rules similar to macros. ABLE provides a wide range of inference engines to process the ARL rulesets.

As illustrated in Figure 5, ABLE Rule Language can be represented in text or Extensible Markup Language (XML) formats. The AbleRuleSet class parses the text or XML source into a set of AbleRuleBlock and AbleRule objects and instantiates an associated inference engine based on the inference method specified in the RuleSet. The ARL supports very tight integration with Java classes and objects allowing instantiation, access to data members on objects, and invocation of methods on objects from rules. Processors include:

- *Boolean forward chaining*—processes if-then rules using forward chaining
- *Boolean backward chaining*—processes if-then rules using backward chaining
- *Fuzzy forward chaining*—processes if-then rules containing linguistic variables and hedges and several types of fuzzy sets, and supports multistep chaining

Figure 5 AbleRuleSet bean and inference engines



- *Pattern Match engine*—processes when-do pattern match rules using forward chaining against a working memory
- *Pattern Match network*—processes when-do pattern match rules using the Reté network forward chaining algorithm against a working memory
- *Predicate engine*—processes predicate rules using a backchaining algorithm with backtracking (similar to Prolog)
- *Scripting engine*—processes assignments, if-then, if-then-else, while-do, and do-while rules in sequential order

Having a single ABLE Rule Language with pluggable inference engines provides many advantages. A single rule authoring and debugging environment can be used for multiple styles of inferencing. The same knowledge representation can be used for scripting agent behavior, dynamically constructing and configuring agents, and explicitly representing domain knowledge. Alternate implementations of the inference engines can be developed for use in specialized environments where memory or processing resources are constrained.

One of the most powerful aspects of the ABLE Rule Language design is the ability to seamlessly mix symbolic rule-based reasoning with subsymbolic neural network and other machine learning algorithms. A common view is that the subconscious processes of the human brain correlate to neural network approaches and that conscious thought is similar to symbol processing. A single ABLE rule set can reason symbolically about the outputs of multiple neural components. For example, sensory data can be fed into a neural network for clustering into similar groups, for classification into categories, or for prediction of trends. Rules can then process the outputs of the neural network, assign semantic labels to those outputs, and reason about the outputs. Rules could decide to kick off a learning episode in one or more neural networks or to take overt actions to change the external environment.

The ability of rules to invoke other AbleRuleSet beans allows hierarchical configuration and natural partitioning of knowledge into individual rulesets. This ability eases the burden on rule authoring and maintenance. The example AbleRuleSet in Figure 6 shows the Java-like syntax and structure of the ABLE Rule Language. Arbitrary Java classes can be imported into a ruleset, domain-specific function libraries can be loaded, and a variety of built-in and imported variable types can be defined and instantiated in the variables section. Data are passed into and out of the ruleset bean via the input and output statements. Rule methods or ruleblocks can be used to

Figure 6 A sample AbleRuleSet

```
ruleset AbleScriptExample {
   import com.ibm.myClass;   // use myClass as data type in ruleset
   library com.ibm.myLibrary; // each public method becomes a function

   variables {
      myClass myTypeVar = new myClass();  // creates an instance
      myClass myTypeVar2 = new myClass("name", "age", "whatever");
      Object BeanVar = new Object();
      Object Result = new Object();
   }
   inputs { myTypeVar } ;
   outputs { Result };

   void init() {
      // one time initialization rules here
   }

   void main() using Script {
      A1:  ObjectVar = createInstance("com.ibm.able.beans.rules.AbleBooleanRuleSet");
      A2:  ObjectVar.name =  "myRuleSet";
      A3:  Result = instantiateFrom(ObjectVar, "d:\\joe\\myRuleSet.arl");
      A4:  BeanVar = getBean(parent, "aBeanName");
      A5:  Result = init( parent, "aBeanName");
      A6:  Result = processBean( BeanVar);
   }
}

void idle() {
   // idle rules run when the main ruleblock quiesces
}
```

define one-time initialization rules (the init( ) block). The main( ) block allows the user to specify the inference engine that will process the rules. The order of evaluation is entirely determined by the inference engine that is selected. When the main( ) block completes, the optional idle( ) block is run.

Function-specific AbleAgents. In addition to the core AbleBeans, the ABLE component library provides a set of function-specific AbleAgents. Data and learning AbleBeans are combined to create neural classification, neural clustering, and neural prediction agents that can be used for lightweight data mining tasks. The set of standard function-specific agents provided with the ABLE framework includes:

- *Genetic search agent*—manipulates a population of genetic objects that may include AbleBeans
- *Neural classifier agent*—uses back propagation to classify data
- *Neural clustering agent*—uses self-organizing maps to cluster or segment data
- *Neural prediction agent*—uses back propagation to build regression models
- *Script agent*—uses the ABLE rule language to define complete agent behavior
- *JavaScript** agent*—uses JavaScript to define agent behavior

The NeuralPredictionAgent uses two Import beans to read training and test data from text files, two Filters to preprocess and postprocess the data, and a back propagation neural network to perform the regression function (shown later in Figure 8). The agent provides high-level functionality through its Customizer as it orchestrates the operation of the five Able-

Beans it contains. The user only needs to specify the source data files and corresponding meta-data files. When initialized, the NeuralPredictionAgent scans the source data and automatically generates the scaling and transformation templates used by the Filters to preprocess and postprocess the data passed through the neural network. Based on the number of input and output fields and their data representation, the neural network architecture is automatically configured. Data-flow connections between the AbleBeans pass data from the Import through the input Filter, through the neural network, and then through the output Filter. The user also specifies target error rates and the maximum number of training epochs. The asynchronous thread of the AbleAgent is used to automatically train the prediction model on a separate background thread and halts when the user-defined termination conditions are met.

Once trained, the NeuralPredictionAgent can be used to process data synchronously. If the model becomes stale after it is deployed, the application could easily force a retraining of the prediction agent, because the training process is automated.

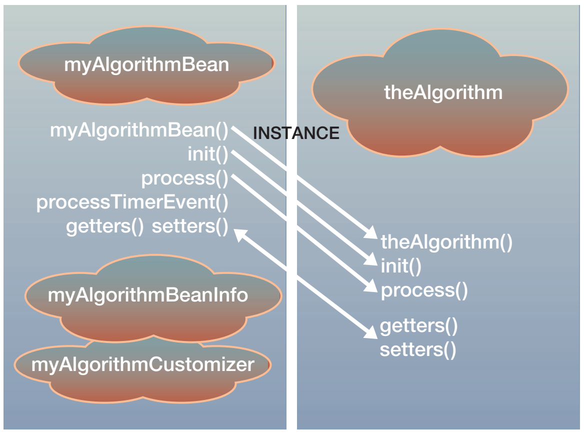
The NeuralClassifierAgent and NeuralClusteringAgent are constructed using multiple AbleBeans in a manner similar to the prediction agent but provide classification and clustering functions, respectively. The neural learning agents provide basic data mining functionality for use in other AbleBeans, giving agents the ability to segment, classify, and make predictions about their environments.[13]

Extending ABLE using custom beans

In addition to the core AbleBeans provided in the ABLE component library, users can easily wrap new or existing algorithms to create their own beans. Sets of domain-specific beans can be added to the ABLE Agent Editor and dynamically loaded from a JAR (Java ARchive) file. A simple design pattern requires that the algorithm object be wrapped by an AbleBean instance, a BeanInfo file be created to specify any members to be externalized, and a GUI Customizer class be provided to allow users to set any algorithm unique attributes.

This approach was used to incorporate the DecisionTree and NaiveBayes classifier learning components into ABLE. As shown in Figure 7, the three Java classes required for ABLE integration are the AbleBean class itself, the BeanInfo that defines bean properties and accessor methods, and the bean Customizer that provides the GUI used to set configuration properties on the bean. The AbleBean must contain an instance of the algorithm class, map the init( ) and process( ) methods to call functionally equivalent methods on the algorithm object, and also wrap any getter and setter methods used by the Customizer. This approach allows the algorithm code to remain unmodified while allowing it to be used as part of any ABLE solution.

Figure 7 AbleBean wrapper design pattern

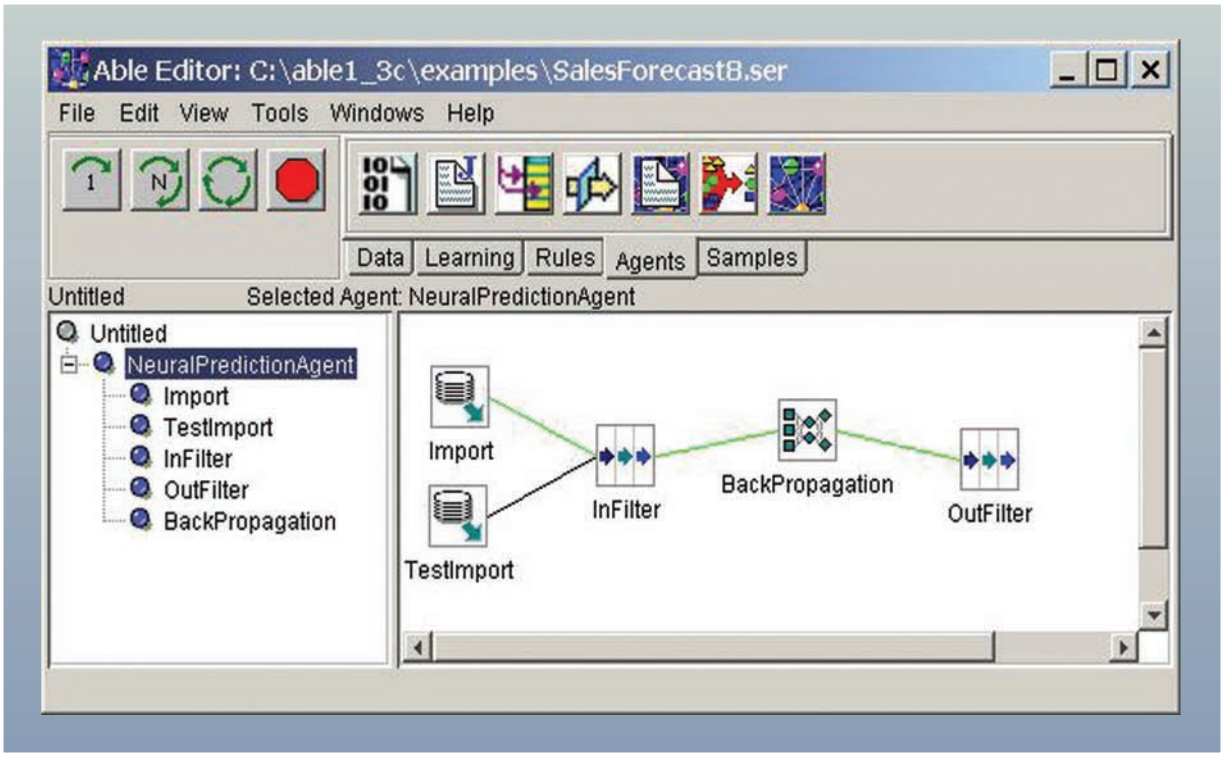


ABLE development tools

The ABLE Agent Editor is a Swing-based interactive development and test environment. It provides a tree view of the agent with drill down into contained beans as well as a canvas view of the agent with corresponding data, event, and property connections. Agents can be loaded, edited, and saved to external files using Java serialization. ABLE Inspectors provide text and graphic views of object data using Java introspection. Support is provided for adding custom inspector views in addition to standard line, bar, *x-y* plot, and pie charts.

The Agent Editor can graphically construct AbleAgents by using the library of core AbleBeans and AbleAgents as building blocks. Data-flow, event, and property connections can be added using the GUI environment. Agents can also be hand-coded and then tested in the ABLE Agent Editor. Each AbleBean provides a Customizer dialog that is used to configure it and to set property values. Figure 8 shows the ABLE Agent Editor with a single NeuralPredictionAgent bean loaded into a default agent. The user

Figure 8 The ABLE Agent Editor


Able Editor: C:\able1_3c\examples\SalesForecast8.ser
File Edit View Tools Windows Help
Data Learning Rules Agents Samples
Untitled
Selected Agent: NeuralPredictionAgent
Untitled
NeuralPredictionAgent
Import
TestImport
InFilter
OutFilter
BackPropagation
Import
TestImport
InFilter
BackPropagation
OutFilter

has drilled down into the NeuralPredictionAgent and is viewing the five AbleBeans it contains. These beans are displayed in the right-side canvas.

When the Agent Editor is started, it loads AbleBeans from JAR files. These beans contain properties specifying the page on the toolbar palette where the bean should be placed. Thus, users can easily provide their own custom AbleBeans and AbleAgents for use with the Agent Editor in combination with the core AbleBeans.

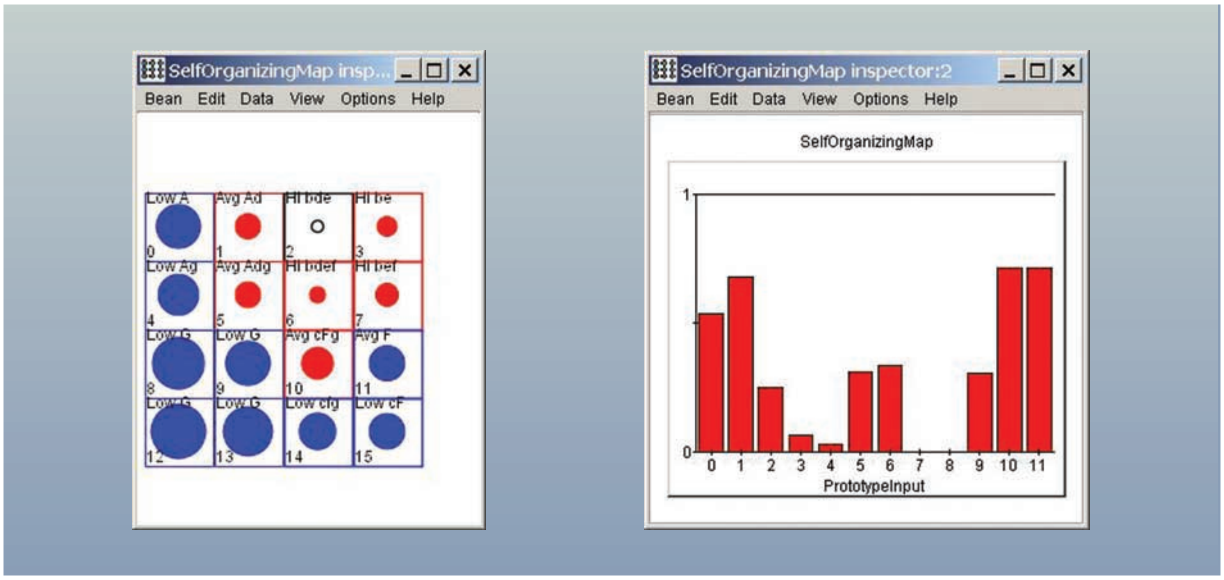
ABLE Inspector windows use introspection to display bean data members and state information. Inspectors provide text views as well as graphical views of the bean data. Views such as bar charts, line plots, and *x-y* plots are provided. Users can select one or more bean properties to be displayed, or one or more indexed properties such as arrays or vectors of objects. In addition, Inspectors allow users to select multiple data items for use in time-series displays. The Inspector caches the data points at each time step and displays the desired number of points. This function is very useful for observing time-series predictions and agent controller behavior over time.

Figure 9 shows two Inspectors from the MarketAnalysis example provided with the ABLE toolkit. The Inspector on the left shows the clusters of a SelfOrganizingMap neural network with labels and categories assigned to each cluster. The Inspector on the right shows a bar chart of the weights of the winning cluster.

ABLE agent platform

The ABLE agent platform provides a set of services for AbleAgents that form multiagent systems. The services include standard agent life-cycle transitions (e.g., create, suspend, resume, quit) as well as directory facilitator and agent communication functions. The ABLE platform is a distributed agent platform supporting agents on multiple physical systems that communicate using Java Remote Method Invocation (RMI). The ABLE agents can communicate with one another using the mechanisms described ear-

Figure 9 Example ABLE Inspectors



lier (AbleEvents or direct method calls) or with other FIPA-compliant agents and agent platforms through the use of the FIPA agent communication language.

As shown in Figure 10, the ABLE distributed agent platform corresponds to the FIPA abstract architecture. The current ABLE platform conforms to the FIPA 97 specifications. Work is in progress to adapt it to the more recent FIPA abstract architecture and conform to the Java Agent Services (JAS) being developed under the Sun Microsystems Java Community Process JSR 87. The JAS provides a set of Java interfaces and a reference implementation of the platform services required for a distributed agent platform that complies with the FIPA abstract architecture. The ABLE platform includes additional functionality covering agent life-cycle management, service registration, and agent security. The following services are provided as part of the standard services supplied by the ABLE agent platform:

- *Booter and Service Root*—provides the startup and root services to agents that want to communicate with the agent platform services and agents running on the platform
- *Naming Services*—provides a unique name for each agent registered with the platform
- *Transport Services*—provides a mechanism for agents to communicate via multiple underlying communication transports including Java RMI and HyperText Transfer Protocol (HTTP)
- *Directory Services*—provides a means for agents to register descriptions of themselves to allow other agents to find them and to find other agents
- *Life-cycle Services*—provides a Factory service that allows new types of agents to be added to the platform and for an administrator to create/start/suspend/resume/stop agents running on the platform. Support is also provided to move agents from one system to another on the platform.

The agent Console provides a centralized graphical user interface for administrators to access the directory services and life-cycle services on the platform. Agents can be created, configured, and deployed using the Console. Additional systems can be added or removed from the agent platform using the Console. Directory services can be queried to find the status of individual agents or of collections of agents based on service attributes.

ABLE application designs

The ABLE toolkit is quite flexible and can be used to add intelligence to applications in a variety of ways. One or more core AbleBean components could be used in an application to provide specific functions. Additional domain-specific AbleBeans such as novel

Figure 10 The ABLE distributed agent platform

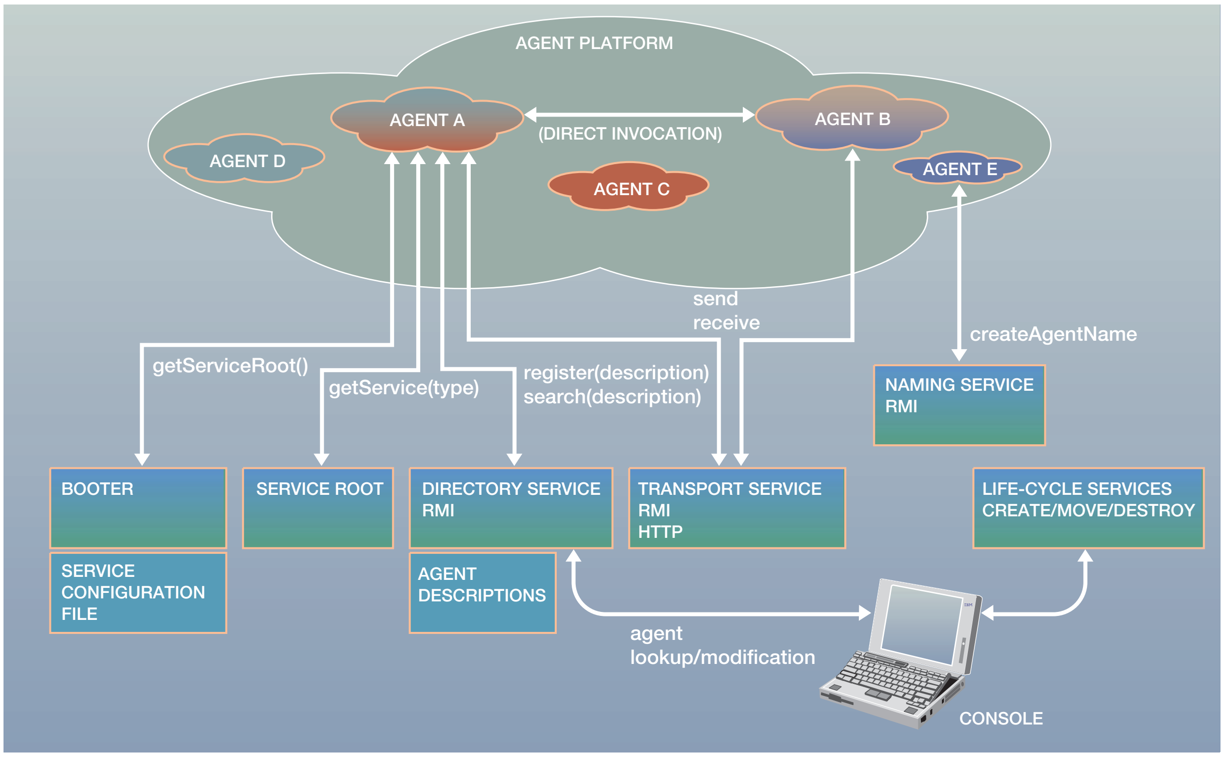
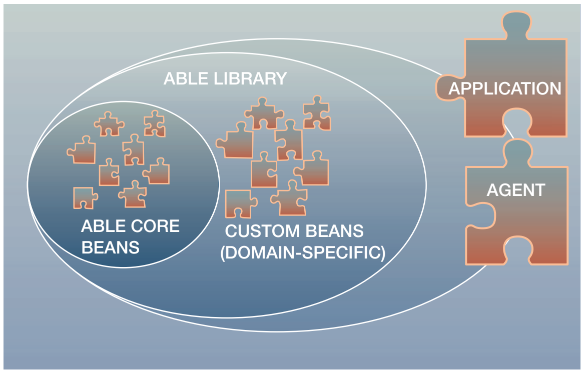


Figure 11 ABLE application design example

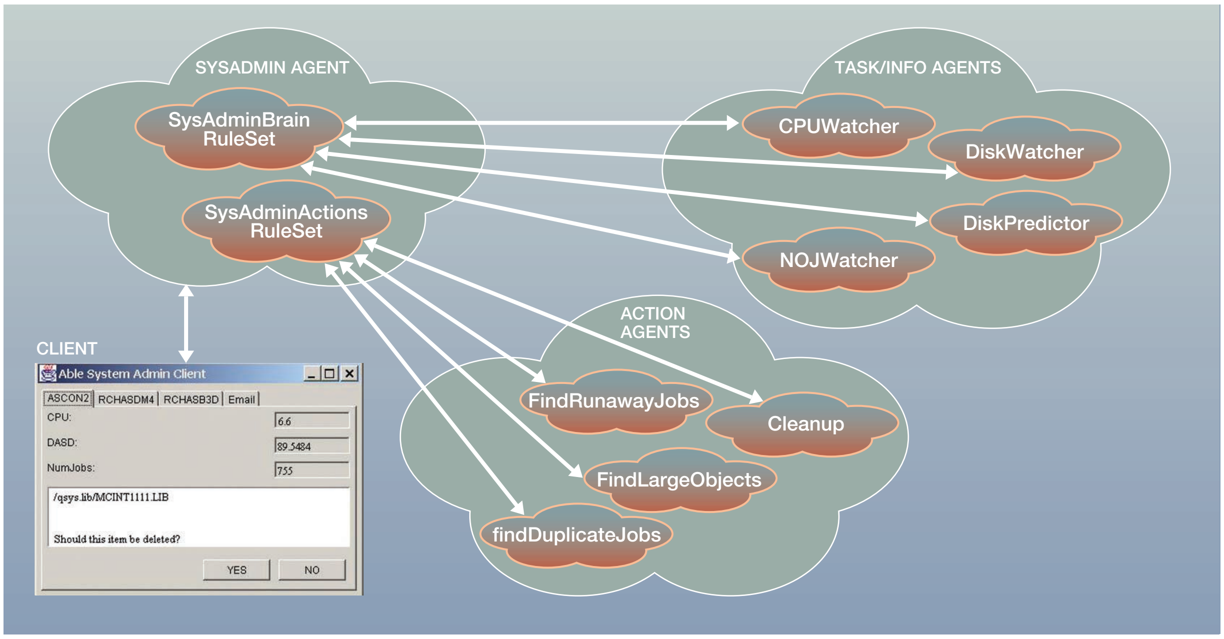


optimization or data processing techniques can be developed and mixed with the core AbleBeans. Autonomous agents composed of multiple AbleBeans could be used to provide function to the application. As illustrated in Figure 11, AbleAgents can bridge the gap between the ABLE component world and the application world by extending the AbleDefaultAgent and implementing the application-specific interfaces, or by extending a base application class and implementing the AbleBean interfaces. This approach allows all of the power of the AbleBean component library to be used to add new function to the application environment and is highly recommended.

In the following subsections, we present two application case studies that illustrate how ABLE components can be used to quickly develop solutions. The case studies are used as concrete examples of the various ways in which the ABLE framework, component library, development tools, and agent platform combine to enable development of multiagent autonomic applications.

Case Study 1: System administration using ABLE. A basic system administration multiagent system was developed for the IBM eServer iSeries* system using ABLE (see Figure 12). The goal was to provide an overall view of system health using multiple agents

Figure 12 System administration using ABLE



CPU: 6.6
DASD: 89.5484
NumJobs: 755

/qsys.lib/MCINT1111.LIB

Should this item be deleted?

to monitor CPU utilization, the workload as indicated by the number of jobs running on the server, current disk utilization, and the expected disk utilization. The monitor agents are autonomous and use the built-in ABLE timer event processing to monitor the associated system resources at selected time intervals. When any one of the monitor agents detects a significant situation, it sends an event that is processed by the SysAdmin agent. When the SysAdmin agent receives the event, it is processed by an internal AbleRuleSet agent (called SysAdminBrain) that invokes other agents to gather additional information.

The SysAdminBrain AbleRuleSet can invoke one of the task agents to perform operations such as finding duplicate jobs, finding runaway jobs, finding large objects or files, and cleanup. The FindLargeObjects task agent uses the ABLE DBImport bean to perform a query to find the largest objects or files on the system. The Task agents are simply information gatherers. They do not take direct actions. The actions are performed by the SysAdminActions agent that contains another AbleRuleSet agent that either prompts the user to approve an action or automatically takes a remedial action such as killing a runaway job or deleting a system object.

A rudimentary SystemAdmin client GUI, shown in Figure 12, was developed using Java Swing. It communicates with the SysAdmin agent using the RMI connectivity that is built into the ABLE agent framework. The SysAdmin agent also sends a report of any actions it has taken to the client for display to the user. The client can also send these findings to a list of e-mail addresses so users can keep track of system management agent actions.

The SysAdmin agents were developed over a period of two weeks. The developers were new to the Java language and to the ABLE toolkit. They made use of the ABLE Agent Editor and AbleRuleSet editors to develop and test the agents and associated rulesets defining their behavior. They used the distributed RMI capability to build an application that runs on a single client system and can monitor multiple server systems. This case study demonstrates the productivity gains made possible through use of a standardized agent toolkit as well as the ingenuity of the developers.

Case Study 2: A diagnostic application. Another use of ABLE technology in IBM products is in the area of server diagnostics. The iSeries electronic support team is constructing a set of ABLE agents to perform data collection, problem determination, and problem source identification tasks. The scenario is that a customer call comes into the support center. The customer support representative asks a series of questions describing the situation, symptoms, and other relevant information. One or more ABLE agents are then dispatched to the customer iSeries machine to help diagnose the problem.

Several steps are required to automate the diagnosis of machine problems. These steps include data collection, data formatting and preprocessing, data analysis and problem determination, and finally, advice or automation of the problem resolution. The ABLE toolkit provides beans and tooling to aid in all of these tasks. The ABLE Rule Language enables agents to call external programs to collect data. The AbleImports allow an agent to collect data and pass the data to another for analysis, using an external database or text file as the storage medium. An AbleDataTable bean provides a view over the external data in column major order. This view allows individual metrics to be analyzed as a time series and for groups of metrics to be correlated and analyzed in a time-series fashion.

Another example of a diagnostic application is an automotive diagnostic prototype developed in partnership with IBM Global Services. In this application, the ABLE Rule Language was used to perform time-series analysis over a 40-second time series, looking at multiple engine sensors to identify misfire conditions. In the production application, we foresee development of a comprehensive set of diagnostic agents, each capable of detecting the presence or absence of a particular fault or closely related set of faults. A diagnostic manager agent will coordinate the activities of the individual diagnostic agents, deriving the higher-order diagnosis (for example, three faults identified by individual agents are related and point to a single point of failure) and providing a diagnostic tree to find the root cause of the failure.

There are several advantages to the multiagent approach. Diagnostic agents can be developed incrementally to resolve the most common (or most difficult) problems first and, over time, can be combined to cover more and more of the problem space. The ABLE Rule Language can be used to define the diagnostic reasoning and data analysis tasks, providing additional flexibility. Agents can be developed by the support team and deployed at any point in the release cycle, as opposed to being tied into the system release schedules. For example, if an unexpected problem was found after the release of a new system, diagnostic agents could be made available to assist customers and support representatives at any time. Flexibility is one of the major advantages of an agent-based solution.

Autotune agent

One of the basic operations required by autonomic systems is closed-loop control, where the state of a target system is monitored, compared to some desired goal state, and then adjusted as required to move toward the goal state.[14] As computer operating systems, middleware, and applications have become more complex; literally hundreds or thousands of parameters must be configured in order to keep everything running smoothly. A practical autonomic system would be composed of many controller agents, distributed across many computer systems, and operating at various levels in a hierarchy. Individual controller agents would receive high-level goals from above and, in turn, control resources and applications at lower levels in the hierarchy.

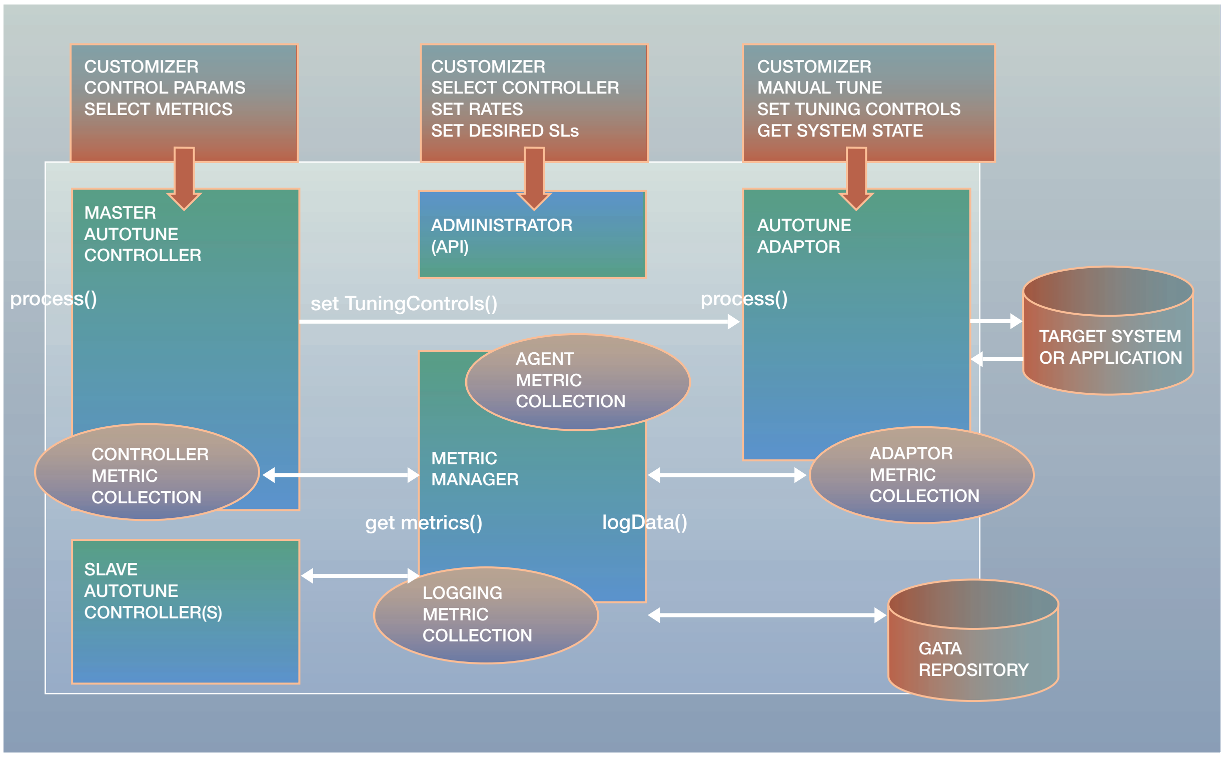
A generic AutotuneAgent has been developed that addresses many of these requirements. This agent extends the AbleDefaultAgent class but completely overrides the agent behavior. The Autotune agent contains one or more AutotuneController beans that provide control strategies and one or more AutotuneAdaptor beans to interface with target systems or applications.

A set of AutotuneMetric classes is defined to represent the state of the target system. Configuration-, Workload-, and Service-level indicators are read-only metrics that provide state information to the AutotuneAgent. TuningControl metrics can be dynamically set by the agent. The AutotuneAdaptor defines the set of metrics supported by a target system. These metrics are managed as a collection by the agent and can be logged to a historical data repository.

Figure 13 shows the architecture of an Autotune Agent. Each Controller bean provides its own Customizer GUI to allow the user to configure its parameters. Each Adaptor bean provides a data panel that allows the user to see and set target system metric values. The AutotuneAgent Customizers allow the user to select which Controller bean is the master (if there is more than one) and to set polling rate and other parameters.

The AutotuneAgent supports both distributed and hierarchical control: distributed by virtue of the agent

Figure 13 Autotune Agent architecture



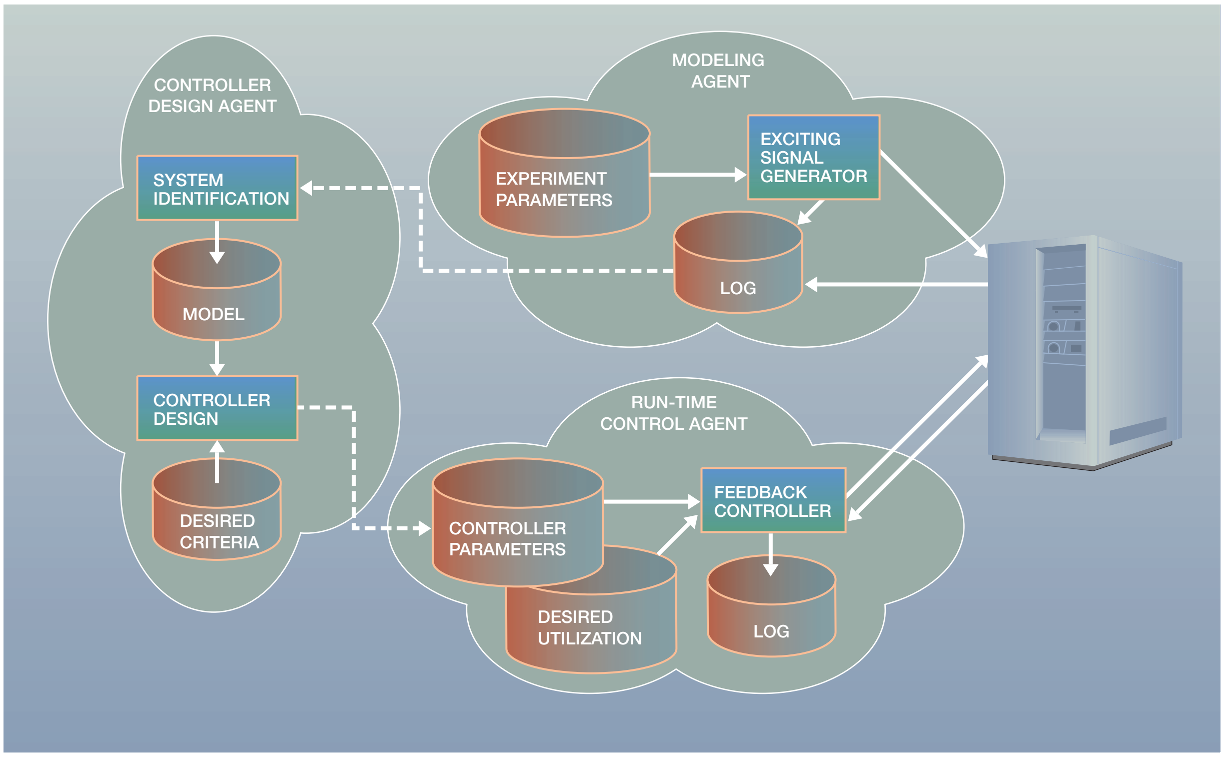
being an AbleBean, hierarchical when the target system is another Autotune agent. Autotune agents have been applied to tuning Lotus Notes* servers, Apache Web servers, and DB2* (DATABASE 2*) utilities.

Case Study 3: An Autotune agent for Apache Web servers

In this application, a multiagent feedback control system based on ABLE Autotune agents was developed for automatically tuning the Apache Web server parameters. Typically, the Apache tuning work is done by the system administrator. The objective is to maintain the system CPU and memory utilization at a desired level so as to avoid overload or to reserve certain resources for other applications. This objective requires significant effort because the relationships between the desired CPU and memory utilization levels and the available tuning parameters (namely, MaxClients and KeepAlive timeout) are not clear.[15] Moreover, this tuning work must be done frequently since these relationships are affected by the workload, and the workload can vary over time.

To automate the Apache server tuning process, three Autotune agents were designed and built for the three phases in automatic feedback controller design and deployment (as shown in Figure 14). In order to understand the dynamic behavior of the server, a modeling agent is first applied to generate time-varying signals for the tuning parameters MaxClients and KeepAlive. Sine waves are used with magnitudes and frequencies specified through the Customizer GUI. The server behaviors (CPU and memory utilizations) under these exciting signals are recorded and passed to the controller design agent through text files and ABLE Imports. The controller design agent uses system identification techniques to extract a first-order linear model from the collected data. Based on this model, a linear quadratic regulation (LQR) controller is designed in order to meet certain design criteria specified by the user through the Customizer GUI, such as minimizing the difference between the desired and measured utilizations and minimizing the changes in the tuning parameters. The output of the controller design agent is a set of controller parameters that are passed to the run-time control agent. The desired utiliza-

Figure 14 Structure of the multiagent system for the Apache Web server



tion level is also specified as the control goal from the system administrator through the Customizer GUI. On the basis of this information, the feedback controller interacts with the Apache Web server to dynamically adjust the MaxClients and KeepAlive tuning parameters to meet the desired CPU and memory utilization levels.

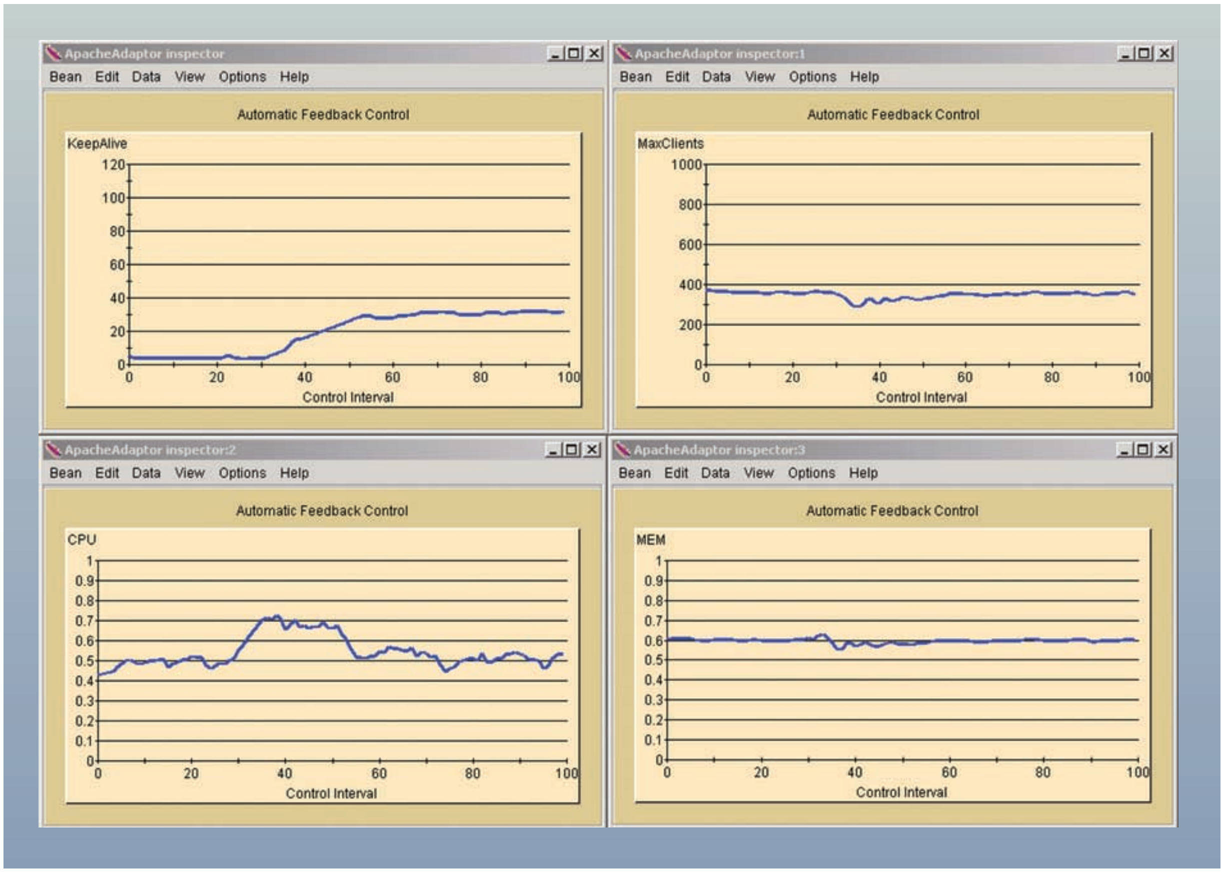
Figure 15 shows an example control run. The control interval is five seconds. Around the twentieth control interval the workload increases as a group of heavy users start to access dynamic Web pages (in contrast to normal users visiting static Web pages). This workload consumes more system resources, causes large increases in CPU utilization, and slightly increases memory utilization. In order to maintain the desired utilization levels (e.g., 0.5 for CPU and 0.6 for memory), the tuning parameters MaxClients and KeepAlive are automatically adjusted by the feedback controller. In particular, a larger KeepAlive value is used to decrease the CPU level, and the MaxClients value is adjusted temporarily according to the dynamics of the server.

Subsumption agent

One of the basic tenets of artificial intelligence over the years has been the symbol system hypothesis, posited by Simon in 1969. His assertion is that people are intelligent because we process symbols and that only symbol-processing capabilities are required to produce intelligent machines. But this hypothesis begs the question of how those symbols become grounded to sensory inputs and perceptions from the real world.

In response to this problem, Rodney Brooks, who was working on robots at the Massachusetts Institute of Technology, proposed an architecture that relies on representations that are grounded in the physical world. Brooks says, "The key observation is that the world is its own best model." His subsumption architecture[16] was used to build a series of mechanical robots. This architecture uses the notion of layers of behaviors, each built upon the lower-level competencies and each responding directly to sensory inputs via effectors on the world.

Figure 15 Performance of the Autotune controller for the Apache Web server under dynamic workloads

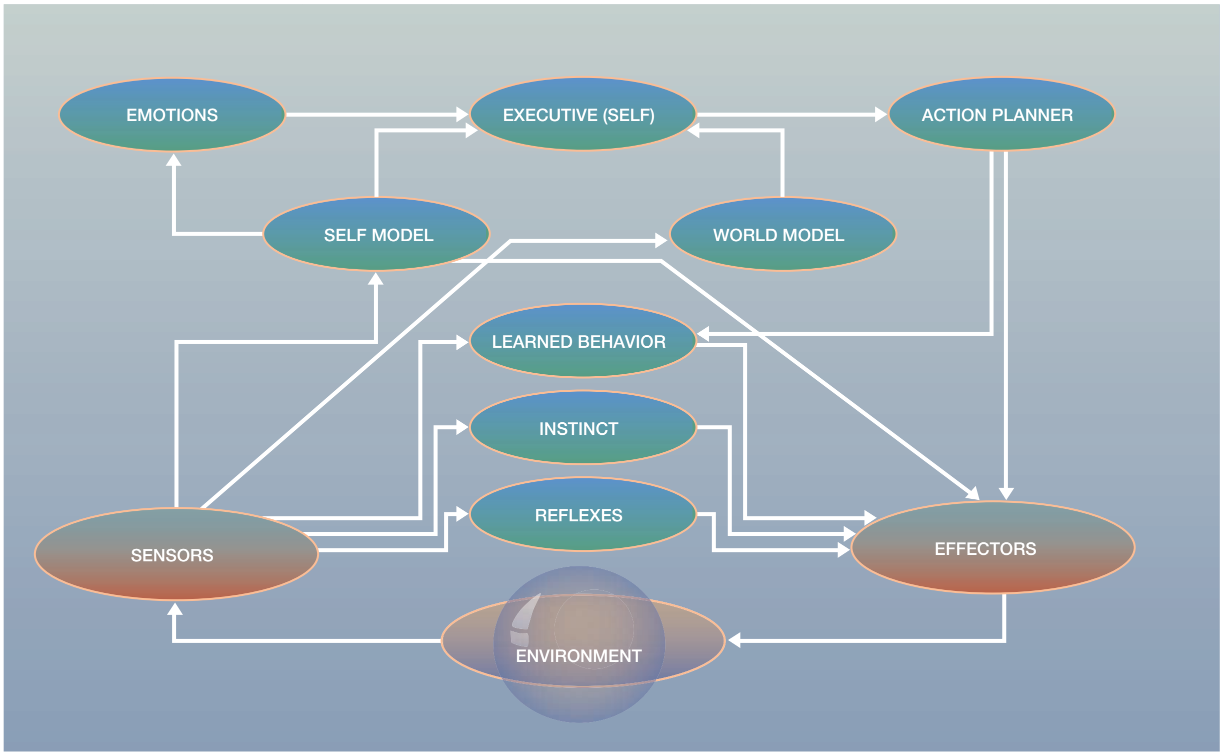


The AbleSubsumptionAgent extends the AbleDefaultAgent class and introduces the properties and behavior necessary to define new capabilities in such a system. There are three types of AbleSubsumptionAgent: reflexive, reactive, and adaptive. Reflexive agents are simple and fast. They respond quickly to changes in sensory input and usually use simple rules to define their behavior. Reactive agents are more complex. They take more time to evaluate and respond to changes in inputs, and often use data-driven forward inferencing or goal-directed backward inferencing. Adaptive agents learn from experience and modify their behavior based on past actions and subsequent feedback. All of these Subsumption agents contain the notion of levels or fixed priority as defined in the subsumption architecture. Although subsumption had a strict hierarchy, multiple AbleSubsumptionAgents can reside at the same priority level and represent alternative cooperative or competitive approaches in responding to inputs.

Autonomic agent

In this section, we outline an architecture and methodology for building an autonomic agent capable of playing a role in a future autonomic computing infrastructure. The AbleAutonomicAgent extends the AbleDefaultAgent and contains multiple autonomous or semiautonomous AbleAgents from the ABLE component library. The internal agents cooperate and compete to take control of the intelligent system and make the appropriate response, much like Minsky's *The Society of Mind*. The base agent architecture is shown in Figure 16. The architecture contains a central pool of subsumption agents similar to that of Brooks[16] with sensors and effectors providing inputs from and outputs to the external world. Three defined behavior layers implement the basic reflexive behaviors, the complex instinctive reactive behaviors, and the more complex behaviors learned from interactions with the world. These three

Figure 16 An autonomic agent architecture



levels are implemented by AbleAgents that are containers for multiple AbleSubsumption behavioral agents that implement domain- or situation-specific behaviors.

Our intelligent autonomic agent must build and maintain a model of the external environment and of its own components. A top-level executive component makes decisions based on the models and its current emotional state. A planner component is used to create multiple step scripts or sequences of actions necessary to achieve the high-level goals being pursued by the executive. Like the behavior levels, the models, emotions, executive, and planner components are all implemented as AbleAgents that in turn may be composed of other AbleAgents and AbleBeans. The high-level architecture defines the data and event flows between the components in the autonomic system. By constructing the base agent using ABLE self-similar components, intelligent autonomic systems of widely varying complexity could be built using this architecture. The system retains all of the advantages of the reactive behavior-based subsumption architecture while adding internal mental states, including models of the self and world, emotions, learned behaviors, planning, and meta-level decision-making.

Our thinking has been influenced by prior work in this area, most notably Sloman and Minsky. Riecken has implemented the M system that corresponds to Minsky's architecture with multiple reasoning agents, blackboards for communications, rule-based inferencing, and semantic networks for representing domain objects.[17] Butler et al.[18] have implemented an object-oriented version of Brooks' subsumption architecture.[19] Sloman was one of the first to discuss computer-generated emotions, and his three-layered model with reactive, deliberative, and reflective processing levels is somewhat similar to this architecture.[3] Caulfield and Johnson sketch an architecture for a "conscious" system whose components correspond to the Autonomic agent architecture.[4] Jonker and Treur[20] present a multiagent architecture intended to simulate animal behavior. Picard[21] gives a good overview of the computational issues related to machine generation and recognition of emotional states.

The novelty in our approach is the use of an agent structure with well-defined functional components, where those components themselves are multiagent systems. The ABLE framework and agent platform make constructing systems like this feasible. The existing ABLE component library provides us with a rich set of machine learning and reasoning capabilities on which to base our implementation. Although we have just started down this road, our experience with building other higher-level agents, such as the Autotune agent, gives us confidence. Our firm belief is that any truly autonomic system will require one or more agents of this type as part of the architecture.

Concluding remarks

Our objective in this paper was twofold: to describe the set of functionality provided in the ABLE toolkit and to demonstrate its utility via real application case studies. Although we selected three examples, they are just a few cases where ABLE has been used. We have applied the ABLE agents to multiple problems in systems management, including event processing, performance monitoring using adaptive thresholds, system health monitoring using hierarchies of fuzzy rules, and time-series prediction for service-level agreement management using neural networks.

ABLE components have been successfully applied to e-commerce, including computing complex discounts in a business-to-business environment using IBM WebSphere* Commerce Suite. The ABLE rule engines have been used in conjunction with the BRBeans component in the WebSphere Application Server Enterprise Extensions. The ABLE framework and component library will be shipped as part of an upcoming iSeries operating system release. Application agents for performing communication traces and data collection are slated for production use by the iSeries eSupport organization. Additional systems management agents are also in development.

We continue to add new algorithm beans to the ABLE component library. The development of the AbleSubsumptionAgent and AbleAutonomicAgent will take place over the next year. We plan to add the IRIS (information, representation, inferencing, sharing) hypergraph knowledge representation to use as an integrated method for encoding and reasoning about domain knowledge.[22]

We have described a series of agents that could play the role of intelligent nodes in an autonomic computing system. One could easily imagine networks of distributed intelligent agents managing storage, operating systems, network resources, database and file systems, middleware, and applications while simultaneously being managed by other agents in the hierarchy. At various points in the network, we may require relatively simple agents, dominated by reflexive behaviors. At higher levels we may require complex reactive behaviors, learning, and adaptation. It is unlikely that we will reach a fully functional autonomic computing system in one giant leap. We will need to incrementally expand the depth and breadth of intelligent behaviors available to the individual agents as well as to the entire distributed autonomic system.

In the future, we plan to leverage our work on the ABLE toolkit to further explore the world of autonomic computing. This grand challenge will likely attract a large number of researchers using a wide variety of technical approaches. We intend to attack the autonomic computing problem from an agent-based perspective. We plan to build on this work by adding higher levels of abstraction and sophistication to our agents and our agent platform as we pursue the goal of building truly autonomic computing systems.

Acknowledgments

The authors would like to acknowledge and thank the iSeries and eServer teams, including Kristi Schultz, Mike Smith, Bob Hansen, Jack Corcoran, Earl Emerick, Carol Egan, Kenton Lynne, Jason Illg, Mark Masbruch, and Tony Dunbar. Special thanks to Jake Kugel and Jeff Awe for their work on the System Administration agents.

We would like to thank the Tivoli and WebSphere teams, including Mark Johnson, Carole Corley, Kristof Kloeckner, Mark Linehan, and Rob Will.

We thank our collaborators at the IBM Thomas J. Watson Research Center, including Joseph Hellerstein, Sujay Parekh, T. S. Jayram, Mark Squillante, Irina Rish, Ricardo Vilalta, Michelle Zhou, Rosario Uceda-Sosa, and Isabelle Rouvellou.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc., IntelliOne Technologies, or Telecom Italia Lab.

Cited references and notes

1. M. L. Minsky, *The Society of Mind*, Simon & Schuster, New York (1985).
2. R. A. Brooks, "Intelligence Without Representation," *Artificial Intelligence Journal* **47**, 139–159 (1991).
3. A. Sloman, "Damasio, Descartes, Alarms, and Meta-Management," *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, San Diego, CA (1998), pp. 2652–2657.
4. J. H. Caulfield and J. L. Johnson, "Softest Computer," *Proceedings of the Society for Optical Engineering*, Bellingham, WA (2000).
5. *Autonomic Computing: IBM's Perspective on the State of Information Technology*, IBM Corporation (2001), available at http://www.research.ibm.com/autonomic.
6. A. B. Damasio, *Descarte's Error: Emotion, Reason, and the Human Brain*, Putnam, New York (1994).
7. V. S. Ramachandran, *Phantoms in the Brain*, Morrow, New York (1998).
8. J. P. Bigus and J. Bigus, *Constructing Intelligent Agents Using Java*, Second Edition, John Wiley & Sons, New York (2001).
9. The Foundation for Intelligent Physical Agents, http://www.fipa.org.
10. J. P. Bigus, "The Agent Building and Learning Environment," *Proceedings of Autonomous Agents 2000*, Barcelona, ACM Press (2000), pp. 108–109.
11. J. P. Bigus and K. Goolsbey, "Integrating Neural Networks and Knowledge Based Systems in a Commercial Environment," *Proceedings of the International Joint Conference on Neural Networks*, Vol. 2, Washington, DC, IEEE Press (1990), pp. 463–466.
12. Agent Building and Learning Environment, http://www.alphaWorks.ibm.com/tech/able.
13. J. P. Bigus, *Data Mining with Neural Networks*, McGraw-Hill, Inc., New York (1996).
14. S. Parekh, N. Gandhi, J. Hellerstein, D. Tillbury, T. Jayram, and J. Bigus, "Using Control Theory to Achieve Service Level Objectives in Performance Management," *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA (May 2001).
15. Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. Tilbury, "Using MIMO Feedback Control to Enforce Policies for Interrelated Metrics with Application to the Apache Web Server," *Proceedings of Network Operations and Management Symposium*, Florence, Italy (April 2002).
16. R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation* **RA-2**, 14–23 (April 1986).
17. D. Riecken, "M: An Architecture of Integrated Agents," *Communications of the ACM* **37**, No. 7, 107–116 (July 1994).
18. G. Butler, A. Gantchev, and P. Grogono, "Object-Oriented Design of the Subsumption Architecture," *Software—Practice and Experience* **31**, No. 9, 911–923 (July 2001).
19. R. A. Brooks, "Intelligence Without Reason," *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Sydney, Australia (1991), pp. 569–595.
20. C. M. Jonker and J. Treur, "Agent-Based Simulation of Animal Behavior," *Applied Intelligence* **15**, No. 2, 83–115 (September/October 2001).
21. R. W. Picard, *Affective Computing*, MIT Press, Cambridge, MA (1997).
22. R. Uceda-Sosa, "Proactive Information in Distributed Knowledge Environments," submitted for publication in 2002.

Accepted for publication April 1, 2002.

Joseph P. Bigus *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York, 10598 (electronic mail: bigus@us.ibm.com).* Dr. Bigus is a Senior Technical Staff Member at the Thomas J. Watson Research Center, where he is the project leader on the ABLE research project. He is a member of the IBM Academy of Technology and is an IBM Master Inventor, with over 20 U.S. patents. He was an architect of the IBM Neural Network Utility and Intelligent Miner™ for Data products. Dr. Bigus received his M.S. and Ph.D. degrees in computer science from Lehigh University and a B.S. in computer science from Villanova University. His current research interests include learning algorithms and intelligent agents, as well as multiagent teams and their applications to adaptive system modeling and control, data mining, and decision support.

Don A. Schlosnagle *IBM Rochester Custom Technology Center, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: daschlos@us.ibm.com).* Mr. Schlosnagle is an advisory software engineer in the Rochester Custom Technology Center. He studied computational linguistics and expert systems at the former IBM Systems Research Institute and wrote, in Common Lisp, a PC-based natural language DB2 query program that was successfully marketed by IBM as a PRPQ. Mr. Schlosnagle has since worked on the Neural Network Utility and on Intelligent Miner for Data, where he contributed the fuzzy logic inference system for evaluating proposed neural network architectures. A greatly enhanced version of the fuzzy system found its way into ABLE. His current interest is in combining neural networks with fuzzy logic.

Jeff R. Pilgrim *IBM Rochester Custom Technology Center, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: pilgrim@us.ibm.com).* Mr. Pilgrim is an advisory software engineer contributing to the Agent Building and Learning Environment. His prior development work includes Intelligent Miner for Data, Neural Network Utility, wide area wireless computing, iSeries Management Central, and configurators for rack-mounted systems such as the 9221. Mr. Pilgrim joined IBM in 1979 at the former IBM facility in Owego, New York, where he was responsible for forecasting workload for defense contracts. Previously an APL zealot, he is now a Java bigot. He received his M.S. in industrial engineering and operations research in 1980 from the Pennsylvania State University.

W. Nathaniel Mills III *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: wnm3@us.ibm.com).* Mr. Mills joined IBM Research in 1996 after running a successful consulting and software development business for 10 years. His initial work in systems management led to the development of WebSphere Studio AE Page Detailer—a tool to measure and display Web page download performance. He is now a Senior Technical Staff Member working with ABLE on various projects involving automated problem diagnosis and systems management. These projects focus on "after sales" product support or are related to the product families of WebSphere and Tivoli. In particular, Mr. Mills helps build ABLE-based applications for use as embedded "rules engines" to help analyze and mine data, to make real-time assessments of various application states to drive problem diagnosis, and to recommend corrective action. Mr. Mills graduated from Trinity College in 1979.

Yixin Diao *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: diao@us.ibm.com).* Dr. Diao is a postdoctoral Fellow at the Thomas J. Watson Research Center. He received his Ph.D. degree in electrical engineering from Ohio State University in 2000. He has conducted research on fault tolerant control system design using an adaptive fuzzy/neural approach, intelligent reasoning for robust fault diagnosis, and nonlinear dynamic system modeling with hierarchical learning structure and multivariate statistical methods. He joined IBM Research in 2001. His work involves increasing the adaptive and learning abilities of computing systems to unknown and time-varying environments by exploiting techniques from control theory, machine learning, and distributed agents. In particular, he is working with ABLE to develop generic adaptive agents for automated server tuning. Dr. Diao has authored around 20 technical papers, and his research interests include computer performance management, intelligent systems and control, adaptive systems, and stability analysis.