Supporting Communication of Heterogeneous Distributed Agents with Agenda Management Systems

Eric K. McCall, Lori A. Clarke, Leon J. Osterweil

University of Massachusetts Amherst,
Amherst, MA 01003

As software engineering efforts move to more complex, distributed environments, coordinating the activities of people and tools becomes increasingly important. While groupware systems address user level communication needs and distributed computing technologies address tool level communication needs, few attempts have been made to synthesize the common needs of both.

This paper presents a framework for generating agenda management systems (AMSs) from specifications of application requirements and describes how such systems address communication needs of heterogeneous agents. The framework supports a variety of application requirements and produces a customized AMS that is based on replicated distributed objects. The framework and generated AMS support evolution in several ways, allowing existing systems to be extended as requirements change. Also discussed are experiences using this approach and lessons learned, primarily from use in supporting a process execution environment and a laboratory coordination environment.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering*

General Terms: Design

Additional Key Words and Phrases: agenda management, agent communication

1. INTRODUCTION

In the last half-century, one of the most important trends to emerge from the study of computer science is reliance on increasingly higher-level abstractions. In the earliest days, machines were programmed with plugboards and toggle switches, but these soon gave way to machine instructions and assembly languages, which surrendered in turn to COBOL, FORTRAN, and progressively to higher-level languages such as Ada95, C++, and Java. Simultaneously, low-level machine functions – the underlying implementation details – were hidden behind standardized high-level operating system interfaces. These abstractions have increased software portability and decreased development and maintenance costs.

Recently there has been rising interest in building applications as collections of connected software components [Boehm and Scherlis 1992] that provide specific services through well-defined interfaces. This surge in interest is due in part to the rise in object-oriented programming languages and the success of reusable component libraries. Additionally, the growth of the Internet and a concomitant emphasis on network computing have encouraged the notion that the components in an application do not have to reside on one physical machine or in one address

This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation, the Defense Advanced Research Projects Agency, the Air Force Research Laboratory/IFTD, or the U.S. Government.
Name: Lori A. Clarke, Leon J. Osterweil
Address: Department of Computer Science, University of Massachusetts, Amherst, MA 01003
Name: Eric K. McCall
Address: Hewlett-Packard Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94304

space, but may exist throughout a network of connected computers. Extending the notion of an application as a set of composite services to include not just services provided by computers, but services provided by computers and people, seems a natural step and points the way to where the trend of higher-level programming language abstraction is leading.

Thus, modern computing systems are increasingly being viewed as collaborations among groups of humans and software systems that must communicate about and share tasks. One common direction of previous work has focused on how to support interoperability among software systems. Such approaches have focused on low-level interprocess communication protocols and mechanisms. While providing a useful substrate, these mechanisms are at too low a level of abstraction to support clear exposition in application programs. We believe that higher level abstractions are needed. A higher level view of communication is commonly sought by the computer supported cooperative work and related communities, but much of this work is aimed exclusively at communication for humans and does not provide the range of capabilities needed to support the next generation of computing systems.

To better support the development of modern computing systems, we seek to establish abstract concepts that address the communication needs of both humans and computing systems, collectively referred to as *agents*. To do this we have formalized the metaphor of agendas, or to-do lists. The use of agendas to solve agent communication problems seems to be nearly ubiquitous, having been applied to problems in such varied contexts as software engineering, the factory shop floor, and office automation. Thus specialized instances of this metaphor have already been used to coordinate people with each other and software systems with each other. We believe that this metaphor, appropriately applied, can effectively coordinate humans and software systems for a wide range of applications. This paper explores the problem of building agenda management systems (AMSs) that are capable of doing exactly that.

There are two primary objectives that must be addressed: *creation* of new AMSs specifically designed to address a specific agent communication problem, and *evolution* of previously created (and perhaps previously evolved) AMSs to address changes in the problem. Creation refers to the production of a new AMS to meet the requirements of an application. Evolution refers to the process of changing an existing AMS to address changes in the specification for that system.

Our approach addresses these objectives by providing a framework that supports three activities: *generation*, *regeneration*, and *adaptation* of agenda management systems. Throughout this paper, the approach is referred to by the acronym *GRAAMS*. To evaluate this approach, we have developed a prototype agenda management framework called *Grapevine*. By evaluating Grapevine's strengths and weaknesses in meeting applications' agent communication needs, we draw conclusions about the GRAAMS approach in general.

Grapevine has two major parts that allow it to be used cost effectively to address the agent communication requirements of a variety of applications. The first part is the distributed object substrate and root object classes. These components form the foundation for application-specific agenda management systems that are produced by the second part, the AMS builder. The AMS builder uses a specification written in an input language to produce an agenda management system that meets the application requirements. We refer to an AMS that has been created and/or evolved by Grapevine as *instantiated*; unless otherwise indicated, all references to an "AMS" refer to an instantiated agenda management system.

Grapevine has been used to produce agenda management systems for three applications, each of which has a different mix of agent communication needs. The first application-specific agenda management system supports communication between a mixture of humans and software systems in the Little-JIL process language interpretation environment. The second AMS supports communication among humans and allows them to collaborate to schedule meetings and maintain task lists. The third AMS is used to support tool communication in an image processing application. These three agenda management systems were evaluated with respect to meeting the communication requirements of the applications for which they were created. We found that, in general, the agenda management approach was effective at meeting the agent communication needs of the application and provided significant support in building each application.

Determining that it is feasible to use this approach to create and evolve software systems that support interoperability of human and tool agents is the focus and contribution of this research. In this paper, we present the design of a flexible, iterative generation tool that is used to generate and regenerate AMSs, the design of a software

architecture for adaptable instantiated systems, and an evaluation of the tool and architecture that comprise this research. Our observations have shown that the GRAAMS approach can provide software system designers with an important piece of the communication and coordination infrastructure needed in large systems of interacting human and software agents.

The next section highlights recent research in a variety of related areas. Section 3 describes the design of the major components of Grapevine, and the following section describes the design of the experimentation performed and justifies the choice of case studies that were conducted. The case studies are described in section 5, which concludes with a description of experimental results. Section 6 offers some conclusions based on this experience.

2. RELATED WORK

Our interest in "agendas" and "agenda management" is hardly unique. Systems supporting agent communication through agendas have appeared before, in a variety of domains, but the problems of agenda management have typically been confronted and solved differently for each specific system. In essence each system seems to have its own, somewhat idiosyncratic, notion of agenda management and an implementation to match. In contrast, the main focus of our study is on defining and solving the problem of agenda management itself, in a way that is applicable in a variety of domains.

Research related to agenda management can be found primarily in the areas of process execution environments, workflow systems, tool interoperability environments, and asynchronous computer supported cooperative work. Considerable overlap between some of these areas exists [Georgakopoulos et al. 1995], and the theme of facilitating agent communication through agendas can be discerned in each. The remainder of this section compares the approaches to agenda management taken in each of these areas. For each area, we identify some assumptions made and point out some idiosyncrasies of the various approaches to agent communication. By abstracting the problem of agenda management and providing a solution that is usable in a variety of areas, we hope to allow researchers and practitioners to refocus on issues of when and why agents should communicate, rather than focusing on low-level details of how to implement such communication.

2.1 Interoperability

Agendas are easily conceptualized as a tool for allowing humans to manage their work. We have proposed extending this common metaphor to allow its use in coordinating the activities of both people and software tools. Thinking of a tool as having an agenda may seem counterintuitive at first, but upon reflection it should be clear that "agenda" is an appropriate metaphor here too. In systems with asynchronous communication, it is common for messages to be queued; this queue may be thought of as an agenda for the recipient.

Existing approaches to tool interoperability can be classified into several broad categories: event bus mechanisms, remote procedure call (RPC) mechanisms, queueing mechanisms, and blackboard systems.

Event bus mechanisms include systems such as **Desert** [Reiss 1996; Reiss 1999], **C2** [Taylor et al. 1996], Lotus and Sun's **InfoBus** [JavaSoft], Tibco's **TIB/Rendezvous** [TIBCO Software], **JEDI** [Cugola et al. 1998], and many others. Systems such as these allow tool components to register and unregister interest in receiving user-defined event types generated by other components. Events are usually immutable and describe an action that has already occurred. In some systems, events may be queued within the bus, but the queues are usually simple FIFO queues that lack complex operations such as sorting. In contrast, items on an agenda are often expected to change state and are often used over a period of time to track the execution of a task.

Commercial distributed object computing standards such as **CORBA** [Object Management Group 1996], Microsoft's **COM+** [Kirtland 1997], **OpenDOC** [Apple Computer 1994], Java's **RMI** [Sun 1997] and others extend the remote procedure call mechanism to object-oriented systems. Typically, objects are stored at a central server. Client-side stub objects then make remote procedure calls to the server object's (public) methods. Distributed object computing (DOC) technologies also typically include some sort of name server or object registry that allows software components to locate server objects. A major focus of remote procedure call mechanisms has been on standardizing data representations for shared objects; typically each DOC or RPC system has an associated language, such as IDL [Stone and Nestor 1987] or QDR [Maybee et al. 1996], for specifying the structure of passed data to allow the RPC mechanism to properly serialize parameters and return values.

In an RPC call, we can view the client and the server as communicating agents in an application; the notion of an agenda is a simple FIFO queue of procedure invocations. No complex functions exist that would allow, for example, the server to reorder the queue or the clients to dynamically monitor the state of pending invocations. RPC and DOC systems by themselves lack the functionality we seek for agenda management.

Queueing mechanisms have been developed commercially for integrating legacy systems with new applications. The best known, IBM's **MQSeries** [International Business Machines Corporation 1995], supports using queues for one-way or two-way communication as well as for one-to-one, one-to-many, and many-to-many relationships between queues and agents. Queues can be viewed as a specialized kind of agenda; they provide an indirect means of communication. While MQSeries may be used to interconnect highly heterogeneous applications and computing platforms, it does not provide adequate support for modeling the data that are queued.

In typical blackboard systems[Nii 1989; Corkill 1991], a variety of knowledge sources (KSes) solve a problem by using a shared data repository (the blackboard) to represent and store knowledge. When a problem statement is posted to the blackboard, one or more KSes are selected by the blackboard control component to become active. The activated KSes perform computations on the blackboard data and may post results that may be later consumed by other KSes, similar to rule-based computation. Blackboard systems are designed so that agents of certain kinds are directed to collaboratively solve a problem. Agenda management systems, on the other hand, are passive; there is no control component that directs the activity of agents using the system. While there are conceptual similarities between a blackboard and an agenda, agenda management systems represent a more general concept; most blackboard systems can be viewed as domain-specific, tool-centric, static agenda management systems.

These interoperability approaches are designed for integrating tools, not people, and provide little automated support for defining complex data types or for defining complex functionality for selecting or ordering among tasks assigned to agents.

2.2 Process Environments

Thinking of an agenda as an invocation support mechanism that helps a person to "invoke" functions of distributed components that provide services in an application makes it easy to see that this metaphor goes the other way as well. Agendas can be and have been used as the means for process environments and workflow systems to invoke functions performed by people.

Process environments support the execution of codified software development activities and other types of processes. By virtue of the need to coordinate the activities of autonomous execution agents, several such environments have developed concrete notions of agendas and agenda management (sub)systems that support agent communication. Process environments, however, typically have application-specific notions of agenda management, such as prioritized tasks, built in to them. In process environments, when a cleanly separated agenda management component exists at all, it is generally designed only for a particular process environment and is too tightly coupled to the process language and environment to be easily reused to build unrelated applications. Of particular note in many process support and execution environments (PSEEs) is the strong coupling between details of the process environment and the design of the agent communication mechanism. While an abstract notion of "agenda" used for agent communication is frequently present, agendas are typically used only for communication with people, with tool communication falling under direct control of the process environment. We believe this design sacrifices tool autonomy and introduces unnecessary coupling in the PSEE without tangible benefit.

One such system, **Process Weaver** [Fernström 1993], has an integrated agenda component that is used to coordinate the activities of humans. "Work contexts" may be posted and delegated to human agents' agendas. The work context and agenda of Process Weaver are, however, specific to Process Weaver, and were not designed to solve more general agenda management problems. Moreover, there appears to be no way of using the agenda mechanism in Process Weaver to integrate software tools into the process environment.

Process Weaver includes a process modeling language that has notions of concurrency with semaphores, message passing, distribution, and modularity built in. These language constructs are used both for control over *how* agent activities are coordinated and *when* such coordination should occur (the process). Thus the coordination mechanism is coupled to the specification of agent coordination. While many process environments exhibit this characteristic, we believe that such coupling decreases understandability and reusability of the PSEE and should be avoided.

The importance of decoupling process state representation, which can be seen as data that are managed by an AMS, from process modeling language, was noted in **ProcessWall** [Heimbigner 1992]. ProcessWall is a process state server that is intended to be used to facilitate process execution, however, it is still a process-centric tool and, from our point of view, unnecessarily mixes application-specific features in with its support for agent coordination.

The **Marvel** software development environment [Ben-Shaul et al. 1992] has addressed the issue of assigning tasks to users, though this ability is tightly integrated with the process environment. In Marvel, one language (MSL) is used to specify both the (rules for the) development process and the data model. This specification is loaded into Marvel to instantiate a project-specific environment, similar to the way an application specific agenda management system is instantiated in our approach. The issue of future evolution of instantiated systems is not addressed.

In the **SPADE-1** environment [Bandinelli et al. 1996], support for asynchronous cooperative work is "based on the principle of separation of concerns between process model enactment and user interaction environment." The SPADE-1 environment consists of a number of tools, among them an Agenda tool. Agenda uses a configuration file to define its behavior (an event-based transition model) and the structure of tasks (the name and type of attributes of the agenda items) that will appear on the agenda, thus providing some adaptability. The Agenda tool is invoked by each human user and allows humans to send and receive information, however, a separate tool, SPADEShell, is used to send requests to the process execution environment. Thus, while the notion of a somewhat configurable "agenda" exists, it is used by humans only.

Similarly, in **OPSS**, the ORCHESTRA Process Support System that is implemented on top of the JEDI event bus, *agents* are defined as either human or tool, however only human agents have agendas. Software agents connect directly to the event bus. We believe some of the problems with writing software agents [Cugola et al. 1998] could be solved if the software agents used an agenda abstraction too.

2.3 Workflow

Like process environments, workflow management systems (WFMSs) are designed to support the codified execution of (business) processes. Also like PSEE's, WFMSs generally use the concept of an agenda to coordinate the activities of humans. While the concept of an agenda (as a worklist) is present in many WFMSs, there has typically been no attempt at generalizing the concept and, like PSEEs, agenda management functionality is typically specific to and integrated with each WFMS.

WFMSs are typically implemented atop database systems and frequently have transaction semantics built-in, meaning that instances of workflows are eventually either committed or aborted. Workflow management systems address issues such as data consistency in addition to coordinating the activities of agents involved in a workflow. The model of coordination presented in [Georgakopoulos et al. 1995] characterizes workflow management systems (WFMS) along a continuum, with human-oriented WFMSs such as CSCW systems at one end and system-oriented (tool-oriented) systems such as commercial transaction processing systems at the other. [Georgakopoulos et al. 1995] also points out issues to be considered at each end of this spectrum, but goes on to ascribe different requirements to a WFMS depending on whether the system is human-oriented or system-oriented. For example, it is asserted that because the human-oriented WFMS cannot know the semantics of the information humans are sharing, data consistency is not the responsibility of the WFMS, but application-specific semantic consistency is the responsibility of a system-oriented WFMS. To contrast, our research aims to satisfy requirements such as semantic data consistency arising from a particular application regardless of the balance of agent types. We believe it is better to allow designers to create a communication infrastructure that will support application requirements throughout the spectrum.

The Workflow Management Coalition (WfMC), a coalition of companies interested in facilitating the interoperability of workflow management software, has defined a reference model [Hollingsworth 1994] for workflow management. The model consists of a central workflow enactment service and five APIs that define the interface between the central service and process definition tools, workflow client applications, invoked applications, other workflow enactment services, and administration and monitoring tools.

The WfMC's reference model defines *worklists* to be groups of work items. A work item is a task for a workflow participant, or agent. This definition corresponds roughly to out notion of an agenda, however, it was initially

expected that worklists would be used by humans only, with the tools needed by a workflow being called directly via a separate interface to the workflow engine. Sometime around 1996, the third and second interfaces were amalgamated, corroborating our belief that it is beneficial to treat humans and tools (applications) similarly in such systems. The API of the amalgamated interface is responsible for a variety of jobs unrelated to coordination of the activities of agents. This interface is used for worklist management, management of process execution, and other workflow management activities unrelated to communication. Furthermore, workflow engine semantics (e.g., the notion of work items being "completed" via the "`WMCompleteWorkItem`" function) are built into the interface, and do not allow for application-specific semantics.

The WfMC's reference model clearly has an abstract notion of an agenda and agenda items that are used by agents to coordinate their activities, as well as having standardized APIs that allow both humans and tools to access them. The WfMC reference model, however, also incorporates components (e.g., workflow schedulers, process definition and control) that are not part of agenda management. We believe that there should be another abstract layer between the level of RPC or DBMS calls and the level of workflow data. This layer is the layer at which agenda management should occur so that agent communication is supported at a high level of abstraction. GRAAMS defines abstractions that are sharply focused on supporting coordination in a variety of application contexts, cleanly separating the mechanism used for agent coordination from application-specific details.

One promising commercial WFMS is **InConcert** [Marshak 1997]. InConcert is described as an extensible "object-oriented client-server workflow management system." A process (workflow) in InConcert is defined by tasks, roles, and references that are specialized by a "solution developer" to the particular process and environment in which the process is to be executed. InConcert allows run-time extension of objects via the InConcert API, thus providing a high degree of customization, but, again, no clean separation between the coordination mechanism and the process enactment mechanism is made. For example, the process management interface is described as being used for "initiating new processes, assigning work, monitoring progress, and deleting completed processes." Work assignment, or the metaphor of putting an item on an agent's agenda, seems to be an activity that is independent of the particulars of any process, yet in InConcert it is part of the process management interface. We see such idiosyncrasies as arising from the lack of separation between the agent coordination mechanism and the process enactment mechanism.

METEOR [Krishnakumar and Sheth 1995; Sheth and Kochut 1997] is a research WFMS that has been evaluated in an immunization tracking application [Sheth et al. 1997] and other healthcare-related applications. METEOR is a "multi-paradigm transactional WFMS" that the authors claim addresses some of the scalability and reliability concerns that remain in many existing WFMSs. METEOR supports a distributed and heterogeneous runtime environment through use of CORBA objects (the ORBWork runtime) or web protocols (the WEBwork runtime). Users interact with a running METEOR workflow through an automatically generated web browser GUI.

The ORBWork workflow enactment component of the METEOR runtime described in [Sheth and Kochut 1997] distributes task managers (and their associated tasks) and task scheduling among many CORBA objects. Tasks themselves are written by a task programmer. The task specification includes macros to indicate to the task manager the state of the task as well as code to define the task's interface, to control its execution, and to handle failure conditions [Krishnakumar and Sheth 1995]. METEOR apparently lacks the concept of an agenda (a collection of tasks that an agent is responsible for). This deficit is noted by the authors [Sheth and Kochut 1997]:

> What we believe is that on top of the *lower level* middleware supported by such technologies as Web, CORBA, database access APIs (ODBC, JDBC), etc. we will in the future have a *higher level* middleware, that provides the abstractions needed to seamlessly engage in coordination, collaboration, and information management activities.

We believe the GRAAMS approach to agenda management can provide the higher level middleware that METEOR currently lacks.

Many early WFMS were not very flexible, either in terms of the processes they were able to capture or in terms of the data they represented. One WFMS-like system that does not fit this characterization is **Lotus *Agenda*** [Kaplan et al. 1990], a personal information manager described as a new type of database, an "item/category database." End-users could add items to the database and assign them to categories either manually or automatically through

user-defined selection rules. These categories could then be viewed in a variety of ways. *Agenda* gives enormous flexibility to the end-user, but in so doing loses the benefits of having strongly typed data and a common, centralized data schema. In addition, Agenda is not designed to be used by tools and does not have facilities for controlling how data are used (e.g., to assure semantic consistency of agenda data), nor does it support shared data. It does represent a reification of the agenda concept without an integrated workflow context. Where Agenda lets each end-user define the context of its use, we propose having a system designer define a major architectural component – the agenda management system – that can be used in, for example, a WFMS.

2.4 CSCW

Much research in asynchronous computer-supported cooperative work has focused on building toolkits for creating collaborative applications, but the applications that the toolkits are designed to create are typically for human users only and do not have high-level data abstractions designed specifically for agenda management. Despite this difference in focus, some issues relevant to agenda management, such as evolution of existing systems and data consistency models for cooperative work, have been explored.

Oval [Malone et al. 1992; Lai et al. 1988] is a tool for cooperative work that offers adaptation abilities that are well beyond those described in this paper. Oval is a "radically tailorable system" that allows users to create applications from semi-structured objects, user customizable views, rule-based agents, and links. The objects are formed from templates of named, but not typed, fields and grouped in a type hierarchy. System customization is performed by users in an ad-hoc way; there is no system designer. Folders, which appear similar to agendas, exist for grouping together collections of objects. Furthermore, a user can create rule-based agents that perform automated actions on the user's behalf. But, Oval is not intended to be used as a tool integration mechanism. Also, because of the lack of system-wide data structures, data consistency and sharing issues are not well addressed.

Recent work in the CSCW community [Dourish 1995; Dourish 1996; Greenberg and Marwood 1994] has examined the problem of managing data consistency in human-centric CSCW systems. In many CSCW systems, traditional methods of managing inconsistency is inappropriate. For example, using transaction rollback techniques can result in a confusing experience for human users of shared whiteboard systems. Because agenda management is an asynchronous coordination mechanism, these problems are less acute. Data consistency issues are still very real, and any agenda management system must provide some consistency management mechanisms. Several CSCW systems, such as **Habanero**[Chabert et al. 1998], **Sync**[Munson and Dewan 1997], and **DistView**[Prakash and Shim 1994], tackle the problem of data sharing via object replication. Such an approach to data sharing and consistency appears promising for agenda management, as a number of technical issues, including maintaining syntactic and semantic data consistency and distributed event processing, must be addressed by both agenda management systems and CSCW systems.

2.5 Summary

We propose that a higher-level agent communication abstraction would assist in the development of modern software applications. There are several important characteristics that current approaches to agent communication apparently lack, including

—the separation of the communication mechanism from application specific semantics,

—accommodation of both human and tool agents (heterogeneity), and

—automated support for the creation and evolution of application-specific communication mechanisms.

Our research has addressed the agent communication needs of applications as a separate problem.

3. GRAAMS APPROACH

As noted, the concept of agendas has appeared in several domains, so it seems natural to approach the problem of meeting applications' communication requirements by extending this concept to provide the basis for a high-level framework that will let application developers meet their communication requirements. There are three main hypotheses behind the GRAAMS approach to providing a high-level agent communication framework. The hypotheses are that

(1) agendas are an appropriate metaphor for the asynchronous communication that occurs among heterogeneous agents,

(2) application-specific agenda management systems (AMSs) can help solve agent communication problems in applications, and

(3) a framework-based generation and evolution approach simplifies development and maintenance of agenda management systems.

The first hypothesis has not been independently tested as part of this research, however, the survey of related work in section 2 has shown that "agendas" have already been successfully used in conceptualizing and implementing a wide variety of systems. The second hypothesis is closely related to the first, but implies that agenda management can be a separate component of an application. The second and third hypotheses have been evaluated by building a prototype framework, called Grapevine, and evaluating it and the agenda management systems it generates in three different application contexts.

3.1 Agenda management system requirements

Before describing the GRAAMS approach and Grapevine in detail, we present some major communication requirements that are suggested by related work on large-scale distributed systems of heterogeneous agents (which we will call *agent community* applications). By keeping in mind a set of common requirements, it will be easier to illustrate how the approach in general, and how Grapevine in particular, can be used to meet the specific requirements of such applications.

We expect agent community applications to differ in scale, in balance, and in the specific functional requirements they have. Scale refers to the number of agents communicating in an application as well as the number and size of objects (AMS data) in the application. Balance refers to the ratio of human to software agents. Thus, our approach should be scalable and should support a varying balance of agent types.

A variety of functional requirements commonly found in agent community applications should also be met with a minimum of effort. A list of some such requirements is shown in Table 1. We show in this paper that Grapevine succeeds in addressing these requirements.

3.2 Design overview

The GRAAMS approach provides the ability to *generate*, *regenerate*, and *adapt* agenda management systems that support high-level agent communication in some particular application. Generation refers to the automated building of an application-specific AMS according to some specification of the application's agenda management requirements. An AMS *designer* is responsible for capturing the requirements of the agenda management problem in the Grapevine specification language and using the specification to initiate AMS generation. Regeneration is the automated extension of an instantiated AMS with additional or altered functionality. To regenerate an instantiated AMS, a designer modifies a specification to meet the changed requirements and initiates regeneration. Adaptation is the customization of an AMS's run-time data structures and/or agent interface code. Adaptation is performed by the *user(s)* of an instantiated AMS. The users of an AMS are the people who collaborate or who write software agents that collaborate.

Any approach to agenda management should meet an application's requirements while still accommodating subsequent changes in those requirements, including individual users' demands. Our approach, while based on generation, is also designed to accommodate a range of different sorts of subsequent regeneration and adaptation activities. Providing support for both generation and subsequent evolution is important for three reasons. First, automated instantiation of an AMS that meets an application's requirements can greatly reduce the effort required to create an agenda management system. Second, ensuring that the capabilities of a created system can be incrementally extended to meet changes to requirements can reduce the effort required to maintain the AMS. Third, allowing users of an AMS to adapt or customize it for their personal preferences can improve its acceptability.

The approach taken to generation in Grapevine is based on object-oriented extension of a set of root classes and policies via a language that is used to specify how an agenda management system should be generated and

Requirement	Description
hierarchy	Support dynamic decomposition (and composition) of agenda items into sub-items and agendas into sub-agendas.
priority	The ability to prioritize agenda items within an agenda and to change these priorities.
prerequisites	The ability to allow attributes of an agenda item to depend on the values of attributes of other agenda items or agendas.
alternative items	The ability to allow an item to be composed of alternative items, meaning there is an exclusive-OR relation between them.
AMS data persistence	Support for varying persistence requirements, handling both transient data and data that should persist for long durations.
access control	Support for the ability to control which agents may invoke operations on particular agendas and agenda items.
auditing	The ability to record an agenda's or agenda item's history (e.g., which agent created, modified, or moved it to another agenda), and associated item information (e.g., what other items depend on an item). An agent should be able to retrieve a specified audit trail.
reflection	Support for responding to queries about the nature of the data in the system and about the system itself.
a variety of views	Support for the customization of views of AMS data.
sharable	The ability to share agendas and agenda items, where an object is considered sharable if it may be accessed or modified by more than one agent.
distributed, concurrent	The ability for an agenda management system to be distributed across different platforms and to support multiple agents concurrently.
coherent	The ability to provide coherent views of agendas, requiring that if data on agendas are accessible and modifiable by several agents simultaneously, the representations are kept globally consistent.
evolution	Support for the changing needs of agents.

Table 1. Common application requirements

regenerated.[1] Four root classes (attribute, agenda item, agenda, and attribute collection) form the basic extensible structure upon which the customized classes needed for a generated AMS are built. Agendas conceptually represent to-do lists, and agenda items conceptually represent tasks on those lists. Attributes are used to describe agendas and agenda items, and attribute collections collect related attributes. The structure of these classes is conceptually simple, yet provides designers with enough expressive power to specify the design of customized data for many different AMSs. The root classes in Grapevine are described in section 4.1.

Generation of an AMS is depicted in Figure 1; regeneration (that is, iteration of the generation process with additional specifications) is shown in Figure 2.

The agenda management system designer in Figure 1 is responsible for expressing the actual requirements for agenda management. By specifying extensions to the basic capabilities provided by the root classes, the designer crafts an AMS to meet the requirements of a specific application area. Once the specification is produced, it is provided to the generator, represented by the oval in Figure 1. The generator then instantiates an AMS (which can then be adapted by end users). The AMS consists of subclasses that extend the capabilities of the root classes, a generic substrate that implements the root classes, as well as agent interfaces and optionally a generic human user interface. The substrate is a set of underlying facilities that support concurrency, distribution, and other fundamental needs; it may be thought of as an extended operating system library designed to support agenda management. The root classes and substrate are not modified during AMS generation, but remain constant regardless of the specification.

After generation, the AMS is adapted if necessary and then deployed to meet the requirements provided to the designer. Agent specific interface code may be linked with the application programming interface (API) to the

[1]While Grapevine has been implemented in Java and reuses a variety of language features (e.g., inheritance, reflection), our approach is designed to be language-neutral.

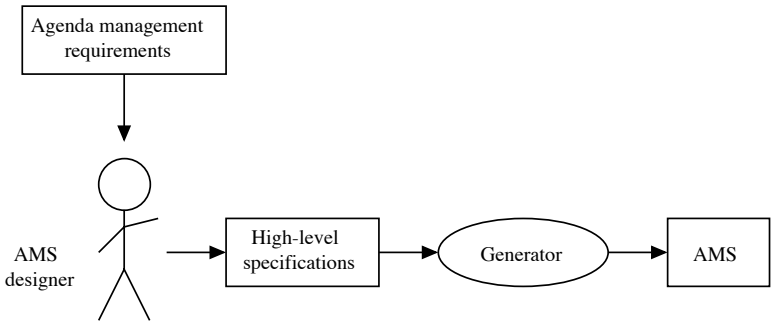


Fig. 1. Conceptual diagram illustrating the agenda system generation process

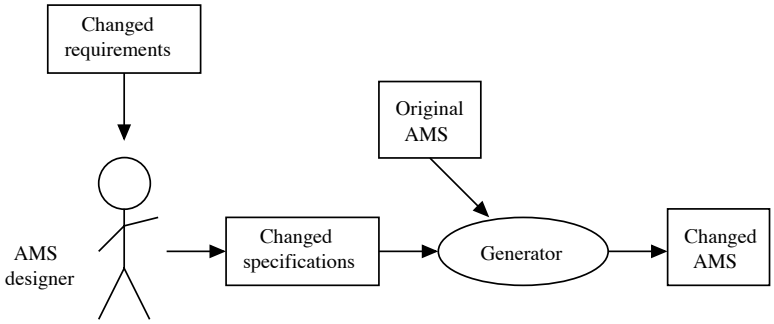


Fig. 2. Agenda system regeneration process

AMS to provide agents with views of, and a means of modifying, AMS data. In the case of human agents, this interface code will usually be written with a GUI tool. For software agents, the viewer can be considered the "glue code" necessary to integrate the agent with the AMS.

Regeneration of instantiated agenda management systems may be performed by iteratively using the generation process, as depicted in Figure 2. To add support for additional required capabilities, a designer can write a specification that describes additional classes that support these capabilities. This specification is provided to the generator, along with some representation of the original AMS, either the original specification or previously generated AMS components. The generator then generates additional object classes that augment the AMS with new object classes. The generated code can be dynamically linked with the existing system to provide the additional capabilities. Because any additional code may be linked dynamically, no changes to previously generated AMS code must be made (hence the use of the word "incremental"), and it is thus possible to augment a running system without discarding existing code or persistent AMS data. These kinds of changes can be characterized as global changes to the AMS because the newly generated object classes are available system-wide. It is easy to generate new classes to replace existing ones, provided the run-time environment provides the appropriate support for dynamically changing class implementations.

In addition to regeneration, a second kind of support for evolution, agent adaptation of object instances (AMS data), is provided through the use of dynamic data structures. By providing attribute collections, a very flexible dynamic data structure, users have a great deal of flexibility in creating instances that are composed of objects of existing classes. This kind of adaptation supports changes that require additional information for only a subset of the instances or agents associated with a particular AMS. If, for example, two humans need to use a previously generated AMS to communicate about a web page they are working with, they could use an existing field, such as a description or notes field, to convey the URL of the page.

A third kind of support is the decoupling of "view" from the AMS itself: agents can be provided with a customized viewer that presents AMS information in the way each user or tool desires, as shown in Figure 3. For

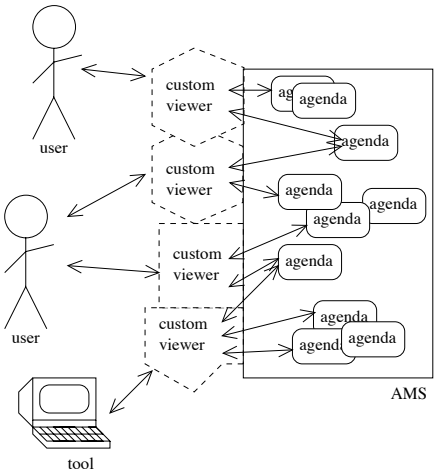


Fig. 3. Agents using an adapted AMS

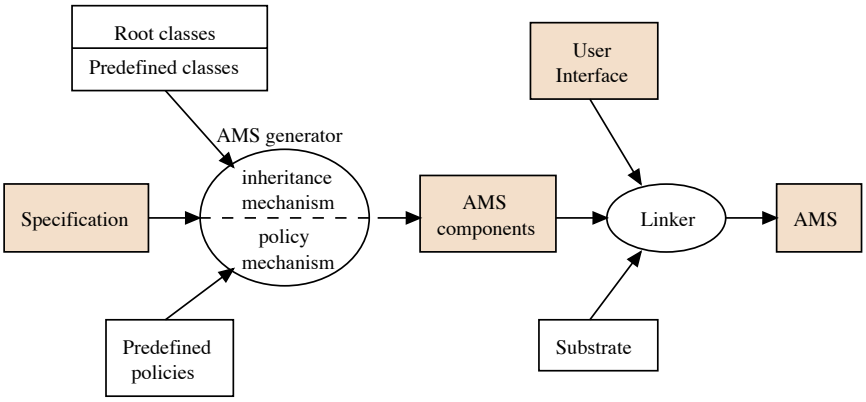


Fig. 4. Components of Grapevine. Shading indicates an application-specific part.

example, if a viewer were customized to identify URLs when rendering AMS data, it might present clickable links for any URLs it found in a description field. While automating viewer adaptation is outside the scope of the current GRAAMS approach, it is important to note that the GRAAMS architecture has been designed to support customized viewers. In fact, decoupling viewers from the rest of an AMS greatly aids efforts to treat human and software tool users of an AMS uniformly.

These three kinds of support for change allow substantial flexibility in meeting changes in requirements: global changes that require new data types and implementations may be met with specification additions and regeneration; local requirements on instance data may be met through the use of flexible dynamic data structures; local requirements on the use of instance data may be met though agent interface customization; and local requirements on instance data and use of data may be met though a combination of the latter two. While these evolution and adaptation mechanisms may not provide sufficient flexibility when complex changes to an AMS must be made, they were chosen because we believe that they accommodate a rich and interesting set of realistic changes in application requirements and can be supported at reasonable cost.

4. GRAPEVINE

The major components of Grapevine, depicted in Figure 4, include the root objects, the inheritance and policy mechanisms, and a set of predefined objects and policies. This section shows how these pieces work together to

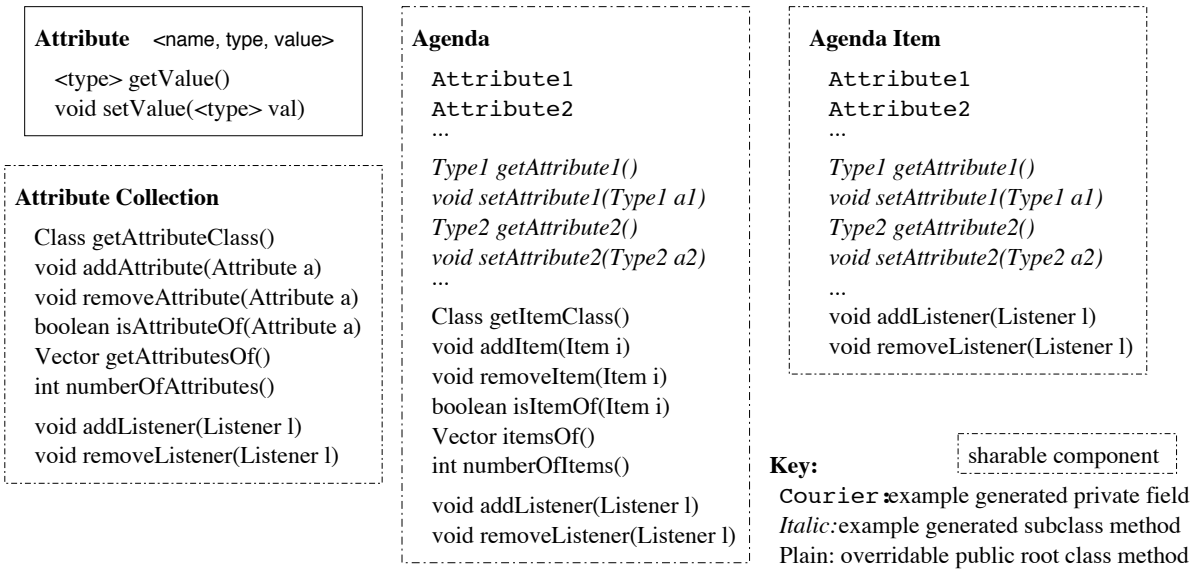


Fig. 5. Root class and subclass template definition (not all methods are shown).

form an AMS that meets the designer's requirements. Supplemental discussion of how Grapevine accommodates changes in applications requirements appears throughout.

It is useful first to introduce some hypothetical but plausible requirements for a simple example agenda management system that will be used to illustrate how the components of Grapevine are used. The example AMS has five primary requirements, as follows.

(1) Users should be able to name agendas and items in the system.
(2) Each agenda and item should have an auditing log associated with it where an audit entry should be appended to the log whenever the name of an agenda or item is changed.
(3) The AMS should support distributed, consistent access by both humans and tools.
(4) Items should be decomposable into a directed acyclic graph (DAG) of subitems.
(5) The AMS should support the assignment of items to groups of agents.

In the discussion that follows, we show how these requirements are met.

4.1 Root Classes

Figure 5 shows the fields and methods of the root classes and the way in which subclasses are generated from them through the inheritance mechanism. Each root class and how it is extended is described in detail below. Dashed lines in the figure indicate that the class and subclasses are sharable, i.e., their instances have global identity in a distributed system.

An **attribute** is a class that is the fundamental building block of an AMS, forming a field of agenda item and agenda classes. Each attribute class consists of a name, a data type, and a value. Methods are provided to get and set each attribute's value and to get each attribute's name and type fields via reflection.

The type field of an attribute subclass may be any standard data type in the underlying implementation language [2] (e.g., integer, float, character, and arrays of these) plus any user-defined type, and is represented by "*Type1*" in Figure 5. Because the attribute root class has no type, it cannot be instantiated. Attributes are defined as root objects to provide a uniform, reflective way to store all information in an agenda management system. They

[2] In Grapevine we have also introduced a pseudo-root class, **enumerated attribute**, that compensates for the lack of enumerated primitive types in Java.

are uniform because they allow any type of data to be stored, and they are reflective because query methods are available to answer queries about the type and name of the data. Attributes are not sharable; they are contained within exactly one agenda, agenda item, or attribute collection.

An example of an attribute is the `status` attribute, which has the name "status", is of enumerated type, and has some descriptive information, e.g., "Complete" or "Aborted", as its possible values.

An **attribute collection** is a class that is a collection of attributes whose membership may change dynamically. Methods are provided for clients to manage the contents of the collection and to receive events (described in section 4.4) generated by the collection. Attribute collections are used for grouping together related AMS entities and for dynamic adaptation of AMS data.

If the designer specifies that an attribute collection subclass may only contain attributes of a particular class, the add and remove method implementations are changed to throw exceptions when a client attempts to add an item of the wrong class to the collection.[3] This allows the designer to trade flexibility for stronger type checking.

Methods could be provided to associatively construct sub-collections based on member attributes' name, type, or value, or some such combination, however the current design does not provide these or any other methods that allow associative access to the attributes in a collection. Instead, we assume the client application will use a Visitor design pattern [Gamma et al. 1995] to construct the queries that are appropriate to it.

An **agenda item** (also referred to as just "item") is a class consisting of zero or more attributes, none of which has the same name. An item conceptually represents a task that must be performed by an agent. The root agenda item class has no attributes and consequently no attribute-specific methods. If the designer adds specific attributes, indicated by italics in Figure 5, to subclasses of the root agenda item class, methods to get and set the value of each such attribute are generated. An item may be queried to determine its type. Items also have methods that allow clients to receive events (described in section 4.4) that are generated by the item.

Attribute collections and agenda items differ in two important ways. First, an agenda item's attributes are statically typed. The number and type of an attribute collection's members may vary dynamically. Second, an item may not contain two attributes having the same name, while an attribute collection has no such restriction.

For example, an agenda item subclass might have two attribute fields, the `status` attribute mentioned previously, and a `name` attribute that has a name of "name," a string type, and a value that describes the agenda item. This agenda item might also provide additional constructor methods that take status and name values as arguments.

The type of any attribute of an agenda item can be an attribute collection. This definition affords enough dynamism for items to provide required functionality, such as annotation and auditing, while still being strongly typed. For example, consider an auditing log for some agenda item. The log attribute's type might be an attribute collection consisting of any number of log entries. Log entries may thus be dynamically added to an item. This is exactly the behavior we require.

An **agenda** is a class with zero or more attributes, none of which has the same name, and a collection of agenda items whose members may change dynamically. An agenda conceptually represents a collection of tasks that are somehow related to one another (e.g., they should all be performed by the same agent, they are all subitems of the same item, etc.). Methods that get and set the value of each attribute and methods for managing the items on the agenda are provided. Agendas also have methods that allow clients to receive events (described in section 4.4) that are generated by the agenda.

If the designer specifies that an agenda subclass may only contain agenda items of a particular type, the add and remove method implementations are changed to throw exceptions when a client attempts to add an item of the wrong class to the agenda. This allows the designer to trade flexibility for stronger type checking.

As with subcollections, methods that return subagendas (agendas whose items are a subset of the items of the parent agenda) could be provided, however the current design does not provide these or any other methods that allow associative access to the items of an agenda. Again, we assume the client will use a Visitor design pattern [Gamma et al. 1995] to construct queries appropriate for the client application.

Details of the root classes are somewhat language dependent. Instantiated AMSs typically depend on reflective

[3]Changing the signature of the method would not override the inherited methods, so we rely on run time type checking in the method implementation.

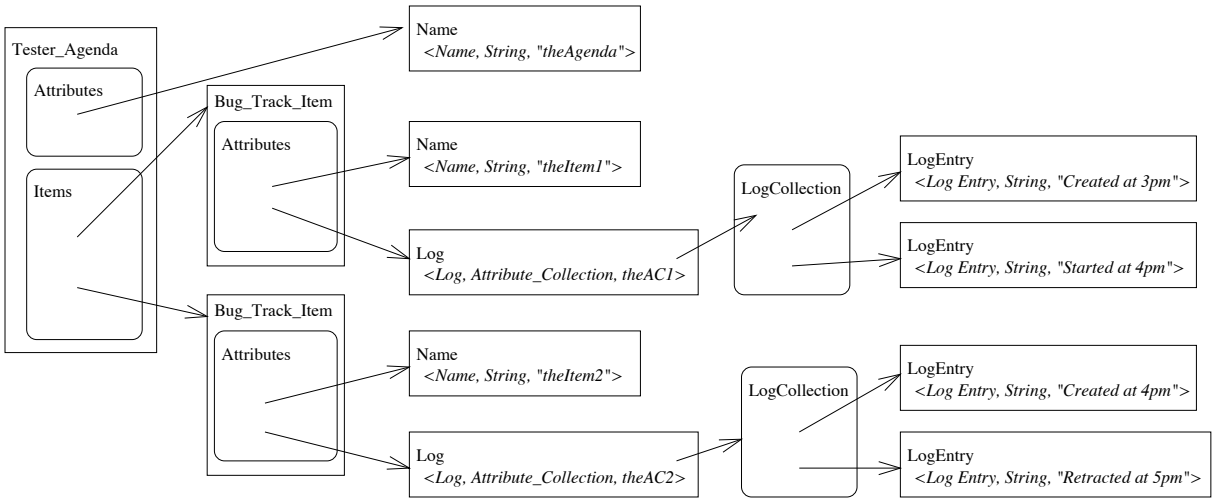


Fig. 6. Example instantiated subclasses diagram

operations to determine representation information, such as the names of an item's attributes. Grapevine has been implemented in Java, a language that provides adequate reflection, and so the necessary reflective methods have been omitted from the specification of the root classes. Only two reflective methods, `getItemClass` and `getAttributeClass`, have been added to the Agenda root class and the Attribute Collection root class, respectively.

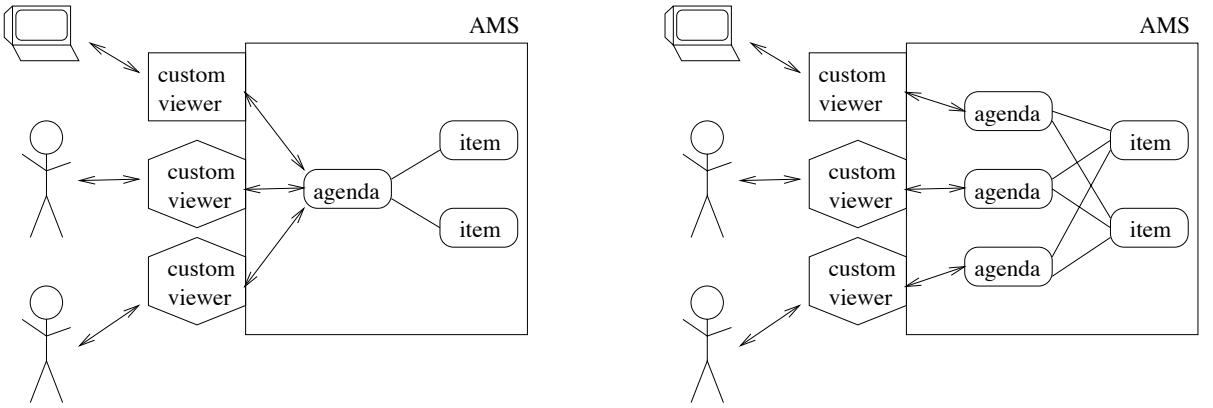
In addition, two root classes to represent iterators would normally be defined. Again, because Grapevine is implemented in Java, the `Vector` and `Enumeration` classes sufficed to provide agents with a way to iterate over all attributes in a collection and all items on an agenda. In languages without iterators available for reuse it is expected that attribute collection iterator and agenda iterator root classes would need to be defined to allow iteration over an attribute collection's attributes and an agenda's items, respectively, to meet requirements such as hierarchy naturally.

It is important to observe that these root class definitions encourage the notion that *every collection of items is an agenda*. Conspicuously absent is the class "Item Collection," having been replaced with "Agenda." We have taken a minimalist approach in the specification of the root agenda class to avoid specification and performance penalties in the implementation of what we believe is the most common type of agenda, namely a single collection of items (usually of the same type) with some associated attributes that describe the collection.

While having only one collection of agenda items per agenda is prescriptive, it is not restrictive. If there is a need for a type of agenda with one or more fixed agenda items or with more than one associated collection of items, it may be constructed through the use of the agenda's attributes. To define a subclass of an agenda with a fixed number of items, the AMS designer would include specific attributes that refer to each desired agenda item. To define a subclass of an agenda with additional item collections, attributes may refer to other agendas or to attribute collections whose attributes refer to items. Appropriate access methods would then be written by the designer, using the extension mechanism described in the next section.

Another important observation is that both agenda items and agendas are sharable, meaning that a single instance of an item or agenda may be referenced by more than one agent. Because every group of agenda items is an agenda and because agenda items may appear simultaneously in many groups, they must also be sharable. This allows, for example, the representation of prerequisite items and sub-items, another common requirement of agenda management systems.

These four root classes, in conjunction with the extension mechanism described next, allow an AMS designer to create customized classes of agenda items. A way in which subclasses of these root classes might be used to organize data in an AMS is shown in Figure 6.



Shared agenda: formal group

Shared items on separate agendas: informal group

Fig. 7. Two ways of supporting assignment of items to groups.

Looking back at the requirements of our example AMS, we can see how the "group" requirement is met by the characteristics of the root classes alone. Allowing both agendas and items to be sharable actually permits multiple ways of implementing an agenda that is shared by a group of users in an organization, as shown in Figure 7. One way is to create an instance of an agenda that all the members of the groups share. Assigning an item to the group would consist merely of adding the item to the shared agenda. Another way is to post a single instance of an item to each group member's personal agenda. In both cases, a single item can be viewed and modified by a specific group of people. A key difference between these two methods of posting the item is the degree to which the "group" is formalized. Intuition and experience with another form of collaboration, e-mail, indicates that it is natural to use more than one way of designating a group. Indeed, most e-mail systems allow messages to be sent to multiple recipients as well as sent to system-wide mailing lists or posted to newsgroups.

4.2 Extension Mechanisms

The extension mechanisms used to customize the root classes are an inheritance mechanism, which allows for type-strong extension of the root classes, and a policy mechanism, which provides enforcement of user specified constraints over the use of objects of generated classes. These mechanisms are used to create the customized classes required by specific AMSs. In Grapevine, inheritance is the extension of a class with additional fields and methods, as in a typical object-oriented language with single inheritance. A policy is a specification of constraints on the use of AMS objects. The implementation of inheritance can be provided by the object-oriented inheritance found in a reasonable implementation language (e.g., Grapevine uses Java), but code to implement policies must currently be synthesized during AMS generation.

Grapevine can also be used to support changes to an instantiated AMS, as well as additions, by changing the specification and regenerating replacement classes. Class replacement can alter the structure or behavior of an AMS, but opens the door to two potentially serious problems. The first problem is that when the structure of AMS data is changed, there are potential type incompatibilities. If, for example, a field in a class is removed or its type is changed, when objects of the old class are loaded from persistent store the AMS or run-time support system must know how to transform objects of the old class into objects of the new class. The second problem arises because AMSs are typically used in a distributed application. When (only) the behavior of a class is changed, new executable code must be loaded to replace the old code, thus the underlying operating system must support dynamic reloading of code. In addition, the code must be reloaded in a consistent manner throughout a distributed system.

Type evolution [Lerner 1997] and dynamic code replacement [Beugnard et al. 1999] are active research areas that are outside the scope of this paper. Lack of extra support does not mean that classes can never be changed,

however. Changes are merely restricted to cases in which there is no running AMS and no type evolution problems (e.g., no persistent data). It should be noted that this case is common when an application is being developed incrementally.

4.2.1 *Inheritance mechanism.* Extension of the root classes by inheritance is specified through use of the keywords `extends`, `type`, `enumerated`, `attribute`, `method`, `attributeclass`, `itemclass`, and `private`; the full grammar is specified in the appendix. These keywords are used to customize the root classes for use in a particular application. The keyword `extends` names a subclass, `type` specifies the type of the contents of a class, `enumerated` is used to define an attribute with enumerated values, `attribute` specifies that an attribute should be part of an object, `method` adds a designer-defined method to the class, `itemclass` specifies the type of items in an agenda's collection, and `private` is used to control which methods are not visible outside of the specification, i.e., to the agents.

The AMS generator creates code that defines the subclasses specified by the designer. The specification of attribute, attribute collection, attribute iterator, and agenda iterator subclasses serves mainly to define their type. These classes override or replace methods inherited from their superclasses with methods of the corresponding signature for the subclass. Agenda and agenda item subclass implementations are generated by directly copying specified methods, creating private fields with the name and type of the specified attributes and generating methods to get and set the values of these fields.

Generated classes and subclasses are strongly typed. By providing the `type` and `itemclass` keyword and by generating appropriately typed subclasses, the designer may create a diverse set of agenda item and agenda types for an AMS while ensuring that these types do not inappropriately intermix. Allowing attribute collections to be included as attributes of agendas and agenda items provides much of the adaptability that is lost due to the generation approach.

The specification sample in Figure 8 provides simple examples of extensions of the attribute, attribute collection, agenda item, and agenda root classes. Instantiation of these classes in an AMS could produce the example graph of objects shown in Figure 6.

When the specification is provided to the extension mechanism, the interface to `My_Item` that is generated includes methods for getting and setting the values of its two associated attributes. In this example, the designer's additional method, `AddLogEntry`, is also included in the generated subclass. An example generated subclass implementation in Java for `My_Item` is presented in Figure 9. The agenda subclass would be generated similarly.

Thus an agent might add a log entry to an instance of My_Item by invoking `addLogEntry("A new log entry");` the agent could not, however, access the Log attribute directly because the Log attribute's access methods have been hidden, through use of the "private" keyword. The reason for hiding the attribute access methods will become apparent in the next subsection, where we describe how policies for the use of classes are supported.

The designer may also use the specification to override methods that a subclass inherits from its superclass. If a generated method would have the same signature as a superclass method, the Grapevine extension mechanism signals an error condition (rather than silently overriding the superclass method). For example, if the designer defined a method named "setLog" in class `My_Item`, then created a subclass of `My_Item` called `My_Logged_Item` with an attribute named "Log", an error would be reported when the specification was processed because the mechanism could not generate a `setLog` method for the added "Log" attribute without overriding `setLog` of the superclass.

We now see how several other requirements of the example AMS are met. The inheritance mechanism provides a way of associating an agenda of subitems (conceptually, a subagenda) with an item, allowing items to be decomposed into a DAG. We also see how we may associate name and auditing information with each agenda and item. We have not yet described support for automatically maintaining an audit log; that support is provided by the policy mechanism.

4.2.2 *Policy mechanism.* Grapevine allows the designer to specify *policies* that provide the designer with a rudimentary yet powerful mechanism to enforce semantic consistency among the objects that comprise an AMS and to discipline their use. Policies may also play a role similar to that played by the process-constrainers of ProcessWall [Heimbigner 1992], which are used to allow a process designer to specify some constraints on the

```
attribute Name extends Attribute {
  type String;
}

attribute Owner extends Attribute {
  type AMS.Predefined.UID;
}

attribute LogEntry extends Attribute {
  type String;
}

attribute LogEnabled extends Attribute {
  type boolean;
}

collection LogCollection extends AttributeCollection {
  attributeclass LogEntry;
}

attribute Log extends Attribute {
  type LogCollection;
}

item My_Item extends Item {
  attribute Name;
  attribute Owner;
  // private prevents the Log from being visible outside the class
  private attribute Log;
  private attribute LogEnabled = true;
  method void addLogEntry(String theEntryString) {
    println("addLogEntry");  // real implementation goes here
  }

  method String printLog() {
    return "printLog";       // real implementation goes here
  }
}

agenda My_Agenda extends AMS.Predefined.NamedAgenda {
  itemclass My_Item;

  method void moveItem(My_Item anItem, My_Agenda toAgenda)
  throws java.lang.RuntimeException {
    println("moveItem");     // real implementation goes here
  }
}
```

Fig. 8. Example inheritance specification code

data in a process.

For example, suppose a designer wants to design a system that adds a log entry to an item every time the name of the item is changed, as in our example AMS. The designer could use the log attribute introduced previously to store the log, but the generated `setName` method would have to be rewritten to ensure that a log entry was made. The Grapevine policy mechanism automates this task.

A policy is specified with a named collection of constraint, action and method name triples (based on the ECA model [Dayal et al. 1990]) that is to be applied to class methods. As illustrated below, the keyword `policy` names a policy and the keywords `enforce` and `for` bind to specific methods the constraints that make up the policy. The keywords `constraint`, `action`, `check as`, `precondition`, `postcondition`, `prepostcondition`, `prerequisite`, and `prepostrequisite` are used to specify the contents of a policy. The full grammar for specifying and binding policies is specified in the appendix. The bodies of the constraints and actions making up a policy are written in the implementation language of the AMS. Constraints return a boolean value, indicating whether or not the action should be taken. Actions do not return a value; they either successfully complete or cause an action to be taken in addition to or instead of the method over which the policy is enforced. A policy's actions may use arbitrary external code that is linked with the generated AMS. This code can be used to modify the run-

```
import AMS.Root.*;

public class My_Item extends Item
{
  void addLogEntry( String theEntryString ) {
    println("addLogEntry");  // real implementation goes here
  }

  String printLog(  ) {
    return "printLog";  // real implementation goes here
  }

  public String getName(  ) {
    return _Name.getName();
  }

  public void setName( String v ) {
    _Name.setName(v);
    firePropertyChange("setName", null, v);
  }

  public AMS.Predefined.UID getOwner(  ) {
    return _Owner.getOwner();
  }

  private void setOwner( AMS.Predefined.UID  v ) {
    _Owner.setOwner(v);
    firePropertyChange("setOwner", null, v);
  }

  private LogCollection getLog(  ) {
    return _Log.getLog();
  }

  private void setLog( LogCollection v ) {
    _Log.setLog(v);
    firePropertyChange("setLog", null, v);
  }

  private boolean getLogEnabled(  ) {
    return _LogEnabled.getLogEnabled();
  }

  private void setLogEnabled( boolean v ) {
    _LogEnabled.setLogEnabled(v);
    firePropertyChange("setLogEnabled", null, new Boolean(v));
  }

  Name _Name = new Name();
  Owner _Owner = new Owner();
  private Log _Log = new Log();
  private LogEnabled _LogEnabled = new LogEnabled(true);
}
```

Fig. 9. Example generated code

time behavior of the policy. In addition, while not currently supported, the generator could be trivially modified to allow a particular object's policies to be enabled and disabled at run-time. Thus, the AMS could provide dynamic control over how a policy is enforced and over whether the policy is enforced at all, affording greater run-time adaptability.

In the previous example we have also seen that the classes for which a policy should apply is not determined by the inheritance hierarchy, but instead by which classes contain certain attributes and are subject to external design constraints. Policies therefore also provide a way of uniformly applying control decisions to classes orthogonal to the inheritance hierarchy. This need might also be met with multiple inheritance mechanisms, as in C++, or with subject- and aspect-oriented programming approaches described in the literature [Harrison and Ossher 1993; Kiczales et al. 1997].

When a policy has been enforced for an object's method and that method is invoked, each constraint designated as

a precondition is evaluated in the order the `check` clauses appeared in the specification of the policy. If a constraint is false, the action is taken, then the next constraint is checked or the method itself is executed. In the case of prerequisite and prepostrequisite constraints, the constraint also determines whether or not the remaining enforced constraints are checked and whether the constrained method executes at all. After the "pre-" constraints, actions, and method execute, the postconditions are evaluated in similar order, and their actions are taken if necessary. If an exception is thrown by an action, it will appear to the caller of the constrained method to have come from the method itself. Pseudocode for this execution sequence is given below.

```
for(each precondition constraint) do
 if(condition is false) then take action
 if(policy binding is a requisite) then goto post
endfor
invoke method
post:
for(each postcondition constraint) do
 if(condition is false) then take action endfor
```

An important semantic restriction on the way the policy mechanism is defined is that the arguments of the constraint, the action, and the methods over which the policy is enforced must all match. If a policy is to be enforced for several methods with differing parameters, several sets of constraints and actions should be provided. Only those constraints and actions whose parameters match those of the method will actually be enforced. A policy typically works together with one or more attributes of objects of the class whose methods have the policy enforced. When a policy requires the presence of certain attributes and is enforced for a method of a class that does not have the attributes associated with it, the generator issues an error. In the Grapevine prototype, the error is actually generated by the compiler at the time that it fails to compile the generated class. An example of how policy parameters are used appears later in this section. Currently, we do not directly support applying policies to the constraints and actions of other policies, since this raises additional issues of order and termination. Such support may, however, be provided in the future.

When a method of a subclass has a policy enforced on it, the extension mechanism renames and hides the method, then creates a new method that will be used in its place. The new method contains code that evaluates the constraints in their specified order, taking the required actions, and then invokes the renamed method. It is important to note that the interface of a generated class can be the same regardless of whether or not a policy is enforced on it. The effect of a policy is on a class's implementation, not its interface. Thus, one way in which an existing AMS can evolve is by adding or removing policy enforcement.

When a subclass is defined with methods that override superclass methods that have had policies enforced, the overridden methods no longer have the policy applied.[4] The designer must explicitly re-enforce the policy for the overridden method. This allows a designer to define a subclass that has a different policy, or no policy at all, enforced for the same method.

If multiple policies are enforced for a method, several alternative execution sequences exist. We considered two of them. First, the composite precondition constraints could all be evaluated (effectively in parallel), then all required actions could be taken, then the method could execute, then all composite postcondition constraints could be evaluated and finally their required actions could be taken. Alternatively, each policy's constraints could be evaluated in the sequence in which the policies are enforced. In this case, if an action defined in one policy were to raise an exception or if a prerequisite condition failed, policies whose enforcement clause appeared later in the specification for the method would not be executed at all. The semantics of the former alternative are appealing, however there are design and implementation concerns with respect to livelock detection and policy commutativity when a policy's conditions have side-effects or when actions can affect the outcome of condition evaluation. We have therefore adopted the simpler semantics of strictly sequential policy execution. Further experimentation may

[4]These semantics were chosen partly to match the inheritance semantics of the "synchronized" keyword in Java.

```
policy logging( String opname ) {
  constraint boolean isLoggingDisabled( String s ) {
    return !getLogEnabled();
  }
  action void addLogEntry( String s ) {
    LogCollection ac = getLog();
    LogEntry entry = new LogEntry(opname+"("+ s +") invoked.");
    ac.addAttribute( entry );
  }
  check as precondition;

  constraint boolean isLoggingDisabled( AMS.Predefined.UID u ) {
    return !getLogEnabled();
  }
  action void addLogEntry( AMS.Predefined.UID u ) {
    LogCollection ac = getLog();
    LogEntry entry = new LogEntry( opname + "(" + u.toString() +
                     ") invoked.");
    ac.addAttribute( entry );
  }
  check as precondition;
}
```

Fig. 10. Example policy specification code

necessitate re-evaluation of this decision.

Policies touch on a number of issues related to consistency management in software systems. The policy mechanism described here is meant to provide rudimentary consistency management capabilities; recent work [Tarr and Clarke 1998] shows a number of directions in which the policy mechanism could be augmented.

The example in Figure 10 should help clarify the way in which policies are generated. It specifies a logging policy, with the bodies of the constraints and actions written in Java. In this example, the `logging` policy consists of only one action, constraint, and checkpoint triple, though in practice many such triples might be required to describe a policy.

An example of binding the example policy to an item's methods is provided below.

```
enforce logging("setOwner") for My_Item.setOwner;
enforce logging("setName") for My_Agenda.setName;
```

This example also demonstrates how parameters are used. The policy parameter ("String opname" in the example) is provided to pass to the constraints and actions information about the method on which a policy has been enforced. The names of the policy parameters may be used within the constraint and action implementations as regular identifiers. The method parameters ("String name" and "UID owner" in the example) are used to pass to the constraints and actions information about the arguments to methods on which a policy has been enforced. Multiple constraint and action definitions may be provided in a policy definition. Only those constraints and actions whose parameters match the method parameters will actually be bound to the methods. If there are no matching constraints, an error will be signalled during AMS generation.

Policies may be applied both to methods defined in the specification with the `methods` keyword and to the attribute access methods that will be automatically generated from the specification. For example, a policy may be enforced for the `setName` method generated for the item defined in section 4.2.1.

The Java code produced by the generation mechanism for the example My_Item from the previous subsection with this policy enforced appears in Figure 11 (only the setName method is changed).

An alternative implementation would be to have the methods associated with the logging policy stored in a singleton object [Gamma et al. 1995], whose methods are invoked by each "bound" method. This implementation would allow a policy to be stored in one class, rather than many. The implementation of a policy could be changed by replacing that class. This scheme offers problems associated with the visibility of private methods, however. In the example above, the constraint must check the value of this object's LogEnabled attribute, which has been marked private; the policy would not be able to access the field of the object. In addition, dynamically loading classes, as in Java, can create complications for regenerating parts of an AMS. It is also not clear whether enabling

```
public class My_Item extends Item
{
  private void private_loggingaddLogEntry_setOwner(AMS.Predefined.UID v) {
    _Owner.setOwner(v);
    firePropertyChange("setOwner", null, v);
  }

  public void setOwner( AMS.Predefined.UID v )   {
    if(!loggingisLoggingDisabled(v))
      loggingaddLogEntry(v);
    private_loggingaddLogEntry_setOwner(v);
  }

  boolean loggingisLoggingDisabled( AMS.Predefined.UID u ) {
    return !getLogEnabled();
  }

  void loggingaddLogEntry( AMS.Predefined.UID u ) {
    LogCollection ac = getLog();
    LogEntry entry = new LogEntry(opname+"("+u.toString()+") invoked.");
    ac.addAttribute( entry );
  }

  Name _Name = new Name();
  Owner _Owner = new Owner();
  private Log _Log = new Log();
  private LogEnabled _LogEnabled = new LogEnabled(true);
  private String opname = new String("setOwner");
}
```

Fig. 11. Example generated code (unchanged methods omitted)

policies per-class versus per-object is sufficient. Here too, further experimentation will provide clarification.

We see now that the policy mechanism allows us to meet a remaining requirement of the example AMS; the last binding in the above example shows how a policy can be used to automatically maintain an audit trail of changes made to the name attribute of an item and an agenda. Only one policy has to be written; it can be applied to both the item and agenda subclasses because both have the two attributes that the policy requires, `Name` and `Log`. Maintaining such an audit trail is likely to be a common requirement of applications that are built with an agenda management approach. In the next section, we show how predefined policies and classes aid developers in meeting common agenda management requirements.

4.3 Predefined Classes and Policies

Grapevine provides a set of predefined classes and policies. By providing an ever-growing library of common high-level components, we hope to ease application development further. Currently, Grapevine provides attributes for name, priority, deadline, status, description, owner, log, notes, and several others that are commonly required to describe agenda management system data. Two interesting attributes that are also provided are the alternatives attribute, which designates items that are alternatives to the one containing the attribute, and the prerequisites attribute, which designates an agenda that holds items that must all be "completed" before this one is "started" (as reflected by the status attribute). Thus, these two attributes support "exclusive or" and sequential "and" semantics.

Grapevine also provides one predefined agenda class, `NamedAgenda`. `NamedAgenda` is simply an agenda with the Name attribute; it has been added to the set of predefined classes because it works with a special predefined class, `NamedAgendaDirectory`, which will be described later.

Several predefined policies are also provided and described in Table 2. Many of the predefined attributes are complemented by one or more predefined policies.

By supplying a library of predefined classes and policies, we not only provide ways of letting AMSs evolve, we provide a way to allow the framework itself to evolve over time. As more agenda management systems are created, this library will grow, providing further assistance to those who are creating and extending AMSs.

Name	Description
LogNameChanges	This policy creates a new `LogEntry` attribute that records the time and new value of the Name attribute, and adds it to the log attribute collection.
LogAttributeChanges	This policy creates a new `LogEntry` attribute that records the time and new attribute value adds it to the log attribute collection. Unlike LogNameChanges, above, this policy does not verify that the new value is actually different from the old value.
AccessControl	This policy uses an external access control table and the owner attribute to look up access privileges, providing control over who may invoke a method. This policy relies on external access control table lookup functionality which has not implemented.
Alternatives	This policy, used with the alternatives and status attributes, automatically removes all alternatives to an agenda item once that item has been started.
Prerequisites	This policy, used with the prerequisites and status attributes, prevents an item from having its status modified until its prerequisite items have been completed.

Table 2. Predefined policies

Class	Method
Agenda	addItem
Agenda	removeItem
Attribute Collection	addAttribute
Attribute Collection	removeAttribute

Table 3. AMS events of root classes

4.4 AMS events

In addition to providing a rich interface for accessing data in an agenda management system, AMS objects are designed to interact with application code by generating events whenever an AMS object's value changes. The event mechanism provided by an AMS allows user interface components and software agents in an application (event "listeners") to register interest in an object so that they later receive callbacks that deliver events that the object generates, loosely following the standard Java Beans property change event model [Flanagan 1997].

The methods in the root classes that generate events are shown in Table 3. When agenda and item subclasses have attributes associated with them, the generator also creates code that sends an event to all registered listeners when the attribute's value is set. The event that is sent conveys the source of the event, the name of the method that was invoked, and the argument of the method. AMS events allow, for example, a user interface component to respond to updates to an item's status attribute by redrawing the item. In addition, designers can write method bodies that generate events, allowing both designer-specified AMS object methods and the action and constraint parts of enforced policies to issue events.

AMS events are a general feature of Grapevine rather than being provided only for some AMSs because it is expected that use of an AMS tends to be event-driven; an agent wants to be notified when another agent communicates with it via the AMS so that it can immediately take appropriate action. This expectation has been met for all applications that have been implemented with Grapevine so far.

4.5 Substrate

Previously we have described how a designer can meet many specific agenda management requirements by using the inheritance and policy mechanisms of the generator component of Grapevine. The architecture of the substrate on which the root classes (and consequently the rest of the system) are built must allow an AMS to meet other requirements commonly found in an agenda management context. The substrate must address requirements such as concurrency control, AMS data persistence, distribution, coherence, and scalability of the AMS.

The requirements of the substrate are described in two categories: general and specific. General requirements must be met to ensure proper implementation of any agenda management system. Specific requirements must be met only if particular applications have requirements that cannot be met by the inheritance or policy mechanisms

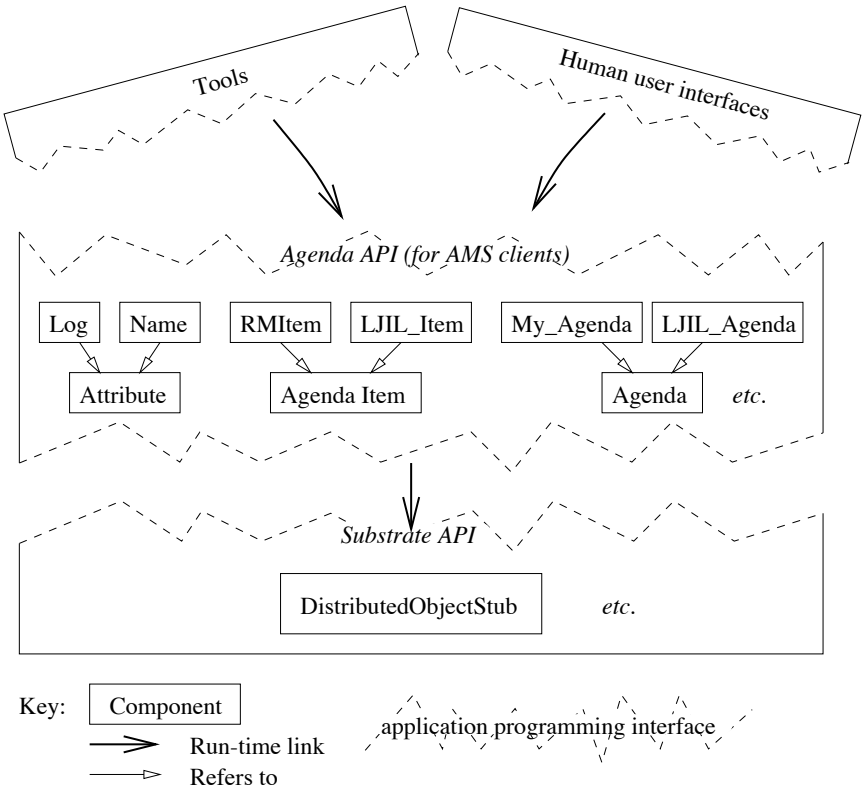


Fig. 12. APIs in an AMS

and must instead be addressed at the lower substrate level.

We have identified two general requirements that must be met by an AMS substrate: access to shared objects (including global object identity) and notification (event delivery) when an object's state is changed. By meeting the first requirement, a substrate allows AMS class instances to be stored in different operating system address spaces yet remain accessible to all clients of the AMS API. By meeting the second requirement, a substrate allows the AMS to notify clients of events. For example, some agents might need to be notified when an item is added to an agenda or when the status attribute of an item changes value. This requirement is general because AMS events are not specific to a particular AMS, but are provided by all instantiated AMSs.

By polling object state, AMS events can be implemented atop a substrate that does not support events [McCall et al. 1998], but such a design can reduce scalability and increase the system resources consumed by an AMS. Therefore it is preferable that the substrate provide a native event mechanism so that AMS events may be efficiently implemented.

Examples of specific requirements that the substrate might need to meet include persistent storage of AMS data, transactional or atomic access to AMS data, support for disconnected operation, flexible concurrency control (e.g., explicit object locking operations), scalability, and high performance. The role of the substrate in Grapevine is to insulate designers from the details of distributed operation, simplifying system design and the AMS instantiation process, while meeting the general and specific requirements. We hypothesize that AMSs are likely to have similar enough substrate requirements that providing a one-size-fits-all solution will make the GRAAMS approach more efficient in the end, even if it means that, for some AMSs, the substrate is excessive. In trading flexibility for simplicity, we allow a designer to worry about modeling AMS data, not the details of an arcane interface description language. Thus, all generated subclasses are implemented atop the substrate while agent specific interfaces use only the AMS API, as illustrated in Figure 12.

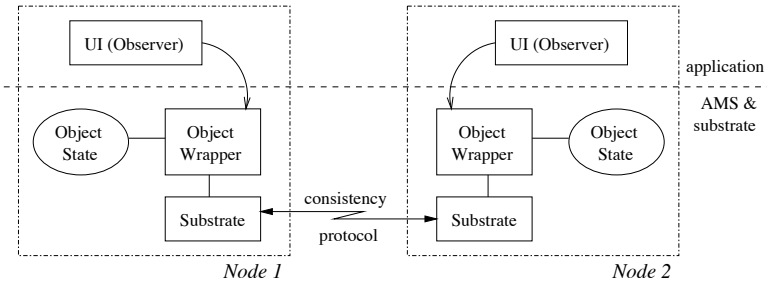


Fig. 13. Application view of substrate

Although we considered several designs for the substrate, we decided on a substrate composed of distributed, cached objects. Caching objects raises issues of object coherence that do not appear in a client/server case. This model of data distribution, however, seems a far better reflection of the agenda metaphor; agenda information is reproduced close to the agents who are interested in it, and agenda items that are frequently accessed by many (e.g., hold a meeting) are replicated, increasing the likelihood of meeting the performance objectives. Having shared attributes does not impact performance as dramatically as when objects are stored on a remote server because attributes will typically be cached locally and invocations of the fine-grained attribute accessor methods will not result in high-latency remote procedure calls. In addition, notification is easily and efficiently supported by reusing infrastructure that is used for cache coherence. Therefore, a substrate that implements an object caching layer beneath the root object classes was designed and implemented to serve as the substrate component of Grapevine.

Figure 13 depicts an example application's view of the object caching substrate. In the example, two user interface components are observing a shared distributed object. The state of the object is replicated in each address space and the object's methods execute locally. When the user interface in one address space invokes a method that changes the state of the object in that address space, the wrapper passes the invocation through to the local copy of the object, then executes a consistency protocol that updates the state of the object in the other address space. If the object generates an event, the event is delivered locally to the wrapper that then forwards it to any remote listeners.

4.6 Agent Interfaces

To use an instantiated AMS, software and human agents access AMS data through the AMS API as shown at the top of Figure 12. As noted before, both software and human agents use the system similarly; the AMS API is a uniform agent interface that is accessed indirectly by humans via graphical user interfaces (GUIs) and directly by software components. Agent interfaces are linked with the AMS components and substrate to form the running application, shown in Figure 4.

In this section, we briefly describe how software tools and human user interfaces use the AMS API. We describe a generic human user interface that works with all generated AMSs, and discuss some design issues related to the creation of application specific human user interfaces.

4.6.1 *Software tool interface*. Using an instantiated AMS when writing a software agent is straightforward. The AMS API is used just as any standard operating system interface or support library is used and allows software agents to have full access to AMS data. Furthermore, software agents may register to receive AMS events from objects. Documentation for the AMS API can also be generated by Grapevine.

To integrate legacy tools into an AMS-based application, it is necessary to write an AMS proxy for the tool that translates AMS data into tool invocations or procedure calls. If, for example, a legacy command line tool that ran regression tests on a particular package is to be integrated into an AMS-based application, the proxy agent is implemented by having it monitor an agenda for the regression test tool. When an item is placed on the agenda, the proxy translates the data about which tests should be run, found in the item's attributes, into command line arguments that are passed to the tool. After the tool is run by the proxy agent, its results are collected and translated into attribute values that are stored in the agenda item, or perhaps stored in a new agenda item that is then placed on some other agenda.

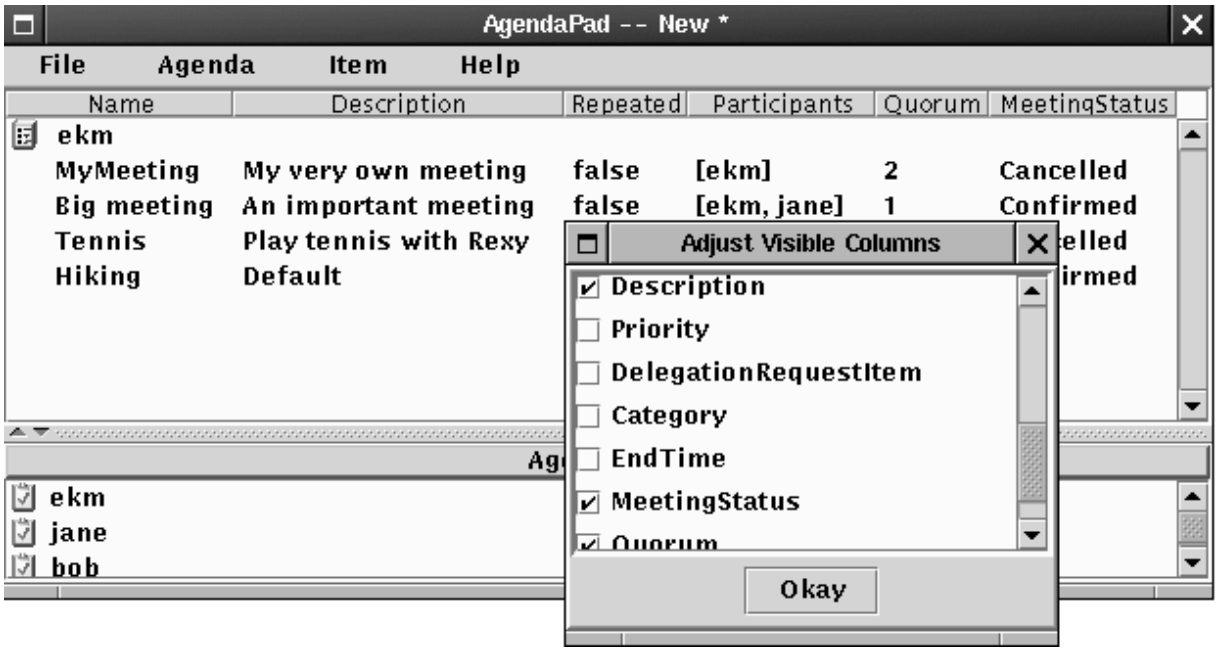


Fig. 14. The AgendaPad user interface

This integration technique has been used with success to integrate a COTS tool (Rational's Rose) and several AMS-unaware pieces of software (a resource manager, image processing tools) into several AMS-based applications (Juliette, described in section 5.1, and a parallel image processing application, described in section 5.3). Many software agents that access the AMS API directly have been successfully written; proxies can be seen as a special kind of general software agent.

4.6.2 *Human user interface*. A human user interface can also be viewed as a special kind of software agent, one that is responsible for interacting with a computer's I/O hardware to inform a human (a "legacy" tool) about the state of AMS data in which the human in interested. This section discusses the generic human user interface that is provided as a component of Grapevine and describes how application-specific user interfaces may use properties of Grapevine-generated code effectively.

Because AMS classes are type-strong and already have a variety of reflective methods available, and because of the potential complexity of generating customized UI components, Grapevine offers an application-independent user interface that adapts at run-time to correctly interact with application-specific AMS data. Development of a generic user interface is simplified by several characteristics of Grapevine's design. Because an instantiated AMS is object-oriented and provides good support for AMS events, we can leverage common UI construction techniques, frameworks, and tools. For example, the common model-view-controller user interface OO design pattern [Gamma et al. 1995] has worked well. The generated AMS classes also comply with the Java Beans specification [Flanagan 1997]. This fact, combined with the "itemclass" keyword in the specification, means that the generic UI can use the introspection capabilities in the Java Beans package to correctly infer the names and types of the attributes of an agenda and of the items on that agenda. Because the user interface can dynamically find out about and render the attributes of an agenda or item, it can be designed to work with any generated agendas or items. Providing a way for the values of attributes to be dynamically edited is more difficult unless developers define editors for each new attribute type.

The main window of the generic Grapevine user interface, called AgendaPad, is shown in Figure 14. The large

lines	words	characters	filename
27	134	1149	policies.ams
90	201	1690	predefined.ams
117	335	2839	total

Table 4. Predefined package: specifications

window in the background displays a view of an agendam in this case named `ekm`.[5] The background window's lower section displays the names of all available agendas and the upper section (the main view) displays a subset of those in greater detail. Agendas in the main view may be opened and closed by double-clicking on the appropriate row. The figure shows one open agenda that has four items on it. The values of the attributes of all the items on the open agenda(s) appear in the columns of the main view. In the foreground is a window that allows the user to select which attributes will appear in the main view. The AgendaPad user interface also allows a user to save current agenda selection, visible column selections, and other session data. Double-clicking on an item in the main view will display an edit window for it.

Application-specific human interfaces are implemented similarly to the generic user interface, except that UI components are customized for the semantics of the application. Because instantiated AMSs are designed for use with a familiar UI model, namely the Model-View-Controller design pattern, and because AMS components are JavaBeans, off-the-shelf tools can be used to aid in the construction of application specific user interfaces. For example, the user interface for the laboratory coordination environment case study application was constructed with NetBeans, an all-Java integrated development environment. That user interface and other application-specific user interfaces are discussed further in section 5.

4.7 Grapevine implementation experiences

The components in the `Root` package were implemented in Java by hand and were wrapped with an object caching substrate wrapper tool to support distributed operation. The implementation of the root classes consists of straightforward implementation of the methods shown in Figure 5, in addition to methods that implement the Java Beans PropertyChangeListener interface that is used by the substrate to deliver AMS events.

When writing agents that use the AMS API, the software needs some way of finding at least one object in the system. Distributed systems usually support some kind of naming service, and the substrate we used was no different. We enabled client code to find AMS objects by creating two extra classes, Directory and NamedAgendaDirectory in the `Predefined` package. These classes implement a globally accessible agenda collection and have been used in every agenda management system to date. The Directory class provides methods to add and remove agendas from the directory and a method to return a list of the agendas in the directory. The NamedAgendaDirectory class complements the NamedAgenda agenda subclass in the Predefined package by extending the Directory with methods to return an agenda of a particular name and to get-or-create an agenda with a particular name.

4.7.1 *Predefined components.* Predefined classes and policies were created and used by developing two specification files. One specification file contains the specification of the predefined classes in the `Predefined` Java package, and the other contains the definition of the predefined policies. The predefined classes are regenerated whenever the predefined specification changed, and are loaded on-demand by the generator whenever an application-specific AMS uses a predefined class.

The sizes of the predefined package specification files (including comments) appears in Table 4. All three of the case study applications reused at least some of the predefined classes from these packages, and all reused most of the classes in the root package. The total specification length of the predefined package is 117 lines, which produces 462 lines of generated, wrapped code. In addition, the two agenda directory classes are provided and wrapped, adding 438 lines of code to the predefined package. The ObjectCache package, on which predefined classes, root classes, and generated subclasses rely, is 5,805 lines of code, with 854 lines of that forming the

[5]The AgendaPad displays the value of the agenda's "Name" attribute if it exists, otherwise it displays a text representation of the object reference.

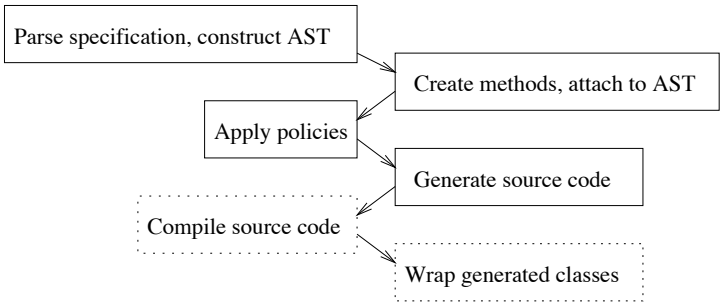


Fig. 15. `BuildAMS` stages

WrapperGen tool, the tool that prepares shared classes for use with the substrate. WrapperGen is not needed by a running AMS and has therefore been removed from the total application size information that appears in section 5.

4.7.2 *BuildAMS, the AMS generator*. BuildAMS generates code in four stages, depicted in Figure 15 (with two optional stages in dotted boxes). First, the specification files are parsed and an abstract syntax tree (AST) is constructed. Second, attribute accessor and designer-specified methods are created and attached to the appropriate AST nodes. Third, policies are applied to the methods for which they are enforced. Finally, Java code is generated. `BuildAMS` by default also invokes the Java compiler to compile generated code, then invokes `WrapperGen`. Performance numbers for code generation are given in section 5 both with and without the optional stages.

Implementation of the BuildAMS generator was straightforward. An LL(1) parser and code generator were written to produce AMS classes from the designer's specification. The generator consists of a combination of a source-to-source compiler that created Java classes to implement AMS classes and policies along with routines that invoke the Java compiler and the substrate wrapping tool.

To support regeneration of an AMS, in addition to processing multiple specification files, the generator uses the Java classloader to load existing AMS component classes on-demand. Thus, BuildAMS supports both of the previously mentioned alternatives for a representation of the original AMS to extend or change. Any classes that are to be changed, however, must have their entire specification (including policies to be bound) provided because BuildAMS does not support in-place modification of already compiled Java classes.

5. CASE STUDIES

Specific applications in three contexts were chosen for study.

—In a mixed human and tool context, a process execution environment was chosen. In the Little-JIL process execution environment, called Juliette [Wise et al. 1998], various human and tool agents interact to execute a preprogrammed process. The agents that must be supported include a variety of human execution agents, a resource management tool, various COTS tools, and the code for the Little-JIL program interpreter itself.

—In a human-centric context, a laboratory worker coordination application was chosen. This application is intended to facilitate collaboration of human agents. The application is required to store both personal agenda information and information for groups of people and to support meeting scheduling activities. This application, called LACE, has more stringent human user interface requirements than the other two.

—In a tool-centric context, an image processing application that uses an agenda management system to implement a distributed worklist algorithm for a coarse-grained parallel program was chosen. This kind of application requires frequent communication between distributed agents via the worklist and has somewhat more stringent performance requirements than the other two applications. The image processing application is not intended to be a communication-intensive parallel application; we have chosen this application to ensure that the GRAAMS approach works in a tool-centric environment, not to advocate the use of an AMS as a vehicle for high performance parallel computing.

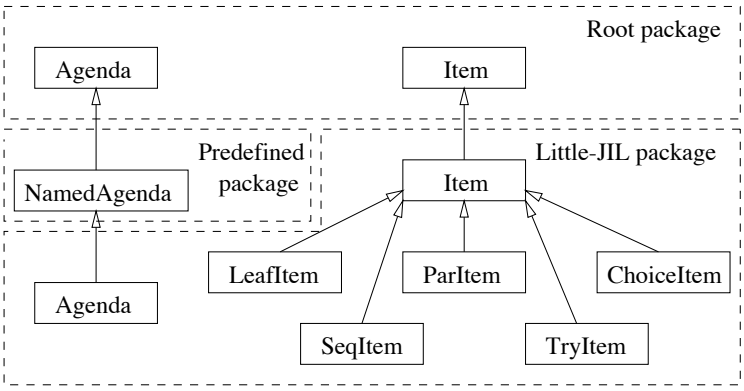


Fig. 16. Inheritance hierarchy of interpreter AMS classes

5.1 Little-JIL

Little-JIL is a language that is used to model and codify processes. The development of Little-JIL has focused on allowing processes to be represented so they may be reasoned about and automatically executed. Little-JIL is sometimes referred to as an agent coordination language because the central focus of programming in Little-JIL is capturing coordination in a process as a hierarchy of steps that are executed by agents. The step hierarchy forms a tree whose leaves represent the smallest specified units of work and whose internal structure represents the way in which this work will be coordinated.

As process programs execute, each step goes through several states. A step transitions into the *posted* state when it is assigned to an execution agent. Its state is later set to *started* by the agent. Eventually the step's state is either set to *completed*, in which case its execution is considered successful, or its state is set to *terminated*, in which case the execution is considered unsuccessful and the step generates an exception. Many other states exist (see [Wise et al. 1998]), but a full description of all states is outside the scope of this paper.

The architecture of the Little-JIL execution environment contains five principal kinds of component: execution agents, the Little-JIL interpreter, a resource manager, an object manager, and a communication system that allows all components to interact. Together, these components support the execution of a Little-JIL process program. For example, if an agent requests that a step be started, the communication mechanism would notify the interpreter of the change of state of the appropriate step instance. The semantics of Little-JIL dictate that resources for the step be acquired at that time, so the interpreter would use the communication mechanism to acquire resources from the resource manager.

The agent communication component is implemented as an agenda management system, hereafter referred to as the Little-JIL AMS. The Little-JIL AMS has two parts, one that defines data that are used for communication between the interpreter, humans, and process-specific software execution agents, and one that defines data that are used for communication between the resource manager and interpreter.

The interpreter part of the Little-JIL AMS defines the data that are used by the Little-JIL interpreter to keep track of step execution. In the Little-JIL execution environment, step execution assignments are made by placing agenda items on an agenda that is monitored by one or more execution agents. This part of the Little-JIL AMS has five types of agenda items: one item type corresponds to each of the four Little-JIL step kinds, and one item type, LeafItem, corresponds to a process step at its lowest level of decomposition. Each Little-JIL agenda item has many attributes, including step name, execution agent, current status, log, step instance parameters, throwable exceptions, and interpreter. Figure 16 shows the inheritance hierarchy of the items.

Attributes of an agenda are used to store an agenda's name and the agenda owner's user identifier. Attributes of an item are used to store a variety of information. A status attribute is used to allow interaction between interpreter instances and the step's execution agent. A log attribute stores auditing information that is maintained through use of the logging policy. An agent attribute stores the name of the execution agent. A ResourceIDs attribute is used

to maintain resource allocation information as the step is executed. An interpreter attribute is provided because, as illustrated below, it was deemed important that the Little-JIL interpreter be distributed, with each step having its own interpreter instance. Handlers, messages, reactions, pre- and post-requisite, parameters, exceptions and thrown exceptions attributes are all used to store semantic information as the process step executes. In addition, a subitems attribute is associated with two subclasses of "Item" to store a parent step's subitems (for the Parallel and Sequential step kinds), and an alternatives attribute is associated with two subclasses of "Item" to store a parent step's alternatives (for the Try and Choice step kinds). Basically, the process program execution state is stored within the AMS, in a way that is similar to the ProcessWall [Heimbigner 1992].

The Little-JIL item types use several policies. Two predefined policies are used to log changes to the name and status attributes. In addition, two Little-JIL language-specific policies have been written and used. The first disallows changing the name of an agenda to one that already exists and the second disallows illegal state changes. This latter policy allows the status attribute's value to be set in ways that correspond only with legal interpreter state transitions. It was created to disallow human users from using the human user interface to set items to illegal states and otherwise interfere with process execution.

An agent typically monitors (i.e., registers to receive events about) one or more agendas to receive tasks (steps) to perform. Multiple agendas are used because an agent may frequently be involved in several disjoint processes or acting in different roles that are logically disjoint. When an item is added to an agenda that an agent is monitoring, the agent is notified by the AMS event mechanism that the agenda has changed. In the case of a human agent, for example, this results in a new item appearing in the person's agenda view window, and in the case of a tool, a known method is invoked to deliver a message describing the change. The agent is then responsible for understanding the item and performing the appropriate task(s). Agents may also monitor items individually by registering to receive AMS events generated by the items. This monitoring feature gives agents the ability to post an item to any agenda and to know, for example, when that item's status subsequently changes.

These mechanisms, provided by the Little-JIL AMS, are sufficient for the Little-JIL interpreter to instantiate and execute multi-agent Little-JIL process programs. When a new step of the process program is to be executed, the interpreter identifies the appropriate execution agent (with the help of a resource management system), creates an appropriately typed agenda item for that step, and posts it on the identified execution agent's agenda. As the step is executed by the agent, its updated status is reflected in the agenda item's status attribute value. The interpreter monitors the status, and as the status changes the interpreter accordingly creates and posts substeps, completes the step and returns "out" parameters, terminates the step and propagates exceptions, and so on. The interpreter for the step's parent also monitors the step, taking appropriate action when it completes or terminates. The Little-JIL AMS thus provides agenda communication facilities that allow the work of agents to be coordinated, while the interpreter encodes the specific coordination semantics of the Little-JIL language itself. This design decouples concerns about why and when coordination should occur from concerns about how coordination should occur.

The second part of the Little-JIL AMS defines the data that are used for communication with the resource manager. The resource manager maintains a resource model that is customized for a process or organization, keeping track of the state of resources as the process executes. The primary operations of the resource manager are to identify resources (in two different ways), to acquire resources, and to release resources. To integrate the resource manager with the interpreter, a different type of agenda and several new item types were defined. An item type is defined for each of the primary operations. As interpreter instances execute a process program, they interact with the resource manager via its agenda to maintain the resource model.

The inheritance hierarchy of the resource manager agenda item types is shown in Figure 17. Agenda item types for each of the primary resource manager operations have been specified. The superclass of all resource management request items, RMRequestItem, has attributes to allow naming, status tracking, and recording of resource identifiers for the request. Identification and acquisition subclasses add a specification attribute that allows further specification of the details of the request. The RMIdentifyAllRequestItem adds a result field used to determine the output of the resource request and RMAcquireRequestItem adds a field that is used to return the identifier of an acquired resource.

The resource manager agenda types used no policies because there were no constraints on the data and because interactions between the resource manager and the interpreter are relatively straightforward.

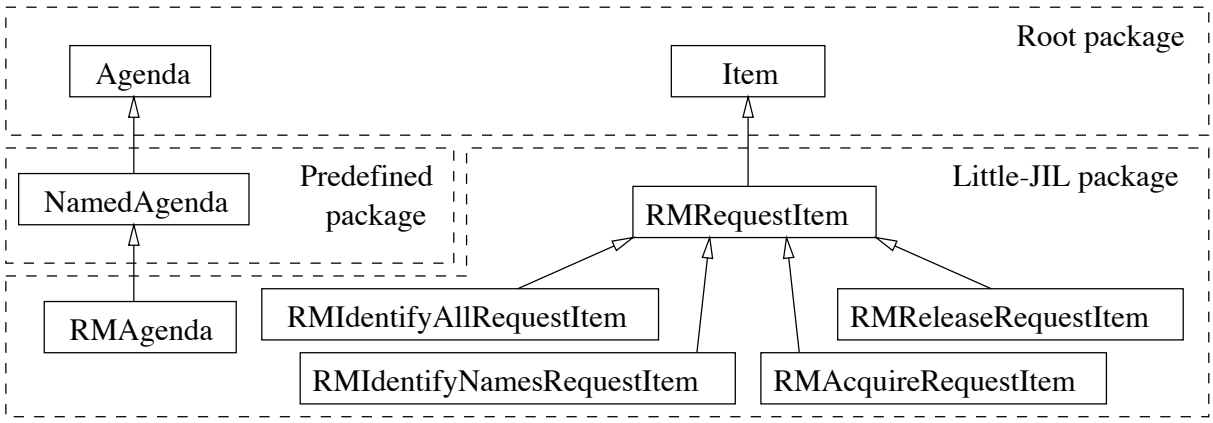


Fig. 17. Inheritance hierarchy of resource manager AMS classes

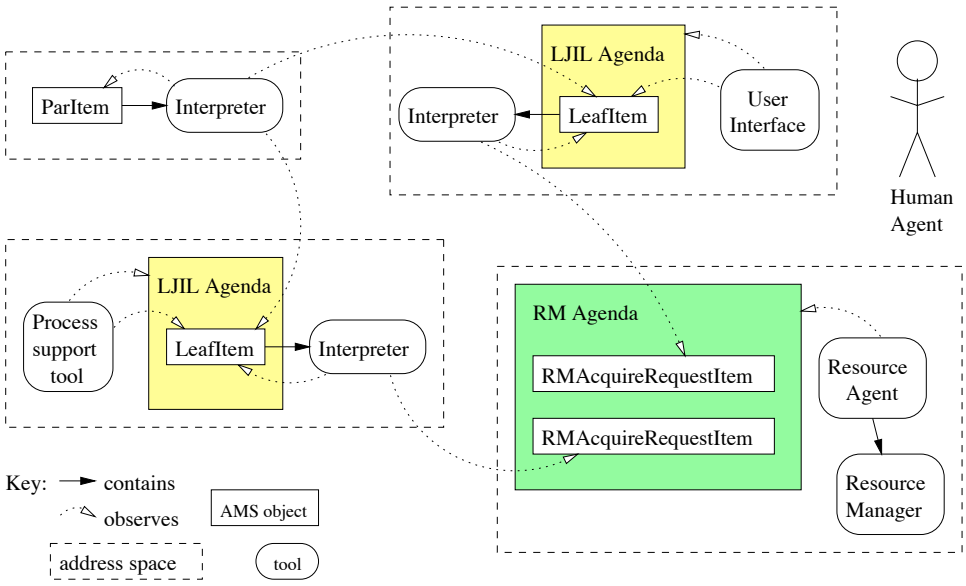


Fig. 18. Example of Little-JIL AMS during execution of a process

The resource manager and interpreter instances interact via the RMAgenda and its items as the execution agents and interpreter modify the state of a Little-JIL step instance's item. When an interpreter instance must identify, acquire, or release a resource, it creates an appropriate agenda item, registers to receive events from the item, and places it on the resource manager's agenda. Because the resource manager was implemented as a monolithic, separate component, it is integrated with the Little-JIL execution environment via a small wrapper called ResourceAgent. There is only one instance of the RMAgenda class in a given process execution environment; the ResourceAgent wrapper monitors that agenda for work. When an item is added to the RMAgenda, the ResourceAgent is notified of the change and responds by invoking the resource manager to take the requested action. The ResourceAgent returns the results from the resource manager by setting attributes of the item appropriately; these changes are seen by the interpreter that is monitoring the item.

The diagram shown in Figure 18 presents a snapshot of the interactions between the various components that have been previously described as a process program fragment is executed. Dotted arrows indicate "observes" relations between objects in which the source of the arrow has registered with the destination to receive AMS

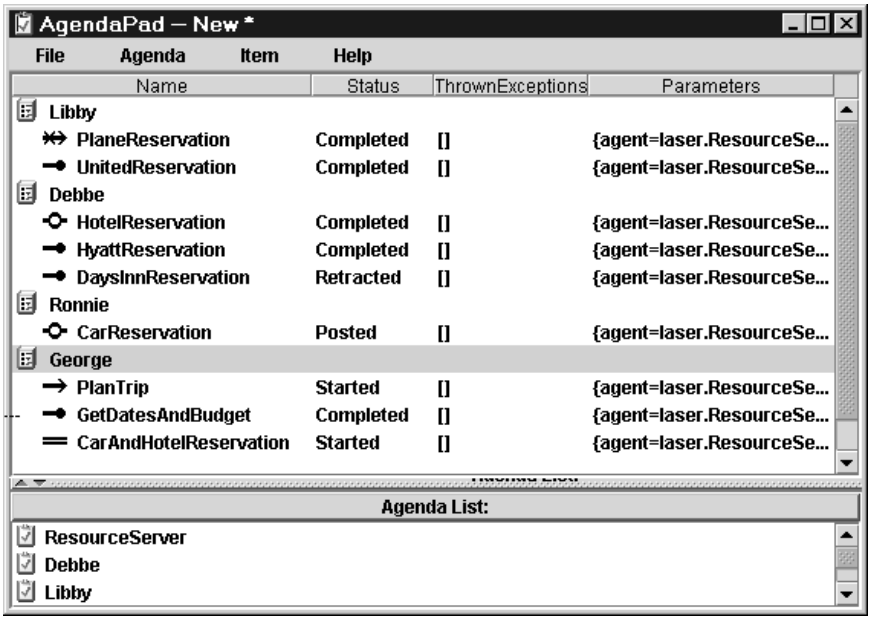


Fig. 19. Human user interface

events. Solid lines indicate a contains relation. Each step contains an instance of the interpreter that has registered to receive events from the step. Four separate address spaces (typically one in each of four different computers) are indicated by dashed boxes and contain the elements involved in the execution of a process step represented by the Little-JIL agenda item in the upper left.

Assume the step is a parallel step with two substeps and all have been previously started. To execute the step, the step's interpreter has created agenda items to correspond to the parallel step's two substeps and acquired agents to execute the steps. It registered to receive events from the substeps and placed them on the agendas for the acquired execution agents. As part of step creation, new interpreter instances are created for each substep; these interpreters also register to receive events. When the items are added to the agendas of the execution agents, events are generated and sent to the observers of the agendas, namely the human user interface for the human agent's agenda and the process support tool for its agenda, which also registers with the items to receive AMS events.

The figure shows the state of the process shortly after both agents have changed the value of the status attribute of their agenda items to "starting," indicated in the figure by the presence of resource requests. The interpreters associated with each item have received an event indicating that the item's status attribute value has changed. They have responded by interpreting the state change. The first part of step execution is to acquire the resources required by the step; to do this, each interpreter has created a resource request item and placed it on the resource agent's agenda. At this point, the resource agent will receive two events from its agenda, telling it that new request items have arrived. It will then service the requests and set the status attribute of the request items appropriately. At that point, the interpreter observing each request item will receive an event telling it the new state of the item, and the interpreter will respond by continuing or aborting step execution, depending on whether resources were successfully acquired.

As seen in the above example, different agents interact with the AMS and consequently with the running Little-JIL process via customized agent interfaces. For humans, this interface may be a GUI that would be used to change an item's status, signal exceptions, change parameters, etc. For COTS tools, this interface may be a wrapper agent that integrates the tool with the AMS, spawning the tool to perform tasks in response to agenda items being posted to the tool's agenda and reporting the results of tool execution by setting agenda item attributes (e.g., parameters, status) as required. This is, in fact, the way in which the resource manager is integrated with the process execution environment.

The human user interface shown in Figure 19 is AgendaPad. AgendaPad allows items and agendas to be viewed and edited. A version of AgendaPad also contains menu shortcuts for common tasks such as starting and terminating steps and has application-specific logic built in that permits execution of Little-JIL processes. For example, when a human user terminates a step, AgendaPad asks which exception is to be thrown by that step.

The Little-JIL interpreter and various incarnations of the underlying AMS have seen daily use for several years. A number of complex process programs have been successfully executed by the interpreter, including sections of a multi-user Booch Object-oriented design (BOOD) process, a trip planning process, a process to coordinate bug tracking, and several other processes that have been used to test interpreter logic and to demonstrate the executability of Little-JIL process programs.

The total specification length of the Little-JIL AMS is 271 lines, producing 2,312 lines of generated, wrapped code. The AgendaPad user interface adds another 3,708 lines, the Root and predefined classes add about 834 and 900 lines, the ObjectCache system adds 4,951 lines, the language interpreter adds 5,300 lines (before being wrapped for use with ObjectCache), the language classes add 3,354 lines, the base tool agent classes add 166 lines, and the resource manager adds 4,997 lines, for a total application size of around 27,000 lines of code. The parts that are written by the application developer are the language and interpreter, the base agent classes, the resource manager, and the user interface. These parts total 17,630 lines, or about 65%, of the total application size.

Of Grapevine's features, only attribute collections, predefined attribute collections and predefined items were not used. Attribute inheritance was used most frequently. Two new policies were defined, and two predefined policies were used.

Overall, our measurements and experiences with Juliette support our belief that an agenda management system provides an appropriate metaphor for coordinating interaction in mixed-agent processes. User surveys indicated that use of the AMS in this application met our requirements and that the underlying AMS abstractions were appropriate, and measurements of time and length indicated that the use of Grapevine reduced the effort required to create and maintain the Little-JIL execution environment.

5.2 LACE

This section describes our experiences developing a groupware application called LACE, an acronym that stands for Laboratory Coordination Environment. The design of LACE is for a fully-featured groupware application intended to provide experimental data about a full-scale, human-centric application. Due to resource constraints, only the meeting scheduling and delegation functionality of LACE was implemented. These parts were chosen to allow further exploration of three major issues of the agenda management approach, namely policies, user interface issues, and support for evolution. The selected functionality requires simple user group support, start and end dates for meeting repetition, meeting delegation, simultaneous viewing of multiple schedules, meeting quorums and automatic cancellation, and the ability to automatically find free blocks of time in the schedules of arbitrary combinations of users.

The LACE user interface is shown in Figure 20. It was a clean-slate implementation that reused no existing AMS UI components. The user interface was relatively easy to implement, thanks to the fact that the LACE AMS was designed to work according to the the Model-View-Controller user interface design pattern [Gamma et al. 1995].

Typical use of the LACE user interface consisted of having the user "log in," as shown in the upper left foreground of the figure. The user is then presented with a view of the calendar that summarizes a set of schedules the user has elected to view. A meeting appears in red or green, depending on whether its quorum has been reached. Double-clicking on a day brings up a meeting edit window, shown in the lower right of Figure 20. Pull-down menu choices allow a user to delegate meetings to other people and to find free times for scheduling new meetings.

The inheritance hierarchy for the classes that were defined for the LACE AMS are shown in Figure 21. Attributes were declared for storing name, owner, duration, etc.

The instantiated lab coordination AMS was deployed in the Laboratory for Advanced Software Engineering Research of the University of Massachusetts. The user interface and AMS for LACE were deployed and used by about nine people in the lab for ten days before a user survey was distributed and results were collected. Of these people, six responded to the survey. Most users used LACE to manage their class and meeting schedules, which included group meetings, and LACE was successfully used to locate a free meeting time and schedule a meeting

Fig. 20. The LACE user interface

for a set of users without requiring individual user interactions.

The LACE specification length is 351 lines of code, producing 2,456 lines of generated, wrapped code. The LACE UI adds 2,447 lines of code to that total, the ObjectCache system adds 4,951 lines, the Root classes adds 834, and the predefined classes that were used add about 800 lines, for a total application size of roughly 11,500 lines of Java. The user interface portion plus the LACE specification file (351 lines), the only parts that are written by the application developer, form only about 24% of the total application size.

As with the Little-JIL AMS, attribute inheritance and predefined attributes were the most heavily used features of Grapevine. Policies were used more than in the Little-JIL AMS, with three new policy definitions and four policy bindings. During development, the LACE AMS was frequently iteratively regenerated as the Meeting class's method bodies and policies were changed and tested.

Based on user survey results and our own observations, we have concluded that the GRAAMS approach can be successful in supporting the communication needs in an application with only human agents, though stronger evolution experience would butress our conclusion.

5.3 Image processing

An image processing application was selected to serve as the third case study because it is a useful tool-centric application. The application and the agenda management system were designed by several graduate students who were not directly affiliated with this research project. The implementation and design details presented here are derived from inspection of the source code of the application and from conversations with some of the authors of

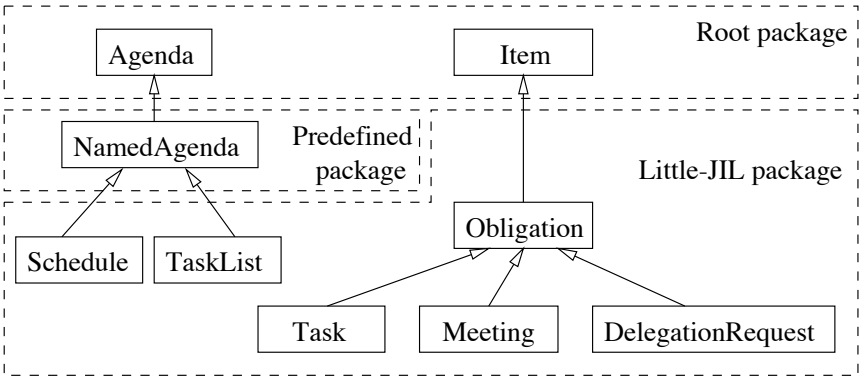


Fig. 21. Inheritance hierarchy of LACE classes

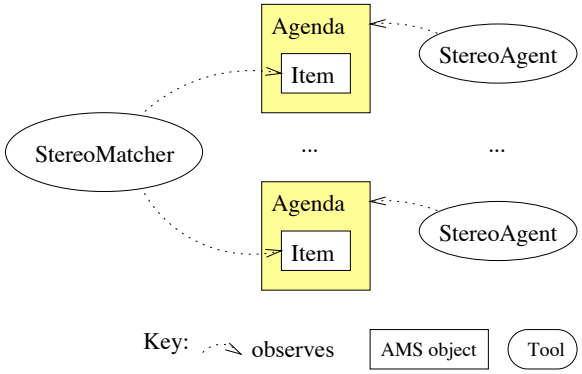


Fig. 22. High level design of image processing application

the application.

The image processing application used an agenda management system to parallelize a stereo matching problem. The high-level architecture of the application is similar to a worklist architecture, in which one "master" process assigns work to many "slave" processes. The architecture appears in Figure 22. The "StereoMatcher" acts as the master; it extends a more general class, "Parallelizer," that can be used to implement many data-parallel algorithms. It divides the work up into many pieces, finds agendas for all the "StereoAgents" in the system, and loops over the list of agendas putting an agenda item that corresponds to one piece of work on each agent's agenda. Each StereoAgent observes its agenda. When an item is added to its agenda, it accesses that item and uses its attributes to calculate a correlation, which it then saves back into the item and sets the status to "Completed." The "StereoMatcher" object monitors each item; when the status is set to "Completed," it reads the result out of the item and merges it with the other results it has received. When the last result has been received, it writes out the calculated disparity (between the stereo pairs) to a file, prints status information, then exits.

The inheritance hierarchy of the items and agendas defined for the stereo matching application appear in Figure 23. Though the agenda subclass has no attributes, the `itemclass` keyword was used to prevent items of the wrong type from being stored on the agenda. Although the root class of agenda could have been used for this application, it would not have allowed any type checking of items. The item subclass contains five attributes. The status attribute is used to store the current state of processing, in much the same way as the status attribute of the Little-JIL AMS. It disciplines the communication between the StereoMatcher master and the StereoAgent slaves. Two attributes contain a string filename that stores the filenames of two images that are to be matched. Correlation and disparity attributes are used to communicate match results between master and slave.

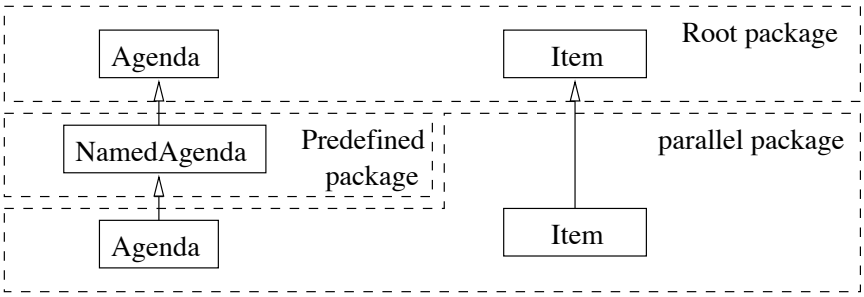


Fig. 23. Inheritance hierarchy of image processing application classes

Specification	compiler	substrate	user time	system time	elapsed time
Predefined	javac	wrap	4.59	0.50	05.25
	jikes	wrap	0.73	0.18	01.01
	jikes	nowrap	0.48	0.12	00.60
	none	nowrap	0.31	0.11	00.41
Little-JIL	javac	wrap	13.33	0.63	14.27
	jikes	wrap	1.75	0.29	02.29
	jikes	nowrap	0.81	0.13	00.94
	none	nowrap	0.61	0.06	00.66
LACE	javac	wrap	12.85	0.61	13.88
	jikes	wrap	2.00	0.27	02.37
	jikes	nowrap	0.87	0.16	01.02
	none	nowrap	0.70	0.06	00.76
Parallel	javac	wrap	4.71	0.62	05.77
	jikes	wrap	0.77	0.26	01.13
	jikes	nowrap	0.45	0.17	00.61
	none	nowrap	0.35	0.07	00.42

Table 5. Raw AMS generation times in seconds.

The stereo matching application was used to successfully parallelize execution on up to six computers. The specification length is 33 lines of code, producing 412 lines of generated, wrapped code. The total size of the package is 996 lines. The ObjectCache system adds 4,951 lines, the Root classes add 834, and the predefined classes that were used add about 350 lines, for a total application size of roughly 7,000 lines of Java source code. The tool classes and AMS specification file (617 lines total), the only parts that are written by the application developer, form only about 9% of the total application size. Note that the actual application logic is implemented in relatively few lines of code.

The parallel AMS was not regenerated since it was created, and users did not report making customizations. The results of a survey given to the AMS designer (and principal user) of the application indicate that Grapevine successfully supported the communication needs of an application whose only agents are tools.

5.4 Evaluation Summary

Timing information for generation of the three case studies is presented in Table 5. All timing measurements were made on a 450 MHz AMD K6-2 based computer with 128 megabytes of PC100 SDRAM and an IBM ATA-33 IDE disk running Linux kernel 2.2.5 compiled with Pentium optimizations. Blackdown's x86 port of Sun's JDK version 1.1.7.1a was used. No JIT compiler was used when collecting timing information, and no compiler optimizations were used when compiling BuildAMS and WrapperGen. An alternate set of numbers shows that when IBM's replacement java compiler, "Jikes", was used instead of the Sun compiler, generation time was reduced dramatically. This indicates that the generation time using "javac" is dominated by the speed of the

Case Study	spec length	generated code	ratio	total app length	percent
Little-JIL	271	2,312	8.53	27,000	35%
LACE	351	2,456	7.00	11,500	76%
Image processing	33	412	12.48	7,000	91%

Table 6. Summary of quantitative measures (in lines of code). Ratio is generated LOC to specification LOC; percent is percent of the application that was provided by Grapevine (includes substrate code).

Java compiler.[6] Times were measured using standard Unix "time" command. The "none - nowrap" lines give the time for generating Java source code without any other postprocessing. The "jikes - wrap" lines give the best times for producing a ready-to-go AMS and are under three seconds in every case, indicating that the generator has met its performance objective.

Table 6 summarizes the quantitative measures that were made on the three case study applications. In the three applications, the designer wrote 65%, 24%, and 9% of the total application, measured in lines of source code. While LACE and the stereo matching application are quite centered around communication, Little-JIL represents an application that has a considerable amount of code devoted to activites unrelated to communication. Even in this case, a large percentage of the final application has been generated or reused, representing a significant savings in implementation and maintenance cost.

The generated agenda management systems were able to meet a majority of the requirements presented in Table 1. Though a few requirements, such as scalability, are untested, we are confident that by extending the substrate, predefined classes and policies, and perhaps refining the policy mechanism, we would be able to meet them all. Most reassuring of all, we found that the people who wrote tools for, or directly used, the generated AMSs found them appropriate and natural to use, indicating that the agenda metaphor is a valid and strong one.

In the remainder of this section, we highlight some experiences we had with specific features of Grapevine.

5.4.1 *Experiences with using instantiated AMSs*. The use of Grapevine seemed particularly appropriate for allowing the Little-JIL resource manager to communicate with the interpreter. In this case, the Little-JIL AMS is used as middleware, but because resource requests are posted to an agenda, there is more indirection than is typically seen in traditional distributed computing systems because requests are translated into method calls by a third party, the resource agent. This indirection allows the agent to potentially defer processing of requests, allowing better resource utilization.

In spite of this success, using an agenda management system for all tool-to-tool communication does not always seem appropriate. In an early version of Juliette, an agent used the agenda management system to dynamically change the resource model. The API that allowed changing the resource model was quite rich and fine-grained, so allowing access to all of the resource model's functionality required either creating an agenda item per method-invocation request, or creating one "procedure call" agenda item with an attribute to identify which resource model function should be used. Both of these alternatives were cumbersome to use. In this case it appears that the traditional, sequential procedure call model of communication was the appropriate one to use. The sentiment that using an AMS for some synchronous tool to tool communication is clumsy was also expressed by the user of the image processing application.

We also found that user interfaces tended to be more useful and usable when they were more application-specific. When a UI is designed for an application, it is able to encode many of the semantics of the application data into behavior of the user interface. For example, the LACE UI does not display the status or time of a meeting directly, but instead draws appropriately colored meetings in different dates to communicate this information more efficiently.

The generic AgendaPad human user interface was originally designed to allow viewing and editing of Little-JIL's interpreter's Agenda objects. Since it uses introspection to discover the attributes of items and agendas, RMAgenda objects and other Agenda subclasses can also be viewed and edited with it. This support allowed human monitoring

[6]The Java compiler is invoked twice when generating an AMS, once by the generator to compile generated AMS classes, and once by the substrate tool when generated classes are converted for use in a distributed environment.

of the resource manager's agenda, which we have found to aid in debugging of processes, in process monitoring, and in other areas. Thus, while a generic user interface appears too clumsy for use by end-users in a production application, it appears quite useful for application development and maintenance.

As Figure 18 has indicated, we relied heavily on the ability to "observe" agenda items and agendas, both when developing human user interfaces and when developing agents. In Juliette, it has become an established rule that agents (both human and tool) should change the value of the "status" attribute only after they have made all desired changes to other attributes. Although agent synchronization via "setStatus" convention has worked well, unenforced conventions should be avoided because they can make software maintenance more difficult and make the learning curve steeper for new users. While the substrate we used lacks a high-level transaction mechanism, it does provide `lock()` and `unlock()` methods that allow a number of methods to be invoked on an object atomically. These methods could be used to allow multiple method invocations by an agent to occur atomically. Object locking was used in the LACE calendar user interface.

5.4.2 *Experiences with Grapevine's extension mechanisms*. The policy mechanism proved adept at handling the quorum feature of LACE. The `checkQuorum` policy contained four constraint and action pairs (ECA tuples), two to cancel a meeting if there were not enough participants (one for an integer parameter and one for a Vector parameter), and two to confirm a meeting if there are enough participants(one for an integer parameter and one for a Vector parameter). It is possible to write a single ECA tuple to handle both conditions, but we felt its function would then be less clear.

It is interesting to trace through an execution of the LACE UI when a user's decision not to attend a meeting causes the meeting to be automatically canceled. When the user clicks the "Change status" button then selects "OK", the LACE user interface invokes each attribute's set method. When the UI invokes the `setParticipants` method, the policy will be run. The policy checks the quorum and the number of participants; the condition part of the ECA tuple is false, so the action is taken. The action simply calls `setMeetingStatus`, setting its value to "Canceled". When these attributes are changed, events are sent out to their observers, i.e., the LACE user interfaces that are currently viewing the Meeting. In response to these events, the LACE UI redraws the Meeting name in red, showing that the meeting has been canceled. We see from this example how the policy mechanism can work quite well with the Model-View-Controller user interface design pattern, via AMS events.

But, we did encounter some problems with the policy definition mechanism. For example, we encountered difficulty with the implementation of the policy in the Little-JIL AMS that was intended to prevent a user from changing the name of an agenda to a name that is already used. We would like the policy to prohibit agendas from having the same name, but we cannot have the policy's action throw an exception because that would require a change in the method's signature. The best we can do with the current policy mechanism is to have illegal name changes silently fail, requiring the user interface to try to set the agenda name, then check to see if the name was actually set. If it was not set, the user interface could report that fact, but it cannot report the reason. This problem also appeared in the LACE user interface code for editing the start and end time of meetings and made its implementation somewhat more clumsy than it should have been. One workaround for this problem is to have the policy throw `RuntimeException`s, exceptions that do not have to be declared in Java, but, if uncaught, policy failure could cause the user interface to abruptly exit, a condition we viewed as the greater of two evils.

We found that method extensions were used quite heavily in all three applications. The most complex designer-specified method is `isDuringPeriodOf()` in LACE, a method which returns a boolean value indicating whether or not a meeting occurred during a particular period of time. This method is used by the viewer to calculate which meetings should be shown on each date in the month view and to calculate which meetings interfere when finding free times. A substantial portion of the AMS specification was devoted to the implementation of this method.

5.4.3 *Experiences with iterative development and evolution*. We found that development of LACE's AMS alternated between refining the AMS specification and refining the user interface. An attribute or method would be added to the Meeting class or another class, then the user interface would be augmented to take advantage of it. The ability to iteratively regenerate part of an AMS was found to be very useful for this sort of development, allowing in this case the Meeting item to be regenerated and tested without having to regenerate the rest of the AMS.

Juliette was developed in tandem with Grapevine, so it is difficult to buttress claims about Grapevine's support for evolution with that case study. During its development, however, there have been concrete examples of all three kinds of evolution support being used, from adding policies or attributes to alter the behavior or structure of the AMS, to customizing a user interface to meet a particular person's requirements.

In spite of some encouraging experience, a weak point in the case studies has been in the evaluation of Grapevine's support for evolution. This weakness results from the limited duration and scope of the studies. Although less evolution was required for these applications than expected, our qualitative experience has indicated that Grapevine does indeed support a variety of interesting and useful changes.

6. CONCLUSION

Modern applications are increasingly being seen as collaborations among groups of humans and software systems. We see agendas as an appropriate abstract concept for meeting the communication requirements of both, and believe that by supporting the creation of application-specific agenda management systems we can aid the development of a wide range of applications.

The specific research goal was to determine whether the triad of generation, regeneration, and adaptation mechanisms as described herein can be used to meet the goals of creation and evolution of specific AMSs that support interoperability of humans and tools in a variety of applications that vary in their human-to-tool ratio. Three applications with a varying human-to-tool ratio were built using this approach. Our experiences indicate that the generation of agenda management systems for each application was cost-effective and that the produced agenda management systems were useful. Though there was less conclusive evidence obtained, our experiences also indicate that generated agenda management systems are evolvable, especially via regeneration of selected portions of an AMS specification. We believe that agendas are an appropriate metaphor for asynchronous agent communication, though they might not be appropriate for synchronous communication. For these reasons, we advocate the use of the GRAAMS approach in building applications that require heterogeneous agents to communicate asynchronously.

This work has raised a variety of issues, many of which deserve further exploration. Four important directions in which this work could be extended are improvements to the policy mechanism, an agenda management system design tool, specific additions to the functionality of generated agenda management systems, and further evaluation.

The policy mechanism does not permit the enforcement of a policy to change the signature of a method. The reason this design decision was made was to allow a designer to change the behavior of an AMS without having to also change all client code. For example, a designer might decide to regenerate an AMS so that all changes that are made to the status attribute of an agenda item are logged. In that case, only the subitem class would have to be regenerated; the signature of the `setStatus` method would remain the same, and the user interfaces and agents could be used without being rewritten or even recompiled. The trouble with the current approach is that there is no clean way for a designer to require client code to respond specifically to a policy failure. As we saw with the "UniqueNames" policy in Little-JIL, the designer might want this policy to cause the method on which it was enforced to throw an exception when the condition fails, thus changing the signature of the method. We might want to extend Grapevine to support two kinds of policies, ones that only affect an enforced method's behavior and ones that affect its signature too.

Another way in which the policy mechanism could be improved is by adding support for dynamic policy enforcement. While it is not clear that policies can be used to represent data consistency constraints in general, allowing policies to be enabled and disabled by applications would certainly increase their usefulness.

Another idiosyncrasy of the policy mechanism is the inability of policies to use method return values in actions or even in postcondition evaluation. An easy way of improving the policy mechanism in this respect would be to do away with the ECA tuple structure of policies, replacing it with a more general syntax for wrapping method invocations (such techniques are sometimes referred to as "interceptors" or "function chains"). A keyword that stands for invocation of the original method would be introduced and used within the body of the policy. This would allow pre- and post-conditions to share variables and have access to return values and exceptions.

Our experiences with the policy mechanism have indicated that policies are useful for changing the behavior of a method but are not generally useful for enforcing data consistency ("well-formedness") constraints in an

application. In many instances, when we have tried to use the policy mechanism in this way, it has been insufficient. Enforcing data consistency appears to be a higher-level problem, and a better mechanism than the current policy mechanism seems to be required to express, as well as enforce, an application's consistency requirements (see e.g. [Tarr and Clarke 1998]).

We initially chose to use a textual specification language for agenda management systems. While the textual specification is adequate for the applications implemented, a graphical tool for specifying agenda management systems could further ease application development. Presenting a designer with a drag and drop graphical user interface could make the use of the inheritance mechanism much easier. If a wide variety of useful predefined policies was available, for example, a designer could simply select those that are apply to each method. Many commercial software development environments include tools for rapid GUI construction. It should be possible to create such a tool that would make AMSs easier to construct and encourage more software developers to use AMSs in application development.

Currently, for any given AMS that is generated, only a generic user interface is provided. If this interface is not appropriate for a particular application, it must be customized by hand or replaced. It may prove useful to automate the creation of custom user interfaces by adding user interface customization keywords to the specification language or by providing other customization tools.

The agenda management systems we have created so far have generally met the requirements of the applications we wanted to build. During development we have occasionally added to the functionality of the generator and the agenda management systems to support certain application requirements, such as persistence and an improved event model. We expect this trend to continue, and there are already a number of directions in which we believe AMS functionality should be increased.

We have previously commented that our experiences have supported our intuition that the agenda metaphor is not appropriate for all kinds of communication. When communication between agents is synchronous the AMS seems to introduce unnecessary indirection and does not help in connecting a response to a request. Future work might explore ways in which the agenda metaphor can be augmented to support a synchronous communication model.

The AMS API could be augmented to support requirements that are likely to be required by production applications. Three such requirements are transactional access to data, associative access to data, and security. The substrate and agenda management system API should be augmented to allow applications to perform complex functions as atomic transactions. An AMS that provided the ACID properties is likely to be much more useful in production environments than the currently generated AMSs. Associative access to data would allow applications to use fairly general queries over agenda management system data. For example, if the Little-JIL AMS supported associative access or queries over AMS data, the resource manager could ask for all items on an agenda that came from a particular agent or those that had the highest priority. Supporting more complex queries will probably reduce application implementation complexity because common queries will not have to be implemented separately by each agent that uses them. Security is becoming an increasingly important part of distributed systems. While the policy mechanism appears to provide a convenient way for implementing access control, common security requirements should be met like any other common requirements, perhaps by providing security policies and classes as part of the set of predefined classes and policies.

One limitation of the AMS event mechanism is that listeners may not register to receive only particular kinds of AMS events from an object. This limitation has not been a problem in current applications because listeners tend not to register for events from many objects, but providing filtering of AMS events based on event type or data might be necessary in the future. It is expected that the current Grapevine design would need only minor modifications to support event filtering.

It is also not clear what sort of object consistency model is appropriate for shared collaborative data such as the kind found in an AMS. There has been some previous research in this area [Barghouti 1992], but further study is warranted because AMSs are intended for use by both machines and people in a variety of application areas. Support for disconnected operation to support mobility may soon also be a serious requirement. Because agenda management systems are used by humans, data consistency models in which the agent has more direct control over when consistency actions are taken might be appropriate.

Study of the possible use of versioning in an AMS as part of a data consistency mechanism would also prove interesting. In some systems, distributed objects are versioned as they are modified; this versioning could be used in an AMS to resolve consistency problems and, perhaps, to aid in type evolution.

Finally, more evaluation would provide valuable feedback and help us improve the design and implementation of Grapevine, and our understanding of the strengths and weaknesses of the GRAAMS approach. Thus, although there is more work to be done in these areas, we believe we have demonstrated that the agenda management metaphor is a useful abstraction for supporting asynchronous communication among users and tools. Moreover, we have created a system that allows the generation, regeneration, and adaptation of application-specific agenda management systems that appear to be functional, easy to use, and significantly reduce the implementation effort for a range of applications.

APPENDIX

The BNF grammar for Grapevine's specification language.

Start ::= (<PACKAGE> Name <SEMICOLON>)? (<IMPORT> Name (<SEMICOLON>
| <DOT> <STAR> <SEMICOLON>))* (SubclassDefinition)* <EOF>

SubclassDefinition ::= (<AGENDA> <IDENTIFIER> <EXTENDS> Name "{" AgendaClassExtension "}"
| <ITEM> <IDENTIFIER> <EXTENDS> Name "{" ItemClassExtension "}"
| <COLLECTION> <IDENTIFIER> <EXTENDS> Name "{" CollectionClassExtension "}"
| <ATTRIBUTE> <IDENTIFIER> <EXTENDS> Name "{" AttributeClassExtension "}")
| <POLICY> PolicyDef
| <ENFORCE> PolicyEnforcement

PolicyDef ::= <IDENTIFIER> FormalParameters "{" ((<DECLARE> Type VariableDeclaratorId <SEMICOLON>)
| (<CONSTRAINT> MethodDeclaration <ACTION> MethodDeclaration <CHECK-AS> (<PRECONDITION> | <POSTCONDITION> | <PREPOSTCONDITION> | <PREREQUISITE> | <PREPOSTREQUISITE>) ";"))* "}"

PolicyEnforcement ::= <IDENTIFIER> LiteralParameters "for" Name ("," Name)* ";"

LiteralParameters ::= "(" (LiteralParameter ("," LiteralParameter)*)? ")"

LiteralParameter ::= Name | (<IDENTIFIER> | <INTEGER-LITERAL> | <FLOATING-POINT-LITERAL> | <CHARACTER-LITERAL> | <STRING-LITERAL>) | <NULL> | <FALSE> | <TRUE>

AgendaClassExtension ::= (ItemClassExpression)? MainExtensionBody

CollectionClassExtension ::= (AttributeClassExpression)? (MethodExtension)*

AttributeClassExtension ::= (<TYPE> Type <SEMICOLON>
| <TYPE> <ENUMERATED> StringLiteralList <SEMICOLON>) (MethodExtension)*

ItemClassExtension ::= MainExtensionBody

MainExtensionBody ::= (AttributeExtension | MethodExtension)*

AttributeExtension ::= ClassAttribute

MethodExtension ::= <METHOD> MethodDeclaration

ItemClassExpression ::= <ITEMCLASS> Name ";"

AttributeClassExpression ::= <ATTRIBUTECLASS> <IDENTIFIER> ";"

ClassAttribute ::= (<PRIVATE>)? <ATTRIBUTE> <IDENTIFIER> (<EQUALS> InitialValue)? ";"

```
InitialValue ::=      LiteralParameter | <NEW> Name <LPAREN> <RPAREN>
javaBlock ::=         java code
MethodDeclaration ::= ( "public" | "protected" | "private" | "static" | "abstract" | "final" | "native"
                      | "synchronized" )* ResultType MethodDeclarator ( "throws" NameList )?
                      ( "{" javaBlock "}" | ";" )
MethodDeclarator ::=  <IDENTIFIER> FormalParameters ( "[" "]" )*
FormalParameters ::=  "(" ( FormalParameter ( "," FormalParameter )* )? ")"
FormalParameter ::=   ( "final" )? Type VariableDeclaratorId
Type ::=              ( PrimitiveType | Name ) ( "[" "]" )*
ResultType ::=        "void" | Type
PrimitiveType ::=     "boolean" | "char" | "byte" | "short" | "int" | "long" | "float" | "double"
VariableDeclaratorId ::= <IDENTIFIER> ( "[" "]" )*
Name ::=              <IDENTIFIER> ( "." <IDENTIFIER> )*
NameList ::=          Name ( "," Name )*
StringLiteralList ::= <STRING-LITERAL> ( "," <STRING-LITERAL> )*
```

ACKNOWLEDGMENTS

The authors would like to thank Peri Tarr for early assistance, Jesse Craig for his work on the AgendaPad user interface, and Aaron Cass for participating in the case studies. Special thanks to Sandy Wise and Barb Lerner for countless hours of interesting discussions and assistance.

REFERENCES

APPLE COMPUTER. 1994. OpenDoc: Shaping Tomorrow's Software. White paper (Sept.), Apple Computer, Inc.

BANDINELLI, S., NITTO, E. D., AND FUGGETTA, A. 1996. Supporting Cooperation in the SPADE-1 Environment. *IEEE Transactions on Software Engineering 22*, 12 (Dec.), 841–865.

BARGHOUTI, N. S. 1992. Supporting cooperation in the Marvel process-centered SDE. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software development environments* (1992), pp. 21–31.

BEN-SHAUL, I. Z., KAISER, G. E., AND HEINEMAN, G. T. 1992. An architecture for multi-user software development environments. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software development environments* (1992), pp. 149–158.

BEUGNARD, A., JEAN-MARC JÈZÈQUEL, NOËL PLOUZEAU, AND WATKINS, D. 1999. Making components contract-aware. *IEEE Computer*, 38–44.

BOEHM, B. AND SCHERLIS, W. 1992. Megaprogramming. In *Proceedings of the DARPA Software Technology Conference* (1992), pp. 63–82.

CHABERT, A., GROSSMAN, E., JACKSON, L., PIETROWICZ, S., AND SEGUIN, C. 1998. Java Object-Sharing in Habanero. In *Communications of the ACM* (June 1998).

CORKILL, D. D. 1991. Blackboard systems. *AI Expert 6*, 9 (Sep), 40–47.

CUGOLA, C., DINITTO, E., AND FUGGETTA, A. 1998. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proc. of the 20th International Conference on Software Engineering* (1998), pp. 261 – 270.

DAYAL, U., HSU, M., AND LADIN, R. 1990. Organizing Long-Running Activities with Triggers and Transactions. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* (1990), pp. 204–214.

DOURISH, P. 1995. The Parting of the Ways: Divergence, Data Management and Collaborative Work. In *Proc. European Conference on Computer-Supported Cooperative Work ECSCW '95* (Stockholm, Sweden, Sep 1995).

DOURISH, P. 1996. Consistency Guarantees: Exploiting Application Semantics for Consistency Management in a Collaboration Toolkit. In *Proc. Conference on Computer-Supported Cooperative Work CSCW '96* (Cambridge, MA, 1996).

FERNSTRÖM, C. 1993. PROCESS WEAVER: Adding Process Support to UNIX. In *Proceedings of the Second International Conference on the Software Process* (1993), pp. 12–26. IEEE Computer Society Press.

FLANAGAN, D. 1997. *Java in a Nutshell* (2 ed.)., Chapter 4. O'Reilly & Associates, Inc.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GEORGAKOPOULOS, D., HORNICK, M., AND SHETH, A. 1995. *Distributed and Parallel Databases*, Chapter An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure, pp. 119–153. Kluwer Academic Publishers.

GREENBERG, S. AND MARWOOD, D. 1994. Real-Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. In *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW '94* (Chapel Hill, North Carolina, Oct 1994).

HARRISON, W. AND OSSHER, H. 1993. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA)* (1993), pp. 411–428.

HEIMBIGNER, D. 1992. The ProcessWall: A Process State Server Approach to Process Programming. In *Fifth SIGSOFT Symposium on Software Development Environments* (December 1992).

HOLLINGSWORTH, D. 1994. The workflow reference model. Technical Report TC00-1003 (Nov), Workflow Management Coalition. Draft 1.0.

INTERNATIONAL BUSINESS MACHINES CORPORATION. 1995. *MQSeries: An Introduction to Messaging and Queueing* (Second ed.). Number GC33-0805-01. International Business Machines Corporation.

JAVASOFT. Infobus. http://java.sun.com/beans/infobus.

KAPLAN, S. J., KAPOR, M. D., BELOVE, E. J., LANDSMAN, R. A., AND DRAKE, T. R. 1990. Agenda: a personal information manager. *Communications of the ACM 33*, 7 (Jul), 105–116.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming* (1997). Springer-Verlag.

KIRTLAND, M. 1997. The COM+ Programming Model Makes it Easy to Write Components in Any Language. *Microsoft Systems Journal*.

KRISHNAKUMAR, N. AND SHETH, A. 1995. Managing heterogeneous multi-system tasks to support enterprise-wise operations. *Distributed and Parallel Databases*.

LAI, K.-Y., MALONE, T. W., AND YU, K.-C. 1988. Object Lens: A "Spreadsheet" for Cooperative Work. *ACM Transactions on Office Information Systems 6*, 4 (Oct), 332–353.

LERNER, B. S. 1997. TESS: Automated Support for the Evolution of Persistent Types. In *Proceedings of the 12th Automated Software Engineering Conference* (1997).

MALONE, T. W., LAI, K.-Y., AND FRY, C. 1992. Experiments with Oval: A Radically Tailorable Tool for Cooperative Work. In *CSCW '92 Proceedings* (Nov. 1992), pp. 289–297.

MARSHAK, R. T. 1997. Inconcert workflow. *Patricia Seybold's Workgroup Computing Report*.

MAYBEE, M., HEIMBIGNER, D., AND OSTERWEIL, L. 1996. Multilanguage interoperability in distributed systems. In *Proceedings of the 18th International Conference on Software Engineering* (Berlin, Germany, March 1996), pp. 451–463.

MCCALL, E. K., CLARKE, L. A., AND OSTERWEIL, L. J. 1998. An Adaptable Generation Approach to Agenda Management. In *Proc. of the 20th International Conference on Software Engineering* (1998), pp. 282 – 291.

MUNSON, J. P. AND DEWAN, P. 1997. Sync: A java framework for mobile collaborative applications. In *Computer* (Jun 1997), pp. 59–66. IEEE Press.

NII, H. P. 1989. Blackboard systems. In A. BARR, P. COHEN, AND E. FEIGENBAUM Eds., *Handbook of Artificial Intelligence*, Volume 4, pp. 1–82. Addison-Wesley. Revised version of 1986 AI Magazine article.

OBJECT MANAGEMENT GROUP. 1996. *The Common Object Request Broker: Architecture and Specification*.

PRAKASH, A. AND SHIM, H. S. 1994. DistView: Support for Building Efficient Collaborative Applications using Replicated Objects. In *Proc. Conference on Computer-Supported Cooperative Work CSCW '94* (Chapel Hill, North Carolina, 1994).

REISS, S. P. 1996. Simplifying Data Integration: The Design of the Desert Software Development Environment. In *Proceedings of ICSE-18* (1996).

REISS, S. P. 1999. The Desert Environment. *ACM Transactions on Software Engineering and Methodology 8*, 4 (Oct), 297 – 342.

SHETH, A. AND KOCHUT, K. 1997. Workflow Applications to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration Systems. In *Proc. NATO ASI on Workflow Management Systems and Interoperability* (Aug 1997).

SHETH, A., WORAH, D., KOCHUT, K., MILLER, J., ZHENG, K., PALANISWAMI, D., AND DAS, S. 1997. The METEOR Workflow Management System and Its Use in Prototyping Significant Healthcare Applications. In *Proceedings of the 'Towards An Electronic Patient Record (TEPR '97)' Conference* (1997).

STONE, D. AND NESTOR, J. 1987. IDL: Background and status. Technical Report CMU/SEI-87-TR-47 ADA188922, Software Engineering Institute (Carnegie Mellon University).

SUN. 1997. Java Remote Method Invocation Specification. Technical report, Sun Microsystems. `http://www.javasoft.com/products/jdk/1. 1/docs/guide/rmi/spec/rmiTOC.doc.html`.

TARR, P. AND CLARKE, L. A. 1998. Consistency Management for Complex Applications. In *Proc. of the 20th International Conference on Software Engineering* (1998), pp. 230 – 239.

TAYLOR, R. N., MEDVIDOVIC, N., ANDERSON, K. M., JR., E. J. W., ROBBINS, J. E., NIES, K. A., OREIZY, P., AND DUBROW, D. L. 1996. A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering 22*, 6 (June), 190–406.

TIBCO SOFTWARE. Tib/rendezvous whitepaper. http://www.rv.tibco.com/whitepaper.html.

WISE, A., LERNER, B. S., MCCALL, E. K., OSTERWEIL, L. J., AND JR., S. M. S. 1998. Specifying Coordination in Processes Using Little-JIL. Technical Report UM-CS-1998-038 (Aug), University of Massachusetts at Amherst.

WISE, A. E., LERNER, B. S., MCCALL, E. K., PODOROZHNY, R., OSTERWEIL, L. J., AND JR., S. M. S. 1998. Little-JIL 1.0 Language Report. Technical Report 98-24 (Apr), University of Massachusetts at Amherst.