Meeting the Computational Needs of Intelligent Environments: The Metaglue System

Michael H. Coen, Brenton Phillips, Nimrod Warshawsky, Luke Weisman,
Stephen Peters, and Peter Finin.

MIT Artificial Intelligence Lab
545 Technology Square
Cambridge, MA 02139
mhcoen@ai.mit.edu

Abstract. Intelligent Environments (IEs) have specific computational properties that generally distinguish them from other computational systems. They have large numbers of hardware and software components that need to be interconnected. Their infrastructures tend to be highly distributed, reflecting both the distributed nature of the real world and the IEs' need for large amounts of computational power. They also tend to be highly dynamic and require reconfiguration and resource management on the fly as their components and inhabitants change, and as they adjust their operation to suit the learned preferences of their users. Because IEs generally have multimodal interfaces, they also usually have high degrees of parallelism for resolving multiple, simultaneous events. Finally, debugging IEs present unique challenges to their creators, not only because of their distributed parallelism, but also because of the difficulty of pinning down their "state" in a formal computational sense. This paper describes Metaglue, an extension to the Java programming language for building software agent systems for controlling Intelligent Environments that has been specifically designed to address these needs. Metaglue has been developed as part of the MIT Artificial Intelligence Lab's Intelligent Room Project, which has spent the past four years designing Intelligent Environments for research in Human-Computer Interaction.

Introduction

Research on highly interactive spaces, generally known as Intelligent Environments, has become quite popular recently. Although their precise applications, perceptual technologies, and control architectures vary a great deal from project to project, the *raisons d'être* of these systems are generally quite similar. They are aimed at allowing computational systems to understand people on our own terms, frequently while we are busy with activities that have never before involved computation. IEs seek to connect computational systems to the real-world around them and the people who inhabit it.

This paper presents what we believe are general computational properties and requirements for IEs, based on our experience over the past four years with the Intelligent Room Project at the MIT Artificial Intelligence Lab. Although many of

the published descriptions of IEs [4] differ in their particulars, it is clear that we have not been alone in confronting some of the frustrating aspects of engineering these complex systems.

Based on this experience, we have developed Metaglue, a specialized language for building systems of interactive, distributed computations, which are at the heart of so many IEs. Metaglue, an extension to the Java programming language, provides linguistic primitives that address the specific computational requirements of intelligent environments. These include the need to: interconnect and manage large numbers of disparate hardware and software components; control assemblies of interacting software agents *en masse*; operate in real-time; dynamically add and subtract components to a running system without interrupting its operation; change/upgrade components without taking down the system; control allocation of resources; and provide a means to capture persistent state information.

Metaglue is necessary because traditional programming languages (such as C, Java, and Lisp) do not provide support for coping with these issues. There are currently several other research systems for creating assemblies of software agents [7,8,9], which provide low-level functionality, e.g., support for mobile agents and directory services. These features are necessary but not sufficient. Because Metaglue provides high-level tools specifically relevant to creating software controllers for IEs, we hope to make it available for more wide-spread use by the IE community.

Much of our discussion will focus on *Hal*, our most recently constructed Intelligent Room [3,5], where approximately 100 Metaglue agents control Hal and interconnect its components. However, we believe the issues raised here extend beyond the particulars of Hal and are important for a wide range of intelligent environments.

Hal is a small room within our lab and is equipped with microphones, seven video cameras, and a variety of audio-visual output devices that it can directly control. Hal was designed to explore a wide range of interactions involving futuristic residential spaces – stressing quality of life – and commercial spaces – stressing information management. We have therefore created applications in Hal that support entertainment, teleconferencing, business meetings, military command post scenarios, and information retrieval.

Next, we expand on our list of computational properties for IEs and examine the reasons behind them. We then discuss design considerations of the Metaglue system, and how it specifically addresses the perceived needs of IEs. In this, we directly trace how the issues raised in the next section are satisfied by capabilities incorporated into Metaglue. Finally, we close with an evaluation of the Metaglue system and directions for future research.

Computational Properties of Intelligent Environments

Intelligent Environments by and large share a number of computational properties due to commonalties in how they internally function and externally interact with their users. Of course, not every IE will identically share all of these characteristics, but we believe examining them even briefly makes concrete many of the issues that IE designers are faced with and should address directly. We not only hope to further

discussion on these issues in the IE community, but to motivate the development of other general purpose tools such as Metaglue.

We note that when multiple people are allowed to interact simultaneously with a single IE, many of the issues discussed below are greatly exacerbated. Space limitations preclude addressing this issue in detail.

Distributed, modular systems need computational glue

Intelligent Environments contain a multitude of subsystems comprising their perceptual interfaces, software applications, hardware device connections, and mechanisms for internal control. Even though each IE is created in its own way for its own purpose, IEs are generally built out of similar components.

Thus, IEs require some way to *glue* all of these components together and coordinate their interactions. These components also generally cannot co-exist on a single computer, due to hardware constraints and the need for environments to respond in real-time to their users. It is not uncommon for individual computer vision or speech understanding applications to consume the resources of an entire workstation, and there is no reason to believe this exclusivity will be diminished as processor speeds increase in the future. Many of these systems perform progressive real-time searches that naturally generalize to consume all increases in available computational power.

Frequently, these components, either off the shelf or research programs in their own right, are not designed to work together, so not only must they be connected, but there needs to be some way of expressing the "logic" of this interconnection. In other words, inter-component connections are not merely protocols, but must also contain the explicit knowledge of how to use these protocols. Thus, viewing the connections simply as Application Programming Interfaces (APIs) is insufficient. For example, consider connecting a speech recognition system and a web browser, so that users can navigate links by speaking the text contained in them. Here, the computational *glue* would include a mechanism that dynamically updates a recognition grammar with the link text whenever the user goes to a new web page; simply having APIs to both of these applications is necessary but not sufficient.

More generally, enormous amounts of control code go into building IEs, much of it dealing with how connections among its pieces should be managed. (See [1,2].)

Resource management is essential

Interactions among system components in an IE can be exceedingly complex. Resources, such as video displays or computational power, can be scarce and need to be shared among different applications. For example, in Hal, multiple computer vision systems share individual video cameras because they are a relatively expensive resource [6]. Conflicts can occur when multiple applications in an IE all want to display information on a single video display or speak to the user simultaneously. One of the largest surprises while developing Hal was that even seemingly simple issues, such as displaying a video, have wide spread repercussions on other parts of the running system because their resources are pulled out from under them. We discuss this at greater length in the next section. Finally, even in an environment with

ample resources for a single user, conflicts can unknowingly arise when multiple people attempt to interact with it simultaneously.

Thus, IEs need sophisticated resource management capabilities, particularly to let them scale properly as new applications and capabilities are added.

Configurations change dynamically

IEs can be highly dynamic systems. In the prescient words of Weiser – referring to Ubiquitous Computing but equally relevant to IEs – *"New software … may be needed at any time, and you'll never be able to shut off everything in the room at once … functionality may shrink and grow to fit dynamically changing needs"* [12] People may come and go at will, bringing with them devices such as PDAs that temporarily connect with an IEs existing computational infrastructure.

In a developing system such as Hal, new hardware and software components are incorporated on a regular basis. It should be possible to add them to a running system without restarting unrelated components. In fact, under many circumstances, new permanent components should dynamically integrate themselves into an IE without interrupting its operation at all.

Even within the confines of a static IE, users may readily switch between different aspects of its functionality. For example Hal supports teleconferencing and information management applications and users readily switching between the two is quite natural; often during meetings the need arises to get more information about something. These "context" changes can have far reaching effects. For example, an IE may need to simultaneously start new underlying applications, activate different speech recognition grammars, and modify the configurations of other perceptual systems.

State is precious

Not only may new components need to be incorporated into a running IE, but pieces of a running system may need to be reloaded into it as well. As with any other software engineering effort, creating IEs require an iterative edit-recompile-run process while testing new features and eliminating bugs. However, if the entire system had to be restarted each time one of its components changed, development would be prohibitively time-consuming. Our Hal environment has literally dozens of hardware and software components connected to approximately 100 Metaglue agents. Having to bring this system completely down to modify it would long ago have made further development a cause of endless frustration.

Furthermore, IEs acquire state through interactions with users. Attempting to trace a bug by forcing a person to repeat a sequence of interactions which may have spanned several hours would be outrageous. To exacerbate the problem, state is acquired not only through human interactions, but through any activities Hal has engaged in, such as information retrieval. A weather report or CNN headline that caused Hal to take some action may have long since changed and is not recoverable.

The most critical part of Hal's state comes from information it learns while observing its users. Hal has several machine learning systems for learning about users' preferences and activities. These systems have no straightforward way to unlearn and return to a previous coherent state. Checkpointing in the style of reliable

transaction systems can partially ameliorate these problems with respect to the local state of individual components. However, when IEs are asynchronous and distributed, repeating a particular *global* state can be, practically speaking, impossible to achieve. (One technique we have been investigating is allowing an IE to essentially simulate itself by replaying previously observed and recorded events.)

Thus, there is a clear need for an IE's software architecture to permit a kind of dynamism rare in conventional computational systems. We would like to stop, modify, and reload components of a running system and have them reintegrate into the overall computation with as much of their state intact as possible.

IEs model the parallelism of the real world

Supporting natural human computer interaction requires that IEs have some handle on multiple ways that a user may interact with them. People speak, gesture, move, and emote simultaneously, and IEs need to have some capacity to cope with this, even on the part of a single user. This is not to say they need to understand the full range of human discourse to be useful. Far from it, IEs that consistently – and most importantly, predictably – understand a small subset of interactions are far preferable from an HCI perspective to ones that always leave users guessing if some particular input will be understood.

Nevertheless, IEs generally have multimodal interfaces which requires they have sufficient parallelism for resolving multiple, simultaneous events. For example, if a user walks to a displayed map, points somewhere and says, *"What's the weather here today?"* and then immediately walks away, the system must be able to discover where they were pointing when they said the word "*here.*" Dealing with multiple users simultaneously again simply exacerbates the problem.

Thus, IEs need at least as much parallelism as the phenomena they are trying to understand. In fact they may need a great deal more for background processing, which leads to the next item.

Real-time response

It almost goes without saying that IEs need to be responsive to their users. Particularly due to the fact that many IEs do not have traditional computer monitors so users can get a handle of the system's inner activity, it is astonishingly frustrating when an IE does not respond quickly to user input. This is another point supporting the basic need for parallel architectures in an IE. The parts of the system that acknowledge and react to users must be immediately responsive even if other parts of the system, for example, in the midst of processing an information retrieval query, require more time to respond.

This also requires that the mechanism for interconnecting an IE's components and processing their data be able to keep up with the underlying external systems. For example, in Hal, five C-language-based computer vision systems, each producing several hundred dimensional data vectors at a rate of up to 30 a second, all connect to

a Metaglue-based visual event classification system which must process all this data in real-time.[1]

Debugging is difficult
Independently of IEs, debugging distributed, asynchronous systems can be a nightmare. If some high-level system event fails to occur, determining which component is to blame is usually a long, involved process. Furthermore, understanding the operation of distributed, loosely coupled components running in parallel – as does the controller for an IE – where different serializations can have different system-wide effects, is best, but rarely successfully, avoided. In an IE, this problem is made all the worse by the presence of many, sometimes exotic, hardware components, such as video multiplexers, that themselves have internal state that may only be imperfectly modeled in their software drivers.

Good software engineering practices go a long way towards dealing with this problem, but a more comprehensive solution would require the development of new types of debugging strategies. (In the next section, we see that Metaglue only makes the most preliminary efforts in this direction.)

Metaglue

We first discuss the design of Metaglue from a programming language perspective, to give potential users a sense of what it would be like to work with it. We then proceed to illustrate how particular features in Metaglue address many of the computational requirements for IEs discussed in the previous section. By necessity, this section is intended only to sketch and motivate the capabilities of the Metaglue system and should not be viewed as a complete description of the language. More detail about Metaglue's internals can be found in [10,11]. (Some examples in this section require cursory knowledge of the Java programming language.)

The Design

Metaglue is an extension to the Java programming language that introduces a new *Agent* class. By extending this class, user-written agents can access the special Metaglue methods discussed below. Metaglue has a post-compiler, which is run over Java-compiled class files to generate new Metaglue agents. Metaglue also includes a runtime platform, called the *Metaglue Virtual Machine*, on which its agents run. The overhead added by this infrastructure to standard Java programs that are turned into Metaglue agents is negligible.

Our goal with Metaglue was to add a very small number of primitives to the Java language to make it easy to write agents. Method invocations between agents, even if they are on different workstations, look exactly like local method calls in Java. Thus, Metaglue agents, minus the few Metaglue-specific primitives, look almost exactly

[1] For completeness, it should be mentioned that this agent is run on a very high end, multiple processor workstation.

like ordinary Java programs. This makes it easy to transform regular Java source files into Metaglue agents, which enormously adds to Metaglue's value as computational glue. We call the process of transforming previously existing programs into agents *wrapping*.

Almost as much time has gone into formulating Metaglue's semantics as to programming the system itself. We sought to provide a focused set of primitives for managing systems of distributed, interacting agents and to avoid the temptation of creeping featurism. By stressing simplicity, anyone proficient in Java can pick up Metaglue very quickly, and the small number of new primitives make it easy to learn, remember, and use the system.

In the remainder of this discussion, it will be helpful to keep in mind that running a Metaglue system first involves starting Metaglue Virtual Machines on all the computers that are involved. Our machines are generally configured to start these when they are booted.[2]

The Capabilities

Metaglue offers the following capabilities, each of which we will address in turn:

1. Configuration management
2. Establish *and maintain* the configuration each agent specifies
3. Establish communication channels between agents
4. Maintain agent state
5. Introduce and modify agents in a running system
6. Manage shared resources
7. Event broadcasting
8. Support for debugging

Metaglue has a powerful naming scheme for agents that is beyond the scope of this document. We will use here the simplest form of it, the name of the Interface file of an agent, which is in the Java class package format, e.g., an agent for controlling a television might be referenced by *device.Television.*

1. Configuration Management

Metaglue has an internal SQL database for managing information about agent's modifiable parameters (called Attributes), storing their internal persistent state, and giving agents fast, powerful database access.[3]

Attributes contain information that might otherwise be hardcoded inside agents and difficult to modify, for example, what workstation the agent needs to run on or parameters that affect its operation. Metaglue has a web-based interface for modifying Attributes, which can be changed even while an agent is running. This is

[2] For reference, this has the computational overhead of running one Java Virtual Machine, which is close to unnoticeable on modern Pentium-based systems.

[3] The use of an internal database helps enormously in dealing with Java's poor file access capabilities. Agents use the database rather than store information in files, which is particularly important because agents can move to different machines while they are running.

one of Metaglue's mechanisms for both configuring a system of agents and interacting with it while it is operational.

This is code an agent would use to get its *location* Attribute from Metaglue's built in database:

```
Attribute location = new Attribute("location");
System.out.println("I run on " + location.getValue());
```

2. Agent Configurations

Metaglue agents can specify particular requirements that the system must insure are satisfied before they are willing to run. These can include the name of a particular computer they must be run on; specifications for particular types of hardware they require access to; and more abstract capabilities that must be available on whichever Metaglue Virtual Machine (MVM) they are run on. These are expressed with the `tiedTo()` primitive, as in:

```
tiedTo(location.getValue());
tiedTo(capability.FrameGrabber);
tiedTo(device.Television);
```

If an agent is started on an MVM that does not meet its stated requirements, Metaglue will move it somewhere else that does. If Metaglue needs to restart an agent due to localized hardware or software failure, it will attempt to find an alternative MVM on which to run the agent that also satisfies these requirements. (See item 5.)

3. Agent Connections

Because Metaglue is intended as computational glue, it needs to establish paths of communication between agents, regardless of where they are running. The `reliesOn()` primitive connects agent with capabilities they can request services from. For example, to use Hal's speech synthesizer, an agent might contain:

```
Agent speechSynthesizer = reliesOn(speech.Synthesizer);
speechSynthesizer.say("Hello! How are you?");
```

The reference to `speech.Synthesizer` refers to an abstract capability, not a particular agent. Because agents refer to each other by their capabilities and not directly by name, new agents can easily be added to the system that implement preexisting capabilities without modifying any of the agents that will make use of them. (A more sophisticated way of obtaining capabilities is described in the Metaglue resource manager below.)

Metaglue will try to locate an agent that provides the requested capability on any of the system's computers' MVMs and return a reference to it to the caller. Metaglue has an internal directory called a *Catalog* that it uses to find agents once they are started. Metaglue agents automatically register their capabilities with the Catalog when they are run.

If no agent offering this capability is found, Metaglue automatically starts one and invisibly insure that it continues running as long as it is in use.

The `reliesOn()`primitive makes it very easy to interconnect agents with a single line of code. Reliance is also a transitive operation. For example, starting the single Hal *demo* agent results in the entire Hal system being loaded because of their chain of reliances. Also, because they have been formally relied upon, Metaglue will attempt to insures that they continue running indefinitely, as discussed below.

4. Agent State

Agents can use Metaglue's `freeze()` and `defrost()` primitives to store and subsequently retrieve their fields from Metaglue's internal SQL database. The standard way of doing this is having an agent directly freeze its state when it is shutting down (or at any other appropriate time), and subsequently defrost itself in its constructor the next time it is started up. Other aspects of an agent's state, e.g., its connections to other agents, are internally maintained by Metaglue and generally do not need to be specifically managed by the agents themselves.

As of yet, we do not have a well-defined schema for capturing the global state of all of the agent's in a running Metaglue system.

5. Modifying a running system

Metaglue will attempt to keep a running system of agents alive. If an agent is manually stopped, for example, during debugging, the agents that rely upon it will by default simply wait for it to return in the event they need to access it. When the agent is restarted by the user, it will reload its frozen state and simply pick up where it left off, first dealing with any pending requests from other agents.

It is also possible to programmatically specify actions, other than simply waiting, that an agent can do if someone it relies upon is stopped. For example, it might temporarily switch to another active agent that offers the same capabilities. Metaglue's resource management can help the system in the event capabilities must be shared due to part of the system being unavailable.

If an agent dies because of unanticipated hardware or software failure, Metaglue will try to restart it automatically, switching to another MVM if necessary, but still meeting the agent's required configuration if possible. It is important to note that unanticipated crashes may cause state information to be lost, and agents who are sensitive to this should refuse to be automatically restarted. For example, an agent that controls Hal's lighting systems may not know whether the lights are on or off if it is restarted after a crash because the state information it defrosts may be inaccurate. However, in that case, it can simply ask Hal's vision agents whether they can see anything, and thereby determine the state of the lights in the room. An agent running part of an application, however, could be started out of sync with the rest of the system, and manual intervention may be required to correct the problem.

Interestingly, the Metaglue system is itself recursively constructed out of a special set of Metaglue agents. These agents have the full functionality of the system available to them, so they can for example, use Hal's speech synthesizer to let users know of internal problems in the system and ask for help resolving them.

6. Managing shared resources

Among the largest and most complex systems in Metaglue is its resource manager. Before it existed, agents in Hal simply grabbed the resources they needed and configured them at will. That a resource management system was necessary became apparent when Hal developed to the point that its multiple applications conflicted with one another and could no longer be run simultaneously. Additionally, for an agent to simply rely on the resources it wants to use, it has to know both what resources exist and are available. As devices and other agents dynamically come and go in the system, this means that every agent would need to keep track of the different resources that offer the sets of capabilities it needs. We discussed above that agents may temporarily make use of substitutes if the agents they generally rely upon are unavailable. Where should that knowledge of possible alternatives come from?

The resource system in Metaglue allows agents to request functionality at a very high-level, without being concerned with how it is provided or resolving resource conflicts among themselves. Metaglue has a hierarchical set of *dealer* agents that are responsible for distributing resources to the rest of the system. There are a wide assortment of different prototype dealer agents available, each of which has its own specified internal logic for performing allocation, substitution, etc. These dealers can be used directly by Metaglue programmers or extended to customize their operation.

Dealers not only give out resources, but they can withdraw previously allocated ones to redistribute them, based on any of several priority and fairness schemes. For example, there are dealers in Hal for allocating televisions, video projectors, and displays in general. An agent must use the dealers to gain access to any of these. If a higher priority agent needs access to a particular display, it will be temporarily withdrawn from the agent who has allocated it until it becomes available again, at which point it will be given back.

Agents can also create filters they pass to dealers to describe precisely the kind of resources they are seeking, e.g. access to a speaker next to a particular display, a video camera in Hal looking at a particular person, etc.

7. Event Broadcasting

In addition to agents making direct requests of one another through method calls, Metaglue agents can pass messages among themselves. Agents can register with other agents, including the Metaglue system agents, to find out about events going on in the system. For example, an agent in Hal interested in greeting people by name when they walk inside the room, simply registers with the vision-based Entry agent to request messages about *entrance* events where the identity of the person can be determined. When these events occur, it receives a message and uses the agent offering speech synthesis capability to say hello to them.

We also use event broadcasts to notify groups of agents about context shifts in room applications to dynamically and uniformly modify Hal's behavior.

8. Debugging

Metaglue has a graphical interface for examining a running system of agents called the Catalog monitor. It displays all running agents and their reliance interconnections. Clicking on an agent brings up a window in a read-eval-print loop, in which users can interactively call the agent's methods.

Metaglue also has a logging facility to manage and centralize agents' textual output. This can be useful for programmers to watch the output of particular agents without worrying about where they are running or where their output streams are being printed.

We have found these capabilities quite useful, but would still prefer source level debugging of remote agents, a dynamic object browser, and ways to set breakpoints over whole groups of agents simultaneously. At least some of these capabilities promise to be available shortly in commercial Java products and we hope to make use of them during Hal's continued development.

Discussion

Evaluating the merits of a programming language can defy objectivity. Nonetheless, Metaglue has been extraordinarily useful in building Hal, and it is highly doubtful Hal would have reached its present level of development with it. Metaglue is a very stable system, and we have left large assemblies of agents running for up to a week without any difficulties. (These systems were eventually stopped for development purposes.) Currently, Metaglue would seem to be as reliable as is Java itself.

We now reexamine each of the previously mentioned properties of IEs in the context of Metaglue.

Distributed, modular systems need computational glue

Metaglue not only provides a channel to interconnect Hal's components, but it also provides the means to build applications for Hal. Rather than use a special communication mechanism, such as CORBA or KQML, separate from the system's internal controller, Metaglue allows us to reduce the amount of infrastructure by providing for both communication and control with a much lighter-weight system.

Resource management is essential

The resource management system in Metaglue not only offers a wide range of default behaviors, but it is easily customizable through Java's class extension mechanism. It is among Metaglue's most developed systems and we are in the process of incorporating it into the applications that predated it.

Configurations change dynamically

Metaglue offers several mechanisms for coping with dynamically changing systems. The Configuration Manager and Attribute system allows users to reconfigure agents while they are running. The fact that agents refer to each other by abstract capabilities means that new agents can be incorporated into a running system without modifying any of the agents that might rely upon them. Metaglue's ability to start and stop agents while leaving the rest of the system running allows us to dynamically "hotswap" components of a running computation. Finally, by substituting new resource managers into a running system, new functionality can be added that previously no agents were aware of.

State is precious

Metaglue offers support for persistent local state in agents via its freeze and defrost mechanisms. Notions of global state, however, remain illusive concepts.

IEs model the parallelism of the real world

Java is inherently multithreaded, which Metaglue inherits from it. Metaglue's resource management allows agents running in parallel to avoid conflicting with one another. The event broadcast mechanism also simplifies communication among interacting groups of software agents running simultaneously.

Real-time response

The amount of overhead Metaglue adds to Java is minimal. Our avoidance of heavy-weight, specialized communication packages allows Metaglue agents to essentially run as quickly as Java's Remote Method Invocation system. Metaglue is now incorporated into "tight" loops in our code, along the most processor intensive critical paths, such as in our computer vision systems. The development of JIT compilers for Java has enormously reduced our need to place perceptory components of our system into external C-language libraries.

Debugging is difficult

Metaglue certainly makes it possible to debug distributed agents systems, but one can hope for more. There is reason to believe the Java community as a whole shares some of this interest and will takes steps in this direction.

Future directions

We are presently incorporating an expert system into Metaglue to allow more sophisticated reasoning about system configuration and resource management. We are also creating a machine learning extension to Metaglue, which will incorporate pieces of the system described in [6].

References

1. Bobick, A.; Intille, S.; Davis, J.; Baird, F.; Pinhanez, C.; Campbell, L.; Ivanov, Y.; Schütte, A.; and Wilson, A. Design Decisions for Interactive Environments: Evaluating the KidsRoom. *Proceedings of* the *1998 AAAI Spring Symposium on Intelligent Environments.* AAAI TR SS-98-02. 1998.
2. Coen, M. Building Brains for Rooms: Designing Distributed Software Agents. In Proceedings of the Ninth Conference on Innovative Applications of Artificial Intelligence. (IAAI97). Providence, R.I. 1997.
3. Coen, M. Design Principles for Intelligent Environments. In Proceedings of The Fifteenth National Conference on Artificial Intelligence. (AAAI98). Madison, Wisconsin. 1998.
4. Coen, M. (ed.) Proceedings of the 1998 AAAI Spring Symposium on Intelligent Environments. AAAI TR SS-98-02. 1998.

5. Coen, M. The Future Of Human-Computer Interaction or How I learned to stop worrying and love My Intelligent Room. IEEE Intelligent Systems. March/April. 1999.
6. Coen, M., and Wilson, K. Learning Spatial Event Models from Multiple-Camera Perspectives in an Intelligent Room. *In submission.*
7. General Magic. *Odyssey (Beta 2)* Agent System Documentation. http://www.genmagic.com/agents.
8. Lange, D. and Oshima, M. Programming and Deploying Java Mobile Agents with Aglets. Addison Wesley. 1999.
9. ObjectSpace, Inc. *ObjectSpace Voyager Core Package Technical Overview (Version 1.0).* December 1997. http://www.objectspace.com/voyager/whitepapers.
10. Phillips, B. Metaglue: A Programming Language for Multi-Agent Systems. M.Eng. Thesis. Massachusetts Institute of Technology. 1999.
11. Warshawsky, N. Extending the Metaglue Multi-Agent System. M.Eng. Thesis. Massachusetts Institute of Technology. 1999.
12. Weiser, M. The Computer for the 21st Century. *Scientific American*. pp. 94-10, September 1991