Agent-Based Software Engineering

Charles Petrie

Stanford Networking Research Center
350 Sera Mall
Stanford, CA 94305-2232
petrie@stanford.edu

Abstract. It has previously been claimed that agent technologies facilitate software development by virtue of their high-level abstractions for interactions. We address a more specific characterization and utility. We believe that it is important to distinguish agent technologies from other software technologies by virtue of a set of unique software characteristics. This is in contrast to much in the literature that concentrates on high-level characteristics that could be implemented with a variety of software techniques.

Agent-based software engineering (ABSE), for at least an important class of agents and applications, can be characterized by both model and inner/outer language components. Our experience in developing applications based on long-term asynchronous exchange of agent messages, similar to typical email usage, leads us to believe these unique characteristics facilitate useful software development practices. The utility derives from a stratification of change among the components, ease of collaborative change and debugging even during runtime due to asynchronous text parsing-based message exchange, and reuse of the outer language as well as generic agents as a programming environment.[1]

1 Agent Development Characteristics

Jennings and Wooldridge have described *Agent-Oriented Software Engineering* (AOSE)[19] [7]. AOSE effectiveness claims are based upon three strategies for addressing complex systems: *decomposition*, *abstraction*, and *organization* and that the "agent-oriented mindset" gives one an advantage in using these strategies. That agents have objectives gives one a clear way to decompose the problem into agents. That agent systems work largely by **emergent behavior** and handle errors gracefully reduces the need for detailed specifications, since not all interactions need be specified in advance, and allows more abstraction to be used in system building. And finally, agent systems are naturally hierarchical organizations themselves. Indeed, it is important to note that agent identity is a fundamental component of all agent languages and methodologies for interactions.

We agree with these points. In particular, the idea of engineering a system so that the correct emergent behavior results is the most critical idea in agent software engineering. We further agree with the notions introduced by Huhns[5] that one aspect of emergent

[1] A shorter version of this paper first appeared in *Proc. PAAM 2000*, Manchester, April, 2000.

behavior is having software modules being able to model themselves and other modules, which leads to modules that are able to attempt different methods of accomplishing a task based upon runtime data and these models, which in turn leads to flexibility and robustness without the necessity of the programmer having correctly considering every possibility. The fundamental idea here is that the programmer focuses on the types of interactions possible without specifying all possibilities in advance and one technique for doing this is modeling interaction behaviors. These notions are important and fundamental. This paper is also important because it points out that software engineering seems to have reached a plateau of results and has been stuck there for some time. Thus the agents approach is potentially very important.

However, we find the AOSE explanations lacking in the detail that would allow a software engineer to decide easily whether an AOSE approach was even being used or not. We describe *Agent-Based Software Engineering* (ABSE) here as a refinement of some aspects of AOSE, based upon our practical experience in agent building. We do not, though, attempt to describe how to practice ABSE. Our goal is rather show that ABSE could be distinguished objectively from other software techniques, which has not previously been done.

This is a topic that is frequently addressed with respect to languages. In fact, in the abstract of a recent paper on agent languages, M. Wooldridge said his intent was to develop "a semantics where conformance (or otherwise) to the semantics could be *determined by the independent observer*" [20]. Whether something is an agent or not, and whether a software engineering technology is agent-based or not, should also be verifiable by independent observers.

Further, we attempt to persuade the user that ABSE could be useful in practice and why, apart from the general potential of emergent behavior. We emphasize some particular aspects of agent models and languages that distinguish agent system development from other software technologies and make it useful. Our intent is to help bridge the gap between agent technology and software engineering.

1.1 Semantics-based Agenthood

Describing agent development requires distinguishing agents from other kinds of software. Unfortunately, definitions of agents most often differentiate the technology from other software technologies by anthromophising agents: ascribing human cognitive traits such as environmental awareness, autonomy, and intelligence. The AOSE view has its foundations in Yoav Shoham's "*Agent-Oriented Programming (AOP)*"[14] in which it is advocated that agents be directly programmed in terms of the mentalistic, intentional notions of agent theories. Similarly, Rao and Georgeff developed the "beliefs", "desires", and "intensions" (BDI)[13] theory of agenthood and developed both formalisms and programming technologies based on them. We call this a *semantic* view because it usually requires ascribing some meaning to the operation of each agent.

And while this early work was formal with distinct software technologies, it is clear that the difficult objectives of creating a programming environment were not completely reached. The programming languages and compilers generated were not generally useful. It has been critiqued as unhelpful by people attempting to use the technology[10]. But more important, this formal work has very limited scope in agent building, much as formal software methods in general have. Since useful languages and compilers were not forthcoming, proponents of agent technologies spoke more of the general characteristics of agents and how that influenced software practices.

These informal descriptions of agents as "autonomous" and "proactive" are useful. Courses on agents, such as "Intelligent Agents" [3], cover much important technical material. For instance, if one intends that agents follow goals without a lot of specific instructions, then one has to turn to the most sophisticated learning, planning, non-monotonic reasoning, and distributed constraint satisfaction techniques that have been developed by the Artificial Intelligence community. This is in fact the most important aspect of agents *per se*: that they perform such sophisticated reasoning, especially in collaboration with each other so that the desired behavior is emergent rather than specifically programmed.

But the less formal AOSE recent work does not describe mentalistic, intentional notions of agenthood in a way that would permit an objective observer to distinguish agents from other software technologies, at least not without a lot of persuasive discussion in each case. Shoham's definition of an agent as "an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments", and the similar BDI descriptions, are of little help in objectively defining agents apart from their specific formalisms. And it does not help to focus on one narrow theory of agents, such as BDI, as this tends not to generalize to a software engineering practice or methodology.

The informal interpretation of mentalistic agent theories is not sufficient to distinguish agent technology from other software technologies, much less provide practical specific techniques for agent building that would constitute *software engineering*. It is at least arguable that, say, object-oriented programming, could be used to implement non-monotonic reasoning, for instance, as well as any other. More to the point, how would one know if one is using agent technology as a software technique? The distinguishing factor cannot be just that the program can be interpreted as exhibiting a kind of autonomy using various sophisticated algorithms. This view of agents says nothing about programming practice, much less software engineering.

However, consider the use of AI techniques in building agents. At one time, "what is AI?" was an oft-asked question. The answer eventually turned out to be that AI was the use of AI techniques. This was a seemingly circular definition but it came to be accepted because they was a definite set of techniques that was developed by researchers and departments explicitly identified as "AI". The same idea can be used in the case of agents. It may be that emergent behavior system design is in its infancy, but based on experience, we can identify several general but distinguishing characteristics of agent software technology.

1.2 Agent Theory Modeling

The BDI theory is portrayed primarily in AOSE as a means of guiding agent software development. But BDI is only one type of agent semantics and to consider only it misses the abstraction of imposing any *agent theory* during development. This is especially important when one is converting legacy code into an agent capable of working with other agents to carry out a distributed computation. What one does in this case is to interpret the workings of the legacy code as an instance of a BDI theory that is then used as the common model upon which agent interactions are based, and which is thus the basis for the integration of the software components.

The Agent-Objection-Relationship model of [17] (similar to other work reported in this volume) is a clear example of an agent model, based more on entity-relationship models than on BDI, used for application-building. In our set of engineering applications, the *ProcessLink* framework used the *Redux* theory of design to characterize the acts of the engineering agents[12]. The Redux theory is quite different from the BDI theory and not at all anthromorphic. However, both are used in modeling: translation from legacy code into agent interactions.

The case is not that different with agents that are built "from scratch" rather than being legacy code conversions. One has a computation in mind and is then is careful to code it so that it ultimately conforms to the agent theory, whether it be BDI or Redux. Another example is Tate's O-Plan[16], which is similar to Redux in many respects, which was used in agents with the "Act" formalism[9].

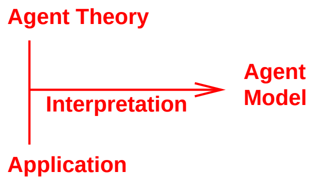

Fig. 1. Agent-based Modeling

As illustrated in Figure 1, the common characteristic of agent-based systems is that some agent theory is interpreted for some application to produce an agent model used for application development. This modeling is characteristic of many, if not all, agent systems, such as WARREN[1] (information ontology imposed on a BDI model), TIE-97[18] (O-Plan and Act), ProcessLink (Redux), and others. All of the ones referenced here are based on some version of hierarchical planning formalisms. For example, WARREN presumes a formal model in which goals are planed, actions to achieve them are scheduled, and execution of the actions is monitored. The links between goals and actions are further formalized, complete with constraints on action scheduling that further guides the development of the agents. The O-Plan and Act formalisms are even more detailed. Redux falls somewhere in between in complexity.

Thus, in both legacy code conversion and the pure agent development, for at least a very important class of agents, *agent construction is partially characterized by a modeling based on a common agent theory.*

1.3 Syntax-based Development

Jenning's ABSE paper[6], based largely upon [7], is also largely lacking in the detail necessary for objective determination of agent-based software engineering, with one important exception. Jennings, as have many others, notes that agents "have their own persistent thread of control" and notes later that this enables agents to select their actions based on "the agent's actual state of affairs". In comparison to remote procedure calls and the similar message passing of object-oriented programming, this is an important distinction, though it is frequently obscured with talk of "self-awareness".

Agents maintain state. This means that they can potentially model themselves and others, as Huhns points out. But the more important point, computationally, is that the response to a query or function call may be based not just on the state conveyed by the caller, but also based upon other information maintained by the agent from dynamic information sources. However, while this may distinguish agent technology from an important class of remote procedure calls, it does not distinguish it from, say, database queries against a database server with its own active thread of control. More important, it is difficult to objectively determine the state of "state". Something more is required.

While Shoham, Jennings, Wooldridge, and many others take the anthropomorphic semantic view of agents, this contrasts directly with the view of Genesereth and Ketchpel in which the distinguishing characteristic is the use of an Agent Communications Language (ACL)[4]. This alternative view is *syntactic* in that it only requires one to know something about the form of the communication among the software modules that are agent candidates. We have previously elaborated on this syntactic view by giving an operational definition for at least one type of agent: *typed-message agents*[11].

Our contribution was to add that if the computation could be performed just as well by a client-server architecture and protocol, then there is no need to call the software components "agents". The *requirement for a peer-to-peer protocol* is really a thesis that this objective indicator corresponds to the otherwise subjectively determined autonomous nature of agents, which maintain their own thread of control and state. This syntactic protocol and behavioral view requires no subjective interpretation in order to decide agenthood. And it clearly distinguishes agents not only from remote procedure calls, but also from databases that do not volunteer information and work with client-server protocols. Admittedly, one must additionally be able to evaluate whether the distributed computation being performed by the agents is better in one case or another, but we argue that without this degree of evaluation, one can say little about the efficacy of either agents or software engineering methods.

This distinction between a syntactic approach to agent definition, which permits an objective distinction between agent software and other kinds, and the semantic approach of ascribing mentalistic characteristics to some software models, is often conflated. For instance, in[17], it is asserted that one difference between the objects in Object-Oriented Programming and agents is that agents "can perceive events, perform actions, communicate, or make commitments. Objects are passive entities with no such capacities." Not only is this a claim that depends upon a subjective interpretation, the paper goes on immediately to point out that the distinction between ordinary object messages is that agent messages are typed. That is, the only objective difference between the two systems is that agent messages are typed, which is exactly the syntactic difference.

In addition to agent models, the language component of agent software plays an important role in development of agent-based systems. Typed-message agents communicate with inner and outer languages that characterizes most Agent Communication Languages (ACLs) today. So, after model translation, the other main ABSE task, for typed-message agents, is the development of the syntax for the outer and inner languages. In theory, this part of the development could require relatively little work. One could use one of the existing outer language ACL standards, like KQML or FIPA ACL, and then use KIF as the inner language, as KIF has all of the expressive power ever likely to be required by any agent application.

This scenario is of course naive. The various ACLs have not converged on a single all-purpose set of performatives and syntax after several years of use. In fact, the long experience of KQML applications and the constant invention of new performatives suggests that trying to define a single set of performatives that everyone will use is either futile or fascist. Similarly, the very expressiveness of KIF makes it sufficiently difficult to use that simpler inner languages are often developed for particular applications.

However, one only needs universal syntax if one really expects a universe of universal agents independently developed with different tools to speak the same language. But this (perhaps overly) ambitious goal is not necessary for agents to be a valuable software tool. And it is also likely that the more universal standards will be the exchange of HTML and XML documents via email and similar standard protocols more prosaic than multi-agent systems that typically are constructed to support some distributed but bounded complex computation. This does not mean that agent languages are not a useful methodology. And their ubiquitous use means that *agent technology is characterized by an ACL and inner language.*

2 ABSE Characteristics and Utility

Previously we have discussed some characteristics of at least a large class of agent systems: that they are based on agent theories, an ACL, and an agent inner language. We also require that the result be an agent system in the sense of requiring a peer-to-peer protocol. These are characteristics that can be objectively determined. We now describe how those characteristics map to characteristics of software engineering for agent systems.

2.1 Using Models for Control

Formalisms for agent control are obviously useful for the development of emergent behavior systems. The Distributed Artificial Intelligence community has been working on such formalisms for a number of years and the problems are far from resolved. This remains a research topic, characterized by Shoham's current economics and decision theoretic work[15]. This is indeed the crucial research topic, but since it is still research, it is difficult to make a claim that such control formalisms are characteristic of agent software development in general.

But we can claim that the formalisms in use today reduce the complexity of the agent interactions in some respect, and thus facilitate the engineering of emergent behavior, in addition to providing a guide for system development. If we look at the formal information ontology and architecture used for WARREN and other multi-agent systems, the planning and scheduling functions clearly organize the behaviors of the various information agents. Indeed, the agents can also monitor their own behavior and decide to clone themselves to offload work.

Another example is Redux. Given a constraint violation, Redux can advise, upon request, the logically consistent solutions to the problem. Further, Redux monitors the state of problem solving. If an agent attempts to make a move in the solution space that will cause a repetition of a cycle of moves by other agents, Redux will reply with a "SORRY" message and refuse the change in order to prevent thrashing of distributed design decisions. Moreover, Redux will tell designers when changes by other designers affect previously made decisions, even advising them to reconsider previously discarded choices.

This is not to say that these control functions could not be done with some other programming environment. But *ABSE is characterized by the commitment to some formal agent theory that reduces the complexity of anticipating the interactions of the agents for the programmer.*

2.2 Using Models and Languages as Guidelines

Modeling provides a template for agent development. It defines that kinds of conversations the agent could possibly have and thus what functions the programmer should provide. It is especially important for the transformation of legacy systems into agents. While this task necessarily requires a lot of intelligence on the part of the programmer, a model does provide a guide for the programmer.

For example, in Redux, a "decision" in a design problem consists of a "goal" and a result that is some non-empty conjunction of "subgoals" and "assignments". It is fairly easy for the programmer familiar with the Redux model to recognize what in the legacy code constitutes a decision being made. For instance, in a program for defining an electrical cable configuration given the parts and electrical requirements, the goal is the text or data specifying these and the result is a set of assignments of connections. The programmer adds software to the legacy system that can shape this data into a well-formed message to be sent to the other agents in the computation. This will occur automatically (subject to optional input from the engineer solicited at runtime) every time a new configuration is generated by the engineer.

This modeling adds an important dimension to especially the AOSE decomposition and abstraction strategies. One does not have merely the agent top-level objective (noted in AOSE) as a guide in agent building: one has a more elaborate model in each case that guides development of the interaction of the agents in the system. *This commitment to a common formal theory to guide development is a distinguishing characteristic of ABSE.*

The ACL also provides a cognitive guide for the programmer in anticipating the *kinds* of interactions among the agents. Some of the research in this is designed to implement strong conversational models, such as [2]. But apart from such conversational models, the profusion of dialects of ACLs and inner languages is confirming evidence that the simple *concept of outer and inner languages is not only characteristic of ABSE but also useful*. For example, many developers notice immediately that they could just use the KQML performatives "ASK" and "TELL" together with whatever inner language they choose to develop for their application. But it is usually the case, as with the ProcessLink (Electronic Project Language) (EPL)[2], that one uses some of the other standard ACL performatives and adds new ones.

However, the inner language is typically much more application-dependent and not as useful a guide to the developer as is the more application-independent ACL.An example of a simple inner language occurs in WARREN. Here is an instance of a query on IBM earnings from the Dow Jones news:

```
(monitor
:SENDER barney
:RECEIVER news-agent
:LANGUAGE simple-query
:ONOTOLOGY news
:REPLY-WITH ibm-query-2
:NOTIFICATION-DEADLINE (30 minutes)
:CONTENT (query news
:CLAUSES
(= $newsgroups ``dow-jones.indust.earnings-projections'')
(= subject ``IBM'')
:OUTPUT :ALL) )
```

Notice that the ACL performative “monitor” is a type of well-defined interaction among various agents in the distributed application. It defines a whole class of actions that can occur among the agents. The inner language (“:content”) refers to more specific types of functions and data. The “monitor” accepts, as content, the keywords “query”, “news” “clauses”, and “output”. Further, “clauses” takes a particular set of feature/variable value assignments. The performative “monitor” is independent of the particular queries being made. *The commitment to some ACL in defining possible interactions and an inner language where domain-specific functions and data are specified is characteristic of ABSE.*

Figure 2 illustrates the basic three ingredients in agent construction, ignoring, for now the communications software. The agent theory is used to model the application. The theory is used to model the specific operations of the application. The ACL guides the formation of the basic agent interactions, and the inner language reflects the basic data exchange and functions. These elements help guide the system developer and also provide a measure of indirect control over the subsequent behavior of the agents.

The fact that one can apply a common theory to several components of an application, as well as at least an ACL, means that ABSE is also a way of integrating legacy applications. Not only can one “wrap” legacy code so that it corresponds to some programming methodology, such as objects, but the wrapping code further should also correspond to some common theory, which provides an extra “glue” for the integration. In our example systems, for instance, TIE97-1 agents included several agents developed at different sites in different countries. Redux has been used to integrate various engineering software in multiple disciplines. WARREN integrates agents working in multiple domains with different expertise and data sources.

[2] http://www-cdr.stanford.edu/ProcessLink/protocol/EPL-syntax.html

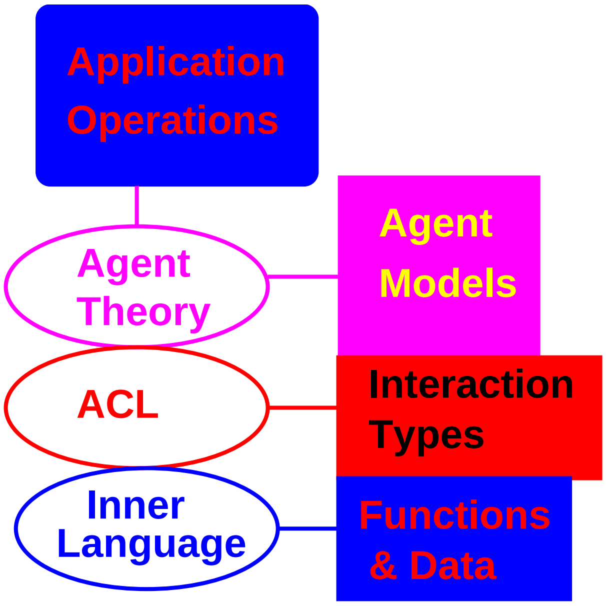

Fig. 2. Agent-based Construction Layers

2.3 Stratifying Rates of Change

The common model, ACL, and inner language are also useful because they allow the developer to stratify the change inherent in the development and maintenance of software systems. If one has designed the agent system well, then the model translation should be changed rarely if at at all, since this is the most difficult and fundamental change.

The set of performatives that comprise the ACL may change more often than the model translation and are, in an agent system, easier to change. Modifying an existing performative, or adding a new one, means changing the parsing of the syntax and possibly calling new functions. A well-designed set of performatives, for the class of applications, will change infrequently.

The inner language provides a more domain-task oriented language in which most of the development and maintenance change for a given application can occur.

As illustrated by Figure 3, changes in the translation into the agent model are fundamental. The degree of difficulty in managing such change in the agent system can be severe. However, typically, if the modeling has been done in an intelligent fashion, these changes are rare.

Since the ACL determines the kinds of interactions that can occur, changes in the ACL may be major, especially if a new performative is introduced. A well-designed ACL will introduce such change infrequently, though inevitably. The inner language provides a layer where change is more frequent but can be handled with less difficulty, realizing no change is trivial in any programming endeavor. In the last two cases, change

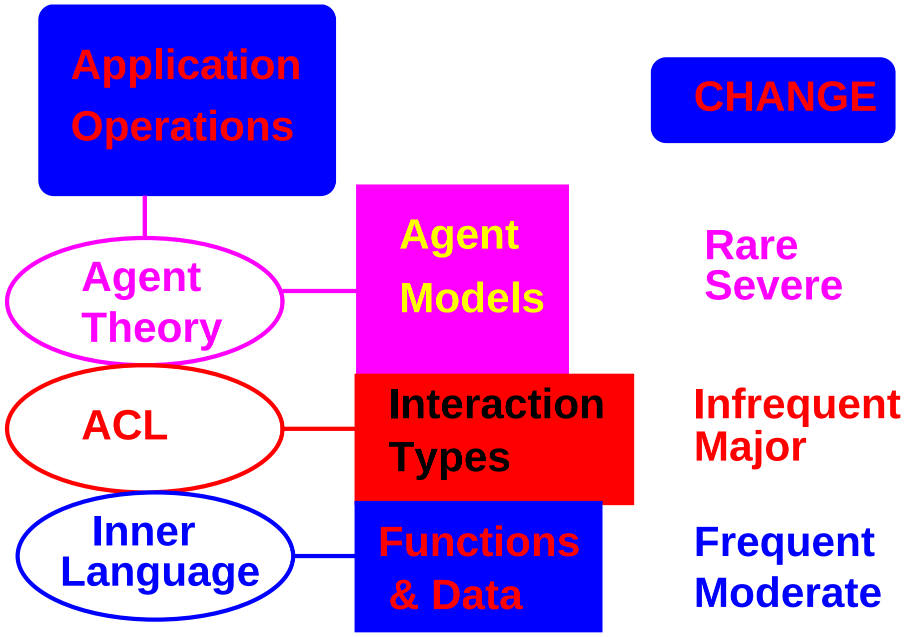

Fig. 3. Change Stratification

is largely communicated by changing the syntax of the languages - either in keywords, values allowed, or more complex changes.

ABSE is thus largely based on the syntax of the ACL and inner language. This means, unlike most other types of software, that message parsing is an intensive run-time computation. Apart from performance, this has a significant impact on distributed development.

Our own development methodology for ProcessLink reflects this. Each of the members of the development team "owns" and agent and may decide independently to change its functionality. However, the interface to each agent is defined by the set of messages it sends and receives and is documented on the web, which messages collectively describe the ProcessLink EPL. This means that the syntax of the ACL or the inner language is changed. Changes to the ACL are usually discussed beforehand in the group as these constitute part of the "jointly-owned" language. Changes to the inner-language are done first and the other team members notified of the change by email, with instructions to see the new web documentation.

2.4 Message-based Collaborative Change and Debugging

Debugging tools for agent systems are generally lacking with a few exceptions. On the other hand, the message-based nature of ABSE naturally provides some debugging functions. This is especially true for applications such as ours that is characterized by asynchronous message exchange among the software agents in which the response time may range from one second to over a day because the software agents depend heavily upon human response time and analysis. This is in contrast to agent applications in which the agents interact in real time without human intervention. The debugging characteristics described below may not be as useful in such applications.

There are three main aspects of agent messages that facilitate ABSE:

- Programming collaboration can occur at a common language level without requiring a common programming language.
- Application-level error messages are not generated by the programming language compilers (though this of course also occurs) but by the agents and their parsers, which provide useful information and quick fixes.
- With a store-and-forward message routing mechanism, message-based interactions provide new methods for debugging.

In ABSE, the programming of individual agents is well-separated from the collaborative programming. Individual agents may be written in the conventional programming language of choice - anything from Java to Smalltalk to Lisp. The collaborative programming is not done in any conventional programming language but rather in the shared ACL and inner language inherited and then refined during development using parsers. This means that there are no compiler constraints or messages of any sort generated by the system except for parser and network software failures, which are not at the level of the shared collaboration. *At the collaborative level, all messages are determined by the programmers,* including error messages.

If another team member does not change their parser to reflect the new syntax, if required, than an important feature of the language-based nature of ABSE is called into play. The parser of the agent receiving the new message will reject the message as being syntactically incorrect. It will return a standard "ERROR" message that includes the string that was not understood. For instance, one can type almost any string and send it to Redux and receive an error message with no code breakage. If one simply types a message with the performative "strange", one receives in reply an "ERROR" message with the text, in the inner language, "Performative 'strange' was unknown to Redux." Or the particular field in the inner language may be wrong: "ERROR RdxType was -*strange*" or even a particular syntax element was wrong: "ERROR in content parsing: content did not end with &. Last value parsed was 'weird'." Or "ERROR in content parsing: no match on any keyword for token 'badsyntax'." These last two illustrate how easy it is to get helpful information almost for free from the parser. It is also easy to have the agents note whether it was an ACL or inner language problem and include other helpful information. The agents do not break and the developers have a clear idea at runtime of the problem.

Some syntactic errors may also be treated the same way as are requests that cannot be fulfilled. Given a message that requires a response or an action, an agent may reply with a "SORRY" message indicating that for some reason, which may be given, the agent cannot properly process the message. This may be because of task data values, the state of problem solving, or even an unrecognized performative. For example, Redux may tell a designer, in the text part of the inner language, that the request to make a particular design decision cannot be fulfilled: " Sorry, this decision is inadmissible. Cause: the fact (IOU-2 is unavailable) is believed." A set of other examples follows:

- SORRY: Process-Controller cannot achieve goal of “schedule-meeting” Constraint-Manager reports “over-constrained”
- SORRY: Ticket-Agent does not understand query “best cost”
- SORRY: You do not have authority to reject decision Use-DOU. Only Optical-Designer can make that change.

Again, the agents do not break, the developers are provided with valuable information at runtime, and the agents can be easily modified to include as much information as desired in such problem messages.

There are many other examples of the ways in which agent messages are valuable for debugging and which use the standard ERROR and SORRY messages. Obviously, because these problem messages are part of the agent software, developers are free to include as much information as may be desired. Because the messages are parsed, providing such information is facilitated.

Agent messages also have other attributes that contribute to debugging. For instance, agents may be programmed to decide not to process a message because the the sender does not have the right authority. For example, Redux may tell a designer, in the text part of the inner language, “ Sorry, but you do not have the authority to reject this decision.”.

This is easy to determine since “SENDER” is a standard part of ACL syntax. Similarly, if there is a problem in even finding the correct name of the agent to be contacted, it is a simple matter to interpret the SORRY or ERROR messages from the agent nameserver, request a list of all valid agents, and then use that information to find the right agent. These sorts of actions are quite typical of agent development, apart from any notions of automatic agent message routing based on content. The point here is that it is very easy to produce agents that degrade gracefully and provide as much information as desirable for development.

The message-based characteristic of agents can also help with debugging if a store-and-forward infrastructure, like that of the JATLite agent message router[8], is used. If a particular agent is having difficulty processing a sequence or set of messages from other agents, those other agents can be taken off-line and the set of messages replayed until the behavior of the agent is correct. We have found this capability to be very useful for debugging and it is dependent upon the unique message-passing nature of agents.

The message-based nature also facilitates runtime error correction. The engineering applications addressed by ProcessLink are long-lived and agent software may die or move during the project, which is characterized by asynchronous communications about design changes over a period of weeks. Errors in message processing are handled the same way as other agent errors or changes. The offending message is rejected by the agent with a “SORRY” or “ERROR” message that typically results in a message being sent to the developer of the sending agent.

At this point, the developers can decide whether the problem is in the sending agent, the receiver, or perhaps another. Now a change is made. If this should result in a different message being sent, the receiving agent simply deletes the last message from the message queue, maintained by the JATLite router, and awaits a new message. If the change is in the way the receiving agent should process the original message, it simply reconnects to the router using the new code and re-processes the original message, which has not been deleted. The computation then proceeds. There is no great

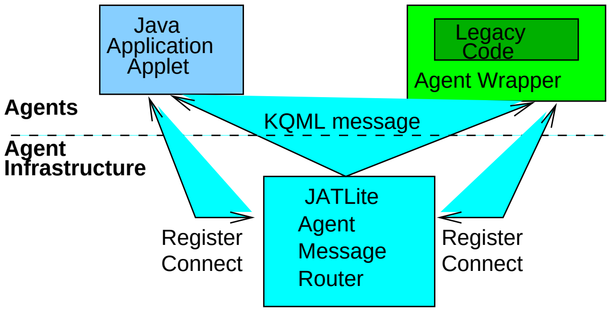

Fig. 4. JAT Agent Message Router

difference between correcting an error during development and during the application execution.

There is another important debugging benefit that can be derived from the asynchronous feature of agent messages if a JATLite-like store-and-forward router is used. Figure 4 illustrates the store-and-forward mechanism of the JATLite Agent Message Router (AMR). All messages go through the AMR and are then sent to the designated recipient. The messages are stored until the recipient signals they should be deleted. Since all messages go through a central point, one can use this characteristic for debugging.

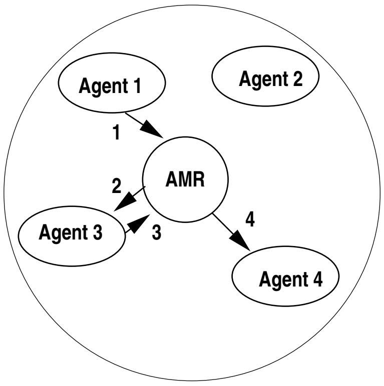

Fig. 5. Single-Stepping Message Delivery

One can build an administrative "agent", as illustrated in Figure 5, that displays the sending and receiving of messages among the agents. Further, since this agent is really a GUI applet tied to the router, one can use it to order the router to "single-step" message transfer and inspect the messages under a variety of conditions, based on regular expression matching on the ACL fields. This kind of debugging is important for system development but is only possible with a store-and-forward router.

2.5 Programming Environment

Since the models themselves are, or should be, general and ubiquitously applicable for many problems, they become a reusable basis for development. A well-designed system also identifies those ACL performatives that can be used largely unchanged not only for the life of a particular application but also for some large class of applications. This set of reusable agents and language components provides a high-level programming toolkit.

There is a well-known set of such performatives that occur and re-occur in different ACLs such as KQML and FIPA, such as “ASK”, “TELL”, “SORRY”, “ERROR” and “REPLY”. There are others that have yet to become standard. One we have discovered is the ProcessLink performative “TRACE”. This performative, similar to the WARREN performative “monitor”, is a request from one agent to another to track the status of an object: changes in values of particular properties or any at all. This is particularly useful for easily interfacing agents. Here is the (partial) BNF for the legal inner language component of this particular
performative:

```
:content := ``(''
   RdxType ``|'' <string> ``&''
   TargetName ``|'' <string> ``&''
   (AgentToNotify ``|'' <string>``&'')
```

where “RdxType” can be one of several types of objects in the Redux ontology. Other agents can respond with information about objects in their own ontology.

As an example in our application, the Redux agent tracks the validity of “constraints” while the Constraint Manager performs constraint satisfaction on those same constraints. Therefore, Redux should tell the Constraint Manager each time the validity of a constraint changes, among other things. Redux could be hard-coded to do so, requiring not only a check to see if there was an agent named “Constraint Manager” in the application, but also requiring that there be only one such Constraint Manager with the special name.

By using “TRACE”, more than one Constraint Manager can dynamically make a request to Redux to track constraints, with no special hard coding other than the general ability to do the tracking. The Redux agent simply notes the name of the requesting agent and adds the request to the list of things to track. Notifications are then sent to the requesting agent as the object properties change with subsequent messages and computations. Adding a new agent means only that the new agent should send Redux a “TRACE” request. Integration does not require rewriting any Redux code. Notice also that both the characteristics of using an ACL and of agents having names make this capability possible.

The other characteristic of multi-agent systems that makes this kind of integration work is that the agents themselves are reusable. The Redux and Constraint Manger agents are two examples of generic agents that provide general functions useful for a large class of engineering applications. Thus a “library” of reusable agents is acquired. WARREN provides another good example of highly reusable generic agents for a large class of applications. However, unlike, say FORTRAN subroutines, the functionality of these software components is quite complex and they can volunteer messages to other software components and users unexpectedly.

3 Summary

We have presented a set of characteristics that may be used to objectively differentiate ABSE from other software technologies. One objection might be that the determination of whether a theory or a language is an "agent" theory or language is not objective. We appeal to the previous definition of AI. Agent theories and languages are those that are explicitly defined to be such by their developers. But we can do even better for agent languages: we can specify that they consist of a typed outer language (with specific characteristics such as explicit representations of the the sender and receiver and errors) and an inner content language, and that the communication protocol be peer-to-peer as previously discussed.

We have not done the differentiation with respect to other software technologies explicitly. We realize that ABSE can be compared to object-oriented programming, for instance, in which messages are also passed via the Internet to distributed code components. Beyond saying that agent messages have the characteristics described that distinguish them from remote procedure calls, which object messages resemble, we leave the detailed comparison of characteristics to experts in all of the other various software engineering technologies. Our goal was only to make such a comparison possible by providing agent software technology characteristics that can be objectively determined. Coverage is outside the scope and expertise of this paper.

We do note that a software technology is defined and differentiated by its commitments. ABSE is characterized by *agent theory modeling, an ACL, and an inner language*. It is defined further by the features of those languages, including the reusable but changeable ACL and the fact that these are text messages handled by agent parsers, as well as requiring a peer-to-peer protocol. Parsing is indeed critical to the utility of using these languages, as described above.

An ACL also has a kernel of reusable performatives, such as "ASK" and "TELL" together with a standard syntax and fields, such as "SENDER", "RECEIVER", and "ONTOLOGY" that reflect particular commitments, such as always declaring the identity of the sender and committing to a conversational mode of interaction that allows any agent to refuse processing of any message and providing for "ERROR" and "SORRY" messages. The import of the latter is that, unlike most software development systems where such errors are incorporated into a lower level of the infrastructure, these messages are generated by the software agents and provide very useful information for system development.

Certainly any programming language and methodology could be used to develop a system with these same commitments, perhaps easily. For instance, one could use the notion of object-oriented programming in combination with agent modeling and language parsing. But if the commitments above are part of the methodology, then we have ABSE.

We have speculated here about the utility of ABSE based upon our experience. We find the useful features of the ABSE characteristics to be:

- use of agent modeling to reduce the complexity of managing the emergent collective behavior of the agents.
- programming guidelines provided by the agent models and the ACL,

- stratification of change into model translation, the ACL, and the inner language,
- programming collaboration based upon asynchronous text message parsing, including error messages, and
- programming tools based upon reusable agents and language components.

Model translation provides both a guide to the developer and an anchor for that part of the software that should change the least. The use of parsed languages for interaction provides for late runtime evaluation of the software that changes the most. Error messages are particularly useful in this regard. And the use of asynchronous messages, together with this parsing, provides for a new mode of collaborative development.

ABSE is probably also a prototyping technique. Once an agent system is working, probably the system would be more efficient if it were compiled into a standard programming language such as one of the object-oriented languages. The agents become objects that pass methods to one another, with no need for an ACL or inner language, or any of the standard syntax and fields, or the required parsers. Parsing is an interpretative computation and once the languages and agent interactions are fixed, there is no reason why these should not be converted into code. Indeed, ABSE is waiting for a production compiler as envisioned by Shoham in the early 90's.

Acknowledgments

The ProcessLink experience is based upon work with Prof. Mark Cutkosky, Theresa Webster, Heecheol Jeon, Sigrid Goldmann, and Andreas Raquet. Barbara Dellen and Harold Holz also contributed to these ideas, Holz being especially responsible for noting that ACL fields such as "SENDER" are a special commitment of agent systems.

References

1. Decker, K., Sycara, K., Pannu, A., and Williamson, M., "Designing Behaviors for Information Agents," **Proc. First Internat. Conf. on Autonomous Agents (AGENTS-97)**, February, 1997.
   See also http://www.cs.cmu.edu/∼softagents/papers/autoag96.ps.gz.
2. R. Elio and A. Haddadi, "Abstract Models and Conversation Policies", **Proc. Workshop on Specifying and Implementing Conversation Policies**, Autonomous Agents'99 Conference, Seattle, May, 1999. See also http://www.boeing.com/special/agents99/elio_a.pdf.
3. B. Faltings, O. Belakdar, S. Willmott,"Intelligent Agents", CS course at EPFL, Lausanne, 2000. See also http://nrc.stanford.edu/∼petrie/agents/falt.html.
4. M. Genesereth and S. Ketchpel, "Software Agents"qq, *Communications of the ACM*, **37** (7), July 1994. See also http://logic.stanford.edu/papers/agents.ps.
5. M. Huhns, "Interaction-Oriented Programming", in this volume.
6. N. R. Jennings, "On Agent-Based Software Engineering", *Artificial Intelligence* **117** (2000), pp. 277-296, Elsevier Press, April, 2000.
7. N. R. Jennings, and M. Wooldridge "Agent-Oriented Software Engineering" in **Handbook of Agent Technology**, (ed. J. Bradshaw) AAAI/MIT Press, 2000.
   See also ftp://ftp.elec.qmw.ac.uk/pub/isag/distributed-ai/publications/agt-handbook.pdf.
8. H. Jeon, C. Petrie, and M/ R. Cutkosky, "JAT*Lite*: A Java Agent Infrastructure with Message Routing," *IEEE Internet Computing*, Mar/Apr 2000. See also http://www-cdr.stanford.edu/ProcessLink/papers/jat/jat.html.

9. K. L. Myers and D. E. Wilkins, "The Act Formalism", Version 2.2, SRI International Artificial Intelligence Center, Menlo Park, CA, September 1997.
See also http://www.ai.sri.com/∼act/act-spec.ps.
10. D. Parks, "Agent-Oriented Programming : A Practical Evaluation", http://www.cs.berkeley.edu/∼davidp/cs263/, 1997.
11. C. Petrie, "Agent-Based Engineering, the Web, and Intelligence", *IEEE Expert*, December, 1996. See also http://cdr.stanford.edu/NextLink/Expert.html.
12. C. Petrie, S. Goldmann, and A. Raquet, "Agent-Based Project Management", **Lecture Notes in AI 1600** pp. 339-362, Springer-Verlag, 1999.
See also http://cdr.stanford.edu/ProcessLink/papers/DPM/dpm.html.
13. A. S. Rao and M. P. Georgeff, "BDI agents: From theory to practice," Tech. Rep. 56, Australian Artificial Intelligence Institute, Melbourne, Australia, April 1995.
14. Y. Shoham, Y., "Agent-Oriented Programming" *Artificial Intelligence*, **60** (1), pp. 51-92, 1993.
15. "Control and Coordination of Multiple Agents through Decision Theoretic and Economics Methods", project. See also
http://coabs.globalinfotek.com/coabs_public/project_overview_pages/stanford.html.
16. A. Tate, B. Drabble, and J. Dalton, "The Open Planning Architecture and its Application to Logistics", in **Advanced Planning Technology** (ed. A.Tate), pp. 257-264, AAAI Press, 1996.
See also ftp://ftp.aiai.ed.ac.uk/pub/documents/1996/96-arpi-oplan-and-logistics.ps.
17. G. Wagner, "Agent-Oriented Analysis and Design of Organizational Information Systems", in **Proc. of Fourth IEEE International Baltic Workshop on Databases and Information Systems**, Vilnius (Lithuania), May 2000.
See also http://www.inf.fu-berlin.de/∼wagnerg/BalticDB2000.ps.
18. The DARPA/Rome Laboratory Planning Initiative,
http://www.ai.sri.com/∼wilkins/mpa/tie-97.html.
19. M. Wooldridge, and N. R. Jennings "Software Engineering with Agents: Pitfalls and Pratfalls" in *IEEE Internet Computing,* **3** (3), May/June 1999.
20. M. Wooldridge, "Verifiable Semantics for Agent Communication Languages", in **Proc. of ICMAS98**, pp. 349-356, IEEE Computer Society Press, May, 1998.