Multiagent System Development Kits: An Evaluation

Elijah Bitting, Jonathan Carter, and Ali A. Ghorbani
Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada

ABSTRACT

The complexity associated with development of Multiagent Systems (MAS) may often be reduced greatly by the use of existing Multiagent System Development Kits (MASDKs). These kits provide the MAS developer with a wide range of powerful agent and MAS building tools, specifically designed to make the task of agent creation easier. The use of a MASDK allows the developer to forget about specific and tedious low-level implementation issues and focus on the desired functioning of the agents themselves.

Few would disagree that these MASDKs are very useful for developers, but which MASDK is best for each of the countless application areas that exist in MAS research? Which MASDK provides the best tools for communication? Which makes the task of creating a distributed MAS easiest and which are even capable of facilitating the creation of such MAS? These questions and others are the focus of this work. Four top MASDKs are discussed and evaluated according to specific evaluation criteria. These criteria are divided into the following five categories: a) Basic attitudes, such as sociability b) Advanced attitudes, such as meta-management and mental attitudes c) Software Engineering support for traditional Software Engineering concepts like architectural description and prototyping d) Implementation issues such as the specific language used and debugging facilities available, and e) Technical issues such as mobility, concurrency and cost.

The MASDKs evaluated are listed here in descending order by their final evaluation results: AgentBuilder, Jack, AgentSheets, and OpenCybele.

Categories and Subject Descriptors

D.2.6 [**Software**]: Software Engineering—*Programming Environments*; I.2.11 [**Computing Methodologies**]: Artificial Intelligence—*Multiagent Systems*

General Terms

Performance, Measurement

Keywords

Agent, multiagent systems, multiagent systems development kits

1. INTRODUCTION

Although several different definitions of multiagent systems exist [18] [12] [11], we chose the following formal definition for the purposes of our work. A multiagent system is one comprised of the following elements: an environment E, a set of objects O, an assembly of agents A, an assembly of relations R, an assembly of operators P, and a set of universal laws L [6]. The environment must be virtual in nature. Objects are defined as things within the environment that can be acted upon and situated within the environment. Agents are a subset of objects that represent the active entities within the system. Relations link objects to one another while operations allow an agent to perceive and manipulate other objects. Lastly, the universal laws represent the application of operators and the reaction of the world to these operators.

Due to the immaturity of the field, several differing definitions of agents can also be found [7, 17, 8]. We choose to use Ferber's general definition of agency [6]. An agent must possess the following characteristics: capacity to act and communicate with other agents, perceive an environment, utilize resources, and be driven by a set of tendencies. The last requirement is a critical factor that distinguishes agents from common objects.

In order to develop a MAS, researchers have several options. Often, they design an environment from scratch which is highly specialized to their research that simulates the phenomena they're studying. In designing the environment, one must consider the definition of the agent and incorporate that definition into an environment that fully supports it. As noted above, there are several different definitions and perspectives of agenthood. Hence, a large number of MAS possibilities result that incorporate these perspectives in as efficient a manner as possible.

Based on our experiences, the environment must be able to efficiently represent the complex internal architecture of an agent. It must address issues of interagent communication (format and protocols), agent mobility (distribution), resource allocation, inference mechanism support for agents, agent knowledge representation, and scalability. There are more issues as well [16].

Researchers have another option when it comes to developing a MAS. They can chose to use a multiagent system development kit (MASDK) that addresses many of the concerns outlined above [5, 2]. In such kits, it is important to still have the flexibility to modify the internal functioning of the underlying architecture to give the researcher maximum control and freedom.

An initial search for existing MASDKs was undertaken only to find over 40 available [5]. We have selected four MASDKs for evaluation. Three of the four MASDKs were chosen according to popularity ratings found at the Agentland web site [2]. AgentBuilder [1], AgentSheets [3] and Jack [9] were selected for evaluation. OpenCybele, although not rated at Agentland, has been used for the creation of MAS testbed environments based on historical consideration of previous work. Once the four were chosen, we set about using the packages and reading their documentation in order to perform a comparative evaluation.

An evaluation of MASDKs cannot be done by simply using a system for a few hours and giving an opinion. A set of criteria must be developed upon which the evaluation will be based for comparison. Using accepted criteria for the evaluation lends credibility to the results and simplifies the evaluation process. Following these principles, we chose to use criteria already developed by Austrian researchers Eiter and Mascardi [5]. The criteria chosen fall into five categories, each of which are more thoroughly described below.

Sociality: Aspects concerning sociality are central to most work related to MAS. Interactions are facilitated by the use of an accepted agent communication language (ACL) and facilities for high-level communication [19].

Advanced Attitudes: Advanced attitudes include such constructs as: mental attitudes, deliberate capabilities (planning), meta-management (reasoning about the self), and adaptivity [19].

Software Engineering Support: Criteria associated with this aspect of evaluation are useful for the evaluation of any software development package. Sound use of software engineering principles promotes the creation of higher quality software [10, 15, 13]. Furthermore, this eases the task of software development for all involved. MASDKs are evaluated according to the extent to which they provide the following: clear development methodology, architectural description, internal functioning of a single agent, documentation, prototyping and simulation capabilities .

Implementation: This dimension of evaluation is concerned with the specifics of the implementation of a development kit and how it implements the MAS it creates. The criteria are based on how the MASDK supports: agent implementation language, the integrated development environment (IDE) GUI for agent and MAS definition, agent skeletons, and debugging [16].

Technical Issues: This category of evaluation criteria concerns itself with important aspects of MASDKs which are not covered by any of the above categories. Included in this section are the following: mobility, concurrency, real-time control, and economic cost.

The following sections present a brief overview of each of the MASDKs that were evaluated. Each MASDKs philosophy of design and agent definition is presented.

1.1 AgentSheets

AgentSheets is an agent-based simulation tool that lets end-users create simulations within an easy-to-use GUI development environment. Through this environment, the focus shifts from agent implementation issues (such as communication methodology and resource allocation) to agent design issues. The developer focuses on the rules and behaviors of the agent and their net effect within a society of these agents. In essence, users can focus on emergent behaviors rather than individual agents.

AgentSheets implements the society through a combination of Java authoring tools, spreadsheets, and agents. Agents are defined as reactive, end-user programmable objects. They react to external events such as mouse clicks, keyboard input, and messages from other agents. This definition is highly simplified and does not address planning, intelligence, mental capacity, and meta-management. Through this definition, agents are seen as a collection of methods. Each method corresponds to one external event. Methods are composed of rules. Rules are if-then-else clauses that execute actions based on some set of condition statements. Methods, rules, conditions, and actions are all predefined structures within a GUI palette and they are drag-and-dropped into agents by the user. The user is responsible for filling in the corresponding parameters.

AgentSheets utilizes the concept of *tactile* programming. Tactile programming is defined by AgentSheets as the next logical step of object-oriented development. In this model, agents are visually developed and must be graphically depicted within their environment. The emphasis is placed on visual development and the communication of the underlying rules, behaviors, and actions of agents during real-time simulation. The programming becomes tactile because of the heavy emphasis on GUIs for developing and manipulating agents.

1.2 AgentBuilder

AgentBuilder makes the task of creating ‘intelligent’ agents a relatively easy one. While slightly more complex to use than other packages reviewed (AgentSheets for example), AgentBuilder is capable of facilitating the production of autonomous, environmentally aware, reasoning, communicating agents, as well as collections or agencies of these agents.

AgentBuilder provides a powerful mental model for its agents, allowing developers to easily specify such things as beliefs, intentions, commitments, and behavioral rules. In addition, use is made of popularly accepted standards such as KQML for the agent communication language along with use of UML-like object modelling facilities to model domain knowledge of an agent. The Knowledge Query and Manipulation Language or KQML is both a language for comunication amongst intelligent agents and a network programming protocal [1].

To ease the agent development process, AgentBuilder provides GUI interfaces for almost all agent development tasks such as: overall project organization, protocol specification, defining agency layout, examining (watching) running agencies, specifying agent behavior, creating running agents, and debugging. These are all tied together under a simple IDE within a Windows environment.

The extendability of AgentBuilder is largely due to its use of external classes, called Project Accessory Classes (PACs). The agent relies on PACs for storage, retrieval, processing, and communication of data. The belief set is composed of PACs as well. These PACs can be written in Java, C++, or legacy languages.

1.3 Jack

Jack Intelligent Agents is a development environment that is built on top of Java and acts as an extension of Java that offers classes for implementing agent behavior. Jack defines its agents as being 'intelligent'. These types of agents model reasoning behavior according to the theoretical Belief-Desire-Intention (BDI) model of artificial intelligence.

When an agent is instantiated in the system, it can do several different things. It can respond to goals that have been explicitly set by another agent or the user. It can also respond to external stimuli that are represented as events. In both cases, an agent will respond if it believes the event or goal has not been satisfied and is appropriate to respond to. Upon receiving the event, the agent searches through all available plans for ones that are relevant to the request and applicable to the situation. Actions within a chosen plan are then executed. At the same time, the agent is able to focus on what is important and the planning process. The BDI model views agents as collections of plans that are executed under specific conditions.

1.4 OpenCybele

OpenCybele uses a different approach to agency than other packages evaluated. Agents are defined as "a group of event-driven activities that share data, thread, and execution concurrency structure." This definition centers the agent on the notion of events triggering activities (stimulus-response). Activities in this case are 'active objects' internal to the agent and act on internal data in response to incoming events. The fact that other agents are incapable of manipulating the internal data of another reinforces the notion of agent autonomy. Instead of directly manipulating another's data or calling an agent's function, OpenCybele agents are limited to generating and delivering events that may or may not trigger the desired response in the other agents of the MAS.

Programming OpenCybele agents requires that developers use Activity Centric Programming (ACP). ACP (described in detail in the OpenCybele user manual) concentrates on agent activities as the core components of a system. This can be contrasted to the traditional functional approach where functions and function calls are the core components of a system.

Cybele is less friendly for the novice user than the other packages evaluated. It requires a substantial knowledge of object oriented programming concepts that are used heavily throughout this approach to agents. In addition, the agent execution states, associated inter-agent concurrency relationships, and state transitions are very confusing without thorough study. A possible reason that this system seems more difficult to learn and use in comparison with the other MASDKs evaluated so far is that OpenCybele is not a MASDK but an agent infrastructure. OpenCybele provides most of the facilities and functionality of a MASDK without the IDE or GUI environment.

2. EVALUATION CRITERIA

In order to obtain objective results from the evaluation described here, we have devised a weighting scheme that utilizes the results summarized in Figure 7. This weighting scheme first ranks the individual criterion on a scale of 1 to 3. A ranking of 1 being a criterion that is *desired* to be included in the system but not necessary or useful. A ranking of 2 means that the criterion is desired and considered *useful*. A ranking of 3 designates this criterion as *necessary*. This weighting can be expressed as a weight vector $\overline{W}$, where

$$\overline{W} = \{w_1, w_2, ..., w_n\} \tag{1}$$

and w_i represents the rank of criterion i. Figure 1 shows our assignment of relative importance to each of the chosen criteria. Note that the agent language and overall cost criteria are not imcluded in this weight vector as no meaningful weight can be asisgneed to them.

Once the ranking of the chosen criteria is complete, a calculation is performed on the basis of the ranking of the individual criteria and the results of the evaluation itself. In evaluating the desired MASDKs by applying the selected criteria, results in the form {supported, partially supported, and not supported} are obtained. We assign the following percentages: supported equals 100%, partially supported equals 50%, and not supported equals 0% to each criterion. From these percentages, we create another vector similar to $\overline{W}$, called $\overline{S}$ where

$$\overline{S} = \{s_1, s_2, ..., s_n\} \tag{2}$$

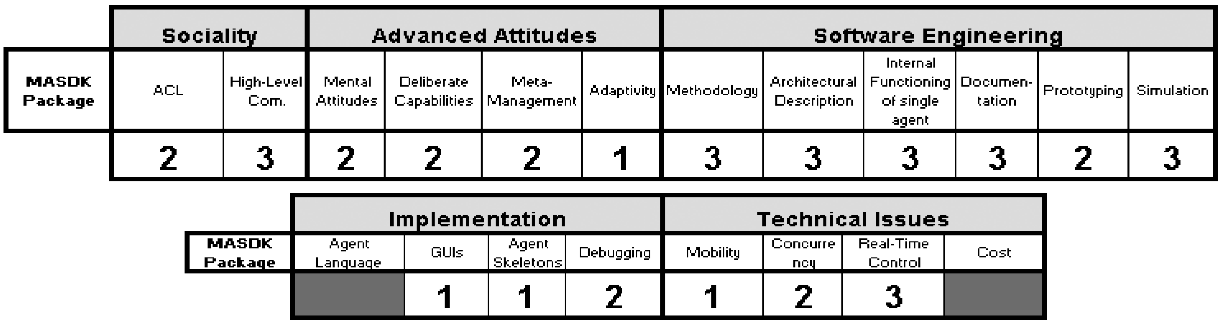
and s_i is the fulfilment percentage associated with criterion i.

Using $\overline{W}$ and $\overline{S}$, we calculate a numeric value associated with each MASDK evaluated. We perform the simple dot product $\overline{W} \cdot \overline{S}$. It results in a scalar value that measures the overall quality of the MASDKs evaluated. The criterion are presented below.

2.1 Basic Attitudes

Basic attitudes are those which determine the basic functioning of an agent and directly influence its behavior. Theorists use these basic attitudes to describe what constitutes an agent. Autonomy, reactivity, and sociability are considered basic attitudes. The problem with discussing these attitudes is that there is no consensus as to what combination of these attitudes constitute agenthood [5]. As a result, only the most fundamental are singled out for use as evaluation criterion. Sociability is considered essential for a MAS because the lack of social interaction amongst agents prevents collaboration of any sort. The resulting MAS is reduced to a collection of independent agents.

Sociability is considered a basic attitude of autonomous agents along with such attitudes as reactivity and autonomy [19]. Sociability is singled out as an important criteria as socialization lies at the heart of most MASs. We consider two aspects of sociability here: the use of an accepted agent communication language (ACL), and the support of high-level communication between agents. The use of an ACL is considered because the use of a standard ACL like Knowledge Query and Manipulation Language (KQML) or Foundation for Intelligent Physical Agents (FIPA) can facilitate the acceptance of a MAS. This facilitates the development of complex communication facilities.



	Sociality		Advanced Attitudes				Software Engineering					
MASDK Package	ACL	High-Level Com.	Mental Attitudes	Deliberate Capabilities	Meta-Management	Adaptivity	Methodology	Architectural Description	Internal Functioning of single agent	Documen-tation	Prototyping	Simulation
	2	3	2	2	2	1	3	3	3	3	2	3

	Implementation				Technical Issues			
MASDK Package	Agent Language	GUIs	Agent Skeletons	Debugging	Mobility	Concurre ncy	Real-Time Control	Cost
	▨	1	1	2	1	2	3	▨

Figure 1: Weight Vector W (the relative importance of each of the chosen criteria; 1=desired but not necessary, 2=useful and 3=necessary).

The ability to exchange high-level information is important for similar reasons. High-level communication allows for an abstraction of low-level communication system details (such as detection of message loss), providing the developer with a simplified transparent communication system.

2.2 Advanced Attitudes

These attitudes go beyond the realm of the basic attitudes described above. Advanced attitudes allow the agent to develop behaviors that are more complex and powerful than the basic attitudes that define agenthood. The advanced attitudes selected as evaluation criteria allow agents to improve their reasoning methodoligies by expanding the factors they consider for decision making [19].
Mental Attitudes: A system supporting mental attitudes would have to include support for such things as commitments, beliefs, intentions, and desires.
Deliberate Capabilities: These are defined as the ability of an agent to act deliberately. These acts include planning in order to satisfy some intention or desire [14, 19]. Planning is useful for almost any AI area, thus making it an important evaluation criterion for MASDK evaluation.
Meta-Management: Meta-management involves the ability of an agent to reason about itself. Such agents must maintain some sort of model of themselves and their actions and beliefs. An agent with these capabilities is said to be reflective.
Adaptivity: Agents who are able to change in order to adapt to a situation or particular environment are said to be adaptive. This adaptation, in a broader sense, is learning. Agents with learning mechanisms are said to be adaptive.

2.3 Software Engineering Support

Software engineering issues are considered important evaluation criteria for MASDKs. Software engineering projects should make use of established software engineering methodologies for consistantly more reliable, correct, and readable code [10, 15, 4, 13]. This dimension of evaluation is the most practical from a development perspective as it concerns itself with the software development life cycle. The general issues surrounding this cycle include the following: methodology, architecture, documentation, debugging and simulation. Issues concerning the documentation of agenthood and agent architecture are also added.
Methodology: Methodology is a clear sequence of steps laid out in order to build a simple agent or agency. The presence of such a development methodology eases the development process and greatly reduces the learning curve for MASDKs. Whether or not the MASDK provides a clear methodology is a point worth consideration while evaluating MASDKs for comparison.
Architectural Description: It is described thorough documentation of the actual architectural design of the development kit in terms of class structures, functional description, and services provided. This information is extremely valuable to anyone wishing to make extensive use of a MASDK for serious development.
Internal Functioning of Single Agent: Some MASDKs provide only the most basic description of how the individual agents operate. It is necessary to provide a detailed description of the agent model, as well as how that agent operates in the MAS environment. Exactly how, and in what order an agent determines the examination of behavioral rules at any time constitutes the internal functioning.
Documentation: For obvious reasons, the provision of meaningful and useful documentation of the MASDK specification is invaluable to the agent developer. Quality documentation can make the difference between a great MASDK and one that is unusable.
Prototyping: Providing standard functional agent definitions, or even standard agencies for common useful applications is referred to as prototyping. The MASDK that makes use of prototyping will provide one or more re-usable prototype definitions.
Simulation: Simulation in this situation refers to the ability of a development kit to provide facilities for the developer to monitor and record certain specific statistics regarding the execution of an agency. The ability to record the number of interactions between two agents in some time interval would require some sort of simulation facilities.

2.4 Implementation

It is important to examine the peripheral issues related to software engineering because they have a big impact on the final product. The impacts are manifested through development time, underlying architecture, the look-and-feel of the product, and the performance. Below, the issues of software language, integrated development environment (IDE), skeleton code, and debugging facilities are addressed in more detail.
Agent Implementation Language: The agent implementation language is the language in which the code for an agent is written. Some packages use standard languages like Java or C++, while others use extensions to languages like these or proprietary agent programming languages.
Agent and MAS definition GUI (IDE): This is a graphical interface provided to ease the development of agents. Instead of having to write out a list of preconditions for a

behavioral rule, a GUI might provide a list of possible preconditions for the developer to select from. Such a GUI, if properly designed, can greatly simplify and speed up the development of agents and corresponding MAS.

Agent Skeletons: Much like prototypes described above, agent skeletons are simplified pieces of agent-code that represent standard agent roles. Such a skeleton might be a buyer agent, or a utility skeleton. A MASDK that provides some of these skeletons can make some MAS development tasks much simpler.

Debugging Facilities: Debugging capabilities are essential for the MAS developer. Debugging helps to create functionally correct programs. It often allows a developer to see the system in a critical way that is not available without debugging facilities. Facilities could allow a developer of an MAS to watch the values inside the belief system of one of the agents as the agent executes in the MAS.

2.5 Technical Issues

This selection of criteria is related to the availability of services provided by the agent environments created using the MASDK. These services can have a dramatic impact on the time of development as the responsibility of providing these used services is shifted away from the developers. Although none of these services are necessary for a MAS, they are often desired.

Mobility: This criterion specifies if agents developed using a specific MASDK are capable of moving from one system to another while maintaining their capability to interact with the rest of the members of its MAS. Mobility can facilitate development in several specific domains, such as distributed computing and information sharing and gathering.

Concurrency: A MASDK capable of creating agents that execute in their own thread (i.e. are not part of a bigger process) is said to have concurrency. This idea facilitates true autonomy and real time computing. Concurrency also allows developers to make use of multiprocessor environments, as different threads can be processed simultaneously on different processors. Hence, true multitasking is established.

Real Time Control: Closely related to simulation and debugging, real time control gives the developer the freedom to set or change certain aspects of the MAS at their discretion during run-time. This control gives the developer the freedom to manipulate the MAS to reflect a desired situation.

Cost: For most developers, the cost of a development kit is of great importance.

2.6 Economic

Economic issues are important when considering the long-term viability of the MASDK. Due to the volatility of the research and the industry, it is important to seek a MASDK that will have long-term support from the original developers. Corporate consideration must be given to attain the likelihood of long-term support. This section addresses the issues of corporate knowledge, history, and support.

Knowledge of Vendor: Knowledge of a vendor providing a MASDK (or the lack thereof) is an important factor in comparing MASDKs. Developers can be more assured of continued support for a MASDK product from vendors they know to be reputable.

Documentation, Training, and Support: As stated above, documentation is a powerful tool for the MAS developer. Similarly, training and product support are equally important factors for the developer as they may be required to solve problems along the way to developing a MAS.

3. MASDK EVALUATIONS

This section is concerned with the results of evaluations of the MASDKs considered. These evaluations are performed based on the criteria outlined in Section 2. Each section gives first impressions followed by a discussion of the criteria.

3.1 AgentSheets

AgentSheets looks very easy to use and highly simplified yet powerful in certain situations. The rule editor makes it easy to create prolog-like rules. The simulations seem versatile and easy to create and modify. As a limiting factor, the system does not support adequate message passing. This restricts the inter-agent communication and collaboration capabilities.

For the modelling of natural phenomena (reactivity to environmental cues), this system seems very appropriate. It is ideally suited for reactive agents as opposed to cognitive ones.

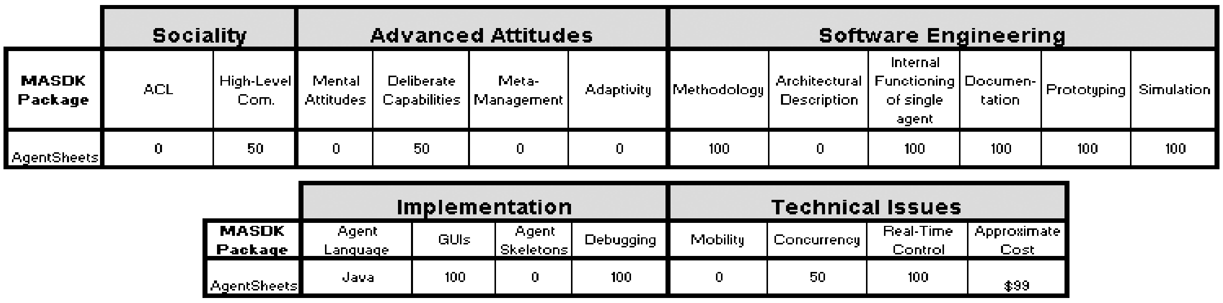
3.1.1 Evaluation

Once again, Java 2 is the choice of language implementation for the development environment. The choice of using an interpretive language for implementation has many implications on the performance of the underlying system. In this case, AgentSheets was slow and required a vast amount of memory in order to run without problems.

This development kit was designed with a non-programmer in mind. Development occurs only through a GUI environment. This dramatically reduces the skills necessary for development as the designer can focus more on design and less on implementation issues. The methods, rules, and conditions are designed through a drag-and-drop interface from the appropriate palettes to the given agent. Naturally, this heavy reliance on a GUI will hinder more sophisticated development related to the internal functioning of AgentSheets.

AgentSheets does not support a default agent communication language such as KQML. This can severely limit the inter-operability of agents generated through AgentSheets with other external agents. Agents communicate with each other through the use of a ‘make’ command. This command is very limited in its functionality. This command will generate a message from one agent to another agent immediately adjacent to it. The message cannot have any parameters and the sender cannot uniquely identify the intended recipient of the message. The event handler of the receiving agent is responsible for then executing actions based on the incoming message. This limitation on message passing is perhaps the greatest weakness of the development environment because it reduces the communicative capacity of the entire society.

No desires, beliefs, or commitments can be directly implemented in any given agent. Agents do have access to a local set of variables. As such, mental states could be represented through the use of the local variables. Furthermore, planning is not accounted for in AgentSheet’s model of an agent. An agent has a very limited perception of its environment and goals. It can only respond to external stimuli. In that sense, a plan of action can be explicitly hard-coded



	Sociality		Advanced Attitudes				Software Engineering					
MASDK Package	ACL	High-Level Com.	Mental Attitudes	Deliberate Capabilities	Meta-Management	Adaptivity	Methodology	Architectural Description	Internal Functioning of single agent	Documentation	Prototyping	Simulation
AgentSheets	0	50	0	50	0	0	100	0	100	100	100	100

	Implementation				Technical Issues			
MASDK Package	Agent Language	GUIs	Agent Skeletons	Debugging	Mobility	Concurrency	Real-Time Control	Approximate Cost
AgentSheets	Java	100	0	100	0	50	100	$99

Figure 2: AgentSheets Analysis Overview.

into an agent in response to these stimuli. Reasoning about plans, however, is not possible. Meta-management is not automatically included within their definition of an agent. It is possible for an agent to monitor its own internal state through the use of local variables. This would have to be explicitly written by the developer.

AgentSheets does not include any facilities for learning mechanisms per se. Agents can only respond to external signals. Hence, learning mechanisms such as neural networks would have to be explicitly written by the developer. These mechanisms would then have to be somehow related to external stimuli in order to execute.

On the positive side, AgentSheets provides an excellent methodology for developing and deploying agents. The steps to do so are clearly laid out and easy to implement as laid out in the "Getting Started" user manual. This product was designed with the end-user in mind. It does not attempt to describe the internal functioning of AgentSheets from an architectural perspective. Through the use of AgentSheet's Risittero, Java applications can be developed that produce the simulations carried out. Reverse engineering could be applied to learn of the underlying architecture of AgentSheets. No attempt is made to explain this architecture within any of the found product documentation.

AgentSheet's documentation is very thorough (from a non-technical perspective) and includes the following: user manual, language reference guide, a guide to getting started, and an example project snapshot. As well, their web site provides abundant support for their product through tutorial movies, teacher guides, and an agent exchange program. The agent exchange program is a listing of real agents that have been implemented at various universities that serve as bigger examples than those provided with the software by default. Although a FAQ is advertised within their support section, it is unavailable for viewing due to unknown technical problems related to their server.

Within the documentation, internal behavior of an agent is examined in relation to the order of rule firing. Rule firing involves the testing of conditions and executing the corresponding actions within the given method of that agent. It is necessary to examine the rule precedence to predict behavior. AgentSheets does a good job of explaining this and other internal mechanisms that govern an agent's behavior.

Agent prototyping is possible in a very non-traditional sense. As a developer defines an agent, various depictions of an agent are possible. Each depiction represents a graphical representation of a given instance of that agent type. It is possible to create a prototypical agent that has several differing states. As such, each state can have its own graphical depiction. In this sense, a standard functional agent can be a prototype for instances of itself. Prototyping is not possible in the more traditional sense because the behaviors cannot be extended or inherited between agent types.

AgentSheets does, however, do an excellent job of ficilitating simulations. This system appears to be based on the premise that simulations are the goal of agent development. Agents are placed on a graphical worksheet and simulations follow. Users can change the types of agents and surrounding environment in real-time. Due to the heavy focus on a graphical environment, the tasks of starting up, shutting down, and manipulating simulations is very easy.

The simulator does a very good job of allowing real-time changes to data internal to agents and the supporting environment. Global and local variable manipulation is possible. Global variables are those that can be shared between all agents while local variables are those that are private to an individual agent.

Run-time support for debugging is provided in abundance through a graphical user interface (GUI). A user can examine the internal variables of an agent as it runs through cycles. Through this same interface, a user can monitor internal rule firing. The debugging facilities of AgentSheets are perhaps one of AgentSheet's best features.

Agent skeletons are not provided by default. While the underlying code for basic functions of all agents (movement, signal passing, variable storage) is pre-written, it is the user's responsibility to define the functionality of all agents within their environment.

Agent mobility is not possible with AgentSheets. Agents cannot communicate outside the immediate environment. Once again, the lack of agent communication facilities limits the potential applications of AgentSheets.

Although no explicit references to concurrency are made in the user manual, it is possible to implement concurrency between agents through the use of signal passing. Resources can be shared through the sending of signals between agents. Due to the fact that there is no central service to handle resource sharing and concurrency, we decided that concurrency is only partially supported through the developer's own means. Once again, the imposed limitations of agent communication reduce the functionality of the overall system.

3.1.2 *Rating*

Figure 2 presents a graphical overview of the evaluation outlined above. The 100, 50 and 0 values correspond to the level of support for the associated criteria in percent. Based on the previously mentioned marking scheme, an objective measure of the overall MASDK can be calculated in a straight forward manner using 2. AgentSheets scores a value of 23.5 out of a possible 39 points (60.2%).

3.2 AgentBuilder

The first observation made is that the installation procedures for the AgentBuilder MASDK are relatively simple. AgentBuilder is Java-based so slowness is expected. Although slow, the GUI is very good and makes navigating the complex environment fairly simple. Very thorough documentation is available as well as various forms of support online including: frequently asked questions (FAQ), examples, listings of reported bugs, and a mailing list.

AgentBuilder makes use of an extensive agent model. The mental component of the agent model is well developed. AgentBuilder also has powerful environment modelling facilities through the use of the Unified Modelling Language (UML). UML designers should find design of agencies simple within AgentBuilder.

3.2.1 Evaluation

Sociality is of the utmost concern for AgentBuilder; the communication mechanisms provided are versatile and powerful. These mechanisms greatly simplify the task of creating agencies where rich structured data is passed from agent to agent. AgentBuilder implements a Project Accessory Class (PAC) called *KqmlMessage* that includes the following attributes:

```
String sender String receiver String performative
String contentType Object content
```

This PAC embodies the knowledge query markup language (KQML). As a standard in agent communication languages, KQML provides AgentBuilder with a stable foundation on which to build agent communication protocols and systems. KQML provides versatile and powerful message passing. Note that the content attribute of a KQML message is a generic Object reference, possibly referring to any data type. This abstraction of message contents allows for the use of existing (or creation of new) generic communication protocols that are given the power to ignore the actual contents of the messages that will be passed.

Advanced attitudes, as defined above, are supported to various degrees within AgentBuilder. Mental attitudes are not modelled explicitly in the system, but could be implemented as part of the existing mental model. In regards to the advanced attitude of deliberate capabilities or planning, a built-in planning mechanism is not implemented in AgentBuilder. The development kit does, however, support the use of external planning modules. The third advanced attitude considered is meta-management. While AgentBuilder does not explicitly make use of meta-management, the agent belief set includes an agent record corresponding to the SELF of that particular agent. This SELF instance contains the beliefs an agent holds about itself. Through the use of this information and its intentions and other beliefs, agents are given the ability to evaluate their own status and change appropriately. The adaptivity attitude or learning capabilities of agents developed with AgentBuilder are great because of the support for external learning modules. AgentBuilder does not, however, implement any learning mechanisms that developers could easily take take advantage of.

The methodology of building either a single agent or a collection of agents is laid out very clearly in the AgentBuilder documentation. The AgentBuilder user manual provides several chapters that are dedicated to the theory and methods of agent and agency development. In addition, several more chapters are dedicated to step-by-step examples of several different types of agents. This includes several examples of creating communicating agents.

A full architectural description of the system is provided in the reference manual, including a full description of all the subsystems (e.g. Project manager, Ontology manager). In addition to a detailed specification of the AgentBuilder MASDK system, the detailed architecture of an AgentBuilder intelligent agent is provided and discussed at length in the user manual. Diagrams and flowcharts help in the descriptive presentation of this architecture. The AgentBuilder user manual (with regards to documentation) is very useful and extensive. In addition, a reference manual is made available to help developers with rich information concerning nearly all aspects of the system.

Like most other MASDKs on the market, agents developed with AgentBuilder are implemented in Java. In addition, the Java GUIs used by AgentBuilder provide most of the functionality of the MASDK. Some of the GUIs defined are for: defining Agent behavioral rules, specifying protocols, defining agencies, watching executing agencies, and debugging. AgentBuilder allows the developer to debug the running agent by examining its mental model. The debugger can perform standard debugging tasks like setting breakpoints and stepping through agent actions. The debugger provides the developer with some limited real-time control of agencies. Execution can be stopped, values changed, then execution can be halted, resumed, or a step-by-step run can be performed.

The AgentBuilder documentation contains no indication that agents built with AgentBuilder are capable of migrating from one system to another. If such a capability exists for AgentBuilder agents, it is not documented. Similarly, no documentation indicating support for concurrent execution of agents in AgentBuilder was found.

3.2.2 Rating

Figure 3 shows a summary of the evaluation results outlined above. Based on the previously mentioned marking scheme, an objective measure of the overall MASDK can be calculated using the assigned weights from Figure 1 and the individual evaluation results for each MASDK. AgentBuilder scores a value of 31.5 out of a possible 39 points (80.8%).

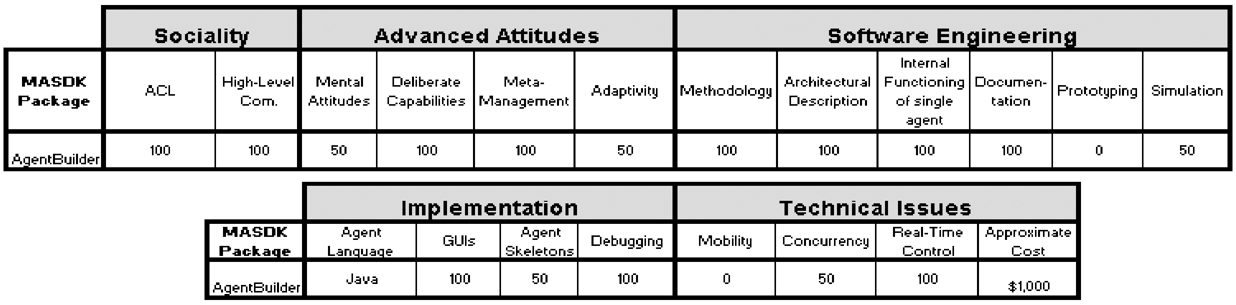
3.3 Jack

The first thing we noticed was that Jack was another Java-based MASDK, which means slow.

3.3.1 Evaluation

Although JACK does address the issue of inter-agent communication and provides a communication network, it does not make use of any pre-existing standard agent communication language.

Standard message passing through the JACK communication layer or ‘DCI network’ is used. The DCI network runs as a network layer just over the transport layer and uses UDP as a transport protocol by default. Because UDP is connectionless and does not guarantee packet delivery, DCI provides a guarantee of delivery on messages.



	Sociality		Advanced Attitudes				Software Engineering					
MASDK Package	ACL	High-Level Com.	Mental Attitudes	Deliberate Capabilities	Meta-Management	Adaptivity	Methodology	Architectural Description	Internal Functioning of single agent	Documen-tation	Prototyping	Simulation
AgentBuilder	100	100	50	100	100	50	100	100	100	100	0	50

	Implementation				Technical Issues			
MASDK Package	Agent Language	GUIs	Agent Skeletons	Debugging	Mobility	Concurrency	Real-Time Control	Approximate Cost
AgentBuilder	Java	100	50	100	0	50	100	$1,000

Figure 3: AgentBuilder Analysis Overview.

The BDI model of an agent is centered about the idea of mental attitudes. It is the beliefs desires and intentions themselves that are considered mental attitudes. Beliefs of the agent are all the pieces of information it has about its environment, other agents or itself. Desires are simply goals, or what the agent wishes to accomplish. Intentions correspond to what we usually call plans.

JACK provides constructs for planning including a plan base class, which is extendable for creating powerful plans. The JACK Agent Language includes several reasoning method statements corresponding to common planning actions.

In addition JACK defines a special capabilities construct as a collection of events, beliefsets, plans, other capabilities and Java code. These capabilities represent modular functional components that can be plugged into or removed from agents. The fact that these capabilities contain plans could alone qualify them as deliberate capabilities, but these constructs also facilitate the chore of planning and reasoning about planning.

The JACK Agent Language provides a declaration: # chooses for event, that allows agents to not only reason about plans that correspond to specified events, but also to reason about what plans to use for a specific event. This is considered Meta-level reasoning.

Jack does not directly provide any high-level functionality that offers support for adaptivity; although there is abundant support for the ability to alter views, beliefs, change in selection of plans, and capabilities. As such, the desire to include adaptivity would be an exercise in designing algorithms that implement these methods. Our view is that Jack supports partial adaptivity.

The Jack user manual lays out clearly how to build agents. In addition several examples are provided with useful Jack agent language code. As a drawback no clear methodology is presented for creating collections of agents that interact in a standard multiagent system.

Jack does, however, provide an extension to the standard Jack package called SimpleTeam which facilitates construction of work-sharing agencies. The documentation for SimpleTeams does certainly outline how to create agencies.

Jack consists of three main components: The Jack agent language, Jack agent compiler, and Jack agent kernel. Jack's user manual gives a description of these three components but only a full architectural description of the Jack agent language including all the statements, declarations, classes, and methods. Since the Jack agent language is such a major component of JACK and it is fully described, we might say that a good architectural description is provided with the system.

The internal functioning of one single agent is addressed in detail in the documentation. The functioning of an agent is the focus of several chapters in the user manual. This description includes discussion of agent theory and how it translates to Jack agents as well as the specifics of how Jack agents use their beliefs, views, capabilities, and plans to act in their environment.

The Jack documentation is extensive. A full user manual is available online. The user manual is extensive and sufficient to lean most of the Jack Intelligent Agents system. In addition an API reference is provided for the runtime, compiler and BDI (communication network) modules. Also a document specifying several Jack examples is available, called practicals.

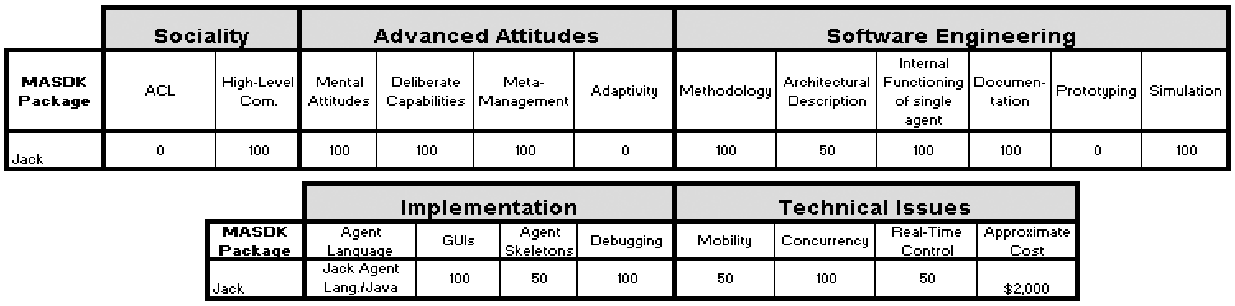
Jack agents are implemented in a proprietary language built on Java called the Jack Agent Language. The Jack Agent Language is actually a superset of Java.

Jack provides an excellent GUI for defining agents within projects. The GUI allows for modification to the agents' views, belief sets, capabilities, and plans. Jack also contains an object browser (JACOB) that provides object modeling for communication of objects between agents and inputting of agents. Both GUIs are very similar to Microsoft's Visual Studio environment and appear well thought out. Due to the choice of language implementation, the GUI is not quick to respond to external events such as mouse clicks and keyboard input.

JACK uses by default an agent class that can be extended by the user using the Jack language. Underneath the Jack language lies Java. Java developers should not have any problem with understanding the Jack language as it is very similar to its underlying implementation language.

The Agent Interaction Diagram showing a graphical representation of agent interaction is available for debugging purposes. In addition the simulation clock mentioned above in the simulation section could be used like a 'step through' command to examine running agents step-by-step. Also by setting certain debugging command line attributes, developers can trace all messages and events that occur in a system. According to the Jack user manual, an integrated debugging environment will be provided with Jack soon.

Partial mobility is provided through Jack's DCI communication network. This network allows agents to communicate through ports using a Jack transport protocol (UDP with guaranteed reliability). Agents can transmit data but code cannot be transmitted per se. In other words, there is not a direct facility for transmitting entire agent definitions from one system to the next. It would be possible to work around this by having standard agents that receive signals from other agents to generate new agents with certain properties. This would essentially do the same thing.



	Sociality		Advanced Attitudes				Software Engineering					
MASDK Package	ACL	High-Level Com.	Mental Attitudes	Deliberate Capabilities	Meta-Management	Adaptivity	Methodology	Architectural Description	Internal Functioning of single agent	Documentation	Prototyping	Simulation
Jack	0	100	100	100	100	0	100	50	100	100	0	100

	Implementation				Technical Issues			
MASDK Package	Agent Language	GUIs	Agent Skeletons	Debugging	Mobility	Concurrency	Real-Time Control	Approximate Cost
Jack	Jack Agent Lang./Java	100	50	100	50	100	50	$2,000

Figure 4: Jack Analysis Overview.

The simulation clock mentioned above could be used as a real-time control. The clock would explicitly control the execution cycles of the system, and the developer would be free to make modifications between any of the cycles.

3.3.2 *Rating*

Figure 4 denotes the summary of the evaluation conducted above. Based on the previously mentioned marking scheme, Jack receives 29.5 points out of 39 available points (76%).

3.4 OpenCybele

OpenCybele is an agent infrastructure written in Java 2. It provides facilities for agent development (in Java) as well as methodologies for creating multiagent systems. While this package is not a bonafide MASDK, we have chosen to include it in this evaluation of MASDKs in order to provide a broader perspective into multiagent development environments in general. Also, use of OpenCybele was considered in the past without any formal evaluation. This provides further motivation to include the package in this evaluation.

3.4.1 *Evaluation*

OpenCybele does not take advantage of any existing ACL standard. Planned updates to Cybele, however, include implementations of FIPA, a widely accepted agent communication language. As far as message passing is concerned, Cybele includes support for both synchronous and asynchronous P2P, multicast, and broadcast, of both communication and message events.

Cybele does not include a cognitive model in its system. As such there are no attitudes included in the model of an agent in Cybele. While Cybele does have the capability to support plans, the system does not provide a facility for defining plans. Nor does Cybele have the ability to plan in terms of dynamically reasoning about what activities are performed in response to an action. Agents are capable of creating self-triggers, which control their own actions, but this is not meta-management. Learning is not supported other than the fact that due to its open nature, developers are given the freedom to add or remove various functionality including learning and Adaptivity.

A very clear sequence of steps for creating and executing a simple Cybele environment and agents is laid out clearly in the user manual. In addition to these simple methodologies, more complex methodologies are presented as part of example systems. The overall architecture of Cybele is based on services and interfaces, very closely related to OOP (Activity Centric Programming [ACP] is presented as an abstraction over OOP). A full description of this architecture is available through the user manual, as well as useful information on the OpenCybele web page.

The internal functioning of a single agent is the focus of much of the Cybele documentation. Cybele uses ACP as a model for developing agents which leads to the functionality of the Cybele agent. This functionality is based on local activities that are executed in response to actions perceived through action messages sent to the agent. The documentation is not extensive, but thorough enough to fully describe the agent infrastructure that is Cybele. A developer's guide is promised at the Cybele web site, which would round out the documentation nicely.

While several examples of simulations are provided with Cybele, no built-in simulation facilities are provided or documented. Similarly, it appears that no standard agents are provided within the OpenCybele environment. The user documentation addresses the roles of the people developing a MAS but does not address any default roles within its agents. Other notable shortcomings of Cybele include the abscence of any standard debugging facilities or an IDE. A lack of these components can severely limit the developer's capacity to quickly learn the functionality of the system.

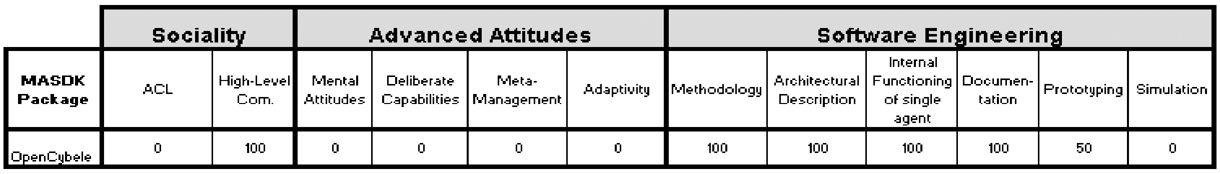
Cybele does support mobility by allowing agents to migrate from one Cybele environment to another. This is facilitated by migration and dynamic class-loading facilities. These facilities, however, are not included in the Open version of Cybele. Concurrency is also supported. Each agent is defined as a group of event-driven activities that share a common thread, data, and code. As such, agents can wait and block on each other in the conventional ways associated with concurrency.

Real time control does not appear to be an option in OpenCybele. Although various configuration parameters for any given host can be set beforehand, it does not appear viable to change these parameters in real time once the MAS is initiated.

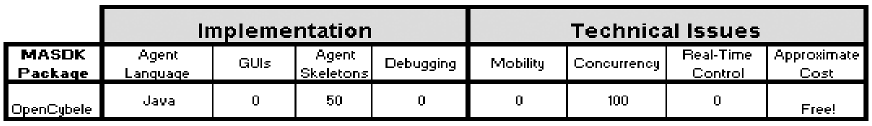
OpenCybele appears to be free for both commercial and non-commercial application. OpenCybele has become an open-source initiative. As such, they retain the rights to demand source code of anything produced using OpenCybele. The documentation implies that there is an exclusive version of Cybele that is not open-source. Unfortunately, the cost of this version could not be found.

3.4.2 *Rating*

Figure 5 shows a summary of the evaluation results outlined above. Based on the previously mentioned marking scheme, an objective measure of the overall MASDK can be calculated using the above table as a quick-reference. OpenCybele scores a value of 18.5 out of a possible 39 points (47.4%).



	Sociality		Advanced Attitudes				Software Engineering					
MASDK Package	ACL	High-Level Com.	Mental Attitudes	Deliberate Capabilities	Meta-Management	Adaptivity	Methodology	Architectural Description	Internal Functioning of single agent	Documentation	Prototyping	Simulation
OpenCybele	0	100	0	0	0	0	100	100	100	100	50	0



	Implementation				Technical Issues			
MASDK Package	Agent Language	GUIs	Agent Skeletons	Debugging	Mobility	Concurrency	Real-Time Control	Approximate Cost
OpenCybele	Java	0	50	0	0	100	0	Free!

Figure 5: OpenCyble Analysis Overview.

4. SIMULATION

As a way of comparing the MASDKs on some equal level, we decided to simulate the same theoretical problems within each MASDK. We do this to get a direct sampling of the MASDKs ability to model a problem and potential solutions. Figure 6 denotes an overview of the problem we implement in each of the considered systems.

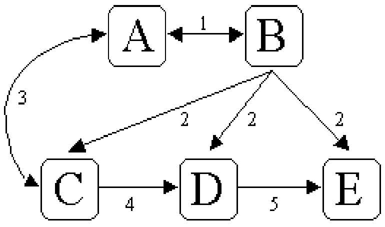


Figure 6: Theoretical Problem.

In step 1, agent A and agent B are involved in a transaction of some sort. In step 2, both agents have completed the transaction and agent B broadcasts a falsehood regarding the event. The intent is for agent B to defame agent A and destroy A's reputation. In step 3, agent C informs agent A of the reported transaction. Agent C is the only friend of A in this scenario. After agent A clarifies the falseness of the transaction to agent C, agent C reports the falsehood to its friend agent D in step 4. In step 5, agent D informs its friend agent E and the defamation has been nullified. Clearly, this is one of many potential scenarios. For our purposes, it represents a typical problem that must be easily representable in the MASDK.

The following subsections outline the highlights and pitfalls of using each development kit in simulating th5s problem.

4.1 AgentSheets

AgentSheets provides a very well thought-out IDE for the average user that does not have any programming abilities. In their documentation, AgentSheets states that the product is intended for the non-programming audience. It shows in the fact that agents can only be developed with rigid designs. Programmers generally strive to make their solutions as generalized as possible. In AgentSheets, generalization of code is made very difficult through the requirement of using the user interface for implementation. The programming concepts of inheritance and polymorphism are not allowed. As such, it is difficult to implement a simulation that is flexible enough to structurally change with the problem.

Due to the lack of communication facilities, only signals can be passed between agents. As such, the ability to respond flexibly to communications is quite limited. Dynamic information cannot be passed between agents through the use of messages. Hence, each event in the sequence of actions required for the problem simulation must be uniquely labelled. This further limits the capacity to freely alter the dynamics of the posed problem. These dynamics include the members involved and the sequence of events.

Overall, the final implementation of the problem was not satisfactory as the programmer could not easily change the sequence of events, the number of members, or the roles of the members. It is desirable to parameterize as many aspects of the problem as possible. In this implementation, it was not possible to parameterize any of the aspects (roles, membership, and actions).

4.2 AgentBuilder

AgentBuilder includes a full IDE interface for agent development. While the GUI seems straight foward to use, there are a few limiting factors that made initial development somewhat difficult. These limitations include the fact that users are not free to simply define a class to use as a PAC (Project accessory class), they must define the class as a concept in the AgentBuilder Concept Modeler. Once the Concept is created it is added to the AgentBuilder Object Modeler; only then the class be instantiated and used by the developer. Another factor that made developing in the AgentBuilder environment somewhat difficult is that performing some simple and common programming tasks such as defining a variable were achieved using non-intuitive steps (*non-intuitive* here refers to our intuition as C++ and Java developers). An example of such a task is accessing a field of the object contained in an incomming KQML message. Such a task includes casting the generic object reference to the appropriate class type. In agent builder, this task requires choosing the appropriate type in the 'casting type' drop down menu. In order to indicate this casting type, the object filed in the KQML message variable must be selected. This is assuming that the desired class type and KQML message variable have already been defined.

The most glaring limitation in developing a MAS with the evaluation version of AgentBuilder is that the package will not allow execution of new code (only the provided examples will run). This makes developing a MAS possible, but makes running, debugging and testing of that MAS impossible.

Once the initial problems with the interface were sorteed out, the development process went rather smoothly (although, since running the project was impossible, we have no idea if the implementation works as desired). The rule editor provides a straight forward interface to define behavioral rules and their components. Those components include previously instantiated PACs, locally defined variables (such as a variable to hold incomming KQML messages), message conditions (when), mental conditions (if), as well as actions, and reasoning statements (then).

Although we did not implement a formal messaging protocol, the message conditions defined as part of the behavioral rules essentially define an ad-hoc protocol to which the agents will conform.

4.3 Jack

In contrast to agent sheets, and in a lesser degree to AgentBuilder, Jack targets the software engineering community. The implications are that Jack includes facilities to allow for designs that would be considered more elegant by the software engineer than the other MASDKs evaluated here. Becuase Jack is Java-based, all the features of Java are carried through, and actually expanded upon in the Jack Agent Language. These features include: polymorphism, inheritance and abstraction.

The BDI model implemented in Jack is an intuitive and easy to use model for simulation of a problem, such as the one described above in section 4. In particular, the relevance() and context() methods used for meta-management (plan selection) allow for the creation of multiple versatile plans. These various plans can be implemented so that agents can react in one of many ways to a single type of event, based on the specific internal and external state of the agent at the time the event occurs.

The Jack IDE was very intuitive and relatively easy to use. Many tasks, specifically associating various components of the system being developed is achieved through drag and drop. For example, in order to access an interface to the agent from a plan, the agent is dragged and dropped on the plan. It is noteworthy that the Jack documentation does not include any mention of the GUI or how to use any part of it. In order to understand how to use the IDE, we took examples and reverse-engineered them within the IDE. This was a very effective process in understanding how to use the IDE. Even though it was effective, it is far more desirable to have quality documentation that outlines the IDE's functionality.

In addition, the documentation is lacking in the description of the use of the relevance() and context() methods. In particular the consequences of the fact that the relavance() method is static are not addressed. The limiation we discovered due to this fact, is that a reference to a non-static interface (such as the SELF agent interface) cannot be referenced from inside the method.

The quality of the final simulation was very high due to the application of the previously mentioned software design features imbedded within Jack. All aspects of the problem became parameterized. This allowed for a wide variety of potential simulations. The architecture was general enough that the defining features of the problem could be specified at run-time.

4.4 OpenCybele

At the heart of Cybele is the activity centric programming (ACP) paradigm. This model of agenthood is a simple yet powerful one; agents are seen simply as a collection of activities. Agent interaction takes place in the form of one agent's activity interacting with an activity of another. Activities are loosely defined as active objects that act on internal data. Cybele's activities can be seen as almost analogous to the plan construct in a BDI development environment such as JACK. The BDI plan includes a specification of what event(s) cause the plan to be executed. Similarly, activities respond to specific types of events. BDI goals and activities are similar in that both are essentially an algorithmic description of a response to some condition or event.

Each activity is considered mutually independent. This means that activities cannot directly affect one another. Similarly, Cybele agents are not allowed to directly affect one another. Cybele implements the idea of autonomy in its agents by enforcing the rule that no agent has a direct reference to any other agent or its methods. This constraint, however, is a particularly artificial one, as programmers can in fact pass references to agents in messages. The user manual simply warns developers not to engage in such reference passing at the risk of destroying the autonomy of their agents.

One of the complicating factors in using OpenCybele involves the use of channels as a communication mechanism. Agents or activities can communicate with their environment only through the use of explicitly defined channels. The channels act as a conduit of activity-generated events that contain messages for other activities or agents. Broadcasting of an event occurs through subscription to a channel with a particular identification tag. Upon receiving the event, a custom method within the agent delegate class is used as a callback mechanism with the event as a parameter. Although OpenCybele maintains that an agent should never be able to call another agent's methods, this communication method allows it by explicitly assigning method invocations to specific events. Hence, it is believed that autonomy is violated through this indirect method of control.

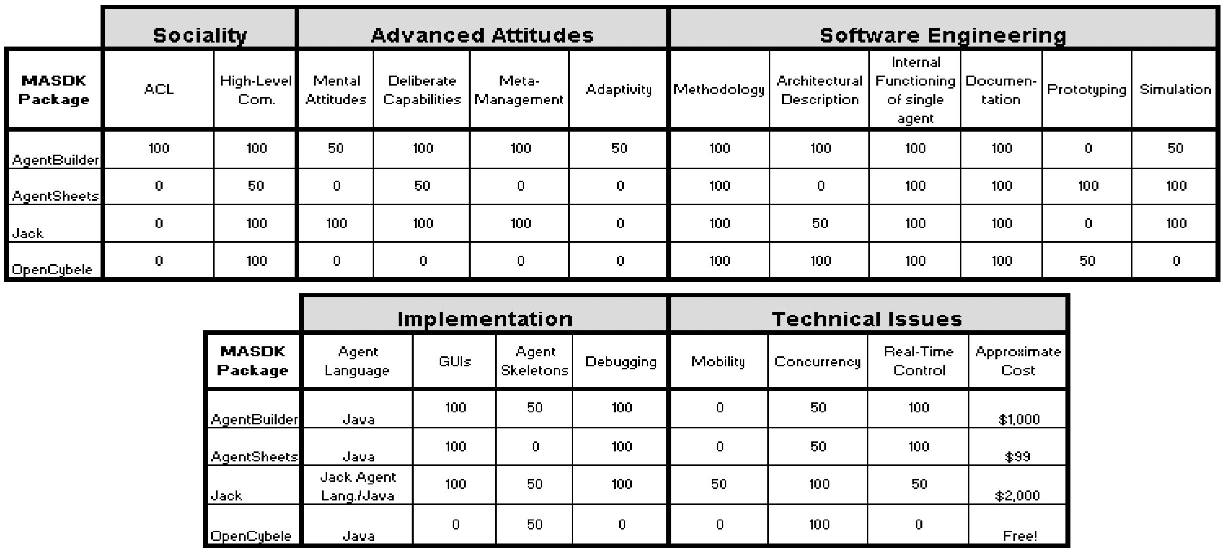
The need to explicitly create channels raises design issues of channel architecture. The need for a channel architecture is complicated by the fact that point-to-point agent communication is not possible without the previous establishment of these channels. Channel handlers are an added component to the architecture that represent a unique identifer that is a combination of the agent's identification, action within the agent, channel corresponding to the action, and specific method corresponding to that action. Point-to-point communication is represented by sending a message to this channel handler. It is the agent's responsibility to attain the channel handler of the other agent. Hence, there is a large overhead in establishing this usual method of communication between two agents.

5. CONCLUSIONS

Figure 7 summarizes our objective findings of the degree to which each MASDK satisfies the specified criteria. These criteria include the following: sociality, advanced attitudes, software engineering, implementation, and technical issues. We take the stance that software engineering criteria are very important along with advanced attitudes.

Figure 8 shows the overall numerical results of the objective portion of our MASDK evaluation. The values in this figure were obtained using the evaluation results shown in Figure 7 and the evaluation scheme described in section 2. The results of this objective evaluation were as expected.

AgentBuilder received the highest evaluation value due to its impressive performance with regards to the evaluation criteria. This is despite the fact that the evaluation version



	Sociality		Advanced Attitudes				Software Engineering					
MASDK Package	ACL	High-Level Com.	Mental Attitudes	Deliberate Capabilities	Meta-Management	Adaptivity	Methodology	Architectural Description	Internal Functioning of single agent	Documentation	Prototyping	Simulation
AgentBuilder	100	100	50	100	100	50	100	100	100	100	0	50
AgentSheets	0	50	0	50	0	0	100	0	100	100	100	100
Jack	0	100	100	100	100	0	100	50	100	100	0	100
OpenCybele	0	100	0	0	0	0	100	100	100	100	50	0

	Implementation				Technical Issues			
MASDK Package	Agent Language	GUIs	Agent Skeletons	Debugging	Mobility	Concurrency	Real-Time Control	Approximate Cost
AgentBuilder	Java	100	50	100	0	50	100	$1,000
AgentSheets	Java	100	0	100	0	50	100	$99
Jack	Jack Agent Lang./Java	100	50	100	50	100	50	$2,000
OpenCybele	Java	0	50	0	0	100	0	Free!

Figure 7: Analysis Overview; the degree to which each MASDK (AgentSheets, AgentBuilder, Jack and OpenCyble) satisfies the specified criteria.

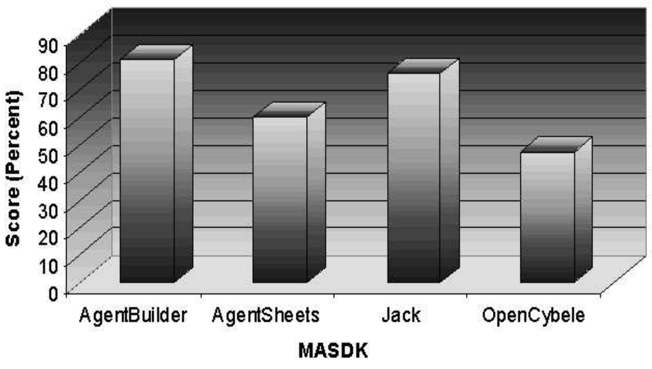


Figure 8: Evaluation results for AgentSheets, AgentBuilder, Jack and OpenCyble MASDKs.

of AgentBuilder will not run user-developed agents. Following closely in second place is JACK. Major differences between AgentBuilder and JACK in terms of the evaluation criteria include: JACK's lack of an ACL and real-time control. In third place is AgentSheets. AgentSheets is severely lacking in many areas of the evaluation, including agent communication, attitudes, and concurrency. In fourth place is OpenCybele. While it appears as though OpenCybele is a powerful development environment for some applications, we discovered that by our evaluation criteria the system leaves much to be desired. Among others, OpenCybele scored very low in the following areas: ACL, attitudes, implementation issues (GUI, skeletons, debugging), and real-time control.

6. REFERENCES

[1] AgentBuilder. Agentbuilder, an integrated toolkit for constructing intelligent software agents, reticular systems, inc., san diego, california, february 1999, http://www.agentbuilder.com/.

[2] AgentLand. Agentland, news and information on intelligent software agents and bots, boulogne, france, http://www.agentland.com; accessed 4-26-02.

[3] AgentSheets. Agentsheets, agentsheets inc., boulder, colorado, 1996, http://www.agentsheets.com/.

[4] F. Bott. *Professional Issues in Software Engineering, 2nd Edition.* University College London Press, London, England, 1995.

[5] T. Eiter and V. Mascardi. Comparing environments for developing software agents. Technical Report INFSYS RR-1843-01-02, Technische Universitat Wien, A-1040 Vienna, Austria, March 2001.

[6] J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence.* Addison-Wesley, 1999.

[7] S. Green, L. Hurst, B. Nangle, P. Cunningham, F. Somers, and R. Evans. Software agents: A review. Technical Report TCS-CS-1997-06, Dublin, 1997.

[8] M. Hoyle and C. Lueg. Open sesame: A look at personal assistants, 1997.

[9] Jack. Jack intelligent agents - version 3.1, agent oriented software pty. ltd., australia, http://www.agent-software.com.au.

[10] G. W. Jones. *Software Engineering.* John Wiley & Sons, New York and Toronto, 1990.

[11] H. Nwana, D. Ndumu, and L. Lee. Zeus: A collaborative agents toolkit, 1998.

[12] H. S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(2):205–244, 1995.

[13] R. S. Pressman. *Software Engineering, A Practitioners Approach, 3rd Edition.* McGraw-Hill Inc., Singapore, 1992.

[14] M. E. Ras, Z.W.; Zemankova. *Methodologies for Intentional Systems.* Springer-Verlag, 1991.

[15] M. Schmidt. *Implementing the IEEE Software Engineering Standard.* SAMS Publishing, USA, 2000.

[16] M. Schumacher. *Objective Coordination in Multi-agent System Engineering.* Springer-Verlag, Berlin, Heidelberg, and New York, 2001.

[17] K. Sycara, K. Decker, A. Pannu, M. Williamson, and

D. Zeng. Distributed intelligent agents. *IEEE Expert*, 11(6):36–46, 1996.

[18] W. Walsh and M. Wellman. A market protocol for decentralized task allocation. In *Third International Conference on Multi-Agent Systems*, pages 325–332. IEEE Press, 1998.

[19] M. Wooldridge. *Reasoning About Rational Agents.* MIT Press, Cambridge Massachusettes, London England, 2000.