A Specialized Macro Language For Specifying the Communication Patterns Within an Agent

Troy A. Shahoumian
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-7
January, 2000

E-mail: troy_shahoumian@hp.com

application management, software agents, CIM, macro-languages

To meet the challenges of application management, our research group in Hewlett Packard Laboratories is prototyping software agents made up of "parts"---lightweight threads that communicate using message-passing. Specifications for these agents are described in CIM and stored in Microsoft's CIMOM object repository. To facilitate the design of these agents, a simple macro language called Agent Generation Tool (AGT) has been developed. Rather than specifying agents directly in MOF, agents can be specified using AGT. The AGT macro processor can then generate the appropriate MOF. AGT allows super-parts, combinations of parts that communicate in a given pattern, to be defined and reused multiple times. The design considerations behind AGT are discussed.

© Copyright Hewlett-Packard Company 2000

A Specialized Macro Language For Specifying the Communication Patterns Within an Agent

Troy A. Shahoumian*

Abstract: To meet the challenges of application management, our research group in Hewlett Packard Laboratories is prototyping software agents made up of "parts"—lightweight threads that communicate using message-passing. Specifications for these agents are described in CIM and stored in Microsoft's CIMOM object repository. To facilitate the design of these agents, a simple macro language called Agent Generation Tool (AGT) has been developed. Rather than specifying agents directly in MOF, agents can be specified using AGT. The AGT macro processor can then generate the appropriate MOF. AGT allows super-parts, combinations of parts that communicate in a given pattern, to be defined and reused multiple times. The design considerations behind AGT are discussed.

Keywords: Application Management, Software Agents, CIM, Macro-Languages

1 Introduction

Work is currently being done at Hewlett Packard in the area of application and service management. The goal is to have a management system that discovers applications, instruments and monitors them. The ultimate goal is to tie these functions in with Hewlett Packard's OpenView products, extending OpenView's ability to control an application's parameters to achieve the best possible application performance.

The system currently being prototyped is based on agents made out of light-weight components called "parts." These parts communicate via a publish/subscribe software bus. Each part runs in its own thread. Exactly how these parts are combined to form an agent is represented in CIM (Common Information Model) [DMTF99a] and stored in Microsoft's CIMOM (CIM Object Manager) [Micro99]. (CIM usually refers to the schemas defined by the Distributed Management Taskforce. In this paper we use the term somewhat loosely to include the extensions we have made to CIM to define our agents, using the CIM core schema as a base.) To store these objects in CIM one specifies them using the MOF language which is compiled into CIM. CIM was not designed for the specification of these agents; writing these specifications directly in MOF is a long and tedious process for even small agents made from a handful of parts.

* Software Technology Lab, Hewlett Packard Laboratories. 1501 Page Mill Road, Palo Alto CA 94301. troy_shahoumian@hp.com

To make it easier to create these agents, a macro language called Agent Generation Tool (AGT) was created. AGT allows the specification of an agent's composition to be authored in a more logical fashion, enabling more sophisticated agents to be built with less chance of errors.

The rest of the paper is organized as follows. Section 2 contains the architecture of the new agents being created in our prototype application management system. The need for a macro language for specifying agents is discussed in Section 3. Section 4 describes the macro language. Section 5 discusses unexpected benefits that resulted from people using AGT. Finally, Section 6 offers some conclusions and possible extensions to this work.

2 Background on Application Management and the Agents Being Prototyped

As corporations rely more on their computer systems for their day-to-day operations, making sure these systems are operating correctly—and quickly triaging and resolving problems when things do go awry—becomes more critical. Not surprisingly, at the same time, a corporation's information technology infrastructure is getting more complex; the number of systems to be managed grows, the complexity of each system increases, and the interactions and interdependencies among computer systems becomes more sophisticated and harder to manage.

Hewlett Packard's OpenView [OV99] line of products is designed to help make computer systems easier to manage. There is an ongoing need to make the agents deployed by the management system more intelligent. As management systems become more complex, more reasoning needs to take place at the agent level rather than having a large amount of data sent back to the management system for processing.

At the same time, there is a need to simplify the implementation of the agents. The agents must be able to handle input from a variety of sources. For example, they must accept data input from numerous outside events such as those generated by monitored processes. They are also responsible for monitoring the lengths of queues in software systems and for determining whether the necessary processes are running. Writing a single-threaded agent to handle all of the different event types would be challenging. The single thread would be responsible for handling the events

coming from all the different sources. As the number of event types grow, implementing the agent as a single thread would becomes more complex.

To meet these challenges, a new architecture for software agents was created. (Some of the details of this new design are described here. More details can be found in [MD+99].) It was decided to create agents out of multiple light-weight threads called parts. Parts are meant to be general-purpose building blocks out of which the management agents can be built. By composing parts in different ways, one will be able to quickly build any of the management agents which would be required. These parts are coded in a programming language such as C++ or Java and because they are given a specific job within the agent, they are easier to implement and maintain. In an agent made up of these parts, each part receives messages, does some processing and then publishes messages when appropriate. These messages can be directed to other parts within the same agent, to parts making up other agents, or to central management systems.

This design for agents should be compared to writing a single-threaded piece of code capable of receiving a number of different message types at any time. The agents made out of multiple threads are better suited to responding to asynchronous events.

It should be mentioned that there will generally be multiple instantiations of a given part within a managed system, perhaps even within a single agent. These parts are coded with the notion that they will be reused multiple times. The key goal in AGT was to allow parts to be logically packaged into larger "groups" of parts which can be deployed multiple times throughout a management solution.

One implementation of our agent architecture is based on COM technology from Microsoft. The rest of this technical report discusses this implementation. Other work in our department is working on an implementation which runs on a variety of platforms and can access data from the CIMOM using XML as an intermediate representation [GGH99].

Parts communicate via several publish-subscribe software buses. (Purtillo [Pur94] gives a thorough introduction to software buses.) This was chosen over a system based on direct notification such as Microsoft's connection points. The publish/subscribe paradigm allows an agent to dynamically subscribe to specific topics of interest and receive messages published onto

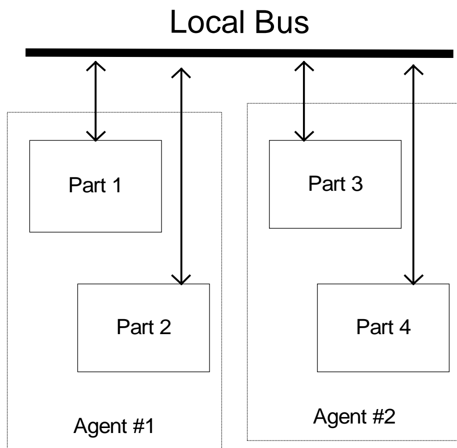

Figure 1 Parts are logically grouped together to form agents. The parts communicate via the software bus. Only the lightweight local bus—which is used for communication between parts residing on the same machine—is shown.

that topic. A lightweight local bus facilitates communication between the parts within a single machine, as shown in Figure 1. A more complex global bus is responsible for communication between agents whether or not they are deployed on the same machine. Which of these two buses an agent is using should be transparent to the agent creator.

The specification of parts within an agent is stored in Microsoft's CIMOM, which is an object repository. An object stored in the CIMOM is similar to a C structure in that it has a name and a number of strongly-typed data fields. In the case of an object specifying a part in an agent, the type of the part is specified, along with the topics which it should monitor for messages on ("subscribe" topics) and the topics which it sends out messages on ("publish" topics.)

Message

machine=camino
user=bob
TotalCPUTime=27
Database=customer.dat
Operation=Read

Figure 2 The events, or messages, which are sent between parts are made up of fields. Each field can be accessed independently.

A field is a typed placeholder within a message, similar to a member variable in a C structure. Fields allow additional information to be carried in the messages. For example, consider a message which is generated when a database transaction is started. A message similar to the one shown in Figure 2 would be sent. The user and machine that initiated the transaction can be stored in two fields of the message. This information will be carried in the message throughout its path through many parts even though not all parts will need to access or change this information. Parts do not read and write an entire message at once, but read from and write to specific fields within the message.

When specifying a part in CIM, besides specifying topics and fields, one can specify parameters for the part. (For an example of this, see the sample MOF file in Figure 4.) This customizes the behavior of the part. In the case of a part which generates a message when a database transaction is started,

parameters may specify which database or type of transactions are of interest. The parameters to a part are specified in a parameter instance object, a separate object from the object representing the part itself. The justification was that parts would be able to easily share parameters by pointing to the same parameter instance object.

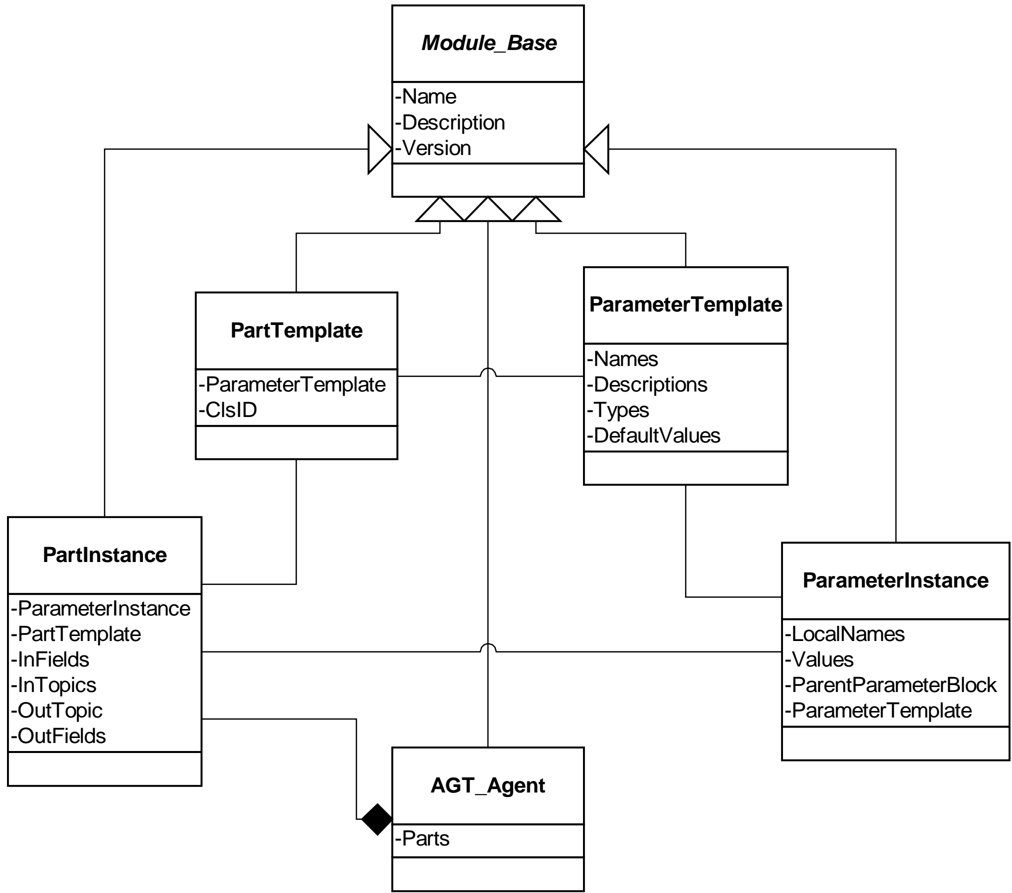

Figure 3 This diagram shows how parts are modeled in CIM. An agent is made out of a number of parts. The specification of the exact part to be used—i.e., where to find the binary representation of the part—is done using a PartTemplate. Each part has an associated ParameterInstance in which all the parameters are stored.

Figure 3 is the UML diagram showing how parts are associated with an AGT_Agent and how each part requires a parameter instance object. When specifying agents by writing MOF files, there was considerable overhead in specifying each part, and keeping things consistent across parts was very hard to do.

To illustrate the complexity of specifying agents directly in MOF, Figure 4 gives the MOF for a sample agent with two parts. This agent is for illustrative purposes only. Looking at the first object described in the MOF,

the first part ($Part1) is an instance of a 'filter' part. It reads all messages broadcast on the input (subscribe) topic `alltransactions', and presumably filters database transactions in some way. It sends out its messages on a private (publish) topic, `topic0_0.' The second object describes the one parameter for this part. When agents are specified directly in MOF, the parameters to a part are stored in a separate object; the advantage is that multiple parts can share the same parameters. (AGT does not take advantage of this feature. For machine-generated MOF it was easier to generate a separate parameter block for each part.)

The next two objects ($Part2 and $pi10002) describe the second part. It subscribes to the same private topic, topic0_0 and publishes to topic `filteredtransactions'. The fifth object specifies that the agent is made up of these two parts.

3 The Need for an Easier Way of Creating Agents

As one can see, specifying even simple agents composed of a handful of parts directly in MOF is very tedious. Examining examples of agents our project team created, it was quickly realized that parts needed to be combined into larger, more complex super-parts which were repeated multiple times within a management system. For example, one super-part might analyze the transaction times for one type of database transaction, and this super-part was repeated several times in a single agent for the different types of database transactions. When writing MOF directly, specifying such a super-part multiple times led to several problems. First, each part required about 10 lines of MOF, and because part definitions look very similar it was easy to get things wrong. Second, because MOF was not designed for defining these parts, there were no embedded clues to help the agent designer keep track of what was going on. When agents are coded in MOF, there was no easy way to tell which parts were communicating together, which group of parts were acting together to achieve as specific goal, and which parts were combined into a "super-part" which was deployed multiple times. (We also explored the use of a visual tool to make it easier to see the grouping and relationships among parts.) Third, as the super-parts were improved, the changes needed to be made multiple times; if a super-part was instantiated five times, one had to change each of the five instances if one was creating agents by directly writing the MOF.

```
instance of PartInstance $Part1   {
        Name = "fm0_temp";
        PartTemplate = "filter";
        InTopics = {"alltransactions"};
        InFields = {"value"};
        OutTopics = {"topic0_0"};
        OutField = {"value"};
        ParameterInstance = $pi10001;
};

instance of ParameterInstance as $pi10001   {
        ParameterTemplate = "filter";
        LocalNames = {"user"};
        Values = {"bob"};
};

instance of PartInstance  as $Part2   {
        Name = "fm0_dbfilter";
        PartTemplate = "transactionrecord";
        InTopics = {"topic0_0"};
        InFields = {"value"};
        OutTopics = {"filteredtransactions"};
        OutField = {"value"};
        ParameterInstance = $pi10002;
};

instance of ParameterInstance as $pi10002   {
        ParameterTemplate = "transactionrecord";
        LocalNames = null;
        Values = null;
};

instance of AGT_Agent as $sample   {
        Name = "sample";
        Parts = {"fm0_temp", "fm0_dbfilter"};
};
```

Figure 4 This is the MOF code needed to specify a simple agent made out of two parts. MOF allows you to give objects names—such as $Part1—so that objects within the MOF file can refer to each other. Even in this simple example, the complexities of encoding agents this way is apparent.

4 The Macro Language

A macro language was designed which made it easier to specify the type of agents discussed above. As mentioned before, the key feature was to be able to logically group parts into a super-part which could be deployed multiple times as if it were a single part. The AGT macro processor was implemented in Java using Sun Microsystem's JavaCC parser generator [Su98]. This

section introduces the features of the language and explains the philosophy behind the design.

4.1 Basics of the Language

Figure 5 shows an example of a simple super-part being defined and subsequently instantiated once.

```
part DatabaseTransactionMonitor (intopics: i)
        part_1 = DatabaseTransactionDetect(intopics: i);
        part_2 = DatabaseTransactionAnalyze(intopics: i, part_1);
        DatabaseTransactionMonitor = DatabaseTransactionRecord(intopics: part_2)
end part;

agent  DBAgent1 = DatabaseTransactionMonitor (intopics: inputs; outtopic: results);
```

Figure 5 An example module and instantiation of that module. In the definition of the module the keyword **intopics** is used in the parentheses to distinguish this construct from other constructs which can also appear within the parentheses. The equal signs in the module definition are not specifying an assignment; they are used to separate a local identifier for the part (i.e., part_1) from the type of part (i.e., DatabaseTransactionDetect.

The super-part DatabaseTransactionMonitor subscribes to one topic and, like all parts and super-parts, publishes to one topic. The intopic name i is not the actual topic name; it is merely a formal parameter. The actual topic name is determined when the super-part is instantiated. The first part, DatabaseTransactionDetect, subscribes to in-topic i and published on its own topic named part_1. This topic name is a local placeholder. In different instantiations of this super-part, the topic name will be replaced with a unique topic name only used in that super-part. The second part, DatabaseTransactionAnalyze, subscribes to the messages published on topic i and to the messages published by the first part, part_1. The third part, DatabaseTransactionRecord, subscribes to the messages published by part_2. The fact that the DatabaseTransactionRecord shares the name of the super-part indicates that it should publish to the super-part's publish topic. This third part generates the output for the entire super-part. The flow of messages in this super-part is illustrated on the following figure.

For this instantiation, the super-part subscribes to topic `inputs' and publishes to topic `results.' The actual topic names are determined when the super-part is instantiated; this is done by the last AGT statement in this example. In the last statement, the identifier DBAgent1 is the name of the

agent being created. It is used by the management system to address the agent.

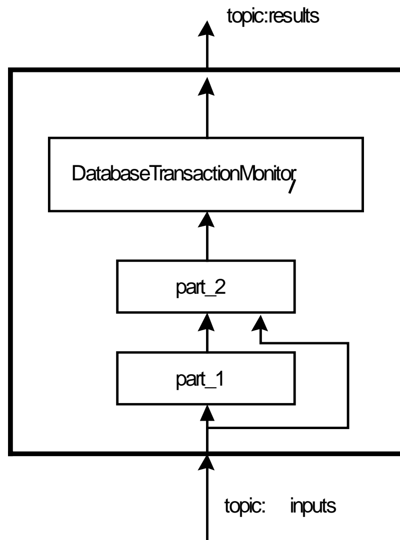

Figure 6 A visual representation of the super-part DatabaseTransactionMonitor and how messages flow through the super-part. A super-part is made up of regular parts—lightweight processes implemented in a language such as Java—and/or other super-parts. The person writing the AGT code specifies how parts are grouped together to form super-parts.

The implementation of parts allows zero or more in-topics and zero or more out-topics. In designing AGT, we imposed the restriction that each part or super-part have only one out-topic and one or more in-topics. Having one out-topic simplified the AGT syntax because there was only one outflow from a part; i.e., by having one out-topic per part, the identifier uniquely determines the out stream of messages from the part. These assumptions are not a serious limitation to the generality of super-parts. For those super-parts that do not need to receive messages, they can subscribe to a "dummy-topic" on which no messages are ever published. An example of such a part is one that determines whether a given process is running. Similarly, one can either have super-parts publish to multiple topics (the push model) or have the receiving super-parts subscribe to all of the required topics. In designing AGT the latter approach was adopted.

Messages published by a part can be read by any other part defined in the same super-part. Other parts defined outside the super-part have no access to these internal messages and can only subscribe to the messages published by the super-part itself. In Figure 5, only messages published by the part DatabaseTransactionMonitor can be read by parts outside this super-part. This was intentional to provide encapsulation. Just as users of a function in a side-effect-free language need only worry about the function's result rather than its implementation, we wanted super-part designers to be free to change their design. So long as the interface stays the same, the design of a super-part can change without breaking anything else. This design allows one to limit at run-time how far message need to be broadcast because internal messages will never be read outside the super-part.

From the above macros, one can not tell whether the three parts instantiated are ordinary parts implemented in C or Java, or if they are super-parts. In other words, the three parts instantiated above could themselves be super-parts composed out of multiple simpler parts. This is intentional. The motivation is to allow agent implementers to create super-parts made up of pre-defined parts and to have be used as if they were ordinary parts implemented in C or Java.

4.2 Customizing Super-Parts using Parameters

The above super-part has no parameters, but parts and super-parts can be modified at instantiation via parameters. This is illustrated in the following example.

```
part DatabaseTransactionMonitor (intopics: i; params: dbname)
        part_1 = DatabaseTransactionDetect(intopics: i; params: TransType = "read");
        part_2 = DatabaseTransactionAnalyze(intopics: i; params: part_1: database = dbname);
        DatabaseTransactionMonitor = DatabaseTransactionRecord(intopics: part_2)
end part;
agent  DBAgent1 = DatabaseTransactionMonitor (intopics: topic1; outtopic: topic2; params:
                        dbname = "CustDatabase");
```

Figure 7 This is a demonstration of parameters being passed to a part. The part 'part_1' is being passed a parameter whose name is TransType and whose value is "read". In the case of `part_2,' the value of the parameter dbname is determined when the part is instantiated.

The super-part has one parameter—dbname—whose value is specified when the super-part is instantiated. This parameter is passed to part_2, the DatabaseTransactionAnalyze, which could either be an ordinary part or another super-part which was specified using AGT. We also pass a constant parameter to part_1, the DatabaseTransactionDetect. Using this simple parameter-passing mechanism allows one to customize each instantiation of a given part.

4.3 Use of Fields

As mentioned before, the use of fields is an important feature of the parts infrastructure. Fields are storage locations in messages. One assumption in the parts infrastructure was that although a part may write to multiple fields within a part, the parts main result would be stored in a single field denoted ReturnValue. This does not limit a part to having a single result. Parts

generating multiple values can store them in an object and return a pointer to that object in the message's ReturnValue field.

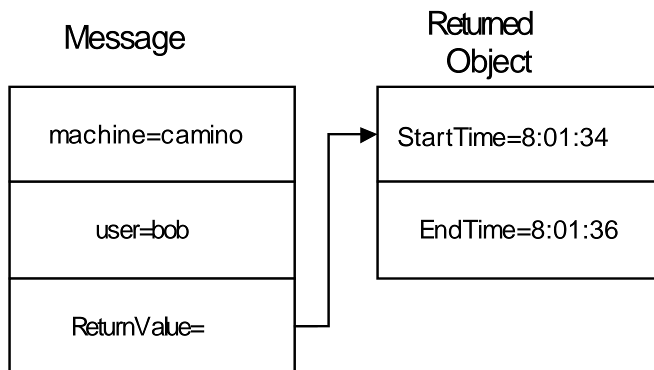

Figure 8 Having a part write its main result to a single field allows for multiple results. The field can have a pointer to an object holding several results.

```
module CheckForProcesses (intopic: i)
        p1 = CheckProcess(intopic: i)
        p2 = CheckProcess(intopic: p1)
        CheckForProcesses = CheckProcess(intopic: p2(f1); outfield: f2)
end module
```

Figure 9 In this example, the part CheckForProcesses is reading from a specific field, \`f1' in all incoming messages and writing to field \`f2'. When no fields are specified, the field \`Value' is used.

By default, all parts read from and write to the field named Value. In the above example, the main result from p1 is written to the field Value and p2 looks for its main input in this field. The third part, p3, reads from the field f1 and writes its main output to field f2.

Parts are allowed to read from and write to other fields besides the ones named. Our guiding philosophy was that almost all parts have a single field they read from and a single field they write to. The mechanisms just described allow these fields to be determined in an AGT specification of an agent.

Note that the fields read from and written to are specified when a super-part is instantiated rather than when it is defined. This choice was adopted to allow more flexibility in how super-parts are used.

5 Additional Benefits of AGT and Related Work

AGT insulates the agent developer from changes in the CIM model used to specify agent composition. As our agent infrastructure is a prototype, the format of the MOF used to specify an agent is very dynamic. It changes as

the needs of our project change. Rather than having people change all of their agent specifications whenever the format of the MOF changes, it is much easier to have them specify agents in AGT. Changing the AGT processor to accommodate a new format for the MOF has proved to be quite easy.

As we noted earlier, the initial version of our prototype used Microsoft's CIMOM to store the specification of agents. Because the use of Microsoft's CIMOM limits us to their supported platforms, we have been exploring the use of XML as additional output style to MOF. As a result, a command-line switch was added to the AGT processor which causes it to output the equivalent XML. This again eliminated rewriting the agents in a new format.

Because one of our goals is to make the agents in this prototype run on multiple platforms, we have explored continuing to use Microsoft's CIMOM to store agent definitions and having non-Microsoft platforms access the CIMOM remotely. The DMTF has published a standard defining how CIM data can be represented in XML [DMTF99c] and accessed remotely using a HTTP stream [DMTF99b]. In our prototype application management system, we are looking into using this standard so that non-Microsoft platforms can remotely access the necessary model data from a Microsoft CIMOM installation. This work is described in Garg [GGH99].

The notion of making agents out of smaller pieces has been explored before in, for example, Denin-Keplicz and Treur [DT94]. This is related to the current work because AGT allows the parts used to construct agents to be built up and used in a hierarchical fashion. An extensive bibliography on agent languages can be found in Wooldridge, Muller and Tambe [WMT95]. Wooldridge and Jennings have a section of their book [WJ95] devoted to agent languages. Bently [Ben88] discusses building small, specialized languages. Batory *et. Al.* [BSTDGS94] discusses the automated generation of software components. Our agents are examples of these types of components. Cleaveland [Cle88] discusses the automatic generation of languages.

6 Conclusion

The software agents in our prototype management system are constructed out of 'parts'—light-weight threads which communicate via a publish/subscribe software bus. To facilitate the construction of complex

agents using this infrastructure, a macro language was needed to package a number of parts with a specific communication pattern. The package could then be easily instantiated multiple times. Having done this, a collection of light-weight processes becomes a sort of `subroutine' whose implementation is not a concern to the caller. The AGT macro language made it easier to specify these collections of light-weight processes which form the software agents of our prototype management system.

Early experiments with AGT have shown that using AGT is more straight-forward than specifying the agents directly in MOF. For the simple agents currently being specified, about half a page of AGT replaced writing three pages of MOF. We expect that as more complex agents are written, the differences will become even more striking. AGT is used throughout our research group and was used to create the agents in a prototype management system.

Issues to be addressed in future research include the following:

- The system for handling parameters to super-modules is not very advanced: The only option is to pass to a sub-module a fixed parameter or to pass a parameter which was received from the parent sub-module. Will this scheme suffice, or will it be necessary to add parameter-manipulating capabilities to AGT?
- The format of the MOF and XML is hard-coded into AGT. An alternative would be to have a format file which can be used to control the format of the output of AGT. If AGT is used on many distinct projects, this could be a worthwhile feature to have.

Acknowledgements
We are grateful to Martin Griss and Jim Pruyne for seeing the value of this work early on. Martin Griss, Jim Pruyne and Pankaj Garg helped with the design of AGT. Special thanks to Vijay Machiraju for assistance with some of the figures.

References

[BSTDGS94] Batory, D., Singhal, V., Thomas, J., Dasar, S., Geraci, B. and Sirkin, M. The GenVoca Model of Software System Generators, *IEEE Software* 11(9), pp. 82-94, September 1994.

[Ben88] Bently, J. Little Languages, *Communications of the ACM* 29(8), pp. 711-21, August 1988.

[Cle88] Cleaveland, J. C. Building Application Generators, *IEEE Software* 4(9) pp. 25-33, July 1988.

[DMTF99a] Desktop Management Task Force (DMTF),*Common Information Model (CIM),* http://www.dmtf.org/spec/cims.htm , February, 1999.

[DMTF99b] Desktop Management Task Force (DMTF), XML Working Group, "Specifications for CIM Operations over HTTP, Version 1.0," July 20, 1999.

[DMTF99c] Desktop Management Task Force (DMTF), XML Working Group, "Specification for the Representation of CIM in XML, Version 2.0," July 20, 1999.

[DT94] B. Dunin-Kaplicz, J. Treur, *Compositional Formal Specification of Multi-Agent Systems,* in *Intelligent Agents, ECAI-94, Workshop on Agent Theories, Architectures and Languges,* Springer-Verlag, pp. 102-17, 1994.

[GGH99] Garg, P. K., Griss, M., and Holland, J. "On Using XML for End-to-End Service Management," Technical Whitepaper, Application Management Department, Software Technology Laboratory, Hewlett Packard Laboratories. August 1999.

[Micro99] Microsoft Developer Network, *CIM Object Manager (CIMOM) guide,* 1999.

[MD+99] V. Machiraju, M. Dekhil, K. Wurster, P. Garg, M. Griss, J. Holland, Towards Generic Application Auto-discovery, Hewlett-Packard Technical Report HPL-1999-80, July, 1999

[OV99] Hewlett Packard's OpenView line of products. http://www.openview.hp.com

[Pur94] JM Purtillo, The POLYLITH Software Bus, *ACM TOPLAS*, 16(1), Jan 1994, pp. 151-174.

[SU99] Sun Microsystems JavaCC (Java Compiler Compiler) http://suntest.com/JavaCC

[WJ95] M. Wooldridge and N.R. Jennings, editors. *Intelligent Agents—Theories, Architectures and Languages,* volume 890 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1995

[WMT95] M. Wooldridge, J.P. Muller, M. Tambe, *Agent Theories, Architectures and Languages—a bibliography.* In proc. Of *Intelligent Agents II: Agent Theories, Architectures and Languages,* pp. 408-31, Springer Verlag, 1995.