Frameworks for Reasoning about Agent Based Systems

Leon J. Osterweil and Lori A. Clarke

Department of Computer Science, University of Massachusetts,
Amherst, MA 01003 USA

Abstract. This paper suggests formal frameworks that can be used as the basis for defining, reasoning about, and verifying properties of agent systems. The language, Little-JIL is graphical, yet has precise mathematically defined semantics. It incorporates a wide range of semantics needed to define the subtleties of agent system behaviors. We demonstrate that the semantics of Little-JIL are sufficiently well defined to support the application of static dataflow analysis, enabling the verification of critical properties of the agent systems. This approach is inherently a top-down approach that complements bottom-up approaches to reasoning about system behavior.

1 Introduction and Overview

The use of agent based systems continues to grow, promising the prospect that important societal functions will be supported by systems of agents [1, 4, 9, 11, 12]. With this growth, however, comes worries about the reliability, correctness, and robustness of systems of agents. We intuitively understand that agents are software components that can "sense their environment", can "negotiate with each other", are logically "mobile", and can "acquire and use resources." Agent based systems are then informally understood to be "communities" of these software items that, acting as a community, can come up with creative and opportunistic approaches to problems. But, while these characterizations provide enough intuition to suggest how such systems might be used to benefit societal objectives, they do not help us to feel more confident that we can keep these systems under control. Indeed, the very flexible and proactive nature of such systems suggests that part of the reason for their creation is that they may indeed behave in ways that may not have been completely planned a priori.

It seems imperative that we establish the basis for reasoning about the behaviors of such systems sufficiently well that we can determine unequivocally that they will never behave in ways that are harmful, while still leaving the systems free to be proactive and innovative. To be able to make such unequivocal determinations, there must be rigorous frameworks not only for describing what agents do as individuals, but also for defining the structure of their collaborations in the context of the larger job to be done. In our work we have defined just such rigorous frameworks, with mathematically precise semantics, and are demonstrating that these frameworks are sufficiently robust that powerful analysis techniques can be applied to agent systems that have been defined in terms of them.

Earlier work has focused on the agents themselves [15] and has attempted to synthesize inferences about overall systems of such agents from a "bottom up" perspective.

While interesting results have been obtained, it seems clear that this approach should be complemented with a more "top down" view. In our work we view the agents as components in a larger distributed software system. We propose to demonstrate that many of the important properties of the overall agent system can be determined more effectively by studying the structure of this overall system.

From our point of view, an agent is an entity (either software or human) that has been assigned the responsibility for the successful execution of a task, where the task has a well defined position in a rigorously defined problem specification, defined in terms of our framework. Within the context and constraints of that overall structure and framework, the agents are free to go about performing their assigned tasks. But the overall structure acts as a set of constraints that limits the activities of the agents. This structure can be engineered to assure that the behavior of the overall agent system never violates critical properties and always adheres to required behaviors.

Our view of agent systems as distributed systems of software components suggests that traditional software engineering development and analysis approaches should be useful in developing agent systems. But the translation of this notion into successful practice is complicated by the fact that agent systems are particularly complex and challenging software systems. As noted above, agents are software components that "negotiate" with each other, are often "mobile", acquire, consume, and release "resources", and exhibit a range of behaviors that traditional software systems often do not exhibit. Thus, successful engineering and analysis of agent systems requires technologies that deal with these behaviors. This, in turn, requires a mathematically sound framework for specifying what is meant by these terms and then building technological capabilities atop these semantics. A "bottom up" approach entails using the semantics of the coding languages in which agents are written as the basis for analysis of their behaviors, and then the behaviors of the overall systems of agents. In practice this is difficult, as the multitudes of details in a coding language complicate analysis and can obscure the larger scale system behaviors that we seek to study.

The "top down" approach that we advocate suggests that we use a modeling language as a framework with which to represent the overall structure of the agent system, and then apply analyzers to models defined through the use of such a language. Specifically, what seems needed is a modeling language that is effective in supporting the rigorous definition of the full range of behaviors of agents operating within agent systems. The language must support, among other things, the modeling of resources and how they are used, the specification of real time constraints, the representation of contingencies, and the specification of a range of communication and coordination mechanisms. Contemporary software modeling languages do not offer this semantic range, nor do they offer the depth of semantic rigor needed to support definitive reasoning.

We suggest the use of our Little-JIL language as a vehicle for defining models of agent systems. We believe that Little-JIL has the semantic range to cover agent behaviors, as well as the semantic rigor needed to reason about systems defined in the language. The semantic rigor derives principally from the use of a finite state machine model of Little-JIL execution semantics. We have demonstrated that this model can be used to translate Little-JIL definitions into flowgraph-like structures that are then amenable to analysis using finite state verification systems, such as our FLAVERS

dataflow analyzer. The overall effect of the application of these technologies is a facility for precisely specifying agent systems as rigorously defined models that can then be definitively analyzed to assure that the systems have desired properties. In addition, as Little-JIL's semantics are executable, it is then possible to translate the Little-JIL model into the structure that actually coordinates agent activities, thereby implementing an agent system.

We now describe Little-JIL, providing indications of why we are confident that it can be used effectively to model agent systems. We then describe FLAVERS, indicating why we believe that it can be effective in the analysis of Little-JIL definitions. Our hypothesis is that this "top down" approach of modeling the overall structure of an agent system provides a valuable complement to existing approaches to gaining confidence in agent systems.

2 Modeling Agent Systems with Little-JIL

In earlier work we defined the overall structure of the coordination of agents in a problem solving activity as a process [8, 14]. From that point of view, we viewed Little-JIL as a process definition languages. Little-JIL is a visual language that supports the view that activities should be viewed as hierarchies of tasks, augmented by a scoped exception handling facility, where completion of each task is assigned to a specific agent. Little-JIL does not support definition of the agents nor how they do their work, only how the activities of the agents are coordinated. Thus, from a slightly different perspective, Little-JIL is also viewed as an agent coordination language. We now provide a very brief overview of some key Little-JIL language features and indicate how this language is a strong basis upon which to build analytic capabilities for assuring the reliability of agent systems.

A Little-JIL step is an abstract notion that integrates a range of semantic issues, all represented by an icon as shown in Figure 1. Each step has a name and a set of badges that represent key information about the step, including the step's control flow, the exceptions the step handles, the parameters needed by the step, and the resources needed to execute the step. Each step is declared once in a Little-JIL process definition,

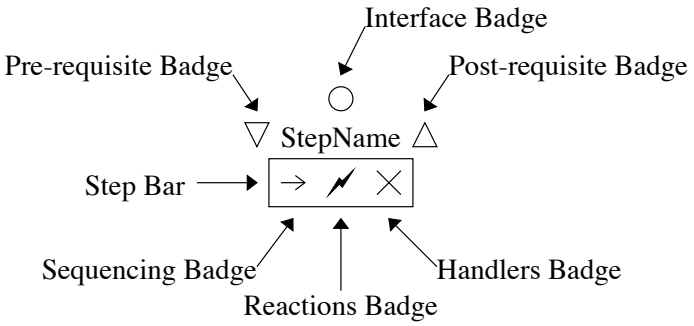

Fig. 1. A Little-JIL step and its constituent parts

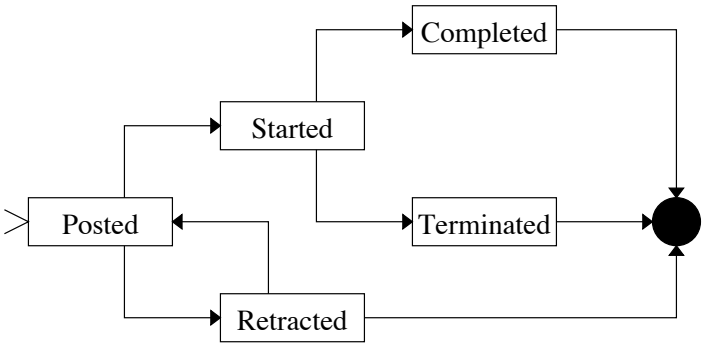

Fig. 2. Finite state machine model of step execution

but a step can be referenced many times. These references are depicted by a step with its name in italics and no badges. This enables reuse and iteration. In addition, a non-leaf step is connected to its substeps and exception handlers by edges, each of which can carry a specification of the flow of artifacts, such as parameters and resources.

The execution semantics of a Little-JIL step are defined by means of a finite state automaton (FSA), with five states: posted, retracted, started, completed, and terminated. Figure 2 shows the normal flow of control for a Little-JIL step. The step's execution can end when it is in any of the three states that have an arrow pointing to the filled circle. A step is moved into the posted state when it is eligible to be started. A step is moved into the started state when the agent assigned that step's execution indicates that the agent wants to start the work specified by the step. When the work specified by a step is successfully finished, the step is moved into the completed state. A step that has been started is moved into the terminated state if the work specified by the step cannot be successfully completed. A step is put into the retracted state if it had been posted, but not started, and is no longer eligible to be started.

A Little-JIL process is represented by a tree structure where children of a step are the substeps that need to be done to complete that step. All non-leaf steps are required to have a sequencing badge. The sequencing badge describes the order in which the substeps are to be performed. There are four types of sequencing badges. A **sequential step** performs its substeps one at a time, from left to right. A **parallel step** indicates that its substeps can be done concurrently, and that the step is completed if and only if all of its substeps have completed. A **choice step** indicates that a step's agent must make a choice among any of its substeps. All of the substeps are made available to be performed, but only one of them can be selected at a time. If a selected substep completes, then the choice step completes. A **try step** attempts to perform its substeps in order, from left to right, until one of them completes. If a substep terminates, then the next substep is tried.

A step in Little-JIL can also have pre- and post-requisites. These are attached to the appropriate requisite badges of a step. A **pre-requisite** is performed after a step starts, but before the work of the step can be initiated. A **post-requisite** has to be done before

a step can complete. Requisites, when they fail, generate exceptions. A step terminates if one of its requisites terminates.

Steps in Little-JIL can throw exceptions, either directly or via requisites, to indicate that their agents were unable to complete the work of the step successfully. Exceptions thrown by a step are handled by an ancestor of that step. Exception handlers are shown underneath the handler's badge and indicate what exceptions the step is able to handle and how to proceed after handling the exception. In Little-JIL there are four different ways to proceed after handling an exception: restart, continue, complete, and rethrow. An exception handler is a Little-JIL step, which may be null. The exception management specification capability of Little-JIL is particularly powerful and flexible. Our experience suggests that this power is necessary for the specification of the kinds of (potentially nested) contingencies that actually arise in complex systems. Little-JIL's scoping and visualization make the exception management easier to understand intuitively. But the semantics of Little-JIL exception management are also precisely defined in terms our FSA model. Thus, exception flow can be modeled accurately using flowgraph models that can then be the subject of the analyzers that we propose here.

Interface badges are used to declare what parameters a step has, what exceptions it throws, and what resources it needs. The resource specification is made using a separate specification language that specifies the types of the resources that are required by the step. The agent for the step is a resource, namely that resource that is needed to assume responsibility for execution of the step. At execution time, the needed resources are requested and a separate resource management module is invoked to match the resource types requested with specific resource instances available for allocation. Should needed resources not be available, a resource exception is thrown, and the Little-JIL exception management facility is used to specify a reaction to this contingency. Little-JIL's facilities for specifying resources also provides the basis for analyses , such as "dead" resource allocations and schedulability.

The semantics of a timing annotation on a step specify that the step's agent must complete execution of the step by the time specified. If the agent fails to do this, then a exception is thrown. Here too, the incorporation of timing specifications as part of the Little-JIL step structure paves the way for potential analysis, such as real time scheduling and planning.

Space does not permit a fuller discussion of the language, but Figure 3 contains an example of a simple Little-JIL definition of an auction agent. Explanation of this example can be found in [3]. A full description of Little-JIL can be found in [13].

3 Analysis of Little-JIL Agent System Specifications

FLAVERS (FLow Analysis for VERification of Systems) is a static analysis tool that can verify user specified properties of sequential and concurrent systems [5, 6]. The model FLAVERS uses is based on annotated Control Flow Graphs (CFGs). Annotations are placed on nodes of the CFGs to represent events that occur during execution of the actions associated with a node. Since a CFG corresponds to the control flow of a sequential system, this representation is not sufficient for modeling concurrent system such as agent systems. FLAVERS uses a Trace Flow Graph (TFG) to represent concur-

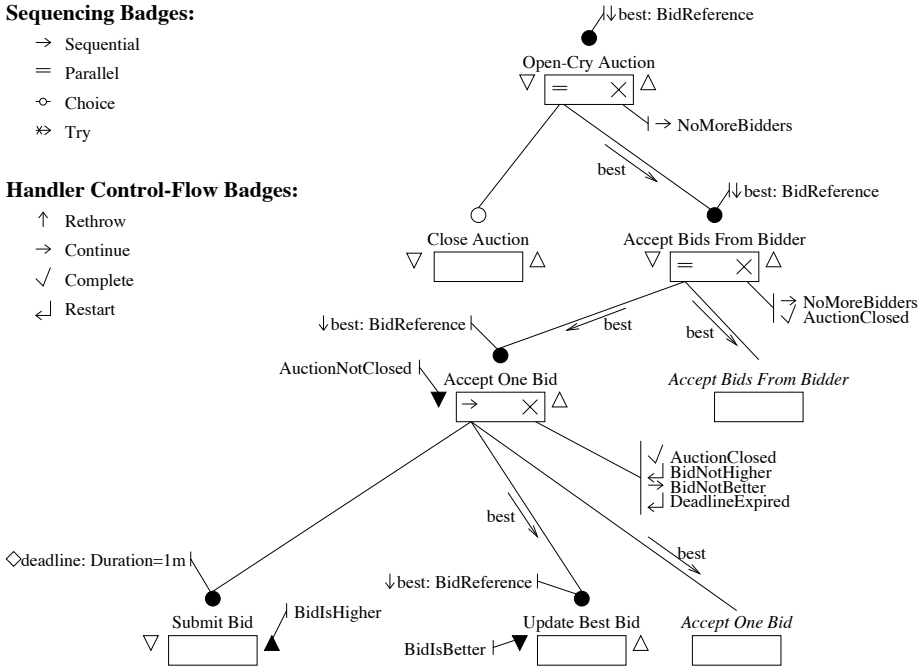

Fig. 3. Little-JIL process definition of an open cry auction

rent systems. The TFG consists of a collection of CFGs with May Immediately Precede (MIP) edges between tasks to show intertask control flow.

As we have indicated above, a Little-JIL agent system definition is translatable into such CFGs and TFG. The fundamental activity here is to build a control flow graph for each of the steps in a Little-JIL definition, to connect these graphs to each other as specified by the language semantics, and then finally to compute the MIP edges. Some details of the complexities involved are described in [3]. Suffice it to say, however, that these complexities can be considerable. While the overall visual appearance of a Little-JIL definition appears straightforward (by careful design), the actual flows of control can be very intricate, involving subtle concurrency, nested exception flow, and dependencies upon resource utilization. Our early experience suggests that some of the subtlety and complexity is often overlooked or oversimplified by humans. This reinforces our belief in the importance of analysis

The annotated TFG is used as the basis for reasoning about properties that are of interest to the analyst. Examples of such properties are livelocks and race conditions. While many such properties can be specified in advance for all agent systems, many other properties are specific to a particular agent system and must be specified by an analyst. FLAVERS supports user specification of properties.

FLAVERS uses an efficient state propagation algorithm to determine whether all potential behaviors of the system being analyzed are consistent with a specified property. Given an annotated TFG and a (possibly user specified) property, FLAVERS will

either return a conclusive result, meaning the property being checked holds for all possible paths through the TFG, or an inconclusive result. Inconclusive results occur either because the property indeed can be violated, or because the conservative nature of FLAVERS analyses causes the analyzer to consider some paths through the TFG that may not correspond to possible executable behavior. Unexecutable paths are an artifact of the imprecision of the model. An analyst can incrementally add constraints to determine whether a property is conclusive or not. This gives analysts control over the analysis process by letting them determine exactly what parts of a system need to be modeled to prove a property.

The FLAVERS state propagation algorithm has worst-case complexity that is $O(S * N^2)$, where N is the number of nodes in the TFG and S is the product of the number of states in the property and all constraints. In our experience, a large class of interesting and important properties can be proved by using only a small set of feasibility constraints. Thus FLAVERS seems particularly well suited to the analysis of agent systems precisely because of its computational complexity bounds. It compares very favorably with model checking approaches (e.g., SPIN [7] and SMV [2, 10]) that have exponential worst case complexity bounds. The FLAVERS low order polynomial bound holds the promise of supporting analysis on the large scale required by complex agent systems.

A major thrust of our work is the application of FLAVERS to verify properties of agent based systems. Our goal is to determine how successful FLAVERS is in definitively verifying the various kinds of properties of agent based systems that are of interest. We have had some success in verifying some user-specified properties of some agent systems and expect to also be able to prove more generic properties, such as the absence of erroneous synchronization and race conditions. We also are interested in how well FLAVERS analysis scales. As noted above, FLAVERS uses analysis algorithms that have low-order polynomial time bounds, but it seems necessary to come to a good understanding on the size of the systems it can be applied to, and characterizations of the sorts of properties to which it is best applied.

4 Conclusions

Our work suggests the possibility of applying powerful forms of analysis to models of agent systems that are particularly comprehensive, yet precisely defined. We are continuing our studies of the applicability of the Little-JIL agent coordination language to the precise specification of agent systems. In doing so, we expect to be able to add substantial power to the arsenal of tools that analysts will be able to apply in establishing the reliability of agent systems.

Acknowledgements

We are grateful to our colleagues, both past and present, in the Laboratory for Advanced Software Engineering Research, for their help with this work. Specifically, Sandy Wise, Stan Sutton, Aaron Cass, Eric McCall, and Barbara Staudt Lerner have all been very active participants in the definition, development, and evaluation of Little-JIL. In addition, Hyungwon Lee, Xueying Shen, and Yulin Dong have been helpful in writing Little-JIL

process programs to help evaluate the language. Jamie Cobleigh led our efforts to apply FLAVERS to Little-JIL process programs.

Finally we wish to acknowledge the financial support of the Defense Advanced Research Projects Agency, and the US Air Force Material Command Rome Laboratory through contract F30602-94-C-0137.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, US Air Force Material Rome Laboratory, or the U.S. Government."

References

1. N. S. Barghouti and G. E. Kaiser. Multi-agent rule-based development environments. In *5th Annual Knowledge-Based Software Assistant Conference*, pages 375–387, Syracuse NY, 1990.
2. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.
3. J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. Verifying properties of process definitions. In *ACM SIGSOFT 2000 International Symposium on Software Testing and Analysis*, Portland, OR, 2000. To appear.
4. K. S. Decker and V. R. Lesser. Designing a family of coordination algorithms. In *First International Conference on Multi-Agent Systems*, pages 73–80, San Francisco, CA, 1995. AAAI Press.
5. M. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 62–75, 1994.
6. M. B. Dwyer and L. A. Clarke. Flow analysis for verifying specifications of concurrent and distributed software. Technical Report 99-52, University of Massachusetts, Computer Science Dept., 1999.
7. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, 1997.
8. B. S. Lerner, L. J. Osterweil, S. M. Sutton, Jr., and A. Wise. Programming process coordination in Little-JIL. In *6th European Workshop on Software Process Technology (EWSPT '98)*, pages 127–131, Weybridge, UK, 1998. Springer-Verlag.
9. V. R. Lesser. Multiagent systems: An emerging subdiscipline of AI. *ACM Computing Surveys*, 27(3):340–342, 1995.
10. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, Boston, 1993.
11. T. Sandholm and V. Lesser. Issues in automated negotiation and electronic commerce: Extending the contract net framework. In *First International Conference on Multi-Agent Systems (ICMAS-95)*, San Francisco, 1995.
12. T. Wagner and V. Lesser. Toward ubiquitous satisficing agent control. In *AAAI Symposium on Satisficing Models*, 1998.
13. A. Wise. Little-JIL 1.0 language report. Technical Report 98-24, Department of Computer Science, University of Massachusetts at Amherst, 1998.
14. A. Wise, B. S. Lerner, E. K. McCall, L. J. Osterweil, and S. M. Sutton, Jr. Specifying coordination in processes using Little-JIL. Technical Report 99-71, Department of Computer Science, University of Massachusetts at Amherst, 1999.

15. P. R. Wurman, M. P. Wellman, and W. E. Walsh. The Michigan internet AuctionBot: A configurable auction server for human and software agents. In *Second International Conference of Autonomous Agents*, Minneapolis, MN, 1998.