

DYNAMIC ANALYSIS OF AGENT FRAMEWORKS IN SUPPORT OF A MULTIAGENT SYSTEMS REFERENCE MODEL

William M. Mongan, Christopher J. Dugan, Robert N. Lass, Andrew K. Hight, Jeff Salvage, William C. Regli, Pragnesh J. Modi
Department of Computer Science
Drexel University
3141 Chestnut Street
Philadelphia, PA 19104
{bill,cjd48,urlass,akh32,jsalvage,regli,pmodi}@cs.drexel.edu

ABSTRACT

Agent systems are unique in their design and functionality; as such, they require unique techniques to profile, reverse engineer, test, and secure. The Agent Systems Reference Model (ASRM) provides a standard for the development and deployment of agent systems. To support the informed construction of the ASRM, we explored existing techniques for obtaining dynamic analysis data for agent frameworks, motivated the development of new techniques providing more concise and cleansed data, and identified mappings of existing agent frameworks and systems to the ASRM. In this paper, we describe our experience and challenges in analyzing agent systems, and with using that data to construct a comprehensive reference model.

KEYWORDS

Agent, Architecture, Design, Reverse Engineering

1. INTRODUCTION

Constructing multiagent systems is a multi-disciplinary task. They are distributed systems; as such, they present many related challenges in terms of program understanding and maintenance. Agents are also proactive and interactive with their environment, and groups such as the Foundation for Intelligent Physical Agents (FIPA) exist to standardize communication and coordination protocols between agents incorporating these features. In addition, they usually contain aspects of artificial intelligence for autonomous decision making based on inputs from each other and from their environment.

This paper outlines our experience using dynamic analysis to profile distributed agent systems. We discuss the challenges we faced using dynamic analysis on these particular systems, and we propose a new dynamic analyzer called Karka that addresses these concerns. Using these profilers, it is our goal to inform and augment the development of the Agent Systems Reference Model (ASRM) (8) (described in Section 2.1) by performing reverse engineering analysis on existing agent frameworks. In this way, we identify high level functional concepts common to most agent frameworks and systems. With this information, we identify how the functional concepts are exercised by subject agent systems through dynamic analysis.

This provides us with specific details pertaining to the implementation of these functional concepts, for example, a publisher/subscriber communication model or the use of a registry service such as UDDI.

The rest of this paper is organized as follows: Section 2 describes Multiagent Systems, the ASRM, and the need for software analysis as the methodology for constructing the ASRM. Section 3 describes related work. Section 4 describes the raw data collection process, while Section 5 deals with our challenges in digesting that data to inform the ASRM. We conclude in Section 6 and discuss future work and research to ease this process in the future.

2. BACKGROUND

Multiagent systems are indeed different from distributed and mobile systems in that agents have beliefs and capabilities as opposed to static functionality (4). Unlike service-based systems, in which functions are invoked statically by the user, agents can pro-actively invoke their own behavior or the behavior of other agents based on their beliefs and understanding of the environment. Multiagent systems are developed with a particular domain-specific application in mind: to simulate cognitive learning processes, to be network-based mobile agents, and so on. According to (6), it can be agreed that most multiagent systems have a “variable number of interacting, autonomous entities that communicate with each other [concurrently] using flexible, complex protocols.” In this way, they are decentralized and highly modular.

There exist numerous *agent frameworks*, which, like their distributed programming counterparts of CORBA and COM, often add overhead to the system (12). However, various agent frameworks provide differing functional components and views on what comprises an agent and how they should interact. In creating the Agent Systems Reference Model, we studied several agent frameworks; we describe the general process and our experience in Section 4. In particular, we describe our experience investigating two premiere and comprehensive frameworks: A-Globe (14) and Jade (1).

As a result, the analysis of the architecture and structure of these types of systems produce representations not only of their distributed structure, but also of their beliefs and capabilities. By dynamically analyzing their behavior at runtime, their internal states and goals were inferred and represented. The potential is twofold: first, it becomes possible to predict what particular agents will do next in their environment; second, commonalities (*i.e.* FIPA compliance) among the agents and their underlying frameworks appear.

2.1 Agent Systems Reference Model

The ASRM (ASRM) is a technical recommendation for a reference model for those who develop and deploy systems based on agent technology. As such, it

- establishes a taxonomy of terms, concepts and definitions needed to compare agent systems;
- identifies common functional elements contained in agent systems;
- captures data flow and dependencies among the functional elements in agent systems; and,
- specifies assumptions and requirements regarding the dependencies among these elements.

The Agent Systems Reference Model allows for existing and future agent frameworks to be compared and contrasted, as well as provided a basis for identifying areas requiring standardization within the agents’ community. As a reference model, the document makes no prescriptive recommendations about how to best implement an agent system, nor is its objective to advocate any particular agent system, framework, architecture, or approach.

A Reference Model derives any number of Reference Architectures that are typically developed using the following process:

- capturing the essence of the abstracted system,
- identifying the few software modules that best describe the abstracted system, and
- identifying or creating an implementation-specific design of the abstracted system.

When performing reverse engineering or software analysis, this process is reversed. It is often the case that several (perhaps similar) subject systems exist but a common abstraction, like a reference architecture, does not. By performing some analysis, one obtains the software modules comprising the subject systems. Data is produced allowing for documentation and understanding of legacy software systems and for verification of existing software documentation. This data can be further analyzed and abstracted to obtain this abstract “essence” of the systems.

The previous processes can execute simultaneously and be used for validation and augmentation. By grouping, abstracting, and querying this data in different ways, information can be gleaned that simple observation may not find. This helps validate that a comprehensive reference architecture was created.

In the ASRM, an agent-based system is divided into several layers:

- **Agent:** a situated computational process that is autonomous, proactive and / or interactive.
- **Agent Framework:** the software component that supports the execution of agents (eg: JADE or A-Globe).
- **Platform:** the software resources atop of which the agent and agent framework run.

- **Host:** A physical computing device on which the platform resides and the agent framework executes.
- **Environment:** the world in which an agent is situated from its point of view; this can be virtual (eg: the web) or an abstraction for the real world.

Based on the analysis of the data generated from reverse engineering (explained later in this paper), a number of functional concepts were identified. In some frameworks these concepts may exist at the agent layer; in others they may exist at the framework layer.

- **Agent Administration:** provides supervisory command and control of agents and allocates system resources to agents.
- **Security and Survivability:** prevents execution of undesirable actions while allowing execution of desirable actions.
- **Mobility:** enables migration of agents among framework instances typically on different hosts.
- **Conflict Management:** enables the management of interdependencies between agents' activities and decisions to avoid incompatible activities, deadlock, etc.
- **Messaging:** enables information transfer among agents in the system.
- **Logging:** enables information about events that occur during agent system execution to be retained for subsequent inspection.
- **Directory Services:** enables locating and accessing of shared resources.

2.2 Motivation for Dynamic Analysis

Like static analysis techniques, dynamic analysis collects data on existing software systems. However, dynamic analysis techniques do so by inspecting that system during execution. This analysis varies widely by implementation, but one approach is to build a data repository of program behavior. This repository holds information on data flow, object instantiation, the call graph, interprocess communication, network or filesystem I/O activity, and so on. If the system contains a lot of "dead code" or other obfuscated constructs, the static analysis results can be inaccurate and deficient in describing the true structure of the system. Dynamic analysis inspects the system as it runs and often breaks the system down into "features." These features are often analogous to the relevant subsystems found during static analysis. In addition, dynamic analysis can obtain data on behavior-specific aspects of the system such as threading and I/O, that could not otherwise be found simply using static analysis techniques. Dynamic analysis assists in cases where source code is not available for static analysis to be performed.

During our software architecture study for the Agent Systems Reference Model, it was difficult to use just static or just dynamic analysis tools to gain a complete understanding of the subject system. Using static analyzers relied on programmer nomenclature. Using dynamic analyzers on these distributed systems resulted in excessive noise and still some reliance on nomenclature. It proved difficult to reduce that noise in some cases because the system simply ran out of memory due to all the calls being monitored during execution. An ideal analyzer uses both static and dynamic analysis tools and conducts noise reduction on the fly to avoid slowdown and excessive memory usage. Finally, this analyzer maps dynamic analysis findings to the static architecture decomposition, independent of the nomenclature used throughout the system. In this way, one could take a slice of the program given a certain feature (for example, mobility), and view the static architecture of the feature as well as the behavioral flow. This information supports system documentation in the *4+1 Model* (see Section 5.1).

3. RELATED WORK

The primary contribution of our proposed tool is to facilitate comprehensive program analysis by augmenting ordinary static analysis with dynamic analysis tools and techniques. Using dynamic analysis, this tool must cleanse the data obtained during static analysis, remove noise from the analysis, and determine the core software entities that describe the abstract features described by the reference model. Related efforts to this goal include (16), which proposes a reduction the runtime time and memory overhead of dynamic analysis by using static analysis techniques to "safely omit" program instrumentation. However, this work focuses on dynamic analysis to determine software vulnerabilities and to validate type checking; it is our goal to instrument a system in its entirety. If one analyzed a distributed system feature-by-feature, it would likely be possible to use static analysis data as a means for targeting specific areas in which to instrument a system, as (16) describes.

(3) underscores the complementary nature of static and dynamic analysis data, not because of the noise overhead of either approach, but because static analysis tends to be more conservative in its results (because it does not evaluate function pointers, specific evaluation over a given input, and so on) than dynamic analysis. On the other hand, (3) argues that dynamic analysis data can be too specific, and fail to capture the general case. They then propose to complement the “conservative and sound” analysis produced by static analysis techniques with the “efficient and precise” analysis resulting from dynamic analysis techniques.

4. RESULTS OF DYNAMIC ANALYSIS OF AGENT FRAMEWORKS

Using in-house and open-source analysis tools, a large data set was gathered from the subject agent frameworks, as well as from sample agent systems running on these frameworks. The majority of raw static analysis data was run through the Bunch clustering system to create GXL tree graphs, which were then viewed and analyzed in ClusterNav to find commonalities described shortly. Dynamic analysis data was filtered and viewed using tools within the SCE package (which in turn uses Bunch for clustering) using SMQL queries common to all of the systems.

The common subsystems identified through the use of static and dynamic analysis tools has informed both the ASRM as well as possible reference architectures derived from it. Our goal, however, was to identify sequences of common behavior exhibited by existing agent systems to both validate and augment our findings.

4.1 Identification of Patterns of Execution

Ideally, all existing agent frameworks map directly to the reference model from an architectural perspective. An analysis of a representative subset is presented in this subsection. The following is the general scenario that is used to exercise the individual frameworks. Any modifications are noted appropriately.

An agent m causes two other agents, s_1 and s_2 , to be created. These agents send a message to each other. When s_1 or s_2 has sent their message, and received the message sent by the other agent, they cease execution. Agent m then migrates to another platform and creates two more agents, and the process repeats.

This scenario is used to investigate the migration and message passing aspects of several agent frameworks. Typically, these components also exercise the other components described in the reference model. For example, migration requires a search of the directory service, has security concerns, needs to deal with agent management functions, and involves coordination. Likewise, the message passing aspect generally exercises communications and security. A brief overview of the scenario from a dynamic analysis is first presented, followed by an in-depth analysis highlighting components and tracing execution.

Execution of this scenario is traced using the EJP (EJP Software Tool) tool. EJP allows the execution to be traced and conclusions to be drawn from analysis of these results. Several figures are included in the next few analysis subsections showing the raw output of EJP. However, we found it difficult to extract quickly just the most important information from these call trace graphs. Although they are accurate, calls such as the standard Java library calls add noise to the graphs. Recursive calls are captured and add multiple extra levels to the graph. Because we are informing a reference model, we are looking to capture features from these method calls at a high level of abstraction. A sample graph output is depicted in Figure 1, after some of this manual filtering was performed to remove Java system calls, etc. Because existing tools did not prove adequate for removing this noise, we performed this filtering and analysis manually.

4.1.1 A-Globe

The A-Globe analysis was broken up into two parts to reduce dynamic analysis noise and for clarity of results. One part involved message passing and the other involved migration—thus deviating from the general scenario. Combining these two, which is certainly trivial, results in the original scenario described.

Overview. A-Globe uses the `Platform` class as the root of its framework. The `Platform` class controls `Containers` that are for all intents and purposes agents. From our analysis, we see that the `Platform` is instantiated followed by an `AgentContainer`. The `AgentContainer` acts as the interface between local agents and the framework instantiation. Its main job is to provide agent-specific resources such as a `MessageTransport` services. An `AgentManager` is used to manage the agents and is also identified from the dynamic analysis performed.

The call trace of the migrating agent while still on the source host is shown in Figure 2. The general procedure for a migrating agent seems to be to run (in this case, the migrating agent does nothing) and then

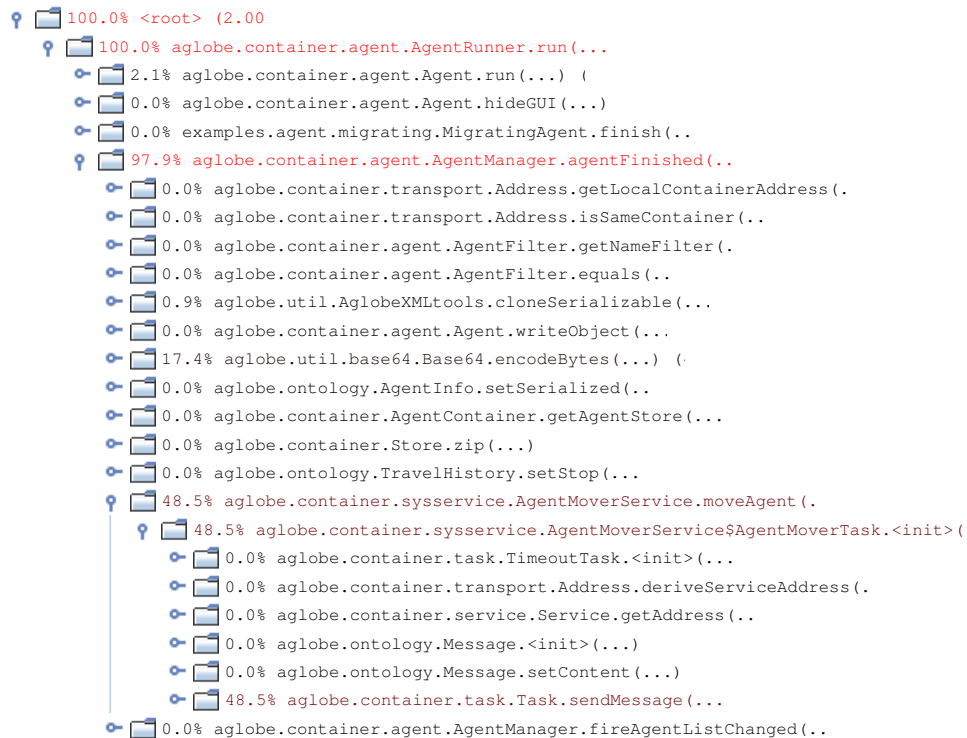


Figure 1: An example of the raw output from EJP. This example shows the compressed call stack of A-Globe preparing an agent for migration.

migrate using the `agentFinished` function. It migrates to the second instantiation of the agent framework.

Mapping The architecture of A-Globe maps directly to the reference model. For clarity, the physical world and infrastructure are not shown in the call graphs in this section. The Java Virtual Machine is the only visible sign of a platform. It occupies the root nodes through any Thread instantiations in the trees.

The framework is represented by the `Platform` class and its corresponding `AgentContainer`. The framework is always used for the agent to access system resources. The `Platform`'s `AgentContainer` provides shared objects, a message transport service, a directory service, a logger (not shown), and various other resources to the agents. An `AgentManager` handles instantiation of local agents on the framework, and initiates migration.

The agents also act in accordance with the idealized agent framework, described in the ASRM. Our analysis starts with the migrating agent. When it is time to migrate, the `agentFinished` function is called which uses XML to serialize the data and send it via the `AgentMoverService` that is part of the `AgentContainer`. This is seen in Figure 2. When the migrating agent arrives at the second framework instantiation, it is re-created by the `AgentManager` and executed. Here, it is simply de-registered from the directory, and is terminated.

Some features were difficult to inspect or to represent graphically because of the abundance of noise generated by the GUI. There is strong evidence from the A-Globe figures and raw dynamic analysis data gathered that there exists a `MessageTransport` management service within the framework that oversees message passing. This `MessageTransport` service registers possible message receivers and then calls a function similar to the `handleMessage` function.

Overall, migration and messaging clearly map to components in the reference model. Each framework makes available all of the important components mentioned in the reference model in addition to the migration, communication, and logging exercised in these experiments.

4.1.2 JADE

As described in the generic scenario description, a mobile JADE agent m creates static agents s_1 and s_2 on a host, terminates the platform after they have completed, then travels to a second host to repeat the process.

Overview. The JADE framework is made up of several classes, including `Boot` and the entire `jade.core` package. In particular, Jade uses containers similar to A-globe. The `AgentContainer` is used as an interface between the agents and the framework. Additionally, resources are delegated through the framework by way

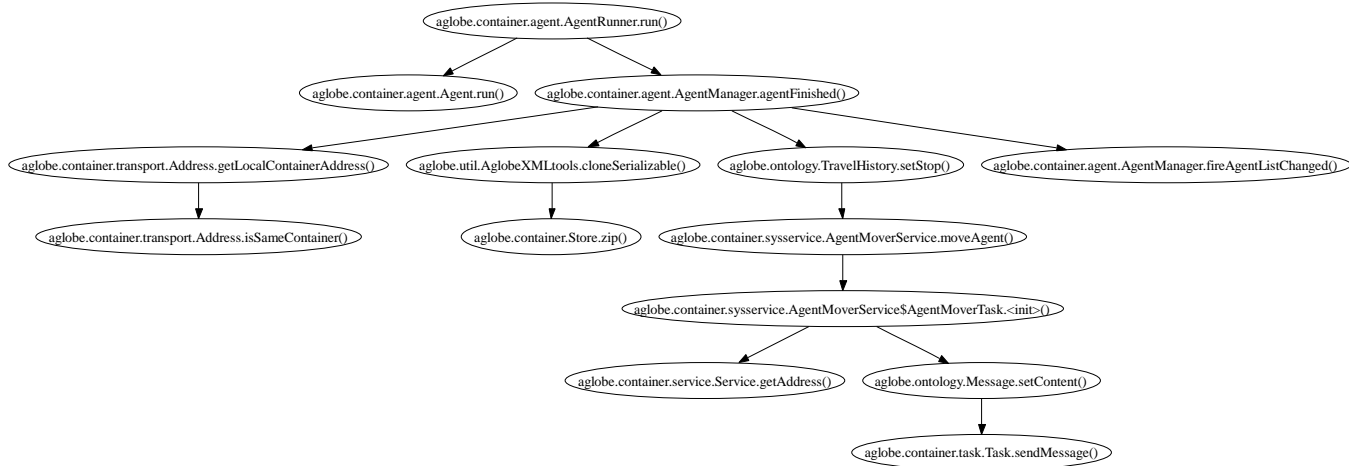


Figure 2: Call graph showing A-Globe migrating agent, m , before migration.

of a *ServiceManager*. The static agents are very interesting in the case of JADE because the progression of message sending is clear in the call graphs.

The JADE data (not shown, due to lack of space) shows the static agent being created, then cloned and sent to the other host. The post-migration snapshot shows how the agent is regenerated and it also creates the two static agents.

Mapping. Again, the environment and host layers are not shown, while the JVM represents a portion of the platform. When the JADE framework is started, an *AgentContainer* is created that acts as the interface between the framework and the local agents. The other agents are started inside the *AgentContainer*. The JADE framework also contains an *AgentManagementService*, a *MessagingService* with a well-defined and FIPA compliant ACL, an *AgentMobilityService* class that oversees mobility, and a *ServiceManager* that manages local resources. A directory service is found in the `jade.domain.ams` package.

First, we examined message passing. Two *JadeCommunicationAgents* are created that send messages to each other and then terminate. The agent sends a message by creating an *ACLMessage*, contacting the directory service with the `getAMS` function call, and using the communication service to send.

The migrating agent exhibits some interesting properties both before and after migration. First, the agent creates the static agents and then clones itself. Migration is processed through the *AgentMobilityService*. The directory service is also contacted in this process. In the after-migration snapshot, the first function call regenerates the agent and it proceeds in a similar fashion by creating the communication agents.

Overall, components in JADE—a good representation of a FIPA-compliant framework—map to the ASRM. Agents operate autonomously, but are monitored through an *AgentManager*. The other components are also present explicitly; security is the only exception that is not obvious in the dynamic analysis, though it is implicit through the JVM and the API.

5. USING DYNAMIC ANALYSIS RESULTS TO INFORM THE ASRM

We decided to focus our analysis on the framework layer for analysis because, of the layers that we defined in the ASRM, the framework most likely contains the implementation of the common agent activities like communication or agent migration. Moreover, this enabled us to inspect a number open-source and licensed frameworks that included such implementation in one place.

We faced some challenges in generating the analysis data, particularly due to the noise included in our dynamic traces, including JVM platform calls or library calls. Because our sample agent system was distributed between multiple framework instances, this problem was even more pronounced and even the simplest method calls resulted in an extensive method trace. Consequently, these traces required too much memory to process and filter; as a result, we found the data inspection process to be a largely manual one.

Our goal was to abstract the method traces into very abstract functional concepts based on common agent system behavior (the “scenario” described in Section 4.1); these functional concepts became the basis of the ASRM.

In reality, our dynamic traces actually informed our static analysis data, which in turn drove the discussion regarding the functional concepts of the ASRM. Our static analyses of the agent frameworks yielded an object

and package level decomposition of the modules present in the system. Often, these packages can be easily identified by simply inspecting the names of and interrelationships between these modules. For example, if two packages are tightly coupled when they should not be or vice versa, it is possible that one or both of the packages do not provide the functionality that the analyst expects (or the package(s) could represent dead code, etc.). This allowed us to make a list of our predictions about the common features of each agent framework. However, due to the potential inaccuracies of static analysis data, it then became necessary to execute a dynamic trace of the scenario described in Section 4.1.

Using the dynamic analysis trace data, we confirmed or modified our predictions regarding the modules obtained during static decomposition. For instance, a package that we originally expected to provide agent directory services actually provided communications functionality. Once we completed our list of common functionality, we grouped and abstracted them into abstract functional concepts similar to those described in Section 2.1. This list served as the basis of an extensive discussion to create the functional concepts portion of the ASRM. We reduced this list to those few abstract concepts that capture the essence of a typical agent system, while still enabling other existing agent systems (for example, an agent system that does not include a framework) to map their functionality to the ASRM. The goal, then, was not to simply create an aggregation of existing agent functionality, but instead to create an abstraction of an agent system informed by existing systems, providing a standard to which agent based systems can be reasonably mapped for comparison and discussion. Ultimately, this reverse engineering data will be revisited when reference architectures or actual agent system designs are derived from the ASRM. At that time, the static decompositions and dynamic traces will be used to identify specific standards regarding interrelationships between the functional concepts in the Reference Model.

Finally, we inspected other agent frameworks and domain-specific agent systems as a case study for mapping systems to the ASRM. To do this, we documented an “idealized agent framework” in which all of the functional concepts described in the ASRM are implemented. We then used static decomposition and dynamic traces as described in Section 4. This inspection was the basis of the case studies and idealized agent framework discussion in the ASRM, and will likely drive the creation of the Agent Systems Reference Architecture.

5.1 Documenting Results: The 4+1 Model

We documented our findings using the *4+1 Model* (5). This model allows for streamlined documentation of the software architecture, and standardizes the metrics for measuring adherence to the architecture from various perspectives. Using the 4+1 model, we produce analyses of systems from an aggregation and abstraction of these views. This abstraction is chosen because of the abstract nature of the ASRM: the reference model defines the high level features commonly found among agent systems, as validated by our study described in Section 4. Specifically, the Development and Physical Views comprise this structural UML description of the ASRM. These descriptions use more abstract UML descriptions than traditional 4+1 structural descriptions (which can exist at the deployment and object level of detail) to match the appropriate level of abstraction found in a reference model.

These descriptions do not prescribe conformance to the ASRM; instead, an idealized example is provided that includes all functionality defined by the reference model, implemented at the framework layer. Similarly, we aggregate the Development, Process and Scenarios Views into the behavioral UML description of the ASRM. These descriptions are found in the ASRM and are thus omitted here for brevity; however, they are primarily made up of UML Activity Diagrams and Sequence Diagrams to show the process flow and time-line (respectively) of the behavior observed during dynamic analysis.

6. CONCLUSION AND FUTURE WORK

It is well understood by the community that static analysis of software systems alone cannot yield a completely accurate view of the system. Dynamic code execution or memory allocation, as well as software evolution and dead code can create noise within static analysis results. Dynamic analysis can “color” the static analysis results to provide an as-built view of the software system. However, this was a manual process.

In this paper, we used static analysis data to define the structural views within the *4+1 model* (5). This is done using UML diagrams such as component diagrams and use cases. Then we use dynamic analysis data to describe the behavioral views and scenarios. This is also described using UML diagrams, including activity diagrams and sequence diagrams.

As described in Section 4, this was a manual process assisted by a number of tools. Even still, we ran into a number of challenges as a result of the noise generated by the platform layer, memory constraints, and nomenclature within the static analysis results for mapping our data to the ASRM. We are working to automate this process by using program slicing to validate structural system descriptions by mapping them to behavioral

descriptions.

We propose a tool called Karka to automate this process using *Reflexion Models* (10) that map static and dynamic analysis data to a reference architecture, freeing the analyst from the need to “filter” the noise from the collected data. In this case of agent systems, the reference architecture will be an idealized agent framework defined by and described within the ASRM.

Using this tool, we hope to explore a number of areas including security trust amongst agents, data flow within an agent system, thread allocations and run-times within a distributed agent system for performance benchmarking, and message passing and beliefs about the world state amongst agents.

REFERENCES

- [EJP Software Tool] Extensible java profiler (ejp). <http://ejp.sourceforge.net/>.
- [ASRM] Agent systems reference model. Technical report, Drexel University Project ACIN, Waterfront Technology Center, 200 Federal Street, Suite 300 Camden, NJ 08103, 2006.
- [1] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE — A FIPA-compliant agent framework. In *Proceedings of the 4th International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-99)*, pages 97–108, London, UK, 1999. The Practical Application Company Ltd. URL <http://sharon.cselt.it/projects/jade/PAAM.pdf>.
- [2] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002. ISBN 0201703726.
- [3] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9, 2003.
- [4] Roberto A. Flores-Mendez. Towards a standardization of multi-agent system framework. *Crossroads*, 5(4):18–24, 1999.
- [5] Philippe Kruchten. Architectural blueprints—The “4+1” view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.
- [6] Jurgen Lind. *Iterative Software Engineering for Multiagent Systems: The Massive Method*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001. ISBN 3540421661.
- [7] Raphael Malveau and Thomas J. Mowbray. *Software Architect Bootcamp*. Prentice Hall, 2001.
- [8] Pragnesh Jay Modi, Spiros Mancoridis, William M. Mongan, William C. Regli, and Israel Mayk. Towards a reference model for agent-based systems. In Nakashima et al. (11), pages 1475–1482. ISBN 1-59593-303-4.
- [9] Balint Molnar, Imre Berenyi, and Bence Siklosi. Function call trap of java codes with the help of aspectj and xml. In *CSMR '02: Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, pages 207–210, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1438-3.
- [10] G. Murphy and D. Notkin. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the Third Symposium on the Foundations of Software Engineering*, 1995.
- [11] Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, and Peter Stone, editors. *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8-12, 2006*, 2006. ACM. ISBN 1-59593-303-4.
- [12] M. Salah and S. Mancoridis. Toward an environment for comprehending distributed systems. In *Proceedings of the 2003 Working Conference in Reverse Engineering*, 2003.
- [13] Abraham Silberschatz, Peter Galvin, and Gred Gagne. *Applied Operating System Concepts*. John Wiley and Sons, Inc., 2003. ISBN 0471263141.
- [14] David Šišlák, Martin Rehák, Michal Pěchouček, Milan Rollo, and Dušan Pavlíček. A-globe: Agent development platform with inaccessibility and mobility support. In Rainer Unland, Matthias Klusch, and Monique Calisti, editors, *Software Agent-Based Applications, Platforms and Development Kits*, pages 21–46. Birkhauser Verlag, 2005. ISBN 3-7643-7347-4.
- [15] T. Win and M. Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving, 2002. URL citeseer.ist.psu.edu/win02verifying.html.
- [16] Suan Hsi Yong and Susan Horwitz. Using static analysis to reduce dynamic analysis overhead. *Form. Methods Syst. Des.*, 27(3):313–334, 2005. ISSN 0925-9856.