Software engineering challenges for mutable agent systems

Ladislau Bölöni Majid Ali Khan Xin Bai Guoqiang Wang
Yongchang Ji Dan C Marinescu
School of Electrical and Computer Engineering
University of Central Florida
Orlando, Florida, 32816, USA
lboloni@cpe.ucf.edu, ma680109@pegasus.cc.ucf.edu,
{xbai, gwang, yji, dcm}@cs.ucf.edu

ABSTRACT

In this paper we address mutability, an important feature of agent societies operating in heterogeneous computing environments. We present the Bond system, a FIPA compliant agent framework, with support for mutability. We propose a set of extensions to the Gaia agent-oriented design and analysis methodology. These extensions allow the methodology to handle certain important classes of mutable systems.

1. MUTABILITY

Mutation is a term used in genetic algorithms and evolutionary computing to indicate random changes of the chromosomes. Throughout this paper we use the term *mutation* to indicate controllable and well specified changes of a program at runtime. The executable of a *mutable program* could be self-modifiable or could be modified by an external entity.

As early as 1975, the Microsoft Basic interpreter for Altair contained self-modifying code, to fit into the limited address space available. Reconfigurable and mutable systems are an increasingly frequent occurrence in today's computing landscape. The ubiquitous web browser consists of a basic framework with multiple API's allowing the user to configure and extend the browser. Extensions such as plugins, codecs, drivers, applets, controls, and themes change the functionality of the browser, e.g., allow the user to view new media formats. Extensions are installed and uninstalled dynamically during the lifetime of the application. Extensions are usually developed by a third party. In addition to the desired changes of functionality, an extension may have undesirable side effects. For example, it may contain spyware. *Spyware* is code built into an extension to report back to some external entity usage statistics and/or private user data. Occasionally, viruses and worms make use of the very same extension API's to penetrate a system.

While web browsers are the quintessential user driven applications, reconfiguration and mutability are even more important for autonomous agents. Several agent systems supporting mutability emerged recently. Varela and Agha [9] proposed the SALSA language based on the actor programming paradigm. The SALSA language is compiled to Java and targets dynamically reconfigurable Internet and mobile computing applications. The SmartApps approach proposed by Rauchwerger [8] takes an approach of "measure, compare, and adapt if beneficial" for scientific applications. Restructuring occurs during various stages from the selection of the algorithm to compiler parameter tuning. The Bond agent system [3, 11] was one of the first Java based agent systems with support for strong mutability. A series of primitive operations performed on a multi-plane state machine supports reconfigurability of Bond agents. This mutation technique is called *agent surgery* [1].

Reconfigurability presents both an opportunity for developing more powerful software systems but can be also a Pandora's box. One way to study the effects of a reconfiguration is to identify and enforce *invariants* which are maintained during an operation. One such invariant is that a successful run in the original agent should also be successful in the modified agent. Other invariants can cover resource management, error handling, and security aspects.

Reconfigurable and mutable agents have a special appeal for highly heterogeneous systems where mutation and mobility are strongly coupled with one another. For instance, the resources available on a laptop and on a cell-phone differ so widely that agents cannot migrate from one to the other without being significantly reconfigured. The solution is to migrate only part of an agent to the new location and replace some of its components with ones compatible with the new environment. Of course, the systems must support a common environment, e.g., Java.

The remainder of this article is organized as follows. The Bond agent system, a FIPA compliant agent system with support for mutation is presented in Section 2. In Section 3, we discuss how one of the popular agent design methodologies, Gaia, can be extended to support mutable agents. A summary is presented Section 4.

2. THE BOND AGENT SYSTEM

The Bond agent system (currently at version 3) is a FIPA compliant agent development environment. It is built on top of the Java Agent Development Environment (JADE) framework and extends its functionality in several ways including: ease of development using a declarative approach, better introspection capabilities, and support for mutability through *agent surgery*. The other important component of an agent is its knowledgebase which contains the agents knowledge about the world and about itself, including its agenda. Bond knowledgebase is implemented using the the Protégé-2000 ontology editor. All Bond agents share a basic core ontology. Individual agents can also use custom domain-specific and agent-specific ontologies. The salient features of Bond system are:

- Support for development of behaviours (called strategies) for multi-plane state machine
- Support for creating and modifying the multiplane state machine using a declarative approach using the Python based Blueprint agent definition language .
- Graphical User interface for monitoring state machine's current status
- Support for Belief, Desire, Intention (BDI) model

A high level architecture of the Bond system is shown in Figure 1. The input provided to the Bond system the description of the agent in the Blueprint agent description language. Each state of this state machine has associated a strategy from strategy database. To enable automatic runtime assembly or mutability, Bond strategies extend the Jade behaviors with meta information concerning their roles, resource utilization, pre- and post-conditions, and other data. The Bond strategy database is an indexed and machine searchable collection of strategies. The multi-plane state machine of the individual agent is assembled from strategies pre-existent in the strategy database based on a description in Blueprint.

2.1 The Blueprint agent description language

A Blueprint specification is a set of declarative statements that describe the internal structure of the agent, i.e. its knowledgebase, set of planes, set of states within each plane and the transitions between these states etc. Besides describing the initial multi plane state machine configuration of the agent, it can also be used to specify surgical operations on the agent by adding or removing states/transitions from the state machine by modifying the blue print description.

An sample Blueprint description for a component of a task engine is presented below. The structure of the multi plane state machine constructed based upon this specification is shown in Figure 2.

```
includeKnowledgeBase("TaskComputation")

createPlane("ApplicationScheduler")
s = bond.strategydb.GreedyApplicationScheduler
                    (agent,'Computation')
addFirstState(s, 'scheduler')
s = bond.strategydb.DummyStrategy(agent)
addState(s,'dummy')
addTransition('scheduler','dummy',SUCCESS)
addTransition('scheduler','dummy',FAILURE)

createPlane("IntentionExecutor")
s = bond.strategydb.SwingExecutor(agent)
addFirstState(s,'swingExecutor')
s = bond.strategydb.WaitAndTransitionStrategy
                  (agent,1000,SUCCESS)
addState(s,'transition')
s = bond.strategydb.DummyStrategy(agent)
addState(s,'dummy')
addTransition('swingExecutor','transition'
              ,SUCCESS)
addTransition('swingExecutor','dummy',FAILURE)
addTransition('transition','swingExecutor'
              ,SUCCESS)
```

2.2 Mutations in Bond: Agent surgery

Agent surgery is a technique to change the behavior of a running agent. As noted earlier the structure of an agent is specified using by a Blueprint. Runtime modifications, add or remove states, and transitions from a plane, can be performed by surgical operations of this structure. For example, the Blueprint presented above specifies an agent whith a Swing based user interface. To adapt this agent for a device which does not support the Swing API we can perform the surgery described by the following surgical Blueprint:

```
setPlane("IntentionExecutor")
s = bond.strategydb.AWTExecutor(agent)
addFirstState(s,'awtExecutor')
addTransition('awtExecutor','transition',SUCCESS)
addTransition('awtExecutor','dummy',FAILURE)
addTransition('transition','awtExecutor',SUCCESS)

removeTransition('swingExecutor','transition'
              ,SUCCESS)
removeTransition('swingExecutor','dummy',FAILURE)
removeTransition('transition','swingExecutor'
              ,SUCCESS)
removeState('swingExecutor')
```

The surgery modifies the `IntentionExecutor` plane. The Swing execution engine is seamlessly replaced by AWT execution engine. Such surgical operations will frequently be required to make an agent conform to a volatile environment in which it has to operate.

3. SOFTWARE ENGINEERING CHALLENGES FOR MUTABLE AGENT SYSTEMS

As agent systems migrate from research laboratories to the world of commercial software development and enterprise computing, new design and analysis techniques must be developed. It becomes increasingly clear that the established software methodologies, such as object-oriented analysis and design, are inadequate or insufficient for the analysis and design of agent systems. Software engineering methodologies specifically designed for agent systems become necessary.

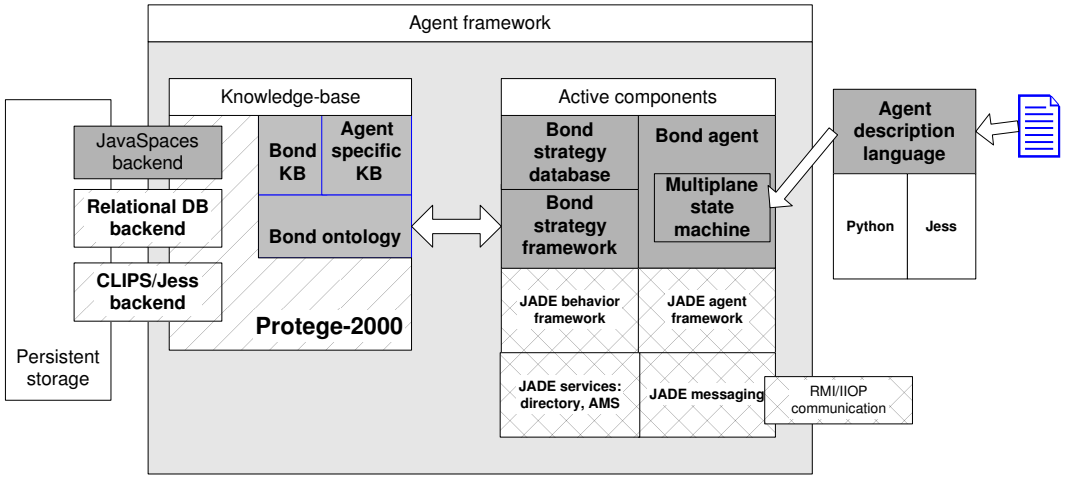

Figure 1: The design of the Bond system

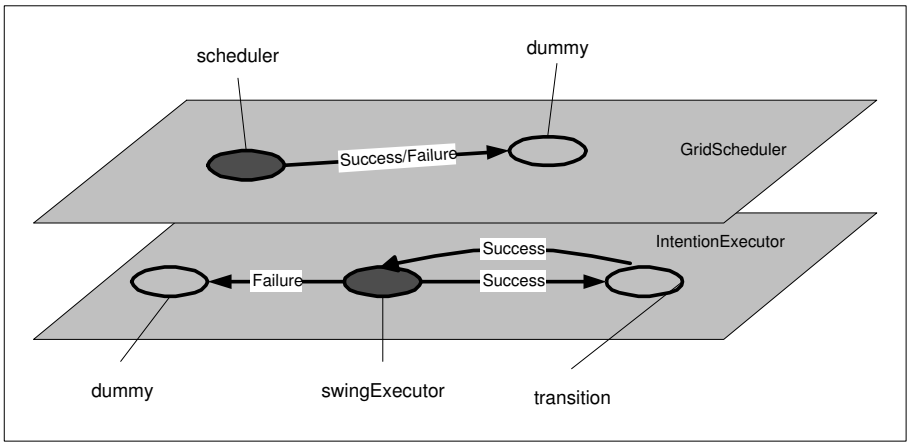

Figure 2: The finite state machine model: Active states are grayed out

Some of these new developments, such as [5, 7, 10], are building upon the existing object oriented methodologies and techniques, e.g., design patterns. Other approaches are building on techniques for knowledge engineering [4] or on formal methods and languages, e.g., the extensions for the Z language [6]. Some agent systems have developed their own analysis and design approaches, targeted to the particularities of the agent system such as Cassiopeia.

The introduction of mutable agents creates new problems for the agent analysis and design methodologies. The analysis step needs to take in consideration of the possibility of the agent being significantly modified during its lifetime. The design step needs to offer information about which agents should be mutated, at what moment of their lifecycle, and what kind of mutation should be performed. In fact, the methodological discipline is more important for the case of mutable agents.

We now discuss the effect of mutable agents on one of the popular agent design methodologies, the Gaia approach [10].

The Gaia methodology, with its roots in object oriented approaches such as FUSION, is a good fit for FIPA compliant agent systems such as Bond, *as long as they do not mutate*. In fact, the authors of [10] explicitly spell out among the applicability requirements that (a) the organizational structure of the system is static and (b) the abilities of the agents and the services they provide are static, do not change during runtime. Our goal is to investigate the feasibility of the removal of these constraints and the changes in the methodology implied by this removal.

We emphasize that no methodology can handle randomly mutating agents. We limit our discussions to the two most important mutation operations:

- Adding new functionality (roles) to the agent.
- Adapting the agent to new requirements or a different set of available resources.

Space limitations prevent us to address important scenarios,

such as: removing functionality, splitting agents, or merging agents.

3.1 Adding new functionality to the agent

In terms of the Gaia methodology, adding new functionality to the agent is equivalent to saying that the agent will be able to function in new *roles*, while maintaining the previously existing ones. More formally, considering the reconfiguration event e, we can say that if the agent was able to fulfill a set of roles R before the event, and after the event, it will be able to fulfill a set of roles R' with $R \subset R'$.

The corresponding structural definition in the AM_1^{mp} model, employed by the Bond system, is that the extending functionality is an agent surgery operation, which transforms agent A to agent A' and for every run R where the agent A is successful, the agent A' will be successful as well. In [2], we demonstrated that elementary surgical operations, such as adding states, adding transitions, and removing transitions labelled with FAILURE, maintain this property. In addition, the surgical operation of adding a plane can also maintain this property subject to a set of disjointness conditions.

There is a good mapping between the Gaia concept of roles and the structural implementation. Just as the agent might not be taking on a certain role, although it would be qualified to do it, the agent might not perform certain runs. Thus we can say that through adding new functionality to the agent by agent surgery, the agent acquires the ability to fulfill new roles. The nature of these roles needs to be clearly spelled out.

The attributes associated with the roles in the Gaia methodology will also be maintained: the *responsibilities*, *permissions*, *activities*, and *protocols*. As these attributes are applied to the agent in a cumulative way, an important goal of the analysis process of an agent surgery operation is to determine that there is no conflict between the attributes of the agents, that is, the invariant of the agents are properly maintained.

A different question, which needs to be answered by the analysis process, is the opportunity and moment in the agent lifecycle when the new functionality needs to be added to the agent. The answer to this question is not an explicit point in the lifecycle of the agent, but a *trigger*, a specific set of conditions under which the mutation becomes desirable.

While one might argue that the agents can be designed so that they can fulfill all the possible roles needed during its lifetime. This argument ignores the cost associated with having such a multifaceted agent. To put this in a different context, not all the workers of a company need to be qualified for all the professions. Nevertheless, it is a frequent occurrence that a worker needs to be sent to additional training so that he or she can fulfill new roles. In many cases, these events can be quite accurately predicted and even planned. Similar considerations apply to agents.

3.2 Adapting an agent to new requirements

Another very important subclass of agent mutations are when an agent is reconfigured to adapts a changing environment expressed with a changing set of resources. The most typical one is that the reconfiguration needed after migration to a new host computer. Another example, which does not involve migration, when an agent running on a host computer needs to adapt between the almost complete control over resources (when the user is not logged in) with only minimal resource allowances (when the user is logged in and working).

Expressed in the terms of the Gaia methodology, the agents will implement the same set of roles with a different set of attributes.

- The *responsibilities* and the *protocols* implemented by the role will remain unchanged during this operation[1].
- The permissions associated with the role will be different. The Gaia methodology collects under the concept of permissions notions such as resource usage and security permissions. The analysis process is responsible to assure that the permissions required by the agent after the transformation are satisfied by its new context. For example, it has to be assured that the memory and processor power requirement are satisfied.

As in the case discussed in the previous section, the goal of the agent analysis and design is to determine the opportunity and the nature of the agent mutation. The opportunity for migration can be expressed in terms of hard and soft triggers. A *hard trigger* is a boolean function which tells us if an agent can not fulfill the requirements of its role in a given context. A *soft trigger*, on the other hand, is a cost-benefit analysis of the agent which might suggest a mutation if this leads to an increased performance of an agent. Generally, hard triggers are leading to changes with implementations with lower resource usage, while soft triggers are biased towards implementations with higher performance and associated higher resource (permission) requirements.

The adaptation scenarios can be described in the terms of a design decision tree. In the Gaia approach (as in many other software engineering approaches) the designer moves from an initial, very high level specification, to an increasingly more specialized choice. These choices for a *design tree*, which will be assumed to be a separate one for every role the agent can play. For the example given in Figure 3, The choices are all valid approach to create a user interface of the program. During the design process, a set of decisions are made to determine the permissions (in the resource usage sense) of the program. The leaf nodes typically correspond to the actual implementation of a program, but of course, not all the potential choices are actually implemented. For an agent which does not require reconfiguration, only one branch of the design tree is explored. Once the decision is made, the design tree is not used in the actual operation of the agent, although it might be kept, implicitly in the design documentation.

For a reconfigurable agent, more than one leaf node is fully

[1]This is a relatively crude approximation which assumes that the functionality of an application is completely specified by the original specification.

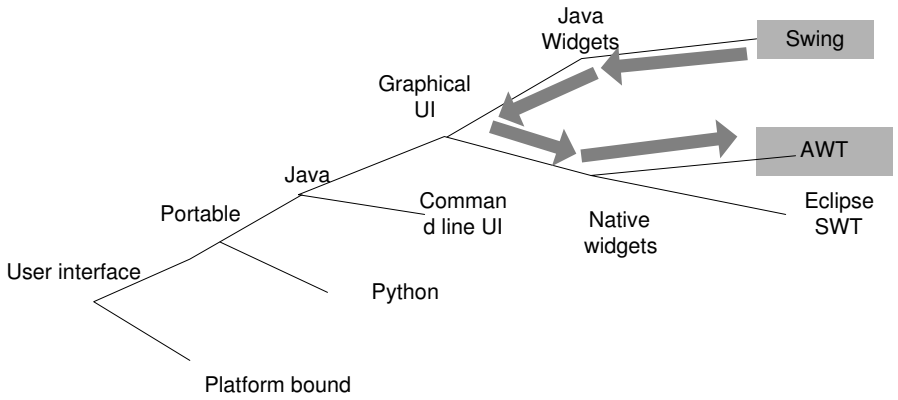

Figure 3: Design decision tree

instantiated. We need to emphasize that this involves the same analysis and programming steps as for any other agent. However, in this case, the design tree has a practical utility during the lifetime of the agent. Let us assume an agent which is executed with a Swing based, fully graphic user interface, and needs to be migrated to a Personal Digital Assistant. A simple check of the permissions of the role will tell us that the current implementation will not work and a reconfiguration operation is needed.

The role, therefore will be moved backwards in the design tree. At every steps releasing the assumptions made at the given point, until the assumptions on the current point in the tree do no conflict with the new context of the agent. Normally, however, we are usually at a more or less abstract specification level, which on its own, is not runnable. Therefore, we will start to move toward the leafs of the decision tree again, this time however making decisions according to the new context. Our goal is to reach a fully implemented leaf node which conforms to the current set of permissions. The required transformation will be, therefore, one which transforms from the original design choice into a new design choice.

4. SUMMARY

This paper explores mutable agents, and the software engineering challenges posed by their design and analysis. The evolutionary aspect of such agent imposes an even stronger requirement for a disciplined development process.

We present an extension to one of the popular agent development methodologies (Gaia) for the design and analysis of populations of mutable agents. We focus on two important subclasses of mutations: (a) add new functionality to an agent and (b) reconfigure an agent to adapt to a new set of resource requirements while maintaining its functionality.

An unexpectedly good fit between the Gaia design methodology and mutable agents seems to exit. While new questions need to be answered and new invariants must be verified, we found no need to introduce new high level concepts. This is a good omen for the reconfigurable agent technology. The populations of changeable and mutable agents might be manageable.

5. ACKNOWLEDGEMENTS

The research reported in this paper was partially supported by National Science Foundation grants MCB9527131, DBI0296107, ACI0296035, and EIA0296179.

6. REFERENCES

[1] L. Bölöni and D. C. Marinescu. Agent surgery: The case for mutable agents. In *Proceedings of the Third Workshop on Bio-Inspired Solutions to Parallel Processing Problems (BioSP3), Cancun, Mexico*, May 2000.

[2] L. Bölöni and D. C. Marinescu. A component agent model - from theory to implementation. In *Second Intl. Symp. From Agent Theory to Agent Implementation in Proc. Cybernetics and Systems, Austrian Society of Cybernetic Studies*, pages 633–639, April 2000.

[3] L. Bölöni and D. C. Marinescu. An object-oriented framework for building collaborative network agents. In H. Teodorescu, D. Mlynek, A. Kandel, and H.-J. Zimmerman, editors, *Intelligent Systems and Interfaces*, International Series in Intelligent Technologies, chapter 3, pages 31–64. Kluwer Publising House, 2000.

[4] F. Brazier, B. D. Keplicz, N. R. Jennings, and J. Treur. Formal specification of multi-agent systems: A real-world case. In *First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 25–32, San Francisco, CA, USA, 1995. AAAI Press.

[5] D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of BDI agents. In W. V. de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038)*, volume 1038 of *LNAI*, page 56. Springer-Verlag, 1996.

[6] M. Luck, N. Griffiths, and M. d'Inverno. From agent theory to agent construction: A case study. In J. P.

Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Proceedings of the ECAI'96 Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents III*, volume 1193, pages 49–64. Springer-Verlag: Heidelberg, Germany, 12–13 1997.

[7] J. Odell, H. Parunak, and B. Bauer. Extending uml for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pages 3–17, 2000.

[8] L. Rauchwerger, N. M. Amato, and J. Torrellas. SmartApps: An application centric approach to high performance computing. In *Proc. of the 13th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC), August 2000, Yorktown Heights, NY.*, pages 82–92, 2001.

[9] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.

[10] M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[11] Bond webpage. URL `http://bond.cs.ucf.edu`.