Software engineering challenges for mutable agent systems

Ladislau Bölöni, Majid Ali Khan, Xin Bai, Guoqiang Wang, Yongchang Ji, and Dan C. Marinescu

University of Central Florida, Orlando FL 32816, USA,
`lboloni@cpe.ucf.edu`,
WWW home page: `http://www.cs.ucf.edu/~lboloni`

Abstract. Reconfigurability and mutability are important features of agent societies operating in heterogeneous computing environment. At the same time, they pose major challenges to the software engineering process. In this paper we review these challenges and discuss their implications towards the agent oriented software engineering methodologies. 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. These results are presented in the context of the Bond system, a FIPA compliant agent framework, with support for reconfigurability and mutability.

1 Introduction

Reconfigurable and mutable systems are an increasingly frequent occurrence in todays computing landscape. As early as 1975, the Microsoft Basic interpreter for Altair contained self-modifying code, introduced to overcome resource limitations (only 4K of memory available for the interpreter). For a closer example, a today's web browser is custom application, consisting of a basic framework with multiple API's for configuration and extension, and a large number of extensions (plug-ins, codecs, drivers, applets, controls, themes and so on). These extensions are usually developed by third parties, are installed and uninstalled dynamically during the lifetime of the application, and frequently changing the behavior of the application in a radical way.

Some of the changes in the functionality of the application are desired or at least approved by the user: an example of such an extension is the ability to view new media formats. Frequently, some of the effects are undesirable from the point of view of the user: some third party extensions contain *spyware*, pieces of code which report usage statistics and other information about the user. Occasionally, viruses and worms can use the very same extension API's. Although software upgrades are usually considered beneficial, the externally initiated upgrade to the version 2.0 of the software in the Tivo digital video recorders have actually removed functionality previously existent on the system, an action considered malicious by users, but beneficiary by the company.

While web browsers are the quintessential user driven applications, reconfiguration and mutability are even more important for autonomous agents. Many popular agent frameworks can be used to implement reconfigurable agents, and a number of recent work explicitly focuses on reconfigurability. Varela and Agha [26] 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 project proposed by Rauchwerger [22] 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 [5, 6] 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* [3].

Reconfigurability presents both an opportunity for developing more powerful software systems but can also be a Pandora's box. One way to study the effects of 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 there is a strong interdependency between mobility and reconfigurability. For instance, the resources available on a laptop and on a cellphone 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 (this leads to the concept of plasticity of user interfaces as discussed in [25]).

In the remainder of this article we will introduce a typical agent framework with support for reconfigurabity and mutation, the Bond agent system (Section 2). Next we overview the challenges posed by reconfigurability for the agent oriented software engineering process and propose a set of extensions to one of the popular agent design methodologies (Section 3). We conclude in section 4.

2 Implementing mutability in software agents

2.1 Methods of implementing mutability in agents

Mutable software can be implemented using a large number of techniques. There are two important classification criteria in this respect: the *granularity of the mutation* and the *mutation technique* deployed by the agent system.

The granularity of the mutation refers to the size of the smallest component being changed. *Source code level mutation* refers to performing changes in the source code of the applications. This is a well known technique for interpreted languages such as Lisp or Prolog. Recently, many popular applications targeting the Java platform, such as the aspect oriented programming tool AspectJ [16] or code instrumentation systems such as JFluid [14] are working in this way. *Machine code level mutation* involves performing transformations in the executable code of fully compiled languages. The mutation can happen either on the executable file as residing on the filesystem or in the runtime executable format in the operational memory. *Object level code mutation* includes transformations at the level of compiled but not linked libraries. Code obfuscators [10] or orthogonal persistency systems are working this way. A special case of these applications are adaptive libraries which can decide during runtime the algorithm to deploy such as Standard Template Adaptive Parallel Library [1]. The most popular way to perform mutation on applications is at the *component level*. Components have the advantage that are sufficiently large and their behavior is well understood and can be isolated in specifications. Also, the interaction between components happens through relatively well documented interfaces. Developers of components are usually encouraged to implement and use components as black boxes. Thus the accidental side effects, which are the most difficult aspect of source code level mutation are significantly reduced at the component level.

Our last classification criteria is based on the techniques used to perform the mutation or adaptation operation. The typical weak mutability applications are implemented through *extension API* through which external components (plugins) can be plugged to the unchanged main application. The advantage if this approach is that the main application can still retain the control through the well controlled application interface. The plugins can run in a controlled environment (such as the sandbox model of the Java applets). Another advantage is that mutation though an extension API are reversible, which is not true in general for other techniques.

Runtime *composition API's* allow strong mutation by changing the structure of the application, through the way in which its components are assembled. In these techniques the components are considered closed primitives. We need to identify two important concepts in this approach: the nature of the components and their assembly model. In order to be suitable for mutation techniques, a component model need to allow the *runtime* assembly of components. For example, the C++ class model can be seen as a component approach but it does not allow for runtime assembly. However, C++ based component models such as Microsoft DCOM, KDE KParts or Gnome's Bonobo does allow runtime assembly. The Java class model on the other hand, allow runtime modification, provided that custom classloaders and special access methods are used. For the Bond agent framework, the primitive components are the strategy objects while the assembly model follows the multiplane state machine model of agenthood.

In absence of a explicit API, mutability can be implemented through *reverse engineering*. Most of the time, this approach is taken for non-cooperative agents, for externally initiated mutations. This is the approach taken by viruses, but also by many legitimate applications, such as code instrumentation applications (for the purpose of debugging or performance profiling), or some persistency applications. One can argue, that the reflection capabilities offered by many modern languages, such as Java or Python are in fact low-level API's towards reverse engineering the code (of course, useable only if code obfuscators were not used).

2.2 The Bond agent system

The Bond agent system (currently at version 3) is a FIPA compliant agent development environment. As the high level architecture of the Bond system (Figure 1) shows, the Bond system integrates a number of high quality open source tools. The communication framework and the strategy model is built on top of Java Agent Development Environment (JADE) framework [29]. The knowledgebase model is using the the Protégé-2000 ontology editor. The scripting support is based on the Jython [30] implementation of the Python language. Reasoning is implemented using the Jess expert system shell. Besides integrating these tools into an easy to use, consistent framework, the Bond system provides an additional set of functionalities and serve as an experimental platform. A screenshot of a running Bond agent is shown in Figure 2.

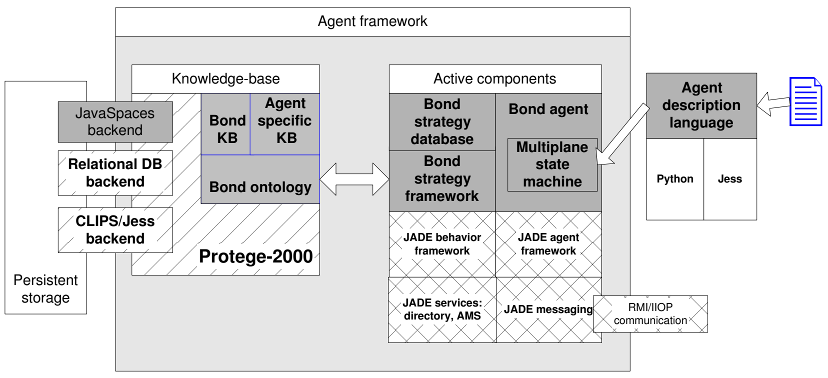

Fig. 1. The design of the Bond system

In this article we will concentrate on those features of the Bond system which provide explicit support for reconfigurability and mutability: the multiplane state machine model and the Blueprint agent description language.

2.3 Multiplane state machine framework

Most agent frameworks are based on the notion of discrete actions generated the agent framework. The components generating the actions are usually called behaviors or strategies, and they are assembled into a structure usually being a variation of the active object design pattern (reactor [23] or proactor [21]).

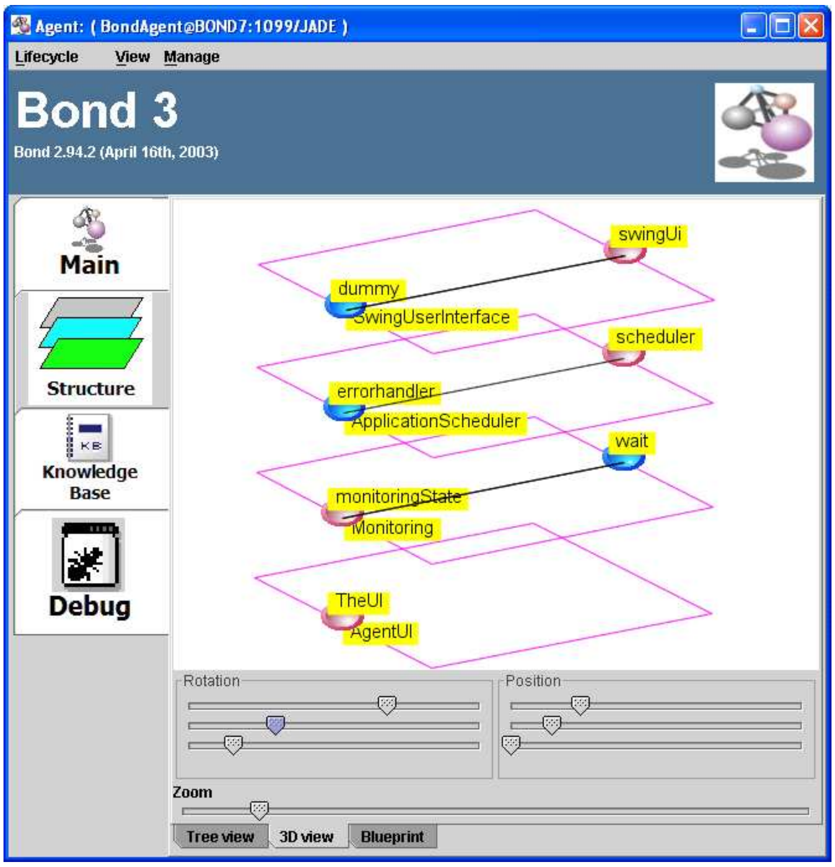

Fig. 2. A screenshot of the Bond agent system

The activation model of the agent framework is an important component in the behavior of an agent, and it is usually a trade-off between expressivity and enforced order.

While agent frameworks such as Jade allow an essentially arbitrary composition of the behaviors, others, such as SmartAgent [12] or the Bond framework enforce a simpler framework, which is easier to manage and model. The data structure enforced by the Bond agent model for behavior composition is the *multi-plane state machine* (Figure 3). The planes of the state machine express the concept of parallelism, the simultaneously active behaviors are described by a state vector.

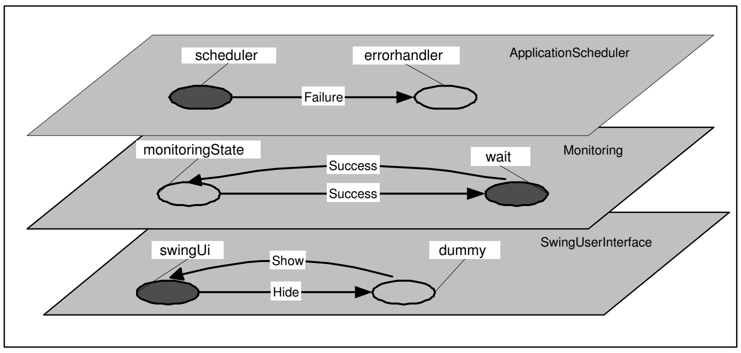

Fig. 3. The finite state machine model: Active states are grayed out

2.4 The Blueprint agent description language. Mutability support

Bond agents are assembled using the Python based Blueprint agent description language. Although strategies can be written directly in Blueprint, the main purpose of the language is the assembly of the agents from components, initialization of the knowledgebase and description of the mutation operations. 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.

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.

Besides describing the initial multi plane state machine 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.

There are seven primitive mutation actions allowed by blueprint language:

– Adding a state (`addState`)
– Removing a state which doesn't have any incoming or outgoing transitions. (`removeState`)
– Adding a transition between two states. (`addTransition`)
– Removing a transition (`removeTransition`)
– Adding a plane (`addPlane`)
– Removing an empty plane. (`removePlane`)
– Replacing the strategy of an inactive state. (`switchStrategy`)

The first six operations are a complete set of operations on the multiplane state machine. In [3] we have proved invariants regarding all these operations. The rationale behind the seventh operation is, that although it can be decomposed into a sequence of the first six, this decomposition is usually very complex (removing the transitions, removing the state, re-adding the state with a different strategy, re-adding the transitions). This sequence of operations, however, breaks the conditions of most invariants which can be proven on the agent. These invariants can be more easily proven to hold in the case of a "switch" operation.

We called the type of mutability implemented this way *agent surgery*. While this name is intended to be taken with tongue in cheek, it is a surprisingly good description of what is actually happening. It is a change, enacted on a living agent (Bond agents do not need to be stopped for this operation), it is performed by a knowledgeable entity (as opposed to random mutation) and it consists of removing unnecessary components or adding new components / prostesis / enhancements.

We now proceed to present an example of specifying an agent through a Blueprint description, and follow up to reconfigure it using a surgical Blueprint script. The agent we are using in our sample is a simplified version of a computation steering agent, a part of a grid application framework. The agent contains an application scheduler, a monitoring system and a user interface. The Blueprint description for this agent is presented below. The structure of the multi plane state machine constructed based upon this specification is shown in Figure 3.

```
includeKnowledgeBase("Grid")

# the application scheduler
createPlane("ApplicationScheduler")
strategy = bond.strategydb.GreedyApplicationScheduler(agent)
addFirstState(strategy, 'scheduler')
strategy = bond.strategydb.ErrorHandler(agent)
addState(s,'errorhandler')
addTransition('scheduler', 'errorhandler', FAILURE)

# the monitoring plane
createPlane("Monitoring")
strategy = bond.strategydb.Monitoring(agent)
addFirstState(strategy, 'monitoring')
addState(WaitAndTransitionStrategy(agent, 1000, SUCCESS), "wait")
addTransition('monitoring', 'wait', SUCCESS)
addTransition('wait', 'monitoring', SUCCESS)

# the Java Swing based user interface
createPlane("SwingUserInterface")
s = bond.strategydb.SwingUi(agent)
addFirstState(s,'swingUi')
addState( DummyStrategy(agent), 'dummy')
addTransition('swingUi', 'dummy', HIDE_UI)
addTransition('dummy', 'swingUi', SHOW_UI)
```

The agent presented here has a Java Swing based user interface. This Java user interface library is known to be a large consumer of resources, and it is currently not available for embedded Java implementations. This would prevent the agent to migrate to portable devices. One solution is to replace the user interface with an interface based on the AWT library - which is relying on the native windowing toolkits of the platform and is available on embedded computers [1]. The change from a Swing user interface to an AWT one can be performed with the following *surgical blueprint script*.

```
setPlane("SwingUserInterface")
if currentPlane()=='swingUi':
    triggerTransition(HIDE_UI)
# removing the old visual strategy
removeTransition('swingUi', 'dummy', HIDE_UI)
removeTransition('dummy', 'swingUi', SHOW_UI)
removeState('swingUi')
# adding the new, AWT based user interface
s = bond.strategydb.AWTExecutor(agent)
addState(s, 'awtUi')
addTransition('awtUi', 'dummy', HIDE_UI)
addTransition('dummy', 'awtUi', SHOW_UI)
triggerTransition(SHOW_UI)
```

An alternative, simpler way of transforming the agent is:

```
if currentPlane()=='swingUi':
    triggerTransition(HIDE_UI)
s = bond.strategydb.AWTExecutor(agent)
changeStrategy(plane = 'IntentionExecutor', state = 'UIStrategy', s)
triggerTransition(SHOW_UI)
```

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 The software engineering perspective of adaptive agents

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.

Some of these new developments, such as [17, 27], are building upon the existing object oriented methodologies and techniques and design patterns. There

[1] But AWT is a considerable less advanced user interface than Swing, and it is almost never used on desktop platforms

are a number of efforts underway to extend the UML language and the associated software methodology for agent oriented programming ([19, 9, 2]) or for modelling the knowledge-base of the agents [11, 13]. Many methodologies are drawing inspiration from the Belief-Desire-Intention model [17, 20]. Other approaches are building on techniques for knowledge engineering [7] or on formal methods and languages, e.g., the extensions for the Z language [18]. The Tropos methodology [8] is adapting ideas from techniques developed for business process modelling and reengineering (the i* notation [28]), at the same time retaining the mentalistic notions of belief-desire-intention and related models. 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 into consideration 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. Generally, 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 [27]. 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 [27] 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. Several extensions proposed to the Gaia methodology extend the scope of the methodology. The ROADMAP methodology [15] extends Gaia with formal models of knowledge, role hierarchies and representation of social structures. It also extends the permission attributes to allow roles to change the definition or attributes of other roles, although it does not cover the issue of how the modified agents are represented.

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. Fortunately, the most frequently encountered operations can be classified in a set of well understood classes:

- Adding new functionality to the agent (understood as the ability to function in new roles).
- Removing functionality from an agent (understood as the inability to function in a set of roles in the future).
- Adapting the agent to new requirements or a different set of available resources.
- Splitting an agent.
- Merging agents.

While many other types of change can be envisioned (for example, transferring a functionality from an agent to a different agent), we will concentrate in this paper on the five mutation types shown above.

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 [4], 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.

We found 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 by adding new functionality to the agent by agent surgery, the agent acquires the ability to fulfill new roles.

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 invariants 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. [2]

For a concrete example of how the diagrams of the Gaia method can be annotated to handle reconfigurable agents, let us turn to the airline industry for

[2] Although 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.

an example. During a flight, the number of (human) agents on the airplane are playing a set of well defined roles: passenger, pilot, stewardess and so on. There are, however, exceptional situations, such as an emergency landing in which case some of the passengers are required to take on new roles, such as to assist the crew in opening the doors.

The agent model diagram of this situation is modelled in Figure 4. In this approach, the exceptional situation is modelled as the mutation trigger. The new state of the agent is modelled in the Gaia agent diagram as a new agent type. Mutated agent types are marked in the diagram with the letter M. The mutation operation is specified using a thick arrow with the "+" label attached (indicating that the mutation retains all the previous roles of the agent [3]).

One of the potential difficulties associated with this model is the potential for an agent type space explosion.

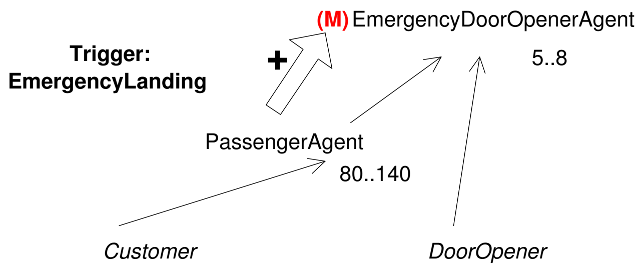

Fig. 4. Agent model for an airplane emergency situation

Another diagram which needs to be adapted to handle the needs of the reconfigurable agents is the acquaintance model. While the Gaia acquintance model does not deal with the details of the interaction, mutations on the agents can frequently change the acquaintances as well. The example presented in Figure 5 also deals with a fictional situation on an airplane. The sudden symptoms of sickness on some passengers and a stewardess triggers a request from a stewardess which makes a passengers step into the role of a doctor. This creates a new interaction pattern, between the doctor and the sick stewardess and the sick passenger. These acquaintance lines would have not existed if the mutation would not have happened.

[3] Strictly speaking, the "+" label is not needed on the agent model diagram, because the operation can be inferred from the role inheritance lines. It is however useful on the other diagrams where the role inheritance is not present.

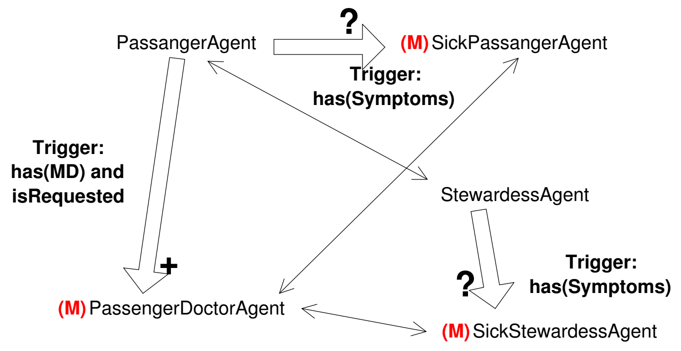

Fig. 5. Acquiantance diagram for the "sickness on an airplane" scenario

3.2 Removing functionality from an agent

There can be several reasons for removing functionality from an agent. One of them is the course-grain equivalent of *garbage collecting*. At some moment in the agent's lifecycle we might find that some of the roles of the agent will never be activated. The ability to perform roles which will not be activated usually implies some kind of waste of resources. Examples are memory and disk space occupied, network bandwidth by polling for messages which will never arrive, processor time for maintaining data structures which will never be queried. It is therefore useful to periodically perform a role-based garbage collection process on the agent. While the process is related to the garbage collection process in programming languages automated programming languages such as Java, there are also some specific differences.

- The garbage collection process happens at the level of components and subsections of the knowledgebase (instead of allocated memory chunks).
- Active components (code) can be also garbage collected.
- The internal structure of the agents can greatly simplify the garbage collection process. For example, for agents based on a multiplane state machine model, the garbage collection process can be reduced to a reacheability analysis on the state machines.
- The probability that a garbage collection step will recover some resources is generally lower, then in the case of garbage collecting memory in applications. Moreover, the benefits of role-based garbage collection will tend asymptotically to zero, unless some other mutation operations in the meanwhile new components.

The role-based garbage collection can happen any time during the agent's lifetime. However, the life cycle of an agent provides some natural points where the side effects of the garbage collection process are minimal. Such points are:

after every transition (for state machine or Petri net based agents), before checkpointing, before moving (for mobile agents) and after every mutation.

Another scenario for removing functionality from agents, is to trigger specialization in groups of large agents. In this case an agent factory generates agents with the ability to perform a set of tasks. The agents are then specialized through removing their ability to perform a certain subset of tasks (Figure 6). The specialization mutation can be either performed under the control of a remote agent, or it can be performed by the agent itself, based on the initial experiences of its lifecycle. This approach is very natural for distributed solution of problems with the "divide and conquer" approach, such as the popular Contract Net protocol [24]. Another application is the emergence of communication patterns. There is solid evidence that the visual and auditory pathways of mammals are generated using a similar method in the early stages of life. The hardware industry had chosen a similar approach for the zone codes of DVD drives. The DVD drives are manufactured as generic devices, which are capable to play DVDs from any zone. During the first several uses, the DVD drives decides on a particular zone coding, and it permanently removes its own ability to play DVD's from other zones.

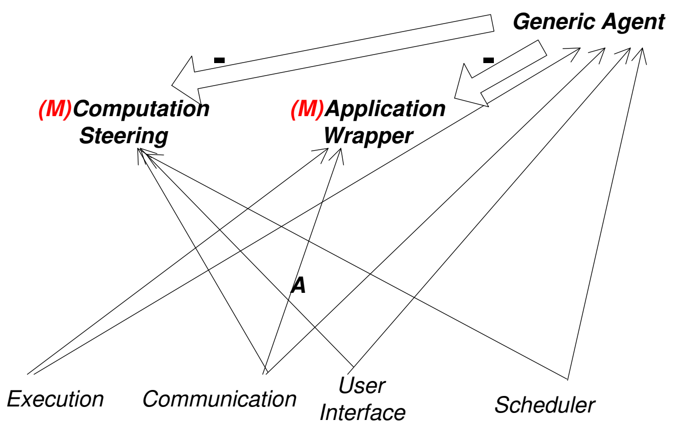

Fig. 6. Specialization of generic agents through removal of roles.

3.3 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[4].
- 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.

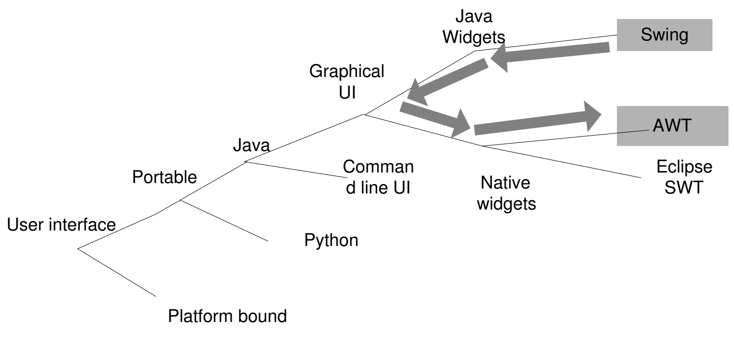

Fig. 7. Design decision tree

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)

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

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 7, the choices are all valid approaches 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 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.

3.4 Splitting and merging agents

Splitting and merging agents are operations which are surprisingly easily implementable in many agent systems. Moreover, very compelling application scenarios can be found for them. The reason why this technique is not more frequently used in applications is because there is no accepted software engineering methodology to specify them. Also, splitting and merging is not a mechanically intuitive concept such as agent mobility[5].

[5] Agent mobility, in practice involves: stopping an agent, serializing it, transfering data and code over the network, signalling back that the transfer was succesful, destroying the agent in the original location, restarting it on the new location. It is quite obvious that this is a complex, and not entirely intuitive process which has little to do with movement in the mechanical sense. However, the power of the metaphor made people accept the notion of mobile agents easier because most of us can visualize it.

The software engineering process for the splitting and merging agents involves most of the notions presented in the previous sections. We need to identify the *triggers* of the split and merge mutation, and the resulting new agent types need to be fit in the agent models. We propose the use of the split and merge operators as presented in Figure 8.

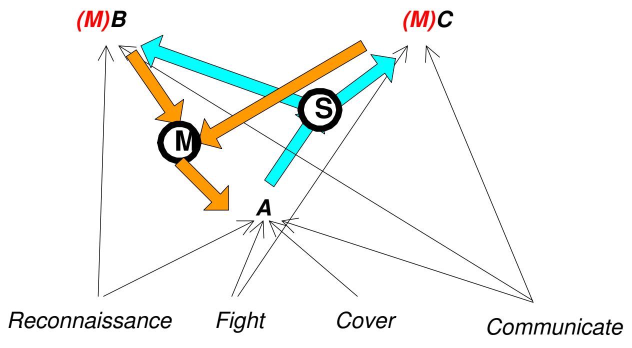

Fig. 8. The agent model for splitting and merging operations

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. An unexpectedly good fit between the Gaia design methodology and mutable agents seems to exist. 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, after all.

5 Acknowledgements

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

References

1. P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In *Pro-*

ceedings of the 14th Workshop on Languages and Compilers for Parallel Computing (LCPC), August 2001.

2. T. Arai and F. Stolzenburg. Multiagent systems specification by UML statecharts aiming at intelligent manufacturing. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 11–18. ACM Press, 2002.
3. 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.
4. 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.
5. 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.
6. Bond webpage. URL `http://bond.cs.ucf.edu`, 2003.
7. 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.
8. P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. A knowledge level software engineering methodology for agent oriented programming. In J. P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 648–655, Montreal, Canada, 2001. ACM Press.
9. G. Caire, W. Coulier, F. J. Garijo, J. Gomez, J. Pavon, F. Leal, P. Chainho, P. E. Kearney, J. Stark, R. Evans, and P. Massonet. Agent oriented analysis using Message/UML. In *AOSE*, pages 119–135, 2001.
10. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, 1997.
11. S. Cranefield, S. Haustein, and M. Purvis. UML-based ontology modelling for software agents, 2001.
12. M. L. Griss, S. Fonseca, D. Cowan, and R. Kessler. SmartAgent: Extending the JADE agent behavior model. In *Proceedings of the Agent Oriented Software Engineering Workshop, Conference in Systemics, Cybernetics and Informatics*. ACM Press, 2002.
13. C. Heinze and L. Sterling. Using the uml to model knowledge in agent systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 41–42. ACM Press, 2002.
14. JFluid web page. URL `http://www.sunlabs.com/projects/jfluid`, 2003.
15. T. Juan, A. Pearce, and L. Sterling. ROADMAP: Extending the Gaia methodology for complex open systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 3–10. ACM Press, 2002.
16. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
17. 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.

18. 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.
19. 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.
20. L. Padgham and M. Winikoff. Prometheus: A methodology for developing intelligent agents. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 37–38. ACM Press, 2002.
21. I. Pyarali, T. Harrison, D. Schmidt, and T. Jordan. Proactor - an object behavioral pattern for demultiplexing and dispatching handles for asynchronous events, 1997.
22. 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.
23. D. C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In J. O. Coplien and D. C. Schmidt, editor, *Pattern Languages of Program Design*, pages 529–547. Addison-Wesley, 1995.
24. R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, 29(12):1104–1113, 1980.
25. D. Thevenin and J. Coutaz. Plasticity of user interfaces: Framework and research agenda. In *Proceedings of Interact'99, vol. 1, Edinburgh: IFIP*, pages 110–117. IOS Press, 1999.
26. C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.
27. 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.
28. E. S. K. Yu and J. Mylopoulos. From E-R to "A-R" - modelling strategic actor relationships for business process reengineering. In P. Loucopoulos, editor, *Proceedings of the* 13^{th} *International Conference on the Entity-Relationship Approach*, pages 548–565, Manchester, UK, 1994. Springer.
29. Jade webpage. URL `http://sharon.cselt.it/projects/jade/` .
30. Jython web page. URL `http://www.jython.org`.