Chapter x

Adaptation and mutation in multi-agent systems and beyond

Ladislau Bölöni and Dan Cristian Marinescu

1 Introduction

Reconfigurable and mutable systems are an increasingly more popular. As early as 1975, the Microsoft Basic interpreter for Altair contained self-modifying code, introduced to overcome resource limitations (only 4K of space available for the interpreter). A contemporary web browser is a custom application, consisting of a basic framework with multiple extension API's and a large number of plug-ins, codecs, drivers, applets, controls, themes and other add-ons. These extensions are usually developed by third parties, installed/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 functionality 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 user's point of view: some third party extensions contain *spyware*, pieces of code which report usage statistics and other information about the user. Occasionally, viruses and worms use the very same extension API's.

While web-browsers are the quintessential user-driven applications, reconfiguration and mutability are even more important for autonomous agents. Recently, several agent systems with support for mutability have emerged. Varela and Agha proposed the SALSA lan-

guage based on the actor programming paradigm (Varela and Agha 2001). The SALSA language is compiled to Java and targets dynamically reconfigurable Internet and mobile computing applications. The SmartApps project (Rauchwerger, Amato and Torrellas 2001) takes an approach of ”measure, compare, and adapt if beneficial” for scientific applications, with the restructuring occurring at various levels from the selection of the algorithmic approach to compiler parameter tuning. The Bond agent system (Bölöni and Marinescu 2000a) was one of the first Java based agent systems with support for strong mutability, introducing a mutation technique called *agent surgery*, which describes the mutations as a series of primitive operations on a multi-plane state machine.

Reconfigurable and mutable agents have a special importance in highly heterogeneous systems, such as ad-hoc networks. In such systems, mutation and mobility are strongly intertwined concepts. For instance, the resources available on desktop computer and a cellphone differ so significantly that agents cannot be migrated from one to the other without being reconfigured, even if both platforms are able to run the same language, such as Java. One solution is to replace the components of the agents with components that satisfy the constraints imposed by the new host. Another choice is to migrate only part of the agent to the new site, using split and merge operations.

Although the ability to change an application at runtime is a programming technique dating back to the beginnings of computer science, there is not yet an universally accepted formal theory or a software engineering model of mutability. The subject of mutability, however is a cross-cutting concern in many fields of computer science. The goal of this chapter is to provide a review of various approaches, and to present original research done by the authors.

We review the contributions from various fields, as well as the concepts related to mutability (or at least, an understanding of the alter-

native usages) is necessary. In Section 2, we propose a set of classification criteria for mutable and reconfigurable applications, and propose a taxonomy based on this criteria.

One approach is based on formal modeling of the change of behavior as a result of mutation. The choice of the formal model of agency greatly influences our ability to describe and reason about change. In Section 3 we compare various models in relation to their ability to model change. We present a set of results obtained in the context of the model of a multi-plane state machine of active objects and discuss the advantages and drawbacks of the approach. As a completely random mutation is unlikely to be of any practical use, we are especially interested in *invariants* relative to operations.

In Section 4 we approach adaptability and mutability from the software engineering point of view. Traditional agent oriented software engineering (AOSE) methodologies are not well prepared to handle mutable agents. In fact, the software engineering process, traditionally seen as the steps necessary to transform an initial specification to a final product, needs to be re-thought and evaluated. The methodologies need to be extended and modified to handle the challenging issues raised by mutable agents. We present a proposed set of extensions to the Gaia agent development methodology which supports the analysis and design of agent systems containing mutable agents.

2 A taxonomy

Mutable and reconfigurable applications can be traced back to the beginnings of computer science. The number of scientific articles dealing with reconfigurability can be counted in the hundreds. Thousands of widely deployed applications are using techniques of mutability. Despite of this, there is no general theory of mutation in applications (and indeed, its desirability and feasibility has not yet been properly investigated). The lack of a common vocabulary of talk-

ing about mutable applications makes it difficult to relate the work done by researchers in disjoint fields, such as workflow management (van der Aalst 1999, Han, Sheth and Bussler 1998), user interfaces (Thevenin and Coutaz 1999), scientific computing (Rauchwerger et al. 2001, An, Jula, Rus, Saunders, Smith, Tanase, Thomas, Amato and Rauchwerger 2001) or agents (Varela and Agha 2001, Decker, Sycara and Williamson 1996, Barber, Goel and Martin 2000). Similarities, which might form the core of a general theory, might go unnoticed, because The vocabulary used to describe an adaptive workflow is significantly different from the one used to describe an adaptive user interface.

In spite of the fact that specifications usually do not list mutability as a desirable property of an application, mutable systems have a noticeable presence. Mutability was frequently introduced in small steps in commercial applications, as an answer to general requirements, and resulting in increased flexibility[1].

The first step towards understanding mutable systems is to attempt to introduce some order in the terminology of the field. The current emphasis on semantics in computer science is a sign that researchers understand the importance of classification (ontologies, taxonomies or even simple terminologies without relations of terms) in the understanding of systems. The next step is to identify some classification criteria which, besides the obvious benefit of categorizing, allows us to identify the most important aspects of mutability. Finally, integrating some of the classification criteria into a taxonomy gives an additional order into the field.

2.1 Alternative names

[1]A good example is the evolution of web browsers. The ability to dynamically update the browser with plugins, applets, ActiveX controls or client side scripting was not part of the initial design of the World Wide Web. These features, under the influence of business and customer pressure, were introduced step by step, with many false starts and intermediate versions.

The requirement to develop software which responds to changes in its local or global environment was handled in many subfields of computer science. As a result, many names were proposed to denote the different concepts, creating a virtual Babel of mutable programs. In this section, we will try to review the different terms used in different fields, without an attempt to completeness.

The adjective *adaptive* appears in many contexts, although with slightly different meanings. *Adaptive workflows* (van der Aalst 1999, Han et al. 1998) are used in terms of workflows, which need to be changed as a result of a runtime event.

Adaptive user interfaces (Thevenin and Coutaz 1999, Pribeanu, Limbourg and Vanderdonckt 2001, Luyten, Vandervelpen and Coninx 2002) were used for describing user interfaces which adapt to the capabilities of the device. Depending on the technique of the adaptation, this can range from a simple data driven, weak mutability process (replacing an image based web-page with a text based one) to component level, hard mutability event (replacing a Swing user interface with a WAP based one). Other terms proposed were plastic user interfaces, migrateable user interfaces, runtime user interface transformations.

Runtime software evolution is used in the recently emerging field of unanticipated software evolution (Gustavsson and Assmann 2002).

2.2 Classification criteria

We propose five classification criteria targeting the most important aspects of mutability. They are based upon:

- the **amplitude** of change: weak vs. strong mutability
- the **granularity**: source code, machine code, library, component level mutation

- **continuity** of interactions: runtime vs. stoptime
- the **initiator** of the change: externally initiated vs. self initiated change.
- the **mutation technique**: extension API, compositional API, reverse engineering, data driven, hardware and others.

In the remainder of this section, we discuss all these possibilities in detail and provide concrete examples. Most of these criteria apply to agents and to non-agent applications such as interactive programs or client-server systems. To allow for a larger pool of examples, we will occasionally refer to non-agent applications as well. This is justified by the fact that many recent versions of applications traditionally considered interactive, are outfitted with autonomous, agent-like behavior. For example media players such as Microsoft Media Player or RealOne have agent-like subsystems.

Additional criteria exist. For example, the classification based on beneficial vs. malicious nature of the change. Unfortunately, deciding on the malicious or beneficiary nature of a mutation is often nontrivial. Viruses are universally considered malicious, while software updates are considered beneficiary. However, the externally initiated upgrade to the version 2.0 of the Tivo digital video recorders removed previously existent functionality. This action was widely considered malicious by users, but beneficiary to the company. Instead of asking if a certain mutation is beneficial or malicious, we need to ask the question: in behalf of whom a certain agent is acting? This question is a fundamental one in agent research, but it does not relate (only) to adaptation and mutation.

2.2.1 The amplitude of the change: weak vs. strong mutability

The first classification criteria refers to the amplitude of the change in the behavior of the system. We call a mutation *weak* if the modified system satisfies the same informal requirement specification as

the original one. Examples are mutations which adapt applications to new resource conditions, extend their capabilities for new data formats or give them new abilities which are in the line of their original specification. Patches which fix bugs and vulnerabilities in software are also included here.

In contrast, a *strong mutation* is changing the specification of the system in a radical manner. Examples are agent systems with strong mutability support (Bölöni and Marinescu 2000a), or hardware devices modified with *modchips* (such as the ones transforming the XBox game terminal in a general purpose Linux computer (XBo 2001).

2.2.2 The granularity of mutation

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 (Kiczales, Hilsdale, Hugunin, Kersten, Palm and Griswold 2001) or code instrumentation systems such as JFluid (JFl 2003) are working in this way. *Machine code level mutation* requires transformations in the executable code of fully compiled languages. The mutation can happen either on the executable file residing on the filesystem or on the runtime executable format in the operational memory. *Object level code mutation* includes transformations at the level of compiled but not linked libraries. Examples include *code obfuscators* (Collberg, Thomborson and Low 1997) or *orthogonal persistence systems*. A special case are adaptive libraries such as Standard Template Adaptive Parallel Library (An et al. 2001) when one decides during runtime the actual algorithm to be used.

The most common approach to mutation involves the *component level*. The coarse grain components have independent and well un-

derstood specifications. Interaction between components happens through relatively well documented interfaces. Developers of components are usually encouraged to minimize the occurrence of side-effects (which are the most difficult aspect of source code level mutation). Component based mutations for interactive applications include applets, controls, themes, plug-ins etc. The behavior or strategy model used by many agent systems can also be the basis for component level mutation.

2.2.3 The continuity of interactions: runtime vs. stoptime

An important classification criteria for agent mutation is the ability of the agent to maintain the set of current interactions. In case of **runtime mutation** if the agent continues its execution during the mutation. For example, conversation protocols continue uninterrupted. When the agent or application is stopped and restarted to perform the mutation we talk about a **stoptime mutation**.

There are instances when the runtime mutation is a basic requirement. This is the case of self-healing fault tolerant systems. For interactive applications, such as media players or web browsers, performing updates during runtime is a matter of user convenience. On the other hand many software update systems require not only to stop the application, but also to reboot the computer.

2.2.4 The initiator of the mutation

Based on the initiator of the mutation, we classify the mutable systems in three categories. The mutation is *user initiated* when the change happens as a result of the direct user action. We assume that the user is fully aware that this action will trigger a modification of the program, such as a software upgrade or installation of a patch. An example of a popular application is the Fortify cryptography patch for the Netscape browser (For 1998). In some cases, the action initiated by the user has undisclosed side effects, as in the case of *spyware* applications. The second case is when the muta-

tion operation is initiated *externally* by an agent. Viruses modifying executables or anti-virus programs removing executable viruses are examples of this approach. Finally, the mutation can be initiated by the agent itself. This is typically the case of agents which can adapt themselves to changing environments such as the SmartApps framework (Rauchwerger et al. 2001), self-healing software or the ubiquitous *self-updating software*. Other examples are *enforced remote updates* such as the case of Tivo digital video recorder, the AOL Instant Messenger, or the Kazaa peer-to-peer file sharing network.

2.2.5 Mutation technique

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

Runtime *composition API's* allow strong mutation by changing the structure of the application. To qualify 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 is assembled during compile time. Other component models such as Microsoft DCOM, KDE KParts or Gnome's Bonobo does allow runtime assembly. The assembly model for these components is a hierarchical document model, held in container applications. The Java class model on the other hand, allows runtime modification, provided that custom class-loaders and special access methods are used. For the Bond agent framework (Bon 2003), the primitive components are the strategy objects while the assembly model follows the multiplane state machine model of agency.

In absence of an explicit API, mutation can be performed through *reverse engineering*. Often this approach is taken for externally initiated mutations on non-cooperative agents. This is the approach taken by viruses, but also by many legitimate applications, such as code instrumentation (for the purpose of debugging or performance profiling), or some approaches to orthogonal persistence. One can argue that the *reflection capabilities* offered by many modern programming languages (Java, Python, etc) are in fact low level API's for reverse engineering[2].

Data-driven reconfiguration techniques exploit the ambiguity between compiled code and data interpreted as how-to knowledge. Examples are knowledgebase applications such as Prolog programs, Jess/Clips knowledgebases, or applications such as the periodic update of the virus databases of anti-virus programs.

Finally, hardware mutation techniques such as modchips or Flash updates are operating at various levels of the computer hardware.

2.3 A taxonomy of mutations

In proposing the five classification criteria, we attempted to make them as independent as possible. Nevertheless, not all combinations represent practical systems. The most frequently encountered combinations are presented in the taxonomy tree of Figure 1. This tree is based on three of the criteria we presented (amplitude, technique and granularity). The remaining two criteria need special treatment. Continuity is a property which depends on the effort of the implementor - runtime mutations are usually preferred, but stoptime approaches are sometimes employed because of simpler and safer implementation. The initiator of the mutation applies to individual changes, not to the mutable system. The initiator of the mutation can be only a human user or a software agent; non-agent applications do not initiate

[2] These are, of course, usable only if code obfuscators were not used.

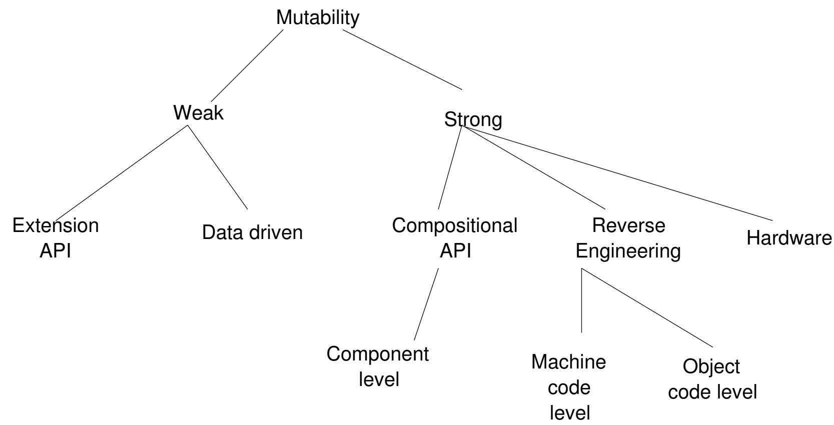

Figure 1. A taxonomy of the most frequently encountered mutability approaches.

mutations.

2.4 Other classification approaches

The classification presented here is a result of a selection of a number of classification methods proposed by various researchers.

Gusstavson and Assmann (Gustavsson and Assmann 2002) propose a classification criteria for runtime software changes based the *technical facet* and the *motivational facet*.

A taxonomy of program transformations for the purpose of code obfuscation is presented (Collberg et al. 1997).

3 A formal description of mutability

3.1 Agent models and mutability

Most researchers agree that autonomous behavior is the determining property of agents. A significant number of constructs have been proposed to describe the behavior of agents. Some of these were custom designed to describe agent behavior (such as the BDI model), while others were co-opted from other fields of computer science. Unfortunately, no single model provides a perfect fit from specification to implementation. There is a consensus of researchers and practitioners that during the development and deployment cycle, multiple models need to be employed.

Additional complexity arises from the fact that the perception about the best use of some of the models have changed in the years since they were first proposed. For example, approaches based on modal logic (Levesque, Cohen and Nunes 1990, Kinny, Georgeff and Rao 1996, Rao and Georgeff 1995) were originally considered as implementation models. Due to the complexity of modal logic, current approaches employ the concept of modal logic mostly as a specification methodology.

Petri nets, with their well established semantics are arguably the best framework to verify important properties such as liveness. They can readily capture concepts of concurrency, synchronization, contention for resources and so on. However, Petri nets are not considered a good fit as direct implementation models. As specification languages, Petri nets (in the form of colored Petri nets) are highly expressive, but many non-technical people find the Petri net descriptions less intuitive than other models (such as the UML activity or sequence diagrams).

The original statechart model (Harel 1987) is a popular approach for describing the behavior of real-time systems. But the expressivity of the model coupled with a lack of consensus on its semantics makes it rarely used as a basis for formal proofs. Statecharts have been successfully used as execution models for real-time systems and they have also been used for the execution models in agent systems such

as SmartAgent (Griss, Fonseca, Cowan and Kessler 2002a).

The UML activity diagrams, inspired both by Petri nets and statecharts have proved to be useful in capturing the semantics of business processes. In their currently standardized form they are not precise enough to be a basis of correctness proofs, although significant research exist towards establishing a precise, unambiguous semantics for UML diagrams (for example, the work of the precise UML group). Although not employable directly as implementation models, tools can be built which are directly generating agents from activity diagrams (for example the DIVA tool generates directly agents for the OpenCybele platform).

Models based on finite state machine based decomposition of active objects represent a trade-off between expressivity and complexity. They can be used to perform formal reasoning, but the results do not cover the activities encapsulated in the active objects. On the positive side, these models can directly serve as implementation models.

3.2 A multiplane state machine model of agent behavior

We consider a formal model of agency based on the decomposition of the behavior of the agent into a set of active objects arranged in a multiplane state machine. In order to describe the behavior of the agents in relation their environment, first a model of the environment (the agent's world) needs to be chosen.

The notation used in the following sections is summarized in the following:

Table 1. Models for describing agents, and their suitability for different stages of agent development, spe

Model / suitability...	for specification	for verification	as an implementation model
Modal logic (BDI etc)	good	fair (complexity)	poor (complexity)
Petri nets	fair (complex, not intuitive)	good	fair (complexity)
Statecharts	good	fair	good (specially for real-time systems)
Activity and sequence diagrams (UML 1 and 2)	excellent	poor	poor
Patterns (reactor, proactor, active object)	good	poor	excellent
State machines	good	fair (limited expressivity)	good

K	knowledgebase
S	strategy
G	goal or agenda
$M(Q, q_0, A, \Sigma, \delta)$	finite state machine
$\sigma(G, K, t)$	scheduling function

There are two views of the world we may consider, (a) the *view of an external observer* with a perfect knowledge of the world, who does not need take any action to acquire its knowledge and (b) the *view of an agent*, with an incomplete and imperfect knowledge who

needs to take actions to improve its knowledge about the world and these very actions change the state of the world. When we define the goal of the agent, we are thinking in terms of the first view, because we want to modify the true state of the world. However, the actions taken by the agent can be defined only in terms of the second view, the only one available to the agent. For example, a *state transformer function* as in Fagin et. al. (Fagin, Halpern, Moses and Vardi 1995) pp. 154 is expressed in terms of the view of an observer with perfect knowledge. Agent strategies are expressed in terms of the view of the agent.

3.3 Modelling agent behavior

We assume that the knowledge of the agent about the world is captured in the *knowledgebase* K of the agents. We usually see the knowledgebase of the agents as being a *model of the environment*; the knowledgebase is an approximation of measurable quantities in the environment. However, a detailed discussion of the representation approaches is outside the scope of this paper.

The **goal** or **agenda** of the agent G is a boolean function applied to the knowledgebase:

$$G : K \rightarrow \{\texttt{true}, \texttt{false}\}$$

The goal of the agent is defined on the knowledgebase because the environment is not accessible to the agent. Of course, this leads to the ability of the agent reach its agenda by self-deceit (modifying the knowledgebase without modifying the environment). Thus, assuring that the knowledgebase is a sufficiently good approximation of the environment is an important part of agent design.

Let $K*$ be the set of possible knowledgebases and $G*$ the set of possible agendas. We define a *strategy* of an agent A as a function which

maps the knowledgebase and agenda into a set of *intended actions*

$$S : (G * \times K*) \rightarrow \alpha*$$

Now, we are ready to introduce a first model of agents:
Definition 1 *The AM_0 model of an agent is the triplet (G, S, K) consisting of a goal G, a strategy S and a knowledgebase K.*
Definition 2 *We call* **concrete agent building** *the following problem: given an original knowledgebase K_0 and an goal G find a strategy function $S(G, K)$ such that the agenda will be satisfied in a finite time.*

3.3.1 Decomposition in the plane. Expressing "change"

From the implementation point of view the AM_0 model is a programmer's nightmare, because of the large monolithic strategy function S. This function is responsible for handling all the events and generating all the actions during the lifetime of an agent.
Definition 3 *The AM_1 model of an agent is the quadruple $(G, S^{(1)}, K, M)$ where G is the goal, K the knowledgebase and $S^{(1)}$ is a set of strategies $S^{(1)} = \{S_1^{(1)}, ...S_n^{()1}\}$. The finite state machine $M(Q, q_0, A, \Sigma, \delta)$ has the number of states $||Q|| = n$. The current state of the state machine q_i determines the currently active strategy $S_i^{(1)}$.*
Property 1 *AM_0 and AM_1 define the same equivalence classes of agents.*
We let the proof of this property to the reader. We define the higher order agent models AM_n recursively. Repetitive application of the property leads to the conclusion that every agent model $AM_n, \quad \forall n \in \mathbb{N}^+$ defines the same equivalence classes of agents as AM_0.

This property raises two questions: if all models are equivalent to AM_0, why introduce more complicated models? The other question

is that the AM_n models can be trivial cases (a single plane with a single state). We don't have guarantees that we can make a decomposition of the agent besides this trivial case.

The answer to the first question is that higher order models capture the natural engineering tendency to assemble solutions from smaller components. The second question is more subtle: one can certainly imagine problems which cannot be decomposed in a meaningful way (for example, none of the model variables has discrete values). What we can say is that most problems in practice are well suited to be decomposed into sub-problems. From an engineering point of view the individual strategies conform to the *active object pattern* (Lavender and Schmidt 1995).

3.3.2 Expressing concurrency

The AM_n model presented in the previous chapter decomposes the unique strategy of the model AM_0 into a number of strategies active one at a time. In this section we propose a method for further decomposition - we decompose the current strategy of the AM_n model into a number of strategies active concurrently.

Definition 4 *We call an m-plane* **scheduling function** *a function*

$$\sigma : (G* \times K* \times t) \rightarrow \{0, 1 \ldots, m-1\}$$

The **scheduling interval of** σ *is a time value* t_{sched} *such that*

$$\forall n \in \{0, 1...m-1\}, \forall K, \forall G, \forall t,$$
$$\exists \Delta t < t_{sched} \text{ such that } \sigma(G, K, t + \Delta t) = n.$$

This definition basically expresses our notion of a valid scheduling - there is a time interval in which every plane will be scheduled.

Definition 5 *We call a* **multiplane strategy** S^{mp} *with the planes* $S^0, S^1 \ldots S^m - 1$ *with the associated scheduling function* σ *a function*

$$S(G,K) = \begin{cases} S^0(G,K) & \textit{if } \sigma(G,K) = 0 \\ S^1(G,K) & \textit{if } \sigma(G,K) = 1 \\ \dots & \\ S^{m-1}(G,K) & \textit{if } \sigma(G,K) = m-1 \end{cases}$$

A multi-plane strategy expresses the idea that a strategy function can perform actions dealing with different parts of the world. The AM_1^{mp} agent model uses multi-plane strategies to express its behavior.

Definition 6 *The AM_1^{mp} agent model is a quintuple $(G, S^{(1)mp}, K, M^{mp}, \sigma)$. G is the goal, $S^{(1)mp}$ is a set of first order multi-plane strategies, K is the knowledgebase. The current behavior of the agent is determined by a multi-plane strategy composed of $S^{m_0^0}, S^{m_0^1} \dots S^{m_0^{m-1}}$.*

Definition 7 *An agent is not sensitive to the scheduling function if it is reaching its goal under one scheduling function, it will reach its goal under any scheduling function.*

From a formal point of view, this property is restrictive and difficult to prove. Nevertheless, this is an assumption made by all the non-real-time software currently in production. We assume that the software is not affected by: different processor speeds, scheduling based on the operating system choices and processor loads, external interrupts or garbage collection sessions. All these tacit assumptions are collected in the assumption that the agent is not sensitive to the schedule. While this does not cover the important subclass of real-time agents, it does cover the large majority of agents developed today. In the remainder of this section we will assume that the agents are not sensitive to the scheduling function.

We will denote a special set of states as *error states* and their associated strategies as *error handlers*. We also assume that the transitions in the finite state machine are labelled, and the labels are associated with internal or external transition events which trigger the corresponding transitions. We typically assume two labels with reserved meanings: `SUCCESS` and `FAILURE`. We assume that a transition

event for which there is no similarly labeled transition is interpreted as a FAILURE transition and their target states are error states. We call a run of an agent *successful* if it contains no FAILURE transition.

3.4 Mutation operators and invariance properties

We introduce a set of *mutation operations* on the multiplane state machines.

O_{as}	Add a state.
O_{rs}	Remove a state with no incoming or outgoing transitions.
O_{at}	Add a transition between two states.
O_{rt}	Remove a transition.
O_{ap}	Add a plane.
O_{rp}	Remove an empty plane.

A *change operation* C is an ordered list of operations $C = O_1 O_2 ... O_n \mid O_i \in \{O_{as}, O_{rs}, O_{at}, O_{rt}, O_{ap}, O_{rp}\}$. The set of operations is *complete*:

$$\forall M_1^{mp}, M_2^{mp} \; \exists C, \;\; M_1^{mp} \xrightarrow{C} M_2^{mp} \tag{1}$$

We can now propose a set of *invariance properties*.

Property 2 *Adding a new state to the agent does not change the behavior of the agent.*

Property 3 *Adding a new transition to the agent does not change the behavior of the agent in successful runs. It might turn some failed runs into successful runs.*

Property 4 *If we add a new plane to an agent and the output set of the strategies in the new plane is disjoint from the input set of the*

existing strategies, for all cases where the original agent achieved its agenda, the modified agent will achieve it as well.

Corrolary 1 *Adding an empty plane maintains the achievability of the agenda.*

Property 5 *Removing FAILURE transitions does not affect successful runs.*

Property 6 *Removing states unreachable from the current state of the agent, or removing transitions going to and from these states does not affect the behavior of the agent.*

Property 7 *Removing states which are reachable only through FAILURE transitions does not affect successful runs.*

One of the difficulties of the model appears when replacing a strategy with an equivalent one. This very simple and frequently encountered operation can be performed using only a series of operations (remove all the incoming and outgoing transitions, followed by the removal of the state, add the new state, re-add the transitions). Unfortunately these series of operations break so many of the conditions of the previous properties that no meaningful invariants can be proved. This dilemma can not be solved only in the terms of the multiplane state machine, but the properties of the associated strategy needs to be considered as well.

We call strategies S_1 and S_2 *equivalent for a run* R, if for all the states of the run, the strategies generate the same action $S_1(G, K_i) = S_2(G, K_i), K_i \in R$. Two strategies are *equivalent for an agent* if they are equivalent for all possible runs of the agent.

We introduce an additional operation:

O_{xs} Replace the strategy S of an inactive state with a strategy S'

For this operation we can prove the following property:

Property 8 *Replacing a strategy with an strategy equivalent for the run does not change the behavior of the agent for the run. Replacing*

a strategy with a strategy equivalent for the agent does not change the behavior of the agent.

3.5 How useful are the invariance properties?

To understand the practical usability of the properties presented in the previous section, we examine three questions. (a) How readily can agents be reduced to the multiplane state machine model? (b) How likely is that the transformations satisfy the conditions of the properties? (c) How strong are the conclusions of the properties?

Regarding question (a) we can state that a large class of agent systems can be reduced to the multiplane state machine model. Systems like Bond use the A_1^{mp} model directly. SmartAgent (Griss, Fonseca, Cowan and Kessler 2002b, Griss et al. 2002a) is using a model very similar which can be easily equated to an A_1^{mp}. Generally, any agent systems which is assembling agents based on active objects (behaviors, strategies etc) can be readily modeled in A_1^{mp}. On the other hand, systems which does not employ a component model will likely contain a less localized execution trace. These systems can still be modeled as state machines, but the resulting model will necessarily be finer grained, and thus less likely to be useful. The secret of successfully applying the invariance properties is that the individual states correspond to the granularity level of the mutation.

Question (b) is more difficult without actually performing a statistical study of the ways in which mutation is used in agents. The best we can do, is to make some observations based on typical usages. The conditions of the properties are relatively relaxed and easily checkable. The notable exception is Property 4, which requires the computations of the input and output sets of components; this is very difficult or impossible to carry out in an automatic way.

The last question is the strength of the conclusions. Some of these results seem trivial, because they promise a complete lack of change on

the global behavior of the agent. Certainly, adding a new state, with no incoming transition does not change the behavior of the agent. We call this result weak, because we know intuitively that this will not be the final state of the changed agent. Nevertheless, results of the presented type has proven to be very useful in many domains of computer science. Invariant transformations are an important component in both static (software) and dynamic (hardware) code optimizations. These techniques perform invariant transformations such as loop unrolling, reordering of independent instructions etc. to achieve faster program execution. In a different example, the extreme programming paradigm (Fowler, Beck et al. 1999, Beck 1999) employs a technique called *refactoring* which leads to improvement in the code structure through a series of invariant operations In static (non-runtime) software evolution, refactoring a system means to change its structure while retaining its semantics. Refactorings are attractive because they are a means to clean up a system, to facilitate maintenance and testing, and prepare functional changes. They have the advantage that they split software evolution in *harmless operations* (refactorings) whose effects can be checked by program analyzers or regression testing, and *difficult operations* that change semantics, but cannot be easily regression tested. Most of the operations in the properties presented can be thought as the equivalent of the refactoring operations.

Another weakness in the results of properties are the limitations of the multiplane state machine model. Although it supports concurrency, it does not take into account resource contentions. One result of this is the rather strong set of conditions of Property 4 regarding the disjointness of input and output sets. Moving to a model which handles resource contentions (such as Petri nets) and developing a set of similar properties for them would significantly extend the power of the theory (but it would create other problems in the practical applications).

4 A software engineering perspective on adaptive and mutable agents

With the gradual adoption of agent systems in commercial software development it became obvious that the established software methodologies, such as object-oriented analysis and design, are inadequate or insufficient for the analysis and design of agent systems. The agent oriented software engineering (AOSE) field emerged to fill this gap.

Some proposed methodologies, such as (Wooldridge, Jennings and Kinny 2000), are building upon the existing object oriented methodologies and techniques, e.g., design patterns. There are a number of efforts underway to extend the UML language and the associated software methodology for agent oriented programming ((Odell, Parunak and Bauer 2000, Caire, Coulier, Garijo, Gomez, Pavon, Leal, Chainho, Kearney, Stark, Evans and Massonet 2001, Arai and Stolzenburg 2002)) or for modeling the knowledge-base of the agents (Cranefield, Haustein and Purvis 2001, Heinze and Sterling 2002). Many methodologies are drawing inspiration from the Belief-Desire-Intention model (Kinny et al. 1996, Padgham and Winikoff 2002). Other approaches are building on techniques for knowledge engineering (Brazier, Keplicz, Jennings and Treur 1995) or on formal methods and languages, e.g., the extensions for the Z language (Luck, Griffiths and d'Inverno 1997). The Tropos methodology (Bresciani, Perini, Giorgini, Giunchiglia and Mylopoulos 2001) is adapting ideas from techniques developed for business process modeling and reengineering (the i* notation (Yu and Mylopoulos 1994)), 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 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 life-cycle, 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 (Wooldridge et al. 2000). 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 (Wooldridge et al. 2000) 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 (Juan, Pearce and Sterling 2002) 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 (roles) to the agent.
- Removing functionality from an agent

- Adapting the agent to new requirements or a different set of available resources.
- Transferring a functionality from an agent to a different agent
- Splitting an agent (for instance for the purpose of load balancing).
- Merging agents

4.1 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 $\mathfrak{R} = \{r_1, \ldots r_n\}$ before the event, and after the event, it will be able to fulfill a set of roles R' with $\mathfrak{R} \subset \mathfrak{R}'$.

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 (Bölöni and Marinescu 2000b), we demonstrated that elementary surgical operations, such as adding states, adding transitions, and removing transitions labelled `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 life-cycle when the new functionality needs to be added to the agent. The answer to this question is not an explicit point in the life-cycle 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.

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 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 cases 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 shown in Figure 2. In this approach, the exceptional situation is modeled as the mutation trigger. The new state of the agent is modeled 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]).

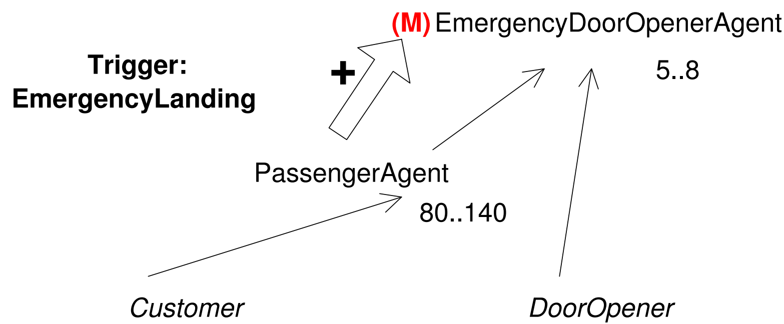

Figure 2. Agent model for an airplane emergency situation

Another diagram which needs to be adapted to handle the needs of the reconfigurable agents is the acquaintance model. While the Gaia acquaintance 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 3 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 stew-

[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.

ardess and the sick passenger. These acquaintance lines would have not existed if the mutation would not have happened.

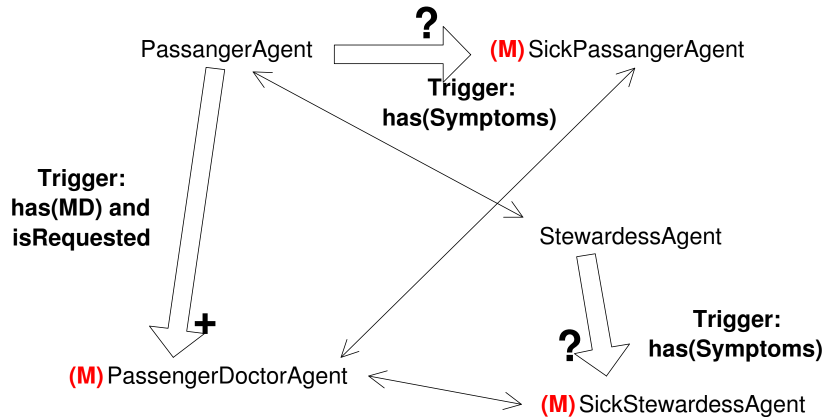

Figure 3. Interaction diagram the situation of a sickness on an airplane

4.2 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 life-cycle 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 reachability 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 in any time during the agent's lifetime. However, the life cycle of agents 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 4). 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 life-cycle. This approach is very natural for distributed solution of problems with the "divide and conquer" approach, such as the popular Contract Net protocol (Smith 1980). 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 DVDs from other zones.

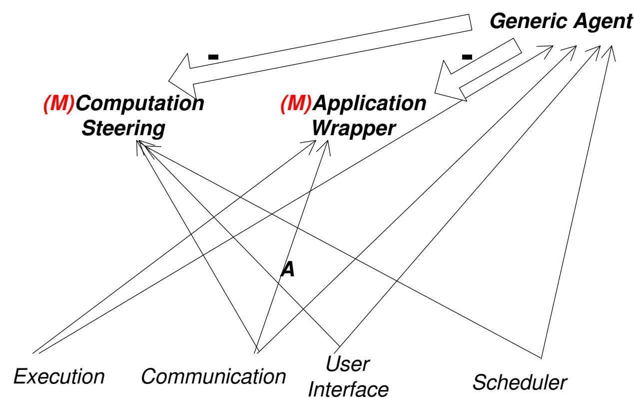

Figure 4. Specialization of generic agents through removal of roles.

4.3 Adapting to new requirements

Another very important subclass of agent mutations are when an agent is reconfigured and adapts to changing environment. A typical case is when the reconfiguration is needed after migration to a new host. 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 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 assures that the permissions required by the agent after the transformation are satisfied.

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 cannot 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.

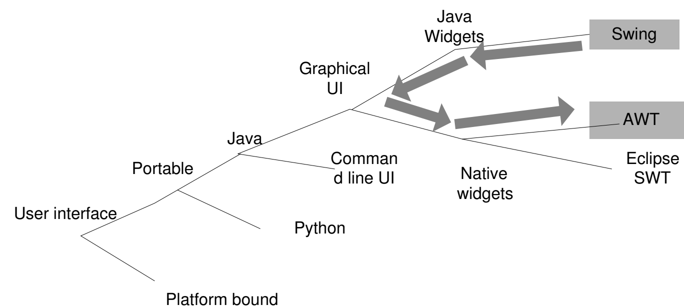

Figure 5. Design decision tree

The adaptation scenarios can be described in the terms of a design decision tree. In the Gaia approach (as in many other software en-

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

gineering 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 5, 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 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.

4.4 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 to justify 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].

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. The agent model identifies agent types involved into the split and merge operations.

In Figure 6 we present an agent model for an agent which represents a military unit [6]. The military unit can split into two components, the reconnaissance unit and the cover unit. This implies a distribution of the roles between the two split units. We should note that some of the roles (such as the communication role) will be are replicated in the two split units.

---

[5] Agent mobility, in practice involves: stopping an agent, serializing it, transferring data and code over the network, signalling back that the transfer was successful, 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.

[6] Many readers might consider that a military unit is better modeled using an agent society. This, however, a matter of choice. From a point of view of battlefield simulation, small military units can be more conveniently modeled as a single agent. Many researchers pointed out the significant similarities between modeling individual agents and modeling agent societies.

We have introduced two new elements in the agent model, the split operator and the merge operator. Each of them are labelled with the trigger of the split and merge operations respectively.

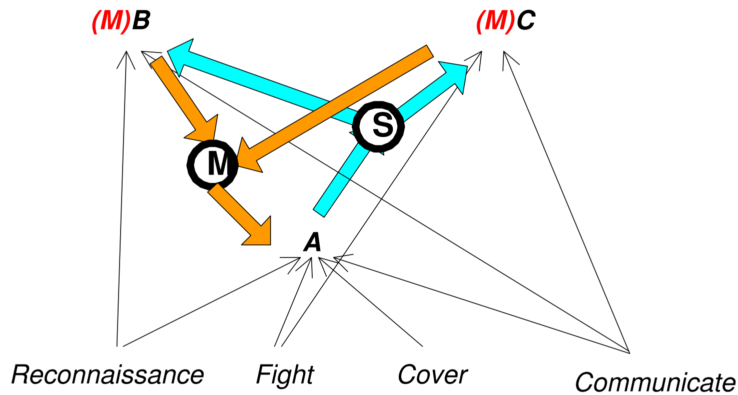

Figure 6. The agent model for splitting and merging operations

5 Conclusions

Mutable and reconfigurable software is increasingly more popular. It is driven by the needs of users to have their applications adapt to the changing global or local environment. These requirements appear in an amplified form for software agents. The software environment of agents is much more dynamic then the relatively controlled environment in which server applications or desktop programs live.

Reconfigurable agents however, are bringing additional challenges to the already complex problems of modeling agents and reasoning about them, and the emerging field of agent based software engineering. We found it interesting, that the actual implementation of mutable and reconfigurable agents is the field which seems the most advanced today - in this case, the practical implementations run ahead the theoretical foundations.

In this chapter we provided a review of the field of mutable and reconfigurable agents from various perspectives. We attempted to organize the various implementation methods by providing a classification for mutable and reconfigurable agents - at the same time tying it to the larger field of reconfigurable software. We discussed the various approaches which can provide a formal model for reasoning about mutable agents. We presented some results based on a multiplane state machine model, and discussed their strengths and limitations. Finally, we discussed the challenges posed by mutable agents for the agent oriented software engineering and have presented an approach for extending Gaia, one of the popular AOSE approaches towards handling mutable agents.

References

An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N. and Rauchwerger, L.: 2001, STAPL: An adaptive, generic parallel C++ library, *Proceedings of the 14th Workshop on Languages and Compilers for Parallel Computing (LCPC)*.

Arai, T. and Stolzenburg, F.: 2002, Multiagent systems specification by UML statecharts aiming at intelligent manufacturing, *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, ACM Press, pp. 11–18.

Barber, K. S., Goel, A. and Martin, C. E.: 2000, Dynamic adaptive autonomy in multi-agent systems, *JETAI* **12**(2), 129–147.

Beck, K.: 1999, *Extreme Programming Explained: Embrace Change*, Addison Wesley.

Bölöni, L. and Marinescu, D. C.: 2000a, Agent surgery: The case for mutable agents, *Proceedings of the Third Workshop on Bio-*

Inspired Solutions to Parallel Processing Problems (BioSP3), Cancun, Mexico.

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

Bon: 2003, Bond webpage, URL `http://bond.cs.ucf.edu`.

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

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F. and Mylopoulos, J.: 2001, A knowledge level software engineering methodology for agent oriented programming, *in* J. P. Müller, E. Andre, S. Sen and C. Frasson (eds), *Proceedings of the Fifth International Conference on Autonomous Agents*, ACM Press, Montreal, Canada, pp. 648–655.

Caire, G., Coulier, W., Garijo, F. J., Gomez, J., Pavon, J., Leal, F., Chainho, P., Kearney, P. E., Stark, J., Evans, R. and Massonet, P.: 2001, Agent oriented analysis using Message/UML, *AOSE*, pp. 119–135.

Collberg, C., Thomborson, C. and Low, D.: 1997, A taxonomy of obfuscating transformations, *Technical Report 148*, Department of Computer Science, University of Auckland.

Cranefield, S., Haustein, S. and Purvis, M.: 2001, UML-based ontology modelling for software agents.

Decker, K., Sycara, K. and Williamson, M.: 1996, Intelligent adaptive information agents, *in* I. Imam (ed.), *Working Notes of the AAAI-96 Workshop on Intelligent Adaptive Agents*, Portland, OR.

Fagin, R., Halpern, J. Y., Moses, Y. and Vardi, M. Y.: 1995, *Reasoning about knowledge*, MIT Press.

For: 1998, Fortify for netscape, URL `http://www.fortify.net` .

Fowler, M., Beck, K. et al.: 1999, *Refactoring: Improving the Design of Existing Code*, Addison Wesley.

Griss, M. L., Fonseca, S., Cowan, D. and Kessler, R.: 2002a, SmartAgent: Extending the JADE agent behavior model, *Proceedings of the Agent Oriented Software Engineering Workshop, Conference in Systemics, Cybernetics and Informatics*, ACM Press.

Griss, M. L., Fonseca, S., Cowan, D. and Kessler, R.: 2002b, Using UML state machines models for more precise and flexible JADE agent behaviors, *Proceedings of the Agent Oriented Software Engineering Workshop, AAMAS*, ACM Press.

Gustavsson, J. and Assmann, U.: 2002, A classification of runtime software changes.

Han, Y., Sheth, A. and Bussler, C.: 1998, A taxonomy of adaptive workflow management, *CSCW-98 Workshop – Towards Adaptive Workflow Systems*.

Harel, D.: 1987, Statecharts: A visual formalism for complex systems, *Science of Computer Programming* **8**(3), 231–274.

Heinze, C. and Sterling, L.: 2002, Using the uml to model knowledge in agent systems, *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, ACM Press, pp. 41–42.

JFl: 2003, JFluid web page, URL `http://www.sunlabs.com/projects/jfluid`.

Juan, T., Pearce, A. and Sterling, L.: 2002, ROADMAP: Extending the Gaia methodology for complex open systems, *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, ACM Press, pp. 3–10.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: 2001, An overview of AspectJ, *Lecture Notes in Computer Science* **2072**, 327–355.

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

Lavender, R. G. and Schmidt, D. C.: 1995, Active object: an object behavioral pattern for concurrent programming, *Proceedings of Pattern Languages of Program Design,*.

Levesque, H. J., Cohen, P. R. and Nunes, J. H. T.: 1990, On acting together, *Proceedings of the Eighth National Conference on Artificial Intelligence*, American Association for Artificial Intelligence, pp. 94–99.

Luck, M., Griffiths, N. and d'Inverno, M.: 1997, From agent theory to agent construction: A case study, *in* J. P. Müller, M. J. Wooldridge and N. R. Jennings (eds), *Proceedings of the ECAI'96 Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents III*, Vol. 1193, Springer-Verlag: Heidelberg, Germany, pp. 49–64.

Luyten, K., Vandervelpen, C. and Coninx, K.: 2002, Migratable user interface descriptions in component-based development, *Proceedings of the 9th International Workshop on Design, Specification, and Verification of Interactive Systems*.

Odell, J., Parunak, H. and Bauer, B.: 2000, Extending UML for agents, *in* G. Wagner, Y. Lesperance and E. Yu (eds), *Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pp. 3–17.

Padgham, L. and Winikoff, M.: 2002, Prometheus: A methodology for developing intelligent agents, *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, ACM Press, pp. 37–38.

Pribeanu, C., Limbourg, Q. and Vanderdonckt, J.: 2001, Task modelling for context-sensitive user interfaces, *Lecture Notes in Computer Science* **2220**, 49–??

Rao, A. S. and Georgeff, M. P.: 1995, BDI agents: from theory to practice, *in* V. Lesser (ed.), *Proceedings of the First International Conference on Multi–Agent Systems*, MIT Press, San Francisco, CA, pp. 312–319.

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

Smith, R. G.: 1980, The contract net protocol: High-level communication and control in a distributed problem solver, *IEEE Transactions on Computers* **29**(12), 1104–1113.

Thevenin, D. and Coutaz, J.: 1999, Plasticity of user interfaces: Framework and research agenda.

van der Aalst, W. M. P.: 1999, Generic workflow models: How to handle dynamic change and capture management information?, *Conference on Cooperative Information Systems*, pp. 115–126.

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

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

XBo: 2001, Xbox on linux, URL `http://xbox-linux.sourceforce.net` .

Yu, E. S. K. and Mylopoulos, J.: 1994, From E-R to "A-R" - modelling strategic actor relationships for business process reengineering, *in* P. Loucopoulos (ed.), *Proceedings of the* 13th *International Conference on the Entity-Relationship Approach*, Springer, Manchester, UK, pp. 548–565.