Agent surgery: The case for mutable agents

Ladislau Bölöni and Dan C. Marinescu

Computer Sciences Department
Purdue University
West Lafayette, IN 47907

Abstract. We argue that mutable programs are an important class of future applications. The field of software agents is an important beneficiary of mutability. We evaluate existing mutable applications and discuss basic requirements to legitimize mutability techniques. Agent surgery supports runtime mutability in the agent framework of the Bond distributed object system.

1 Introduction

Introductory computer architecture courses often present an example of self-modifying assembly code to drive home the message that executable code and data are undistinguishable from one another once loaded in the internal memory of a computer. Later on, computer science students learn that writing self-modifying programs or programs which modify other programs is not an acceptable software engineering practice but a rebellious approach with unpredictable and undesirable results. A typical recommendation is *"Although occasionally employed to overcome the restrictions of first-generation microprocessors, self-modifying code should never ever be used today - it is difficult to read and nearly impossible to debug."* [1]

We argue that self-modifying or mutable programs are already around, and they will be an important component of the computing culture in coming years. In particular, this approach can open new possibilities for the field of autonomous agents.

Software agents play an increasingly important role in today's computing landscape due to their versatility, autonomy, and mobility. For example the Bond agent framework, [2], is currently used for a variety of applications including an workflow enactment engine for commercial and business applications [5], resource discovery and management in a wide-area distributed object-system, [7], adaptive video services [6], parallel and distributed computing [4]. Some of these applications require that the functionality be changed while the agent is running. But to change the functionality of an agent we have to modify its structure. This motivates our interest for agent surgery and by extension for program mutability.

2 Mutable programs

We propose the term *mutable programs* for cases when the program executable is modified by the program itself or by an external entity. We use the term "mutable" instead of "mutating" to indicate that changes in the program structure and functionality are directed, well specified, non-random. The word mutation is used in genetic algorithms and evolutionary computing to indicate random changes in a population when the "best" mutant in some sense is selected, using a survival of the fittest algorithm. In these cases the source of the program is modified. The case we consider is when a program is modified at run time to change its functionality. We distinguish *weak* and *strong* mutability.

Weak mutability is the technique to *extend* the functionality of the application using external components.The application still keeps its essential characteristics and functionality. The trademark of weak mutability is a well defined API or interface between the program to be extended and the entity providing the additional functionality. The mechanism of extension is either the explicit call of an external application or loading of a library, either in the classical dynamic library sense or in the Java applet / ActiveX control sense.

This is the currently most accepted form of mutability in applications. Examples include: data format plugins, active plugins, skins and themes, applets and embedded applications, automatic upgrades etc.

In case of **strong mutability**, applications change their behavior in a radical manner. The APIs for the modification is loosely defined, or implicit.

Strong mutability can be implemented at any level of granularity. At the machine-code level, typical example are viruses and anti-virus programs. Classical executable viruses are attaching themselves to programs and modify the loader code to add their additional, usually malign, functionality. Certain viruses perform more complex procedures on the executable code, e.g. compression, redirection of function calls etc. Anti-virus programs modify the executable program, by removing the virus and restoring the loader. A special kind of anti-virus programs, called *vaccines* modify an executable to provide a self-checksum facility to prevent further modifications. Another example of strong mutability is *code mangling* , performed to prevent the reverse engineering of the code. A somewhat peculiar example is the case of self-building neural networks.

Component-based systems are most promising for strong mutability because the larger granularity of the components makes the problem more tractable. At the same time the cleaner interface of components and the fact that they are usually well specified entities allow for easier modification. The most promising directions are the custom assembly of agents based on specifications, and runtime modification of component-based agents, called in this paper "agent surgery". An example of application-level strong mutability is the case of dynamically assembled and modified workflows.

Several requirements must be met before mutabilility could become an accepted programming technique, they are: well-defined scope, self-sufficiency, seamless transformation, persistency, and reversibility.

First, the modification should have a **well-defined scope** and lead to a predictable change of program behavior. For example a user downloading a new plugin expects that the only modification of the program behavior is to accept a new data type.

Self-sufficiency requires that the change described by the process should be defined without knowing the internal structure of the application. Thus multiple consecutive changes can applied to an application. It also allows one change to be applied to multiple applications.

Seamless transformation of program behavior, the change should be performed on a running program without the need to stop its execution. The alternative to modify the source code, compile it, and restart the program is not acceptable in many cases. As our dependency upon computers becomes more and more pronounced, it will be increasingly difficult to shut down a complex system e.g. the air-traffic control system, a banking system, or even a site for electronic commerce to add, delete, or modify some of its components. Even today it is not acceptable to restart the operating system of a computer or recompile the browser to add a new plugin necessary to view a webpage. A similar hardware requirement is called hot-plug compatibility, hardware components can be added and sometimes removed from a computer system without the need to shut down the system.

Another requirement is to make the change **persistent**. This requirement is automatically satisfied when the file containing the executable program is modified. If the image of a running program is modified then a mechanism to propagate this change to the file containing the executable must be provided.

The reverse side of the coin is the ability to make the change **reversible**. In a stronger version of this requirement we expect to revert a change while the program is running. A weaker requirement is to restart the agent or the application in order to revert the change. This requirement can be easily satisfied for individual changes, by keeping backup copies of the original program. If we allow multiple changes and then revert only some of them the problem becomes very complex as the experience with **install programs/scripts** shows.

We emphasize that not all these requirements should be satisfied simultaneously. Actually only the predictability is a critical property. Satisfying the additional requirements however, broadens the range of applicability of the technique. Thus, when designing the **mechanisms of mutability** we should attempt to satisfy as many of them as possible.

The common feature of every mechanism of mutability is that we treat code as a data structure. This is an immediate consequence of the von Neumann's concept of stored-program computers, and thus applicable to virtually any program. Yet, in practice, modifying running programs is very difficult, unless they are described by a simple and well-defined defined data structure accesible at run-time. For example, a program written in C++ or C has a structure given by the flow of function calls in the original source code. After the compilation and optimization process however, this structure is very difficult to reconstruct.

The object code of compiled languages or the code generated from the assembly language, while it can be viewed as a data structure, it does not allow us to easily discover its properties. This complexity justifies the point made at the beginning of this paper, that self-modifying machine code should not be used as a programming technique.

In conclusion, programs can be successfully modified if there is a high level, well documented data structure [1] that in some sense is analogous with the genetic information of biological structures. If the program designer chooses to have only part of the code described by this structure (like in the case of plugin API-s) weak mutability is possible. If the entire program is described by the data structure then strong mutability is possible. This is the case of Bond agents whose behavior is based on the multiplan state machine and the agent surgery enables strong mutability.

3 Mutability in Bond

Agent surgery is a modification technique employed in the Bond system to change the behavior of a *running* agent by modifying the data structure describing the multi-plane state machine of the agent.

There are several advantages the multiplane state machines offer over other data structures:

-The behavior of the agent in any moment is determined by a well defined subset of the multiplan state machine (the current state vector).

-The multiplane state machine exhibits *enforcable locality of reference*. While most programs exhibit temporal and spatial locality of reference, this can not be used in mutability procedures, because we don't have a guarantee that the program will not perform suddently a long jump, which can be intercepted only at a very low level - by the operating system or the hardware. The semantic equivalent of a long jump, the transitions, however, are executed through the messaging function of the agent, thus they can be captured and queued for the duration of the agent surgery.

-The multiplane state machine is *self-describing*. Its structure can be iterated on by an external component. This allows to make the changes persistent by allowing to write out the runtime modifications to a new blueprint script. On the other hand allows for specifying operations independently from the structure of the agent.

Bond agents can be modified using "surgical" `blueprint` scripts. In contrast with the scripts used to create agents which define the structure of a multiplan finite state machine, surgical blueprints are adding, deleting or changing nodes, transitions and/or planes in an existing multiplan finite state machine.

The sequence of actions in this process is:

[1] Of course, strong mutability is possible even without this high level data structure, if the external entity is able to figure out the low level data structure manifests itself at the object code level - viruses are doing exactly this. However, this cannot form the basis of a generally viable technique.

(1)A *transition freeze* is installed on the agent. The agent continues to execute normally, but if a transition occurs the corresponding plane is frozen. The transition will be enqueued.

(2) The agent factory interprets the blueprint script and modifies the multi-plane state machine accordingly. There are some special cases to be considered: (a) If a whole plane is deleted, the plane is brought first to a soft stop - i.e. the last action completes. (b) If the current node in a plane is deleted, a *failure* message is sent to the current plane.

(3) The transition freeze is lifted, the pending transitions performed, and the modified agent continues its existence.

4 Surgery techniques

The framework presented previously permits almost arbitrary modifications in the structure of the agent. Without some self-discipline however, the modified agent will quickly become a chaotic assembly of active components. A successfull surgical operation is composed on a number of more disciplined elementary operations, with a well specified semantics. In these section we enumerate those techniques which we consider as being the most useful.

4.1 Simple surgical operations

The simplest surgical operations are referring to the adding and removal of individual states and transitions. These operations are executed unconditionally. In the case of direct specification of operations is that the writer of the surgical blueprint must have a good knowledge of the existing agent structure. If new states are added to an existing plane, but not linked to existing states with transitions, they will never be executed. In order to remove existing states, transitions we need to know the name and function of the given states and transitions.

4.2 Replacing the strategy of a state

In this operation the strategy of a state is replaced with a different strategy. The reason of doing this is to improve or adapt the functionality to the specific condition of an agent. For example if an agent is running on, or migrated to a computer which doesn't have a graphical user interface, the strategies implementing the graphic user interface should be replaced with strategies adapted to the specific host, for example with command line programs.

These operations keep the old strategy namespace in which the strategy operates. In certain cases we should replace a group of interrelated strategies at once. For example, running and controlling an external application locally in Bond is done using the `Exec` strategy group. These strategies allow starting, supervising, terminating local applications, but can not run applications remotely. Now if the application requires remote run, we can replace all these strategies with the corresponding strategies from the RExec strategy group, which run applications using the Unix rexec call.

4.3 Splitting a transition with a state

Transitions in the Bond system represent a change in the strategy. One important way of changing the functionality of an agent is by inserting a new state in a transition. In effect, the existing transition is redirected to the new state, while from the new state the "success" transition is generated to the original target. The name of the new state is generated automatically. This can be later used to add new transitions to the state.

Typical application of these techniques are: logging and monitoring, confirmation checks and interagent synchronization.

4.4 Bypassing states

Bypassing a state is the semantic equivalent of replacing its strategy with a strategy which immediately, unconditionally succeeds. Practically the state is deleted, while all the incoming transitions are redirected to the target of the "success" transition originating from the given state.

The bypass operation can be used to revert the effect of the split operation discussed previously. For example, one application of the bypass operation is to remove the debugging or logging states from an agent.

4.5 Adding and removing planes

The planes of the multiplane finite state machine are expressing parallel activities. Thus, new functionality can be easily added to an agent by creating a new plane which implements that functionality. The new functionality can use the model of the world in the agent (the knowledge accumulated by other planes) and can directly communicate with other planes by triggering transitions in them. Analogously, we can remove functionality by deleting planes. There is direct support in *blueprint* for adding and removing planes.

Of course, like any surgical operation adding and removing planes is a delicate operation. Generally, we can safely add and remove planes which represent an independent functionality. Usually, the planes added to an already working agent can be safely removed.

Examples of using this technique are: adding a visualization plane, adding remote control or negotiation planes and replacing the reasoning plane to include a better reasoning mechanism.

4.6 Joining and splitting agents

Two of the simplest surgical operations on agents are the joining and splitting. When **joining** two agents, the multi-plane state machine of the new agent contains all the planes of the two agents and the model of the resulting agent is created by merging the models of the two agents. The safest way is to separate the two models (for example through use of namespaces), but a more elaborate

merging algorithm may be considered. As our design does not specify the knowledge representation method, the best approach should be determined from case to case. The agenda of the new agent is a logical function (usually an "and" or an "or") on the agendas of the individual agents. It is tempting to consider the joining of agents as a boolean operation on agents, and maybe to envision an algebra of agents. While the subject definitely justifies further investigation, the design presented in this paper do not qualify for such an algebra. The more difficult problem is handling the "not" operator, which applied to the agenda would render the current multi-plane state machine useless.

In case of agent **splitting** we obtain two agents, the union of their planes gives us the planes of the original agent The splitting need not be disjoint, some planes (e.g an error handling or housekeeping) may be replicated in both agents. Both agents inherit the full model of the original agent, but the models may be reduced using the techniques presented in section 4.7. The agendas of the new agents are derived from the agenda of the original agent. The conjunction or disjunction of the two agendas gives the agenda of the original agent.

There are several cases when joining or splitting agents are useful: (a) Joining control agents from several sources, to provide a unified control, (b) Joining agents to reduce the memory footprint by eliminating replicated planes, (c) Joining agents to speed up communication, (d) Migrating only part of an agent, (e) Splitting to apply different priorities to parts of the agent.

Joining and splitting of agents is used by our implementation of agents implementing workflow computing.

4.7 Trimming agents

The state machines describing the planes of an agent may contain states and transitions unreachable from the current state. These states may represent execution branches not chosen for the current run, or states already traversed and not to be entered again. The semantics of the agent does not allow some states to be entered again, e.g. the initialization code of an agent is entered only once.

If the implementation allows us to identify the set of model variables which can be accessed by strategies associated with states, we can further identify parts of the model, which can not be accessed by the strategies reachable from the current state. The Bond system uses *namespaces* to perform a mapping of the model variables to strategy variables, thus we can identify the namespaces which are not accessed by the strategies reachable from the current state vector.

If the agenda of the agent can be expressed as a logical function on model variables and this is usually the case, we can simplify the agenda function for any given state, by eliminating the "or" branches that cannot be satisfied from the current state of the agent.

All these considerations allow us to perform the "trimming" of agents, for any given state to replace the agent with a different, smaller agent.

While stopping an agent to "trim" it is not justified for every situation there are several cases when we consider it to be especially useful, like before migration and before checkpointing.

The default migration implementation in the Bond system is using trimming to reduce the amount of data transferred in the migration process.

5 Conclusions

A number of factors, some of them technical, others legal and market-driven motivate an increasing interest in self-modifying programs. Some of the applications of software agents require that the functionality be changed while the agent is running. To support efficiently agent mobility we propose to trim out all the unnecessary components of an agent before migrating it. But to change the functionality of an agent we have to modify its structure. This motivates our interest for agent surgery and by extension for program mutability.

Several requirements must be met before mutabilility becomes an accepted programming technique, mutability should have a well-defined scope, be self-sufficient, seamless, persistent, and reversible. For example we should be able to modify running applications because it is conceivable to stop a critical system to modify its components.

The Bond agent framework is distributed under an open source license (LGPL) and the second beta release of version 2.0 can be downloaded from `http://bond.cs.purdue.edu`.

Acknowledgments

The work reported in this paper was partially supported by a grant from the National Science Foundation, MCB-9527131, by the Scalable I/O Initiative, and by a grant from the Intel Corporation.

References

1. A. Clements *Glossary of computer architecture terms. Self-modifying code.* `http://www-scm.tees.ac.uk/users/a.clements/Gloss1.htm`
2. L. Bölöni and D.C. Marinescu, *Biological Metaphors in the Design of Complex Software Systtems*, *Journal of Future Computer Systems*, Elsevier, (in press)
3. L. Bölöni and D. C. Marinescu, *An Object-Oriented Framework for Building Collaborative Network Agents*, in *Intelligent Systems and Interfaces*, (A. Kandel, K. Hoffmann, D. Mlynek, and N.H. Teodorescu, eds). Kluewer Publising, 2000, (in press).
4. P. Tsompanopoulou, L. Bölöni and D.C. Marinescu *The Design of Software Agents for a Network of PDE Solvers Agents For Problem Solving Applications Workshop, Agents '99*, IEEE Press, pp. 57-68, 1999.
5. K. Palacz and D. C. Marinescu, *An Agent-Based Workflow Management System Proc. AAAI Spring Symposium Workshop "Bringing Knowledge to Business Processes"* Standford University, IEEE Press, (to appear)).
6. K.K. Jun, L. Bölöni, D. K.Y. Yau, and D. C. Marinescu, *Intelligent QoS Support for an Adaptive Video Service*, *Proc. IRMA 2000*, IEEE Press, (to appear).
7. K.K. Jun, L. Bölöni, K. Palacz and D. C. Marinescu, *Agent-Based Resource Discovery*, *Proc. Heterogeneous Computing Workshop, HCW 2000*, IEEE Press (to appear).