An Approach to Autonomizing Legacy Systems

Gail Kaiser, Phil Gross, Gaurav Kc, Janak Parekh, Giuseppe Valetto
Columbia University, Programming Systems Lab
Department of Computer Science
500 West 120th Street
New York, NY 10027
{ kaiser, phil, gskc, janak, valetto } @ cs.columbia.edu

Abstract
Adding adaptation capabilities to *existing* distributed systems is a major concern. The question addressed here is how to **retrofit** existing systems with self-healing, adaptation and/or self-management capabilities. The problem is obviously intensified for "systems of systems" composed of components, whether new or legacy, that may have been developed by different vendors, mixing and matching COTS and "open source" components. This system composition model is expected to be increasingly common in high performance computing. The usual approach is to train technicians to understand the complexities of these components and their connections, including performance tuning parameters, so that they can then manually monitor and reconfigure the system as needed. We envision instead attaching a "standard" feedback-loop infrastructure to existing distributed systems for the purposes of continual monitoring and dynamically adapting their activities and performance. (This approach can also be applied to "new" systems, as an alternative to "building in" adaptation facilities, but we do not address that here.) Our proposed infrastructure consists of multiple layers with the objectives of probing, measuring and reporting of activity and state within the execution of the legacy system among its components and connectors; gauging, analysis and interpretation of the reported events; and possible feedback to focus the probes and gauges to drill deeper, or – when necessary - direct but automatic reconfiguration of the running system.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification - *model checking, reliability, validation*

General Terms
Measurement, Performance, Reliability, Standardization, Verification

Keywords
Runtime monitoring, dynamic adaptation, self-healing framework, autonomic system, externalized reconfiguration

1. Introduction
By its very nature, maintenance is the longest and most difficult phase of the software process lifecycle, for nearly any application domain. Furthermore, regardless of the amount of effort put into the development stages and pre-deployment and testing, a high performance computing application must be able to *adapt* to changing environmental contexts regarding, e.g., contention for resources. Adaptation of on-line, running applications is even more complex than conventional off-line maintenance. Hardwiring the adaptation mechanisms into the application code is the most common approach for "new" systems, but hard to modify post-deployment and tending to result in one-of solutions with relatively little to share or amortize the cost across applications and domains. Also, retrofitting adaptation facilities directly into legacy systems or those composed of third-party components may be impractical when one has limited control over or understanding of the design and implementation of the system components. As a result, many of the methodologies and technologies being developed for building new adaptive systems do not fit well for those built out of legacy components.

Following IBM's "autonomic computing" terminology [1][10], we are investigating an approach to "autonomizing" legacy systems and assembling "autonomic" systems-of-systems from components. Seeing that directly inserting adaptation mechanisms into existing application code is difficult, error-prone and costly, besides being hard to reuse or reason about, we have sought to enable adaptation properties through a solution orthogonal to the legacy systems' main business logic and communication framework. We are now developing an infrastructure for instrumenting running systems with *probes*, and passing the data gathered by that instrumentation to *gauges*, where it can be collected, collated, filtered and aggregated into system level measurements of the system's operation. These measurements form the basis by which analyses are made of the system's execution, in terms of available *models*, which then leads to the feedback phase where runtime *modification*s to the system are (automatically) carried out. This approach to continual dynamic monitoring and reconfiguration of deployed systems is intended to help automate most system management functions with little or no human intervention, whereby the adaptation of the running system can be carried out dynamically without incurring any system downtime.

We propose a common external infrastructure that can be widely applied as a technology for instrumenting, measuring, and dynamically controlling software systems through adaptation and reconfiguration. Our infrastructure becomes an integral part of the system-of-systems' "self," co-existing and cooperating with the systems' native functional mechanisms (as well as any special-purpose adaptation facilities, e.g., for fault tolerance or transaction recovery, built into individual components). At this point in our research, we have the capability to instrument a wide variety of existing systems to perform measurements and convey this runtime information to the monitoring layer. This monitoring layer can then evaluate the performance of the system based on this data according to a wide variety of metric, protocol and architectural, etc. models. We are currently progressing from the few initial special cases towards the generic third and final layer of the infrastructure – the "Decision" layer, and its control of the effectors that carry out its self-healing and self-management decisions.

2. Our Approach

It is generally agreed that a running system's robustness can be improved by dynamic analysis of model-based measurements to determine appropriate modifications and adaptations. However, rather than mixing measurement and analysis code with problem-specific code, or even with separate monitoring code but peculiar to the target system and/or its underlying framework, we have chosen to develop common (or "standard") *externalized* reconfiguration as well as monitoring mechanisms, where the adaptation feedback loop is handled outside of the application. Thus we split the reusable mechanism from the system-specific policies. This is particularly significant, because coordination of multiple independently-developed internal adaptation mechanisms is quite difficult or even impossible without some sort of "global" supervisor.

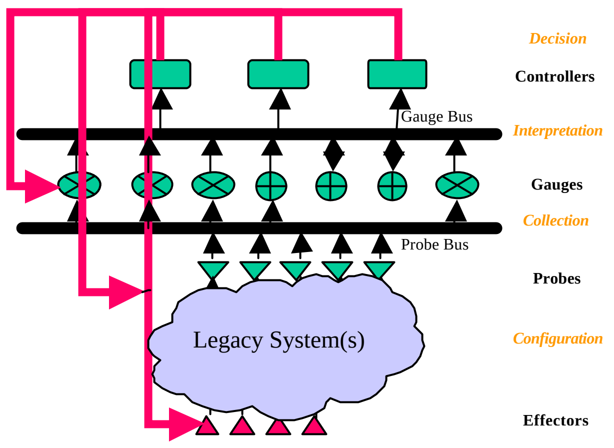

Figure 1. Common Infrastructure

Figure 1 shows our three-tiered infrastructure: Initially, data is collected from the running system. It is instrumented with non-invasive *probes* that report raw data to the higher levels via the *Probe Bus*. The data is then interpreted via a set of *gauges* that map the probe data into various models of the system. The gauges then report their findings to the *Gauge Bus*. Then the *decision and control* layer can analyze the implications of the interpreted data on overall system performance and make decisions on whether to: (1) introduce new gauges in the interpretation layer to analyze further, or disable some as superfluous; (2) deploy new probes to provide more detailed information to remaining gauges, or turn some off to reduce "noise"; and/or (3) reconfigure the system itself, perhaps changing the running system's structure by introducing new modules or modifying system or component parameters. The system reconfiguration would be carried out via deployment/activation of software *effectors* to reconfigure or adapt individual components and/or major substructures of the system. In our approach, these effectors make the actual changes under the control of a decentralized workflow system, which handles coordination of the effectors and contingencies.

We emphasize that the infrastructure is largely independent of the running system. However, this is not to say that the specific probes, gauges, controllers, effectors and models are themselves independent of the running system – they are not. The probes and effectors must often be specialized to the implementation technology; the gauges and decision mechanisms must be specialized to the problem domain and environmental context. However, reuse of common infrastructure facilities should be commonplace, such as probes and gauges geared towards performance, architecture dynamism, system robustness, etc.

Our current implementation of this externalized infrastructure is called Kinesthetics eXtreme, or KX (pronounced "kicks"). KX can be downloaded from [6]. KX is being applied experimentally for load balancing and server replacement for Telecom Italia's heterogeneous instant messaging system [16], and for more complex contingencies in a geographically-based "open information" (e.g., CNN, BBC) analysis system developed at Information Sciences Institute [14] and in use at the US Pacific Command (PACOM). We are restricted from discussing either application in too much detail, so we will discuss our approach more generically.

3. Monitoring

Probing of a running system is a necessary prerequisite for monitoring the execution of the system. We need a minimally invasive approach that can be guaranteed to have zero or negligible effect on the performance of the system, while still offering nearly the equivalence of remote debugging abilities. A probe here is an individual sensor attached to, or associated with (as a monitor of), a running program – or a component or connector of a running program. A probe can sense some portion of the program's, or its environment's, execution and make that data available by issuing *events*. One focus of the DASADA program [13], under which this research is being conducted, has been to develop a "standard" API for controlling and reading (and adding and removing) probes.

Most of our own work has focused on interoperable infrastructure, rather than the probe technology itself. We use a variety of probes developed by other researchers as well as ourselves, and can "drop in" any probe technology meeting the DASADA standard API [3]. Experience both within and outside DASADA has shown that it is easy to adapt or wrap a variety of probe technologies to be compliant with this API. Basically, the standard defines how probes may be **deployed** (probe code situated at a host for subsequent attachment to specific systems at that site), **installed** (attached to a specific target), and **activated** (turned on).

Probes generally fall into one of two categories, either passive or active. In the former, we rely on activity within the system to trigger any probe processing. This method is well-suited for situations where much of the information about the performance of the system can be gleaned from the actual goings-on in the system itself, e.g., to keep track of the number of times a certain service was invoked during the execution, possibly tracking its parameters and cause-effect correlating with invocations of other services. For example, passive probing using the Active Interfaces Development Environment tool [8][9] inserts callback probes into Java source code as part of the instrumentation phase. We have also experimented with ProbeMeister [12], which inserts probes into Java byte code, and are in the process of integrating "instrumented connectors" [2] which replace Win32 DLLs. There are numerous other alternatives.

Active probing, on the other hand, is a potentially more sophisticated mechanism that permits us to perform (resource-limited) computation within the system's context when making our measurements on the system's activity and state. We have developed an active probing technology based on our "Worklets" mobile agent platform [7][11][15], which means that we only need the lightweight Worklet Virtual Machine to be pre-installed within the target system (using any of the above source, or bytecode or DLL instrumentation technologies - or others). We can then dynamically attach and remove what we call "Probelets", mobile agent-based probes, as our monitoring needs dictate.

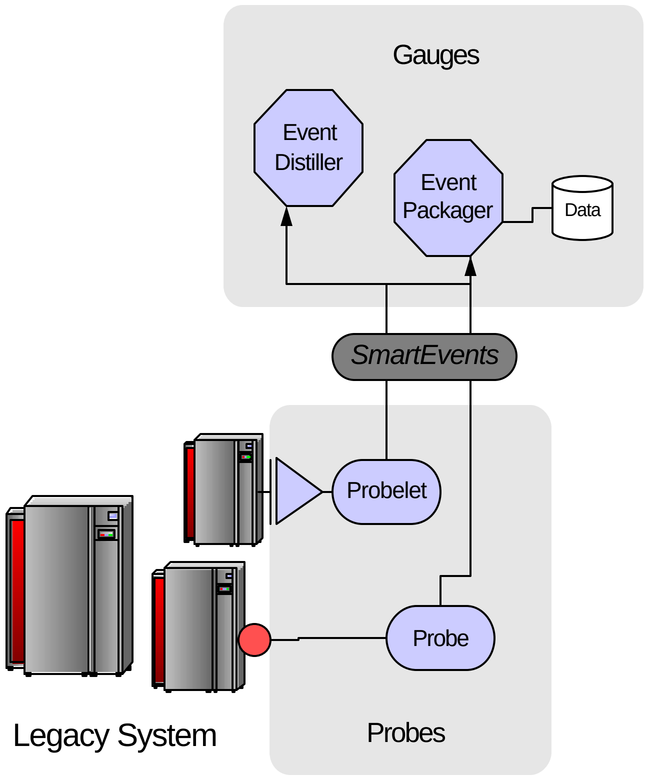

Figure 2 KX Probes and Gauges

We have proposed an XML Schema [5] as a "standard" format for structuring probe output data. Our intent is to unify the disparate ways in which the varied probe implementations describe observed events. This "Smart Event" Schema includes points for extending with generic application models so that (e.g.) arbitrary gauge technologies can use this information to determine what probes to select according to relevant models. This enables that probe descriptions do not need to be customized for a given application or its specific models. In general, Smart Events are composed from standard "blocks" of information. The initial probe might emit simple Smart Events containing a raw data block, but later processing and analysis stages augment this with additional or higher-level information blocks.

Some gauges' activities will be so time-consuming that they can only be used in an offline analysis mode. Such gauges will want to consume events using "event logs". Rather than forcing each such offline gauge to create its own logging facility, a generic persistent event store and "replay" facility has been implemented.

4. Dynamic Analysis

Gauges are software entities that gather, filter, aggregate, compute, and/or analyze measurement information about software systems. In particular, they interpret probe data against various models, to produce higher-level outputs: gauges can emit events just as can probes. These events are typically at a higher level of abstraction, but the "Smart Event" XML Schema has been defined to support both levels. As with probes, a major concern of the DASADA project has been developing standards for gauges to allow interoperability.

Our own gauges work within a framework called XML-based Universal Event Service (XUES) [7], with two major components named the Event Packager and the Event Distiller, shown in Figure 2. The Event Packager can transform the raw-data format of legacy probe output into Smart Event compatible event streams. It also packages and logs these events for possible replaying. The Event Distiller can recognize complex temporal event patterns from multiple probe sources, and it constructs higher-level measurements to reflect the system state represented by the events. It also produces events to interface with the decision layer and gauge visualizers. The Event Distiller is "programmed" by providing a collection of condition/action rules, where the condition specifies the event pattern and the action specifies what to do when that pattern is recognized – typically generation of a higher-level event.

The diverse need for system monitoring and adaptation has resulted in the definition of a wide variety of gauges based on the associated models, the types of values they are expected to report, etc. In order to have a uniform way of interacting with these gauges, we expect them to asynchronously report information on the gauge bus. Gauge consumers can then register an interest in particular types of gauge events. A gauge consumer may also reconfigure a gauge (e.g., to change its reporting frequency). The consumers are automatically notified if a gauge asynchronously reports a value, or if a gauge has been reconfigured or deleted by some other consumer.

5. Feedback Control Loop to Reconfiguration

We see the control problem as follows: gauge outputs are in turn input to a decision process that determines what course of action to take, if any. The decision process can be supported by a variety of tools, including, for example, an architecture transformation tool that reacts to gauges that detect differences in the running architecture from what is nominal. Executing the high-level repair action, e.g., to reconfigure the architecture, will normally involve several activities at the effector level. Some of these activities may fail, so one needs to be able to express the control process as a workflow rich enough to express contingency plans for alternative actions.

We use our Workflakes decentralized workflow system [16] to carry out the actual local adaptations and more global reconfigurations in the running system, as is illustrated in Figure 3. We do not yet employ workflow notation to describe the activities, the workflow is currently coded by hand (in Java). However, we are experimenting with the Little-JIL workflow formalism [4] for specifying the reconfiguration actions, and are also investigating several other workflow notations. The chosen process specification language must support the description of the actions to be applied to repair a system, including at which location(s) they should be applied. The language needs to specify both sequential and parallel execution of actions, and how to deal with unsuccessful actions, e.g., by retrying, attempting alternate actions, or rolling back changes.

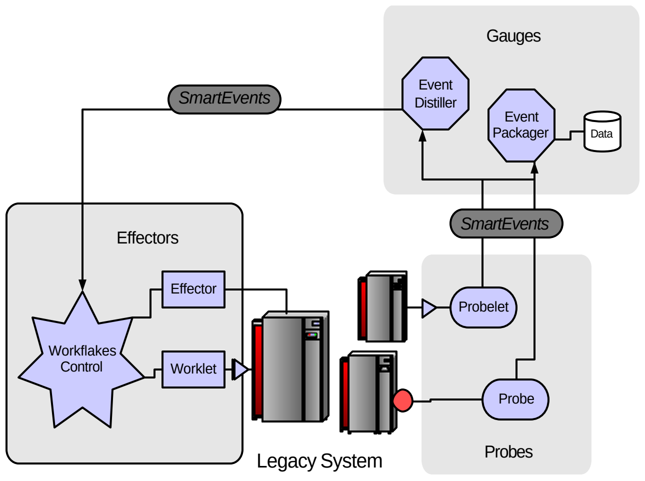

Figure 3 KX Feedback Loop

Workflakes coordinates the actual reconfigurations by invoking low-level effectors attached to the target system. As with probes, effectors could be realized with various technologies. We currently use our Worklets mobile agents as the effectors, analogous to our "Probelets" above. Workflakes conducts a reconfiguration workflow by selecting, instantiating and dispatching Worklets, and coordinating the activities of the deployed Worklets on the target system's components and connectors.

More generally, effector management primarily involves the installation and removal of such low-level modules that cause adaptations and reconfiguration to occur. The control layer might also invoke the management actions of the probe and gauge layers on occasion, for example, to produce refined measurements before proceeding. Effector actions range over a spectrum from simple adaptations – relatively low-level adjustments to a well-defined target system API, e.g., changing a process variable or calling a method – to potentially complex reconfiguration commands that cause structurally significant changes, possibly involving high-level adjustments at the system/environmental level. The latter may involve, e.g., starting, migrating, restarting, or stopping one or more processes, and/or rearranging the connections among components.

Our current feedback loop is relatively ad hoc, depending on manually constructed gauge rules that may trigger canned workflows to perform reconfigurations. The next step is to base reconfiguration decisions on more sophisticated architectural models. Architectural models for a given target system could be created a priori, or generated based on analysis of probed event traffic. We would then be able to build gauges that recognize structural changes based on these models. A variety of mechanisms could be in the analysis, including expert systems and constraint solvers, as well as hard-coded repair rules.

6. Examples

Static Architecture Scenario

A somewhat simplified, but common, scenario that an externally autonomized system will undergo might go as follows:

- Probes and gauges are placed via the control layer.
- Probes emit implementation-level events (ILEs) like "process D006 opened file 'C:\Program Files\log.txt' for write" or "process E001 used 2021."
- Gauges provide interpretations of these events by first determining what logical architectural entities are being referred to – here, perhaps "Radar Tracker" (D006) and "Radar Analysis" (E001), for example. This mapping from implementation terms (process ids) to logical architectural components must be established in the architectural model by the processes that originally set up the system and probes. The gauges additionally interpret implicit information from the probes; for example, perhaps 2021 means 2021 microseconds.
- The gauges are then "read" by the control layer to see if any action should be taken. For example, assume that the ILE for E001 is interpreted as "Radar Analysis took 2021 microseconds to process the last scan." Furthermore, assume that the analysis module is a function of the parameter, ScanGrain. The decision logic may then determine that the ScanGrain for Radar Analysis should be coarsened to 5 degrees / scan.
- The control layer will then use one of its architectural models to determine that process E001 needs to be adapted - i.e., the inverse translation from before, here from logical architecture to physical architecture - and determine what process variable of E001 corresponds to ScanGrain and needs to be reset to reflect the 5 degrees / scan modification. It will select or construct a workflow to be conducted by the effector layer.

Notice that nothing about the architecture itself changed during this scenario; no modules or connections were created or destroyed. Moreover, the repair was effected by a simple parameter change to a running module; no new resources were brought to bear.

Dynamic Architecture Scenario

A more dynamic scenario involving the same kinds of activities might look like:

- Probes and gauges are placed via the control layer.

- Probes emit architecturally significant implementation-level events (ASILEs), such as “process D006 spawned new process E001 of type RAN” and “process E001 requested socket 239.”
- Gauges interpret ASILEs and modify the corresponding physical and logical architectural models. Here, perhaps, because E001 was of type RAN the system knows to identify the E001 process with a (previously unidentified) logical process, “Radar Analysis.” Similarly, the socket may correspond with the Analysis Report socket. We call this process *identification* of physical models with pre-defined, logical architecture models.
- Imagine that some time later the same ILE as above, “process E001 used 2021,” is transmitted by the probes and reported by the gauges. The control layer at this point may want to change the system’s running architecture by issuing to the effector layer a workflow intended to carry out reconfiguration. This time perhaps the adaptation would be to “replace Radar Analysis type RAN with RAAN” (another radar analyzer type, perhaps with a coarser scan rate).
- Now suppose that the component replacement is executed successfully, but a failure of a different component, the Moving Map Display (MMD), is noted.
- On replacement of the MMD, the Radar Analysis component is again observed to fail. On restart, the MMD again fails.
- A higher-level dynamic analysis component notices the oscillating sequence of events, and forces a reconfiguration, instantiating the MMD on a different computational node. Now both services execute successfully, and the event log can be used for later off-line analysis of the causes of the problem.

So there are two separable dynamic architecture activities here: modeling the dynamic architecture as it evolves and reconfiguring the architecture via the control layer. System scenarios could easily require the former without being able to use facilities to do the latter (but not vice versa).

7. Conclusions and Future Work

The proliferation of component-based software engineering has forced us to rethink many of our system maintenance and optimization techniques. It has become harder to tweak the performance of our systems especially when a lot of the components were designed and developed by different sources, including COTS, open source and legacy. We have proposed a mechanism for dynamically monitoring and reconfiguring a system-of-systems built out of a heterogeneous mix of components. Our orthogonal solution makes no restrictions on the system design, hereby freeing us from having to insert application-specific adaptive mechanisms into the code, which would not be reusable elsewhere anyway. Furthermore, it provides the advantage of having a running system continually self-evaluate its performance and fine tune its operations automatically.

We have had considerable success in being able to, both manually and automatically, instrument the source code of components and observe the implementation-level events that were emitted by our probing technologies. We also incrementally construct an evolving model of the system’s execution based on the stream of event data. Our work in the immediate future will focus on better decision-making logic in the feedback and reconfiguration layer based on more comprehensive architectural models of the system.

8. Acknowledgements

Giuseppe Valetto (Giuseppe.Valetto@tilab.com) is also affiliated with the Telecom Italia Lab, Via Reiss Romoli 274, 10148, Turin, Italy.

This work is being undertaken in collaboration with Bob Balzer and Dave Wile, Teknowledge; Nathan Combs, BBN; David Garlan and Bradley Schmerl, CMU; George Heineman, WPI; David Wells, OBJS; and Lee Osterweil, UMass. We would also like to thank the other members of the Programming Systems Lab. The Programming Systems Lab is funded in part by Defense Advanced Research Project Agency under DARPA Order K503 monitored by Air Force Research Laboratory F30602-00-2-0611, by National Science Foundation CCR-9970790 and EIA-0071954, by Microsoft Research, and by NEC Computers, Inc.

9. References

[1] An Almaden Institute Symposium: Autonomic Computing http://www.almaden.ibm.com/institute/2002/

[2] Balzer, R.M., and Goldman, N.M., Mediating Connectors, in ICDCS Workshop on Electronic Commerce and Web-Based Applications, June 1999.

[3] Balzer, B., Probe Run-Time Infrastructure, December 2001. http://www.schafercorp-ballston.com/dasada/2001WinterPI/ProbeRun-TimeInfrastructureDesign.ppt

[4] Cass, A.G., Staudt Lerner, B., McCall, E.K., Osterweil, L. J., Sutton, Jr., S.M., and Wise, A., Little-JIL/Juliette: A Process Definition Language and Interpreter, in 22nd International Conference on Software Engineering, June 2000. ftp://ftp.cs.umass.edu/pub/techrept/techreport/2000/UM-CS-2000-066.ps.

[5] Columbia University Programming Systems Lab, DASADA Probe Event Schema, January 2002. http://www.psl.cs.columbia.edu/kx/smartevent-schema.html

[6] Columbia University Programming Systems Lab, Download PSL Software. http://www.psl.cs.columbia.edu/software.html

[7] Gross, P.N., Gupta, S., Kaiser, G.E., Kc, G.S., and Parekh, J.J., An Active Events Model for Systems Monitoring, in Working Conference on Complex and Dynamic Systems Architecture, December 2001. http://www.psl.cs.columbia.edu/ftp/psl/CUCS-011-01.pdf.

[8] Heineman, G.T., A Model for Designing Adaptable Software Components, in 22nd International Computer Science and Application Conference, August 1998. http://www.cs.wpi.edu/~heineman/PDF/WPI-CS-TR-97-6.pdf.

[9] Heineman, G.T., Adaptation and Software Architecture, in 3rd International Workshop on Software Architecture, November 1998. ftp://ftp.cs.wpi.edu/pub/techreports/pdf/98-13.pdf.

[10] IBM Research, Autonomic Computing. http://www.research.ibm.com/autonomic/

[11] Kaiser, G., Stone, A., and Dossick, S., A Mobile Agent Approach to Lightweight Process Workflow, in International Process Technology Workshop, September 1999. http://www.psl.cs.columbia.edu/ftp/psl/CUCS-021-99.pdf.

[12] Object Services & Consulting, Inc., ProbeMeister 2002. http://www.objs.com/DASADA/ProbeMeister.htm

[13] Salasin, J., Dynamic Assembly for System Adaptability, Dependability, and Assurance (DASADA). http://www.darpa.mil/ipto/research/dasada/

[14] University of Southern California Information Sciences Institute, GeoWorlds Project. http://www.isi.edu/geoworlds/

[15] Valetto, G., Kaiser, G., and Kc, G.S., A Mobile Agent Approach to Process-based Dynamic Adaptation of Complex Software Systems, in 8th European Workshop on Software Process Technology, June 2001. http://www.psl.cs.columbia.edu/ftp/psl/CUCS-001-01.pdf.

[16] Valetto, G., and Kaiser, G., Combining Mobile Agents and Process-based Coordination to Achieve Software Adaptation, Columbia University Department of Computer Science, CUCS-007-02, March 2002. http://www.psl.cs.columbia.edu/ftp/psl/CUCS-007-02.pdf.