A Case Study in Software Adaptation

Giuseppe Valetto
Columbia University and Telecom Italia Lab
Via Reiss Romoli 274
10148, Turin, Italy
+39 011 2288788
Giuseppe.Valetto@tilab.com

Gail Kaiser
Columbia University
Department of Computer Science
New York, NY 10027, United States
+1 212 939 7081
Kaiser@cs.columbia.edu

ABSTRACT
We attach a feedback-control-loop infrastructure to an existing target system, to continually monitor and dynamically adapt its activities and performance. (This approach could also be applied to "new" systems, as an alternative to "building in" adaptation facilities, but we do not address that here.) Our infrastructure consists of multiple layers, with the objectives of 1. probing, measuring and reporting of activity and state during the execution of the target system among its components and connectors; 2. gauging, analysis and interpretation of the reported events; and 3. whenever necessary, feedback onto the probes and gauges, to focus them (e.g., drill deeper), or onto the running target system, to direct its automatic adjustment and reconfiguration. We report on our successful experience using this approach in the dynamic adaptation of a large-scale commercial application requiring both coarse and fine-grained modifications.

Categories and Subject Descriptors
D.3.3 D.2.4, D.2.5 [**Software Engineering**]: Software/Program Verification – *reliability, validation;* Testing and Debugging – *diagnostics, error handling and recovery, monitors.*

General Terms
Management, Performance, Reliability.

Keywords
Dynamic Adaptation, Dynamic Reconfiguration, Perpetual Testing, Distributed Systems, Software Process Enactment, Workflow, Coordination.

1. INTRODUCTION
Our approach to adaptation adds a feedback control loop outside and orthogonal to the legacy system's main computation, control and communication. (Note that by legacy we mean *any* pre-existing software, not necessarily truly ancient software, constructed in, say, COBOL or Fortran.) The only direct interaction with the target system is to insert (or wrap) probes that detect system events, and impose (in some target-specific manner) effectors that can make adjustments and reconfigurations in that system. System models must also be devised based on the target system's functional and non-functional properties, protocols, architecture, domain model, etc., so that higher-level gauges can interpret probe emissions, and controllers can decide upon and enact system repairs and adaptations. Such system models can be developed piecemeal and incrementally, with respect to selected system views or substructures, so a priori full-scale analysis is unnecessary.

Others have also proposed to control the behavior and performance of a running application, either as a generic coordination mechanism [1], or attacking specific aspects of dynamic adaptation: dynamic service composition and management [21], deployment [13], self-modification [8], "perpetual testing" [18]. The distinction of our approach is precisely the externalization of the dynamic adaptation infrastructure, which minimizes interdependencies with the systems that are subject to its control. We see this as a key to generality, with respect to the reach, granularity and kinds of dynamic adaptation that can be exerted.

Previous papers [22][12][14] introduced our concepts, model and system – called Kinesthetics eXtreme (KX, pronounced "kicks") - for applying dynamic adaptation facilities "from the outside" of a given target system. In this paper, we evaluate the model's merits and limitations based on experience gained by putting it to test on a real-world, mass-market Internet service.

WOSS '02, Nov 18-19, 2002, Charleston, SC, USA.

2. THE KX INFRASTRUCTURE
2.1 Overview
Figure 1 shows an idealized view of our infrastructure.

Initially, data is collected from the running target system. It is instrumented with non-invasive *probes* that report raw data to other layers 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 the 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, tune or adapt individual components and/or major substructures of the system.

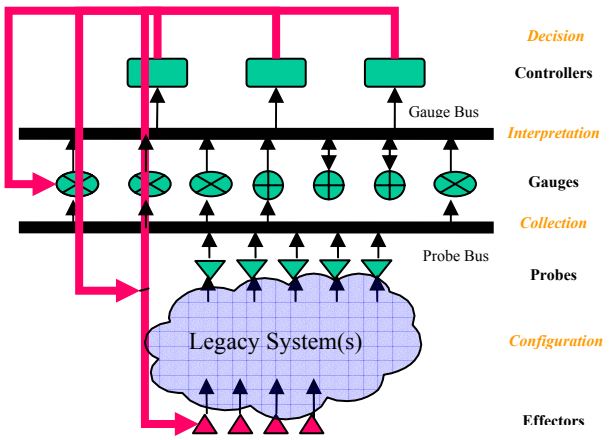

Figure 1: Externalized Dynamic Adaptation Infrastructure.

We emphasize that this infrastructure model 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, we anticipate that reuse should be commonplace, such as for probes and gauges geared towards availability, robustness, network QoS, etc.

2.2 Monitoring

Probing is a necessary prerequisite for monitoring the execution of a running system. We need a minimally invasive approach that can be guaranteed to have zero or negligible effect on the performance and reliability of the system. A probe here is an individual sensor attached to or associated with 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 DARPA DASADA program [19][10], under which KX has been developed to date, has been to agree upon a "standard" API for controlling 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 outside sources as well as ourselves (e.g., the "probelet" in Figure 2, not discussed here), and can "drop in" any probe technology meeting the DASADA standard API [2]. For example, OBJS' ProbeMeister [16] dynamically inserts probes into Java byte code, and Teknowledge's "instrumented connectors" [3] replace Win32 DLLs with pre-instrumented libraries.

We have proposed the "Smart Events" XML Schema [9] as a standard format for structuring probe output data. Our intent is to unify the disparate ways in which the varied probe technologies describe observed events. This Schema includes extension points for inserting additional tag structures appropriate for specific probe and/or gauge technologies. We are aware that XML text is verbose, so we are investigating efficient "wire formats" for XML-based event notations – which would allow the wide base of XML processing tools to be employed at final destinations but incurring relatively little traffic penalty. Our implementation also supports the unstructured attribute/value pairs handled by today's content-based messaging event buses like U. Colorado's Siena [5].

2.3 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 can. These events are typically at a higher level of abstraction, but the aforementioned Smart Events XML Schema has been defined to support both levels. As with probes, a major concern of the DASADA program has been defining a standard gauge API to allow interoperability [11].

Our own gauges operate within a framework consisting of two major components, Event Packager and Event Distiller, shown in Figure 2. The Event Packager transforms, when necessary, the raw-data format of legacy probe output into Smart Events-compatible event streams (using probe- or probe source-specific plugins). It also packages and logs these events in an SQL-based persistent store for possible replaying. The replay can be either "precisely timed" or "fast-forwarded".

The Event Distiller recognizes complex temporal event patterns from multiple probe sources (conceptually similar to Stanford's Complex Event Processing [15]), and constructs higher-level measurements to reflect the system state represented by the events. The Event Distiller is "programmed" by 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 an appropriate higher-level event. These events interface with the decision layer and, optionally, gauge visualizers.

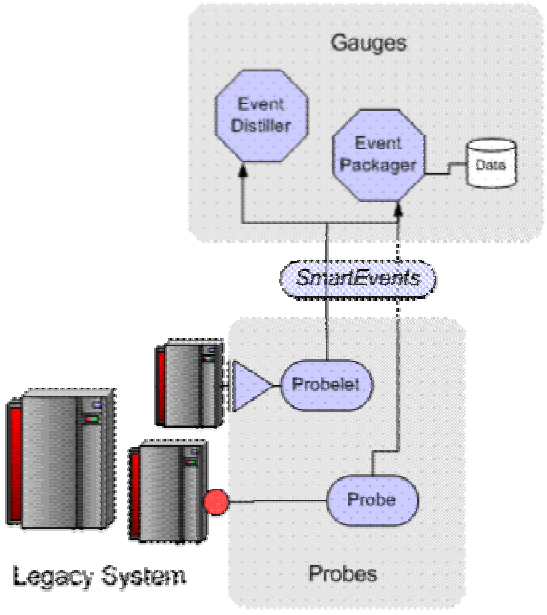

Figure 2. KX Probes and Gauges

Both the probe and gauge buses follow publish/subscribe models with content-based routing, so event producers and consumers do not need to "know about" each other.

2.4 Feedback to Reconfiguration

Gauge outputs are input to a decision process that determines what course of action to take, if any. The decision process may be supported by a variety of tools, including, e.g., an architecture transformation tool (such as CMU's Tailor [20]) that reacts to gauges that detect differences between the running and the nominal architecture. Executing high-level repair action(s), e.g., to reconfigure the architecture, will often involve several activities at the effector (implementation) level. Some of these activities may be conditional or dependent on others, or may simply fail, so one needs to be able to express the adaptation process as a workflow rich enough to express contingency plans. This decision and control layer might also invoke the management actions of the probe and gauge layers on occasion, for example, to induce refined measurements before proceeding with adaptation.

We use our Workflakes decentralized workflow system [23], as illustrated in Figure 3, to instantiate and coordinate all kinds of adaptations of the running system, from local actions to global topological changes. Workflakes is built on top of BBN's Cougaar distributed agents architecture [10].

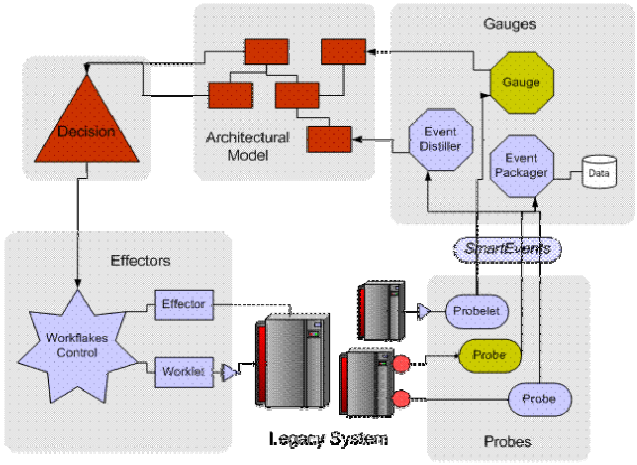

Figure 3. KX Feedback Loop Using Workflakes

We do not yet employ a true workflow notation to describe these activities; the workflow is currently expressed as a set of coding patterns in Java, which are then dynamically loaded into and executed by the Workflakes engine. In the longer term, we will integrate a process workflow specification language, which must support the description of the actions to be applied to repair and adapt a system, including at which location(s) the changes should be applied. The language needs to specify both sequential and parallel execution of actions, and how to deal with unsuccessful actions, by retrying, attempting alternate actions, rolling back or compensating changes.

As with probes, effectors can be realized with various technologies. 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.

Effectors are the most target-specific aspect of our approach, and often must be handcrafted. Workflakes currently conducts an adaptation workflow by selecting, instantiating and dispatching Worklets mobile agents [22], and coordinating the activities of the deployed Worklets on the target system's components and connectors. Worklets carry Java mobile code encased in "jackets" that determine conditional execution and repetition, timing, etc.

3. DYNAMICALLY ADAPTING AN INTERNET MASS-MARKET SERVICE

Our case study concerns a multi-channel instant messaging (IM) service for personal communication, which operates on a variety of channels, such as the Web, PC-based Internet chat, Short Message Service (SMS), WAP, etc. The service is currently offered on a 24/7/365 basis as a value-added service to thousands of users.

Our goals are twofold: We want to achieve *service optimization*, with respect to the overall QoS perceived by the end users, which can be achieved by adapting the functional and/or extra-functional characteristics of the various service components as well as their interactions. QoS requirements include on-the-fly architectural modification for scalability, in response to the detection of host- and component-specific load thresholds, as well as on-the-fly reconfiguration of the server farm hosting the service. We also aim to support *dynamic monitoring and control of the running service*, that is, simplify and resolve a number of concerns related to the continuous management of such a complex distributed application. These requirements include automated deployment of the service code; automated bootstrapping and configuration of the service; monitoring of database connectivity from within the service; monitoring of crashes and shutdowns of IM servers; monitoring of client load over time; and support for "hot" service staging via automated rollout of new versions and patches.

Our case study is organized as a series of iterations, which aim at incrementally fulfilling requirements originating from needs discovered in the field by the service provisioning organization, and elicited from the application development and maintenance team. For each iteration, results are first evaluated in the lab; then new requirements are accepted for the next iteration, while the results produced are delivered and put to test on the field. We report hereby on the results of the first iteration.

The service runtime environment, as shown in Figure 4, consists of a typical three-tiered server farm: a load balancer provides the frontend of the service to all end-users and redirects all client traffic to several replicas of the IM components, which are installed and operate on a set of middle tier hosts. The various replicas of the IM server all share a relational database and a common runtime state repository, which make up the backend tier, and allow replicas to operate in an undifferentiated way as a collective service. Some of the IM servers provide additional facilities, which handle access to the service through specific channels, such as SMS or WAP, and interoperate with third-party components and resources that remain outside of the scope of the service, e.g., the gateways to the cell phone communication network. Those extra facilities wrap the core IM functionality in various ways. Given this kind of modularity, it is possible to achieve continual validation of all of the service components in a server farm in a consistent way, by applying probing, gauging and repair in the first place on the core IM server components, and

extending them as needed to validate any critical features of the additional wrapping components.

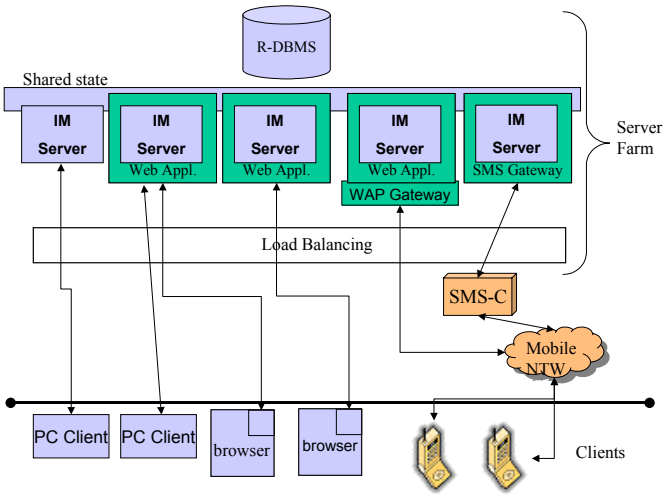

Figure 4. The IM service architecture

The case study addresses all of our requirements using a specific set of probes, gauges and repairs on top of the common facilities provided by the KX platform. Workflakes addresses the manageability requirements by taking responsibility to correctly initiate the service software via a completely automatic process, which replaces the original manual procedures and later scripts for the installation, deployment and bootstrapping of service components. This process is enabled by explicitly integrating knowledge about the service architecture and the runtime environment of the server farm into the logic and data loaded at startup onto the Workflakes engine. Furthermore, Workflakes addresses QoS requirements, responding to scalability needs with a reactive process that orchestrates new deployments of IM servers and opportune reconfiguration of the load balancer (IBM commercial software).

After startup, Workflakes selects one of the hosts from its internal representation of the runtime environment of the server farm and sends out a Worklet mobile agent to it. This Worklet carries and executes bootstrapping code for the IM server and configures it with all the necessary parameters (such as the JDBC connection handle to the DBMS, the port numbers for connections by clients and other IM servers, etc.). Notice that not only the configuration information, but also the executable code of the IM server is deployed and loaded on demand from a code repository made available to the incoming Worklet. This exploits an advantage of a code-pulling feature of the Worklets agent platform, which allows one to do away with any preliminary installation of the application code on all machines taking part in the server farm - greatly simplifying the bootstrapping, staging and evolution of the service. (An analogous approach is followed in U. Colorado's Software Dock [13].)

When the Worklet instantiates an IM server, certain probes are activated to track its initialization. In the event of an unsuccessful initialization, the likely cause is inferred by KX on the basis of the probes' output and reported to a dashboard GUI for the human management of the service, as well as to the Workflakes process. Workflakes may react by deciding to try to bootstrap an IM server on the same machine again, or on another one. Otherwise upon successful initialization, the process dispatches another Worklet onto the load balancer, to instruct it to accept traffic for the IM service and pass it to the initialized server at the right host address and port.

Following the initial bootstrapping phase, Workflakes takes a reactive role, while the KX platform starts monitoring the dynamics of service usage. Certain probes and gauges are activated to track user activity, such as logging in and out of the initialized server. IM servers have an associated load threshold, which in the case of this particular service is most simply expressed in terms of the number of concurrently active clients in relationship with the memory resources of their host. When that threshold is passed, the gauges notify Workflakes, which reacts by trying to scale up the service. It selects an unused machine still available in the server farm, and repeats the bootstrapping process fragment on that machine, including the update of the load balancer configuration. Of course, this scaling-up policy can be repeated as many times as the number of machines in the server farm allows.

Notice that the Worklet bootstrapping a new IM server must carry an extra piece of configuration: an indication of some other alive IM server. This enables the new instance to sync up with the IM server pool and its shared state, and allows it to function as an undifferentiated replica. After a successful initialization of a subsequent IM server, client requests begin to arrive at that server via the reconfigured load balancer, achieving scalability and thus enhancing overall reliability and performance. Other conditions that can prompt new deployments and bootstrapping of IM servers include failures of some existing server replicas, which are inferred by gauges from specific sequences of probe events.

Thus KX together with Workflakes effectively fulfills our deployment, bootstrapping and scalability requirements, supporting both the service monitoring/control and service optimization goals flexibly and dynamically. Minor changes to the bootstrapping process sketched above enable also service evolution campaigns to be expressed as a process with tasks that withdraw old server instances from the load balancer (thus disallowing new traffic to be assigned to them), shut them down when traffic is absent or minimal, and conversely start up, register on the load balancer, and make available to users other server instances with the new code release.

4. THE BOTTOM LINE

- The original manual deployment procedure required 2-3 person-days from scratch on-site, i.e., on the premises of a server farm. Using scripts and assuming DBMS and web application servers already resident, that was reduced this to ½-1 person-day on-site. With KX, that is reduced to a few minutes from a remote location – under the same assumptions.
- The scripts consisted of about 500 lines of csh or other equivalent Unix shell. Using KX, this is reduced to around 220 lines of Java code that runs on Win32 platforms as well as Unix.
- The monitoring and maintenance effort originally required 1 sysadmin on-site 24/7/365, monitoring the state of the service periodically and taking care of trouble tickets as they came, plus 1 technical team on call for further support. KX

enables continuous remote monitoring of major service parameters, with automated alarms, and completely automated resolution of a set of well-known fault conditions.

- Considering one such specific condition: KX recognizes that load threshold is passed in a matter of <1 second, and takes approximately 40 seconds to instantiate a new service instance and load-balance it. Previously, there was no way to detect an overload with direct evidence, and to scale up automatically in response. Performance degradation of IM server(s) was supposedly kept under control by the sysadmin, who would check the number of concurrent users on each server - which is periodically logged - and would manually start up an additional server before such number approached the overload threshold. Such manual inspection was potentially error-prone, risking that resource starvation (e.g., RAM shortage) could remain unnoticed until the server broke down and had to be restarted.

5. FUTURE WORK

Besides continuing to validate KX and Workflakes with this as well as other case studies in different application domains, we have identified a number of areas for improvement and further research that we intend to pursue.

Our current feedback loop is admittedly relatively ad hoc, depending on manually constructed gauge rules that trigger "canned" workflows - albeit with fairly sophisticated instantiation parameters, including access to a "Worklet factory" - to perform reconfigurations. Moreover, when the results reported here were achieved, the decision component of our infrastructure was dispersed between the Event Distiller gauges and the Workflakes workflow engine

We are evaluating workflow languages that can adequately express the coordination logic in Workflakes, and – to that end - we have been experimenting with U. Massachusetts' Little-JIL workflow formalism [6]. We also want to understand and characterize precisely the kinds of the adaptation that can be successfully automated, possibly matching them to different workflow formalisms.

We have also recently begun working with formalized architectural models (for instance, expressed with ADLs) as the base for adaptation decisions. Such architectural models for a given target system can be created a priori by hand (as in [7]), or generated based on analysis of probed event traffic (as investigated by [17]). We are experimenting integration with CMU's Acme toolkit [4], and we can now build gauges that recognize structural changes based on its models, and enact repair processes that are decided upon by Tailor.

6. ACKNOWLEDGMENTS

KX is a team effort of Columbia's Programming Systems Lab. KX components can be downloaded from http://www.psl.cs.columbia.edu/software.html. The general infrastructure model and concepts have been developed 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. Pier Giorgio Bosco, Mario Costamagna, Matteo Demichelis, Elio Paschetta, and Roberto Squarotti at TILAB contributed to the case study. 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 grants CCR-9970790 and EIA-0071954, and by Microsoft Research. The work at TILAB is funded in part by EURESCOM project P-1108 (Olives).

7. REFERENCES

[1] Alonso, G., Workflow Assessment and Perspective, in International Process Technology Workshop, September 1999.

[2] Balzer, R., Probe Run-Time Infrastructure, Teknowledge, December 2001. http://schafercorp-ballston.com/dasada/2001WinterPI/ProbeRun-TimeInfrastructureDesign.ppt.

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

[4] Carnegie Mellon University, Acme Web, The Acme Architectural Description Language. http://www-2.cs.cmu.edu/~acme/.

[5] Carzaniga, A., Rosenblum, D.S., Wolf, A.L., Design and Evaluation of a Wide-Area Event Notification Service, ACM Transactions on Computer Systems, 19(3):332-383, August 2001.

[6] Cass, A.G., Staudt Lerner, B., McCall, E.K., Osterweil, L.J., Sutton, S.M., Jr., Wise, A., Little-JIL/Juliette: A Process Definition Language and Interpreter, in 22nd International Conference on Software Engineering, June 2000.

[7] Cheng, S.-W., Garlan, D., Schmerl, B., Sousa, J.P., Spitznagel, B., Steenkiste P., Using Architectural Style as a Basis for Self-repair, in Working IEEE/IFIP Conference on Software Architecture 2002, August 2002.

[8] Cobleigh, J.M., Osterweil, L.J., Wise, A., Staudt Lerner, B., Containment Units: A Hierarchically Composable Architecture for Adaptive Systems, in 10th International Symposium on the Foundations of Software Engineering, November 2002.

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

[10] Cougaar Home Page, Welcome to the Cognitive Agent Architecture (Cougaar) Open Source Website. http://www.cougaar.org.

[11] Garlan, D., Schmerl, B., Chang, J., Using Gauges for Architecture-Based Monitoring and Adaptation, in Working Conference on Complex and Dynamic Systems Architecture, December 2001.

[12] Gross, P.N., Gupta, S., Kaiser, G. E., Kc, G.S., Parekh, J.J., An Active Events Model for Systems Monitoring, in Working Conference on Complex and Dynamic Systems Architecture, December 2001.

[13] Hall, R.S., Heimbigner, D., Wolf, A.L., A Cooperative Approach to Support Software Deployment Using the

Software Dock, in 21st International Conference on Software Engineering, May 1999.

[14] Kaiser, G., Gross, P., Kc, G.S., Parekh, J.J., Valetto, G., An Approach to Autonomizing Legacy Systems, in Workshop on Self-Healing, Adaptive and Self-MANaged Systems, June 2002.

[15] Luckham, D.C., Frasca, B., Complex Event Processing in Distributed Systems, Stanford University Technical Report CSL-TR-98-754, 1998.

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

[17] Object Services and Consulting, Inc., Software Surveyor Dynamically Deducing Componentware Configurations. http://www.objs.com/DASADA/.

[18] Perpetual Testing. http://www1.ics.uci.edu/~djr/edcs/PerpTest.html.

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

[20] Schmerl, B., Garlan, D., Exploiting architectural design knowledge to support self-repairing systems, in 14th International Conference on Software Engineering and Knowledge Engineering, July 2002.

[21] Shirvastava, S.K., Bellissard, L., Feliot, D., Herrmann, M., De Palma, N., Wheater, S.M., A Workflow and Agent based Platform for Service Provisioning, in 4th IEEE/OMG International Enterprise Distributed Object Computing Conference, September 2000.

[22] Valetto, G., Kaiser, G., 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.

[23] Valetto, G., Kaiser, G., Combining Mobile Agents and Process-based Coordination to Achieve Software Adaptation, Columbia University Department of Computer Science, CUCS-007-02, March 2002.