Retrofitting Autonomic Capabilities onto Legacy Systems

Janak Parekh[1], Gail Kaiser[1], Philip Gross[1] and Giuseppe Valetto[1,2]
[1]Columbia University
Department of Computer Science,
New York, NY 10027 United States
[2]Telecom Italia Lab
V. Reiss Romoli 247 - 10148
Torino, Italy
{ janak, kaiser, phil, valetto } @ cs.columbia.edu

Abstract

Autonomic computing – self-configuring, self-healing, self-managing applications, systems and networks – is a promising solution to ever-increasing system complexity and the spiraling costs of human management as systems scale to global proportions. Most results to date, however, suggest ways to architect *new* software designed from the ground up as autonomic systems, whereas in the real world organizations continue to use stovepipe legacy systems and/or build "systems of systems" that draw from a gamut of disparate technologies from numerous vendors. Our goal is to *retrofit* autonomic computing onto such systems, externally, without any need to understand, modify or even recompile the target system's code. We present an autonomic infrastructure that operates similarly to active middleware, to explicitly add autonomic services to pre-existing systems via continual monitoring and a feedback loop that performs reconfiguration and/or repair as needed. Our lightweight design and separation of concerns enables easy adoption of individual components for use with a variety of target systems, independent of the rest of the full infrastructure. This work has been validated by several case studies spanning multiple real-world application domains.

1. Introduction

The increasing complexity of networked computer systems and applications has led to a tremendous interest in what some have termed *autonomic computing* [1]: in particular, the notion of self-managing software is an attractive approach to reducing the time and effort costs of operating and maintaining software systems, and to increasing their dependability and assurance levels [2]. Related solutions are already being promoted commercially; several major vendors sell enterprise applications that require little help from IT staff to run and maintain [3]. However, most approaches described in the literature for developing autonomic software systems (e.g., see [4]) ignore legacy software and the increasingly common assembly of large scale systems from components supplied by multiple sources, instead assuming the customer or user will be willing and able to migrate to this new generation of systems. A New York Times article [5] describes the "trailing edge" industry, where migration usually is not an option. The article is primarily concerned with electronic assemblies and other hardware used by the military, but the author notes similar factors are at play in civilian telecommunications equipment, medical devices, etc. – many of which also run old software. Even when not mandated by archaic hardware, legacy software may persist indefinitely – even though it was implemented in "unsafe" languages like C, or in languages no longer in common use where many expert maintainers are now past retirement age [6], making the need for autonomic repair even greater [7].

Note that by specifying a legacy system here, we mean any system, no matter how recent, that does not include its own built-in self-management capabilities. Further, some subsystems of the "systems of systems" of interest may indeed include their own autonomic or analogous previous-generation fault-tolerance, dependability, reliability, survivability, etc. facilities, but these alone will not necessarily provide an "end-to-end" self-management capability for the composite system as a whole (this issue is argued in greater detail in [8]).

A few general-purpose facilities have been developed to automate problem detection and/or problem correction for pre-existing software. For example, some new operating systems include engines to automate the collection of crash data [9]; other tools help detect anomalous behavior by monitoring system and application logs [10]; and a few tools provide administrative control over application behavior [11]. However, these tools generally leave analysis of what the system is doing (or not doing), how and why, to a human administrator, who must then determine, plan and carry out the reconfiguration or repair.

In an attempt to do better, we have developed a generic framework for not just collecting but also *interpreting* application-specific behavioral and performance data at runtime. We tailor this interpretation to the application and/or domain by the introduction of system models that can describe expected correct behaviors and possibly anticipate error situations (that can automatically be recognized as having occurred, or not occurred). The models may be relatively simple as well as incremental in the sense that new rules for system behavior (or misbehavior) can easily be added as they are gleaned; deep analysis and formal representation of the target system is not required, but of course would increase value if available. Further, the framework includes a (software) *feedback control loop* [12] to automatically decide when corrections are required, select and instantiate repair plans, and coordinate the execution (and handle contingencies) of the possibly many interdependent elements required for target system reconfiguration – ideally with no downtime, while the system continues operation (possibly at a temporarily reduced level of service).

Our autonomic computing framework consists of four main kinds of components: sensors, gauges, controllers, and effectors – as one rendition of a *reference architecture* we developed together with a consortium of researchers, as explained in [13].[3] The gauges and controllers are informed by *models* of the target system, and thus may be themselves rather generic, with the same components usable over a range of target systems, whereas the sensors and effectors are typically more tightly coupled to the target system and/or its operating environment.

Sensors[4] watch the target system to collect primitive data, while separate *gauges* aggregate, filter and interpret the sensor data according to system models. This monitoring framework can be used with or without a feedback loop that automatically performs dynamic adaptations. Without the feedback loop, gauges would typically generate alerts and/or be visualized on a human systems management console – but providing deeper understanding and more of a "big picture" of the target system's activities than earlier human-oriented systems management.

The automated adaptation framework, which we have previously presented in [14], supplements the monitoring framework with decision, coordination and actuation capabilities. Based on the coalesced and interpreted sensor data relayed by gauges and on modeled information about the target system, a *controller* makes decisions on what adaptations (if any) need to be done. This triggers a controller facility to orchestrate the work of one or more *effectors* – which interact with the target system to carry out the low-level tweaks and tuning, and/or coarser subsystem restarts and reconfigurations, as directed by the adaptation plan. This adaptation framework can also work from a different diagnostic input, such as a traditional management console operated by a human expert, as opposed to the automated monitoring framework presented in this paper.

Our implementation of this approach is called Kinesthetics eXtreme, or KX (pronounced "kicks"). KX runs as a lightweight, decentralized, easily integrable collection of active middleware components, loosely coupled via a publish-subscribe event notification facility. We show how KX can be used to monitor, analyze, and consequently repair a variety of target applications employing models of application-level semantics, protocols and performance requirements, thus effectively achieving the self-configuring, self-healing, self-managing goals of autonomic computing – but for legacy

[3] The consortium included BBN, CMU, OBJS, Teknowledge and WPI as well as Columbia.

[4] In earlier papers we used the term "probe" instead of "sensor", but this terminology became confusing when we embarked on an intrusion detection application – where "probe" refers to an attacker scan of open ports.

systems and/or systems of systems, rather than applying only to new systems with autonomic properties explicitly designed-in.

Of course, our autonomic computing framework, and KX in particular, do not necessarily work for all legacy systems. Our approach is limited by the degree to which the target system enables placement of sensors and effectors and by the availability of application-specific models that can support analysis of sensor data and definition of repair plans using the effectors. Further, the configuration of KX itself *viz-a-viz* the (distributed) target system can become rather complicated under some circumstances. In the full implementation, the controller component is responsible for the initial configuration and possible dynamic reconfiguration of the KX components; in a partial frontend monitoring application, the KX components must be set up manually. Further discussion of this topic is outside the scope of this paper.

This paper starts with an overview and functional model of our approach to an *external* autonomic computing infrastructure, added onto *a posteori* and independent of the computations and communications performed by the target system or system of systems. We then present the architecture and implementation, covering the main components of the infrastructure. We describe highlights of several example applications conducted to date. We compare to related work, and then summarize the contributions of this research and directions for future work. An extended abstract of this paper appeared in [15].

2. Model

The underlying vision of our approach is that of an end-to-end closed control loop superimposed onto a pre-existing software system. The lowest-level observation and actuator components of the feedback loop necessarily rely on selecting technologies, and sometimes *ad hoc* mechanisms – “whatever works,” peculiar to the target system. Our goal is to devise a common platform that abstracts those components and enables a generic infrastructure for the rest of the feedback loop that can operate across a wide variety of target systems.

The model for the common “reference architecture” platform noted above was developed together with other participants in the DARPA Dynamic Assembly for Systems Adaptability, Dependability, and Assurance (DASADA) program [16]. This platform is explicitly conceived to be lightweight and modular. It comprises several first-class entities, as shown in Figure 1. Data and control flow among components are indicated by solid lines and dashed lines, respectively.

The target system refers to the legacy (or new) system, or system of systems, that is being monitored and, when needed, automatically reconfigured or repaired. Note the target system may itself be distributed, in which case the platform components – at least the sensors and effectors - would typically be distributed along with it.

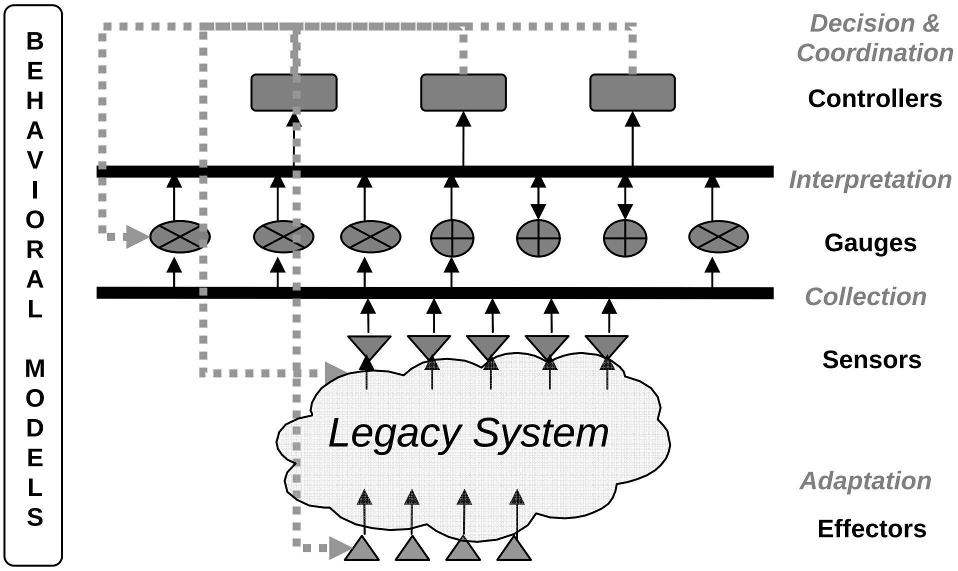

Figure 1. Reference Architecture.

Sensors watch target environment elements to produce time series of data reflecting the state of (that part of) the system, by instrumenting its constituents (as defined by architectural components and connectors, or even finer-grained modules), and collecting local data about their execution. Sensors may emit relatively simple events corresponding to system activities as they occur, more complex events reflecting substantive summaries of log updates, or alerts generated via an internal analysis. Sensors are typically small, constrained, ideally noninvasive pieces of code which get installed in or around the target application system.

Gauges interpret and analyze data originating from one or more sensors. Gauges are intended to recognize abstract semantic events (complex event patterns [17]), which signify whether certain conditions or incidents of interest have occurred (or perhaps are about to occur) in the target system – or alternatively have not occurred within a required time-bound, and in some cases may point out their root cause. Gauges may potentially operate according to an effective hierarchy where higher-level gauges interpret aggregate partial analyses from lower-level gauges.

Controllers encompass decision and coordination capabilities. Decision involves the ability to determine if certain semantic events (as detected by gauges) warrant system adaptation, and selection of the most suitable adaptation strategy amongst those known. Coordination, in turn, involves the ability to recruit, instantiate, direct and orchestrate suitable effectors in an organized fashion as dictated by the selected dynamic adaptation strategy, which might include ordering, synchronization, or other dependencies among the effectors' tasks. Controllers therefore interpret gauge output, perform decision analysis, and coordinate appropriately. While decision and coordination could, in principle, be separated in the architecture (and were separated in one experimental implementation of this reference architecture discussed in [18]), the rationale for combining them is to simplify continual analysis during adaptation: the controller can then consider intermediate outputs from effector activities as well as gauges, possibly leading in some cases to rollback, retry and/or reconstruction of the adaptation plan, analogous to workflow exception handling [19].

Effectors[5] are target-specific code modules that are invoked by controllers once the latter determine an appropriate system adaptation strategy, which may consist of the use of several different effectors. Effectors must be capable of tuning the target system via its exposed configuration parameters. They may also perform partial replacements by initiating and retiring system components, invoke special utilities such as process migration [20] and, in general, carry out nearly any feasible form of adaptation to individual components and connectors of the running system permitted by that target system. Effectors are necessarily more tightly coupled to the target system than the rest of the reference architecture.

Behavioral Models (e.g., architectural models as in [21]) constitute an implicit component necessary to provide relevant information about the target system or its environment: what is its architecture and communication topology, how it is supposed to operate, what are its performance or security requirements or characteristics, and so on, possibly including negative models indicating expectations about what might go wrong. These models are used to customize generic gauges and controllers to the specific target system, e.g., to indicate what to look for and what to do about it. System models could also tailor sensors and effectors, but more typically the sensors and effectors would be customized in selection of technology and/or within their implementations. The models do not necessarily have to be known *a priori*, and could possibly be derived while the system runs (e.g., as in [22]). The set of models is intended to be open-ended, with new ones added or old ones updated or removed incrementally. Behavioral models need not exist as separate runtime entities, but would usually be deployed into the other components.

[5] Some authors instead use the term "actuators"; we consider these terms interchangeable.

Notice how this model, depicted in Figure 1, keeps the autonomic infrastructure logically distinct and largely physically separate from its target – although some infrastructure components may be co-located with target elements. In particular, the sensors and effectors represent the contact points between the autonomic platform and the target system and thus some may run on the same host as some portion of the target system, possibly within the same operating system process. However, since the feedback loop is otherwise entirely outside of the target application, it is possible to maintain a clear separation between the target system specifics and the analysis and adaptation mechanisms employed, enabling commonality and reuse.

The reference architecture is still in its early stages, with respect to both the components and the interfaces among them. We have proposed a common event format for the data flow from sensors to gauges and from gauges to controllers, an XML dialect called "smart events" [23]. Preliminary work has also been done by ourselves and others on target- and implementation-independent APIs for control flow to sensors [24], gauges [25] and effectors [18]. The role of controllers and their interaction with the rest of the infrastructure are less well understood. For simplicity in exposition, as well as generality, we do not discuss or assume any of these formats or APIs in this paper; in any case, it is far too early to evaluate the reference architecture itself, since as far as we know KX is the only complete implementation of the reference architecture. Instead, we focus on our designs and implementations for the infrastructure components, and the infrastructure as a whole, and then discuss our system's application to several experimental case studies.

3. Architecture

Our Kinesthetics eXtreme (KX) architecture, shown in Figure 2, covers the entire "autonomizing legacy systems" reference architecture end-to-end for adding autonomic capabilities to the target legacy system or system of systems. As an externalized platform, KX is not intertwined and tries to avoid interfering with the target system's conventional functional and extra-functional communications and computations. Since the gauges and controllers typically run on separate hosts, the only direct points of possible impact are sensors and effectors; there is more potential of interference with continuously or periodically active sensors than with effectors, which normally remain dormant until invoked to perform repairs. Communications among KX components could possibly affect local resource utilization and available network bandwidth. Careful sensor/effector placement can mitigate this, e.g., placing sensors on the "outside" of components or as taps on inter-component network communications/connectors, although insight into the target system's operations may then be reduced. The only *a priori* knowledge of the specific target system comes from the behavioral models: system-specific models must necessarily be supplied to a given KX instance in order for it to monitor and/or dynamically adapt in accordance with those models. Most runtime knowledge is collected by sensors, although it is sometimes possible for effectors to perform the equivalent of (limited) sensing duties when their repair tasks involve localized checks.

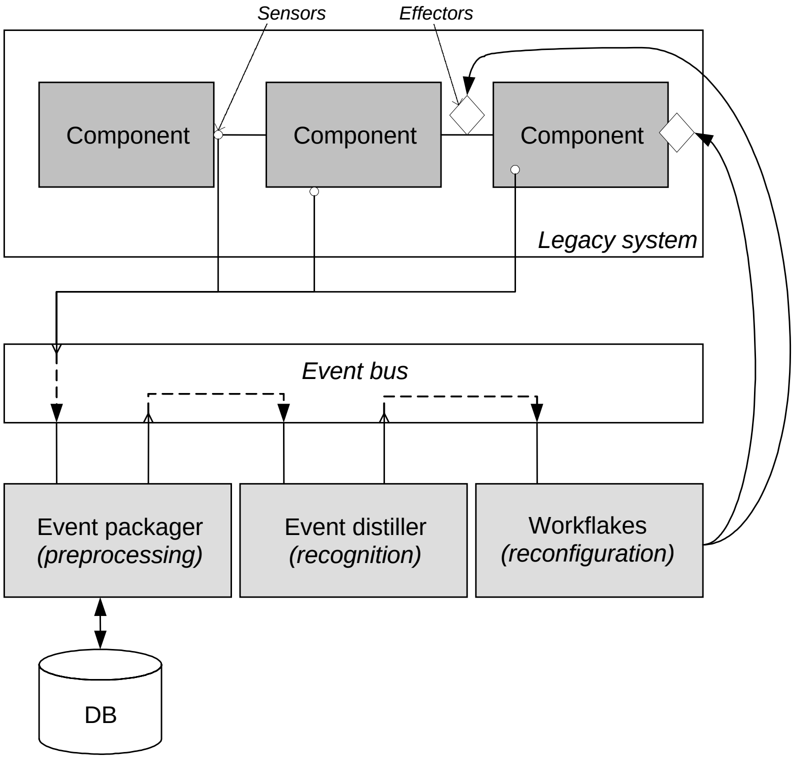

Figure 2. KX Architectural Overview.

Figure 1 depicted the data flow among autonomic infrastructure components as what appears to be a pair of buses, one among sensors and gauges and the other among gauges and controllers. That is a conceptual diagram: these buses might be unified, separate, or there may be no bus at all, e.g., point-to-point connections could be used. The reference architecture intentionally leaves open this issue.

In KX we chose to communicate among sensors, gauges and controllers solely via publish/subscribe event notification, using content-based[6] asynchronous messaging middleware. KX does not separate the buses: in principle sensor events could be subscribed to directly by controllers without gauge intermediaries. By leveraging event notification middleware, KX components can be easily rearranged on-the-fly, with multiple instances of KX gauges and

[6] Subject-based publish/subscribe messaging might have worked just as well for the example applications in this paper.

controllers introduced as needed to address scalability requirements. Any of several available event notification systems could be used [26]. We initially chose Siena (Scalable Internet Event Notification Architecture) from U. Colorado at Boulder [27], as among the most advanced distributed event propagation systems where both source code and support were readily available, but later added support for alternatives (e.g., Elvin [28]). We have recently developed our own event system specifically for KX, discussed briefly in [29], but since it was not used in the experiments presented here we do not consider it further in this paper. Although we have experimented with publish/subscribe communication between controllers and effectors, point-to-point communication is normally employed because the controller usually invokes specific effectors, who must report back unambiguously to that controller to achieve coordination (although this communication may sometimes be asynchronous, as explained in [18]).

No particular sensor or effector technology is formally part of the KX infrastructure, as the best selection among potential technologies must consider the implementation details of the target system and can vary widely. We have to date used mainly our own Worklets mobile agent technology for effectors [30]. Our development of the Worklets platform was originally intended for other purposes [31] and preceded the rest of KX, but it applied nicely to the reconfiguration and repair requirements of our case studies. Other techniques we have experimented with for effectors include JMX management beans [32] and SOAP-based interfaces for synchronous remote calls.

We have employed a number of different sensor solutions developed by others. Some of these inject callbacks into source code (when source code is available and recompilation is feasible), such as WPI’s AIDE [33]. Others modify bytecodes or binaries, such as OBJS’ ProbeMeister [34]; replace DLLs or other dynamic libraries, such as Teknowledge’s mediator connectors [35]; operate in the surrounding environment, e.g., to inspect network traffic, such as System Detection’s Antura [36]; or monitor operating system resource usage. We have also experimented with using Worklet mobile agents to deploy and modify sensors, dubbed “Probelets”. Many other instrumentation technologies are described in the literature, with some commercially available.

Since the various sensor technologies do not necessarily output the event format presumed by our gauges, we introduced the *Event Packager* component as a preprocessor event translation service to transform into a common format. The Event Packager also removes duplicates, timestamps sensor events according to a globally synchronized clock (using NTP [37]) and acts as a “flight recorder” to persistently log the event streams, for later replay or data mining.

The *Event Distiller* is our main gauge component. It performs sophisticated, possibly cross-stream temporal event pattern analysis and correlation across continuous data streams from multiple sensors, to monitor desirable and undesirable behaviors. When undesirable behaviors occur (or desired behaviors do not occur within the requisite time-bound), the Event Distiller generates meta-level events indicating this interpretation; these higher-level notification events also carry information about the lower-level sensor data that contributed to the analysis. The Event Distiller is dynamically configured with the rules defining complex event patterns of interest – that is, the behavioral models regarding what to monitor – so new models can be added and previous models replaced or removed on the fly.

KX fulfills the controller role by employing our workflow technology called *Workflakes* [14,18]. Workflakes is a decentralized process enactment engine, specialized towards automated coordination of software entities as previously suggested in [38], as opposed to the more conventional use of workflow to organize human activities (e.g., see [39]). Workflakes is triggered by gauge outputs to select and tailor a dynamic adaptation plan to the problem at hand, then instantiates and superintends a collection of effectors to enact the tasks specified in the workflow.

Our motivation for employing the workflow paradigm to close the autonomic feedback loop originates from the observation that adapting real-world, complex, systems of systems requires in most cases a multiplicity of fine-grained interventions, impacting separate target elements. Those interventions can be regarded as a set of interdependent activities that may have well-defined causal relationships: they may be conditional (or otherwise dependent) on others; during the course of their enactment, certain activities may fail, calling for some form of contingency planning; etc. In general, the more complex the adaptation policy and the more involved the impact on the target, the more concerted the corresponding plan needs to be. Therefore, a sophisticated coordination mechanism is needed to ensure that the adaptation of the target system occurs in a coherent and consistent way. We argue further in [14] that process workflow technology is a promising approach to addressing that complexity both at the specification and the execution level.

A comprehensive description of Workflakes can be found in [18]. Workflakes is constructed on top of the open-source Cougaar large-scale multi-agent platform [40], adding about 2,200 lines of Java code.

It is important to note that all of the KX components are separately usable. Depending on the problem domain, one or more of these components may be used. For example, if only a few very well-defined repair scenarios are to be performed, or if KX is being used only to do high-level monitoring and report to a human systems management console, one may choose to omit the Workflakes controller component. If only one source of events is being monitored, the Event Packager component may be redundant. For instance, we describe in [41] one dynamic adaptation application using Workflakes, and target-specific sensors and effectors, but without the rest of KX.

The Event Packager and Event Distiller components are discussed in greater detail in the next few sections, which together with the experiments presented in Section 4 constitute the main contributions of this paper.

3.1 Adapting Events from Sensors to Gauges

The Event Packager (EP) component is architected to support a number of event input services, such as duplicate removal and persistent spooling. It utilizes a plug-in architecture to support a broad variety of incoming event formats (*inputs*); a variety of transformations, including timestamping; and a variety of output event formats and other options (*outputs*). New plugins can easily be synthesized; for example, Instant Messaging (IM) messages can be represented as a form of event input.

The various plugins are coordinated via a user-definable metabase (metadata database) that dictates what should be done to the data (*transforms)* and where the data should be sent. Transforms can include single-event processing tasks, such as event clock/timestamp synchronization, static event reformatting/rewriting, augmentation, and selective or complete event persistence. Typically, the goal is to have a number of different input formats streamlined, spooled, and aggregated onto one event stream for the other KX components.

The Event Packager implementation (see Figure 3) was designed from the ground up to be easy to extend. Developing a new input, output, transform or store only requires that one Java class be extended, and some simple methods filled in. This enables the quick and easy creation of wrappers around existing sensors and middleware. The Event Packager uses its own opaque event format container to allow future support for new event formats without breaking compatibility with existing plugins (although for optimal performance, certain plugins might support introspection into event formats for specialized processing). By using opaque event containers, minimal per-event decision-making is needed, which enables the creation of fast pseudo-pipelined datapaths. If more complex processing is needed, a transform can be applied, although this may affect event processing speed.

We have developed input plugins that support and standardize Siena and Elvin messaging, TCP socket streams (transporting both serialized Java objects and XML messages), console input, email (via sendmail), and AOL Instant Messaging (AIM) messaging as event sources. A broad array of output formats closely mirrors these inputs. Transforms include event conversion (from Siena and flat ASCII formats) and event timestamp synchronization (to compensate for distributed clock environments).

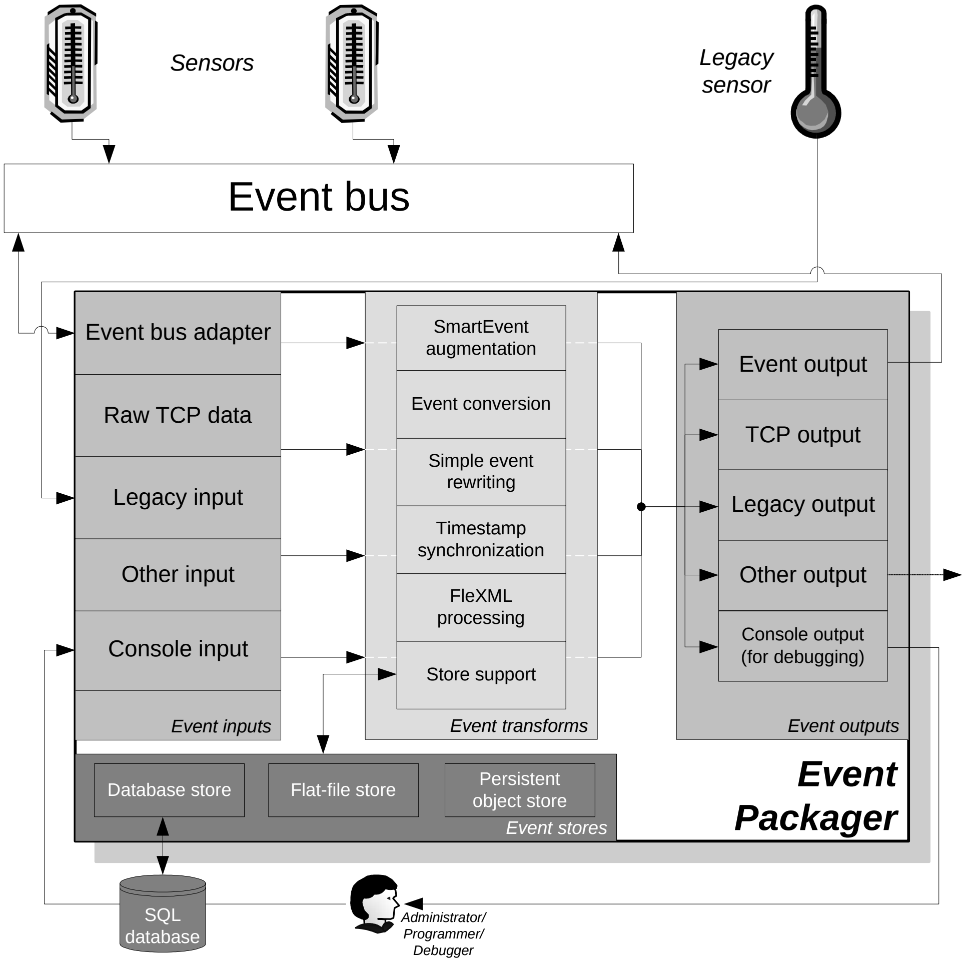

Figure 3. Event Packager Internal Architecture.

EP also supports in-memory, JDBC-backed SQL, and flat-file (serialized object) stores. Persistent logging enables the Event Packager to support "latecomer" analysis, or reanalysis of previously-received event streams, as new Event Distiller gauges are deployed. Multiple persistence techniques may be simultaneously employed via the plug-in model, so that rules can specify persistence to one or more data repositories, such as an SQL database, enabling the use of efficient offline analysis and data mining.

The above components are arranged on-the-fly. Upon startup, the Event Packager reads its configuration file and instantiates the necessary plugins and begins routing events. However, plugins can be added (via Java late-binding reflection mechanisms) and removed during runtime.

EP currently consists of about 9,000 lines of Java code; the core engine that coordinates inputs, transforms, outputs and stores is about 2,000 lines, while the bundled plugins to deal with input, output, transform and store comprise the rest.

Some C glue code handles legacy integration. A typical rulebase is usually a few hundred lines of XML; a small example is given in the Appendix.

Flexible XML (FleXML) is an XML-based technology used by the Event Packager as an optional plugin, to intelligently convert from XML-formatted sensor output to the XML or non-XML event vocabulary expected by the Event Distiller – currently Siena events consisting of unordered sequences of typed attribute/value pairs. Siena supports a naïve translation from XML data into its flat event namespace, but cannot handle hierarchical formats [42]. The next section on FleXML is included for completeness, but can be skipped without loss of continuity. The optional FleXML facility adds 2,200 lines of C++ and Java plus some XML files for the Metaparser and Oracle, and about 4,500 lines of Java for the accompanying Tag Processor architecture.

3.2 FleXML

FleXML (Flexible XML) is an extension of XML that loosens various restrictions of standard XML. It enables XML data from different vocabularies to be treated as if it was all part of the same language for the purpose of cross-application, cross-domain, and cross-version filtering, aggregation and correlation. Our approach is to delay binding of both syntax (schema) and semantics (tag processors) for XML fragments until needed ("on demand"). In particular, FleXML supports what we call "cocktail" XML Schemas, where placeholders (processor instructions) are left to fill in new grammatical fragments on demand, smooth handling of data corresponding to older and newer versions of the same XML Schema, and introduction of special-purpose transformations and processing code for individual XML tags and attributes.

A full treatment of FleXML and its applications is outside the scope of this paper; we address here only the aspects relevant for KX. FleXML's primary use in KX to date has been as an optional plugin for the Event Packager, to intelligently convert from XML-formatted sensor output (such as generated by AIDE [43]) to the XML or non-XML Siena event vocabulary expected by the Event Distiller.

3.2.1 The Metaparser

The *Metaparser* first performs a validating parse of a FleXML message (event). Like typical XML parsers, it performs *syntax* and *validity* checks on the XML document to make sure that it is well-formed and corresponds to a known schema. If either or both of these fail, a typical XML parser declares the document malformed or invalid, respectively, and stops parsing it. While FleXML documents, like XML documents, must be well-formed, the metaparser will not immediately fail in the second error case (e.g., when the next element tag is unknown or not recognized at that point in the schema); instead, it will attempt to resolve the problem autonomically. The apparently erroneous XML fragment, along with namespace and XPath context, is passed to a separate component (the Oracle) that contains a repository of schema information. If a matching subschema is found, it is installed into the Metaparser, which can then continue with its validation of the document. Note the problem fragment could appear in the "middle" of an otherwise valid message, so in effect the subschema is inserted into the parent schema at the relevant location.

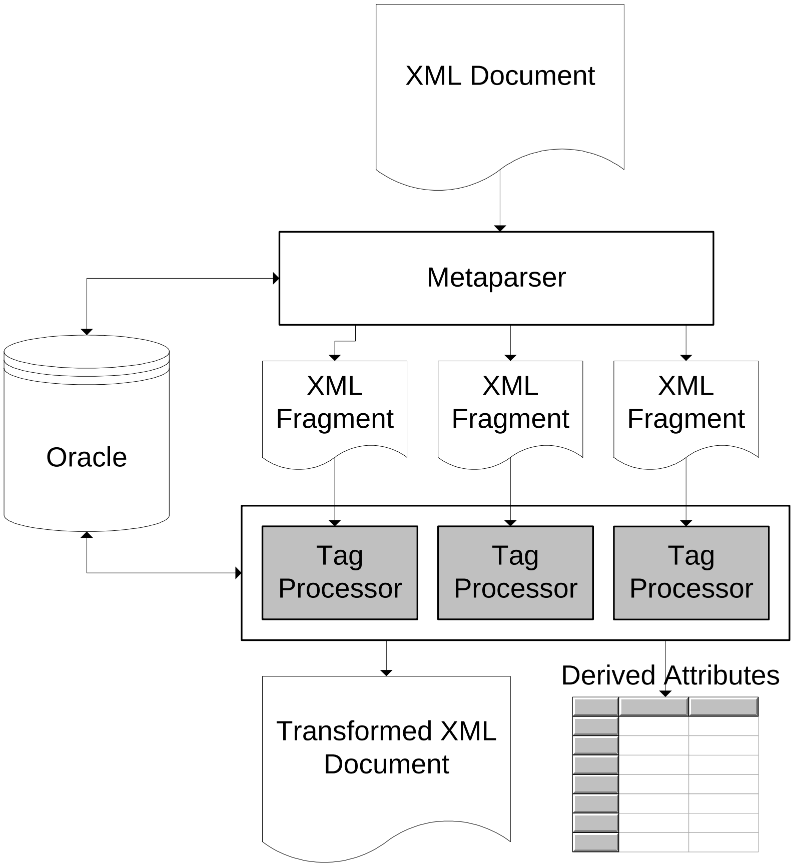

Figure 4. FleXML Internal Architecture.

Composition of multiple schemas within the same message, with dynamic handling of the subschema discovery, is desirable in several situations. Schemas may have been modified, the schema may be inherently "pluggable," as with the SOAP envelope schema [44], or the message format may have been designed as an elaborate composition of separate grammatical components (using the FleXML placeholders mentioned above).

As it parses each message fragment, the Metaparser calls the corresponding Tag Processors, if any happen to be associated with elements or attributes in the message. This is usually to rewrite or augment the original message, e.g., to standardize the format or highlight important attributes. Both tasks use the Oracle to allow adaptive, autonomic behavior in an environment of potentially changing message formats and their meaning. Note this autonomic behavior is with respect to the KX infrastructure itself, as opposed to the target system.

3.2.2 Tag Processors

Tag Processors are XSLT or mobile code components. When the Metaparser hands off a message fragment to a Tag Processor, it also passes an "environment" associated with the message, where the Tag Processor can write its results as well as read results of previous Tag Processors. This allows state to be maintained as the message is processed by multiple components; the Metaparser and Tag Processors are stateless across messages, however, to reduce complexity and size and to improve performance. The Tag Processors, like the schema snippets, are discovered dynamically by querying the Oracle, allowing new Tag Processors to be deployed on the fly.

Tag Processors use a combination of XSLT patterns and XML-specified rules to write values to the environment and possibly rewrite portions of the message. The XSLT "rule template" can apply conventional XSLT transforms and/or add "virtual tags" to the message to identify particular side effects that should occur. An XML rule set describes the various possible effects that should occur, including writing of a particular attribute-value pair to the environment, and/or executing an arbitrary piece of code. Tag Processors are commonly used to standardize different message formats into a single format, to augment messages with higher-level information, and to run arbitrary legacy code modules against messages. The first two uses greatly reduce the burden on the Event Distiller.

3.2.3 The Oracle

The Oracle is a support database for the Metaparser and Tag Processors. When presented with an unfamiliar tag and the context in which it appeared (namespace and XPath), the Oracle will attempt to find and return an appropriate schema or schema fragment, along with associated tag processors. Ideally, an extended context (beyond namespace and XPath) would be provided, and uncertainly and ambiguity dealt with intelligently; multiple matches or "near matches" could be returned. These issues, and other such as caching, replication, eager vs. lazy propagation of updates, centralized vs. decentralized implementation, etc., are outside the scope of this paper.

3.3 Recognizing Event Sequences

The Event Distiller (ED) is responsible for detecting problematic or anomalous system activities by matching (*gauging*) sequences of events emitted by one or more sensors. An *event sequence* is defined here as being a nondeterministic ordering of events ultimately indicating correct vs. incorrect behavior. Such event sequences' transitions (i.e., between subsequent events) will almost always have timebounds so as to emphasize the real-time nature of the application domain and to act as a check on the nondeterminism.

ED rules define the event patterns of interest as derived from behavioral models, in an XML vocabulary; the full notation is given in the Appendix. Note that Event Distiller rules are not related to the Event Packager rulebase specifying plugin configuration. Each ED rule is partitioned into "states" and "actions", where matches amongst the former are mapped to (meta-)events that are emitted corresponding to the latter. This state/transition representation closely corresponds to a nondeterministic finite automaton – the idea is that one event may lead to many different possible subsidiary events, and one wants to match whichever ones are appropriate. Transitions are inherently timebound to provide a control on the size of the nondeterministic matching problem – an expiration implies that a transition is no longer possible, and KX can then garbage collect from the pool of potential matches for incoming events, to reduce the amount of system state required during execution. An alternative approach would be to provide backtracking, but this is impractical given the runtime requirements of such a system and the potentially huge number of events it may witness at any given time.

The Event Distiller internal architecture (see Figure 5) supports several additional first-level constructs as defined in the rule language:

- Rule chaining is accomplished by allowing published actions from one rule to match other rules' states. This late-binding approach enables dynamic rules to be created and to immediately support chaining.
- Looping provides Kleene star-like functionality, but can also match a specific number of times.
- Success and failure actions can be made at any matched state. A success action is published immediately upon reaching the state. A failure action is one where all the transitions from that state to another state are eliminated and no further transitions can be done, and is sent upon successful garbage collection of the current rule instance. Multiple success and failure states can be specified at each state if desired. Such actions may trigger a rule chaining within the Event Distiller architecture, may be used by other interested components (such as controllers that begin applying a repair or reconfiguration workflow), or may even trigger human notification via some immediate communication channel, such as a pager.
- Absorption enables a given state match to be exclusive, e.g., if a particular state of a particular rule enables absorption, all rules below it will *not* match state, even if they specify the exact same criteria as the first

matched rule. Note that this implies a partial ordering upon the rulebase – rules at the top have absorption capability over all other rules in the rulebase, whereas rules at the bottom can declare absorption but such a declaration has no effect.

- Variable binding enables conditional matching – a value can be *bound* by the first match, and further states in a rule may require that value to appear in subsequent events. This is useful for any sequence of events that refer to a common shared value, such as the name or unique identifier of a service being monitored.

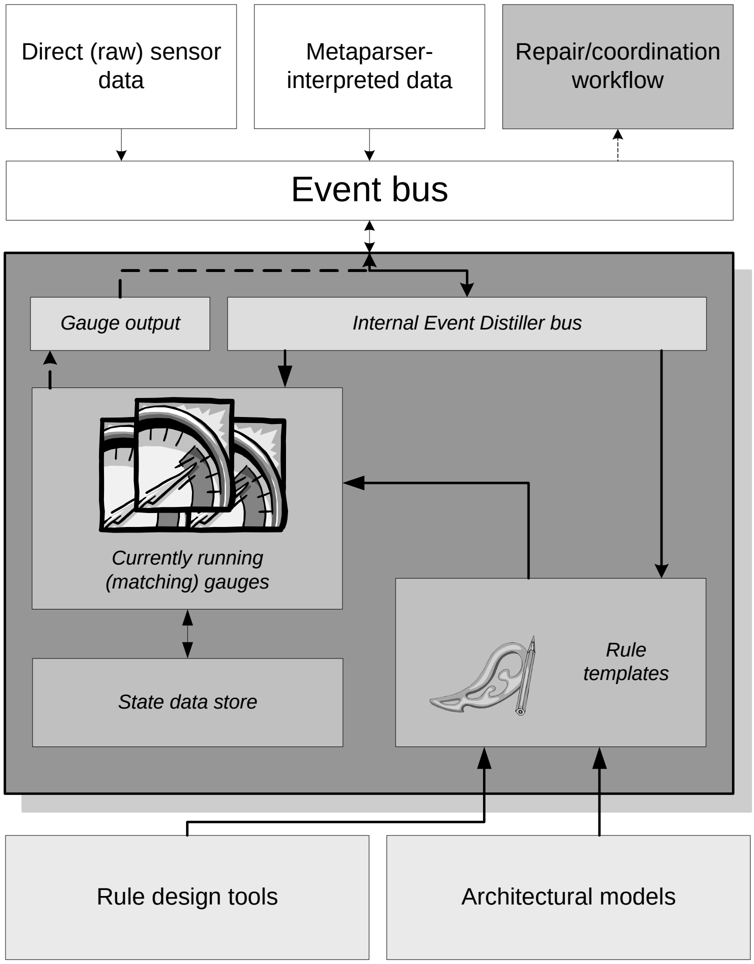
Direct (raw) sensor data
Metaparser-interpreted data
Repair/coordination workflow
Event bus
Gauge output
Internal Event Distiller bus
Currently running (matching) gauges
State data store
Rule templates
Rule design tools
Architectural models

Figure 5. Event Distiller Internal Architecture.

Internally, the Event Distiller uses a collection of nondeterministic state engines for temporal complex event pattern matching. The rulebase is loaded into memory, and forms a series of “state machine templates”; once an event matches the *first* state of one of these templates, an instance of the template is automatically created to keep progress of the matching through the state machine. While this is memory-intensive, it allows a richer representation of event sequences: logic constructs are supported, as are loops, rule chaining, and variable binding as required by the architecture. Memory usage is mitigated by supporting timeouts and automatic garbage collection. Timestamped event reordering is also supported, so if events arrive out-of-order within a certain window (1 second by default), the Event Distiller will rearrange them appropriately so that sequences, and causality, can still be recognized correctly. Note that such reordering, if done with many sources, requires some authoritative time declaration as close as possible to the sources themselves, as network latencies may be unpredictable. If the generator of the events being matched doesn't support timestamping, Event Packagers may be placed at the generation point or at its immediate network peer to create timestamps to enable reordering.

ED's repertoire of event patterns may be populated in one of several ways: First, an XML-format rulebase is supported, where event sequence patterns are specified, along with timebound parameters among sequence elements as well as success and failure notifications. There is also a GUI to assist a KX integrator; it also works as a systems management console for human engineers, although a major goal of the effort has been to automate many repairs within a KX feedback loop (via notifications to Workflakes). Second, the Event Distiller supports dynamic rule generation – messages can be sent to the Event Distiller with XML snippets specifying a rule or a segment of a rule (e.g., to construct new rules on the fly or modify existing rules). Currently, such rule modifications are received through the publish-subscribe channel, potentially containing an XML snippet that contains a full rule (e.g., states and actions) to be matched from hereon. Such rule changes affect *templates* for future matches and not currently-matching rules. Third, as with the Event Packager, other sources can be easily integrated: For instance, support for CMU's Acme architectural description language constraints [21] has been partially integrated: The Event Distiller can act as a "reporting gauge" onto the Acme Gauge Bus [45], thereby providing feedback to the corresponding architecturally-oriented repair tools.

The Event Distiller implementation is currently about 7,000 lines of Java code. The event pattern rulebase may vary in length depending on the complexity of the behavioral model, but in experiments were typically a few hundred lines of XML.

4. Example Applications

In order to validate our approach to introducing autonomic properties into legacy systems, we developed several scenarios and corresponding experiments with real, deployed complex distributed software. We describe three such scenarios in this section, dealing with service failures, load balancing, and quality of service across two different legacy systems (additional applications are discussed in [14] and [41]). Because the authors were not involved in the prior development, deployment and systems management of these real-world systems, it is not possible to offer full “before” and “after” comparisons with detailed quantitative data. However, these case studies demonstrate that it is possible and advantageous to apply our approach to real-world legacy systems. To show the diversity of plausible application domains, we also discuss a simple “toy” example that detects and blocks email spam.

4.1 Service Failures

We integrated KX with a complex GIS (Geographical Information System) intelligence analysis tool developed at the USC Information Sciences Institute (ISI) and used experimentally at the US PACOM (Pacific Command), known as GeoWorlds [46]. GeoWorlds uses a distributed set of services glued together by Jini [47]. While the system generally works well, services sometimes stop running, with no recourse except to wait for the request to time out and to manually restart the appropriate backend subsystem. For example, GeoWorlds’ reliance on harvesting external websites (e.g., www.bbc.co.uk) – for news items that it then maps to locations in the GIS system – is subject to frequent glitches (DDoS, server failure, etc.), requiring restart of the GeoWorlds service that is trying to access the external website, possibly substituting an alternative news site.

Using WPI’s AIDE (Active Interface Development Environment) sensor technology [33], we were able to automatically instrument the GeoWorlds Java source code, and in particular the mechanism that dealt with request-to-service dispatch, with sensors that would monitor the start and end of method calls that were relevant to contacting external services. The Event Distiller incorporated rules to monitor a variety of method calls, making sure that a “termination” call matched up with each “initiation” call within an appropriate timebound (ranging from seconds to a minute). AIDE reports method calls in an XML format; these calls were then translated to a simple attribute/value set via the FleXML Metaparser and fed into the Event Distiller.

```
<state name="Start" timebound="-1" children="End" actions="" fail_actions="">
  <attribute name="Service" value="*service"/>
  <attribute name="Status" value="Started"/>
  <attribute name="ipAddr" value="*ipaddr"/>
  <attribute name="ipPort" value="*ipport"/>
  <attribute name="time" value="*time"/>
</state>
<state name="End" timebound="15000" children="" actions="Debug" fail_actions="Crash">
  <attribute name="Service" value="*service"/>
  <attribute name="State" value="FINISHED_STATE"/>
  <attribute name="ipAddr" value="*ipaddr"/>
  <attribute name="ipPort" value="*ipport"/>
  <attribute name="time" value="*time2"/>
</state>
```

Figure 6. Failure Detection Pattern.

Figure 6 shows an example of a simple event pattern used to perform such failure detection. The incoming sensors reporting Status and State values track method completion. If for some reason a “FINISHED_STATE” is not received within 15 seconds after a method had initiated, the system sends out the “Crash” event; otherwise, the “Debug” notification is emitted, signifying a “success” and acting as a record of the operation for future debugging purposes if desired. Note that the strings prefixed with an asterisk (“*”) designate a variable binding, e.g., the Event Distiller substitutes all instances of “*service” by the first source that it sees for this instance of the rule. Thus, this one rule can match a large number of different sources and subjects.

When the repair system (Workflakes) received a “Crash” event, the repair involved a simple restart of the service as specified in the message generated by the Event Distiller. A more sophisticated repair (which was not implemented) would have coordinated multiple services to prevent having to restart a long transaction from scratch, instead using partial results leading up to the individual service failure. Even with the simple repair, however, we were able to automate a process that previously had been done manually.

4.2 Load Balancing

Several GeoWorlds execution scripts rely on computationally-intensive backend services hosted at ISI, such as a noun phraser that would analyze incoming news articles and extract nouns for mapping to GIS attributes; crash avoidance and performance maximization through request relocation was clearly desirable. To accomplish this, the relocatability of Jini services was exploited to build a load-balancing solution for GeoWorlds. A system monitor sensor was built in C# to measure the overall load on the backend system(s) running the noun phraser, and results were piped into a custom plugin for the Event Packager.

CMU’s Acme architectural description language [48] was used to specify GeoWorlds’ architectural composition and system load constraints on the various services. The Acme Gauge Extractor then constructed Event Distiller rules based on these constraints. During the execution of various services, if this load exceeded a predetermined threshold for an extended period of time, the Event Distiller detected and reported it as a violation of the architectural constraints. The triggered repair caused the service to move to a different Jini-enabled host. We were able to visualize the load and service state using AcmeStudio’s architectural diagram visualization tools [49], so one could watch the feedback loop in action, in concert with the architectural model.

Additional logic was programmed into the Event Distiller rulebase to detect oscillation, utilizing the feature whereby ED can listen to itself (more generally, a hierarchy of EDs could be constructed to analyze meta-level events). In particular, if many (outgoing) Event Distiller messages requesting a load-balance were detected within a short timespan, one of two strategies could be selected: either eliminate load balancing between the two oscillating hosts for future repair plans (by notifying the Workflakes controller), or increase the load threshold in the architectural constraints (either by patching ED's own rulebase or by manipulating the Acme constraints used to generate the rulebase). We implemented only the former, but the latter approach presents a more flexible long-term solution for future work.

4.3 Quality of Service

We had the opportunity to experiment with a commercial J2EE-based multi-channel Instant Messaging (IM) service used daily by thousands of real-world end-users. First, on-demand scalability was added: by monitoring user sign-on events and server request queues, KX was able to determine the load of each member of the IM server farm and take appropriate actions whenever needed. Repairs, selected on the basis of inferences carried out using Event Distiller rules, encompassed modifications to the threading model of active servers, or even on-the-fly deployment and activation of additional server instances and corresponding reconfiguration of the commercial load-balancer of the IM server farm to redirect client traffic to these new servers. Failure detection was also supported from a load-balancing standpoint: information on server failures and interconnections between servers and backend DBMS entities was similarly captured to facilitate load balancer reconfiguration to direct client traffic to still-functional servers. The same set of sensors and effectors, coupled with slightly different Event Distiller gauge rules and Workflakes repairs, were also used to support controlled and graceful staging of the service infrastructure; this enabled automated software release deployment without necessitating a complete shutdown (and service interruption) during transitions.

A set of quantitative results were derived from running and observing the adapted IM system in lab conditions, with both manual and automated traffic simulation that reproduced in-the-field demands on the IM service. These results focus on the improvement via automation in the support, maintenance and management activities typically carried out on the IM service under field conditions. Also, some measurements about the development and integration effort necessary to implement the case study were taken. The most significant results are:

- Substantial reduction in effort for deployment and configuration of an IM service in the field, originally approximately ½ person-days, with locally present experts. With KX, that was reduced to 1-2 minutes from a remote location.
- Reduced monitoring and maintenance effort necessary to ensure the health of the running service. KX completely automated the 24/7/365 monitoring of a set of major service parameters, as well as the counter-measures to be taken for a set of well-known critical conditions.
- Reduced reaction times and improved reliability: for example, KX recognized the overload of an IM server in 1-2 seconds and deployed an additional server replica in approximately 40 seconds. Overload detection was originally manual, starting with accumulated application logs – a clearly error-prone approach, potentially endangering service availability.
- Manageable coding complexity: KX sensors, gauges and effectors were derived from generic code instrumentation templates that were then customized with situational logic. This resulted in rather compact code: 15 lines of Java code on average for sensors, and usually less than 100 lines for effectors. The total code written for this specific case experiment on top of the generic monitoring and dynamic adaptation facilities provided by the KX infrastructure was approximately 2,000 lines of Java and XML.

This study demonstrated the utility of a KX end-to-end feedback loop for service management and application-level QoS in an industrial context. Traditional application management practices report warnings, alarms and other information to some knowledgeable human operator who can recognize situations as they occur and take actions as needed – with very limited automation in the management platform. Instead, our approach offers a high level of guidance, coordination and automation to enforce what is a complex but often repeatable and codifiable process.

4.4 Spam Detecting and Blocking

In order to demonstrate KX's flexibility beyond the more conventional system management cases above, we instrumented Sendmail [50], a popular email Message Transfer Agent (MTA), to capture messages being received in a target network. More precisely, a *Sendmail milter* [51] was created and installed to capture incoming traffic. Specific attributes about each message (such as source address, subject, and Message-ID) were captured by sensors, encapsulated into events by the Event Packager, and sent to the Event Distiller. The Event Distiller rules (see Figure 7) would trigger if multiple (3+) messages containing the same source and Message-ID were received in a very short timespan (less than

10 seconds). Once detection occurred, a mobile agent effector was dispatched to reconfigure the Sendmail MTA in the target network to block all further messages from that source address by rewriting the configuration file and sending a hangup signal (SIGHUP) to Sendmail to reload its configuration.

```
<state name="a" timebound="-1" children="b">
  <attribute name="from" value="*1"/>
  <attribute name="messageID" value="*2"/>
</state>
<state name="b" timebound="100" count="1" children="" actions="A,B" fail_actions="F" absorb="true">
  <attribute name="from" value="*1"/>
  <attribute name="messageID" value="*2"/>
</state>
```

Figure 7. Sample Pattern to Detect Repeated Emails.

This solution worked for simple spam – i.e., one message sent by a spammer to sufficient people in the same organization would verifiably get caught and future communication from that spammer would be blocked. Of course, the organizational newsletter might also be blocked. While this technique is superceded by better spam-specific technologies, such as SpamAssassin [52], which uses dynamic rules and Bayesian learning to distinguish more “stealthy” spam, this example demonstrates the broad utility of our Event Distiller’s timebound-based pattern matching, in this case with email-specific semantics. In essence, we were able to add (limited) autonomic behavior to Sendmail.

5. Related Work

Other projects in the DARPA DASADA program more directly addressed the technical details of specific environments; our main goal was to provide a generic, reusable infrastructure to extend their gauges to on-line operation, while the target system remained running (i.e., without bringing it “down”). For example, Spitznagel [53] discusses static model checking to analyze the compositionality and traceability properties of component wrappers, while Geib [54] combines multiple interacting constraint satisfaction algorithms to verify component composability, validity and adaptability *a priori*. Combs [55], Cobleigh [56] and Wolf [57] were other projects that investigated reconfiguration workflow; we address these and other reconfiguration tools and their relation to the KX controller in [13]. Oreizy [58] also took an end-to-end approach, but applied only to target systems exhibiting the C2 architectural style [59].

A number of other projects also take a middleware approach to autonomic computing. Cornell’s Astrolabe system [60,61] acts as a distributed hierarchical information repository, using a replicated DNS-like infrastructure to support a number of applications. They provide an SQL-like interface as a base model, and define solutions, including management, as manipulations on top of this repository. While they have developed system monitoring solutions on the

Astrolabe infrastructure, their implementation benefits environments where a large number of nodes may need to know specific application-specific information and where short latencies are less critical. KX could be adapted to adopt a solution like Astrolabe to store and replicate its behavioral models across the Internet into specific application domains, within which it would do low-latency, fast-response model and system monitoring. The JAGR project [62] adds self-recoverability to the open-source JBoss J2EE application server, thereby providing a middleware autonomic layer for J2EE business logic components (known as Enterprise Java Beans, or EJBs).

In the commercial arena, OC Systems has an analogous platform to DASADA sensors and monitors with their AProbe [63] and RootCause [64] products, while SMARTS offers their Automated Business Assurance service with "Codebook Correlation Technology." [65] These technologies are generally noninvasive and rely on quickly matching against static or predetermined analysis, as compared to our intent to integrate with application semantics, where new success or failure rules can be introduced on the fly.

Both the network and fault management communities have some autonomic behavior as well. The NESTOR project [66] takes a network-layer approach to monitoring. Additionally, JSpoon [67] is a language developed for the NESTOR project that adds "management" attributes to a network architecture. KX may benefit from JSpoon-like semantics in an attempt to enrich our behavioral models with lower-level network information. Fault management systems [68,69] are also closely-integrated at the systems level, for telecommunications-level reliability. These systems are largely static, designed for vertical solutions, and not for complex distributed "systems of systems".

Intrusion detection systems [70,71] usually focus on system- or network-level security, and are not generally useful for application reliability or self-management. We are actively investigating the migration of our work towards intrusion detection to better support specific application-level security semantics, in particular based on semantic models gleaned from machine learning systems [72,73].

A number of academic and commercial generalized event correlation systems exist [74,17], which correspond, to some extent, to our Event Distiller gauges. These generally use a coding and compilation approach to defining event patterns; in contrast, our dynamic-at-runtime rules are better adapted to embedding solutions in continuously running systems, albeit at potentially lower performance levels. We are in the process of investigating these tradeoffs further.

Several sensor and gauge technologies have been integrated into the event propagation [75] and network layers, often in hardware via SNMP [76]. These tend to be optimized for lower-level, high-volume general-purpose packet streams.

They can easily be utilized by KX, which can provide higher-level semantics to simple matches found in these lower layers.

"Grid Computing" attempts to make distributed computing resources visible as a single virtual computer. Features such as system configuration management and autonomic management have been listed as desiderata for commercial Grid computing [77], but are not currently part of the Grid computing standards. IBM's OptimalGrid [78] approach supports instrumentation and self-optimization in a Grid computing environment, but requires a customized implementation which it can then assemble and coordinate, as opposed to supporting legacy software. Similarly, the AutoMate project [79] defines autonomic system components as being built on a particular framework known as Distributed Interactive Object Substrate, or DIOS, and tightly controls execution of these components. KX does not provide grid computing semantics, and as a result can rely on a legacy framework, instead building a late-binding management layer on top of it. In fact, we consider grid computing to be a natural match for the automated distributed management capabilities of our KX architecture.

A number of systems that address repair, reconfiguration or other adaptation issues exist, either in the context of autonomic computing platforms or on their own account. What distinguishes our work is once again its externalized stance. In fact, many solutions that could fulfill the role of the controller in our model present significant structural dependencies with respect to their target system. One of the most common approaches is represented by an environment or middleware with native adaptation capabilities. Examples include Conic [80], Polylith [81], 2K / dynamicTao [82] and many others; they all offer a set of dynamic adaptation primitives as a premium for applications built with and operating on top of themselves. Georgia Tech [83] defines a concept called *service morphing*, which refers to dynamic service assembly, deployment, and coordination. They also employ middleware and event-based detection to rearrange services for better QoS, but focus on newly-engineered services, whereas KX relies on manipulating existing service architectures.

Minsky [84] defines a formalism to define whether or not a complex, distributed system can be termed self-healing; the principal concept is that the system must have *regularities*. Further research might help determine if the addition of KX to an "irregular" legacy architecture can "smoothen" it and provide the regularities needed to enable such self-healing behavior.

6. Conclusions, Limitations and Future Work

We have presented a general approach to retrofitting autonomic capabilities onto pre-existing systems designed and developed without monitoring and dynamic adaptation in mind. The generic reference architecture, consisting of sensors, gauges, controllers and effectors, and tailored by system-specific behavioral models, can be implemented in many different ways. Our KX implementation provides sample components for gauges (Event Distiller) and controllers (Workflakes), as well as additional components (Event Packager and FleXML) that act as adaptors for diverse sensor technologies developed by others. Our primary experience with effectors utilizes our own Worklets mobile agents. KX has been used to add self-management and self-healing functionality to several legacy systems and systems of systems, spanning service failures, load balancing, quality of service and an experiment in spam detection and blocking.

It is important to note that not all legacy systems are amenable to our approach. It must be possible to place sufficient sensors into and around the target system's components in order to monitor it, and the target system must expose sufficient "control knobs" for effectors to reconfigure it. In some cases, sensors may be limited to probing operating system and network activities that only indirectly indicate target system operations, whereas effectors may sometimes be restricted to coarse-grained component restarts and replacements. The latter was indeed the case for portions of our experimental applications as discussed above.

Although gauges could be restricted to always specify maximum time bounds, event propagation delays and repair plans (allowing for contingencies) may not always be predictable. KX is therefore not currently suitable for hard real-time systems. However, some of the KX components have been applied to a soft real-time multimedia application, where delays were explicitly accounted for (see [41]).

Further, most of the behavioral models employed to date have been relatively *ad hoc*, based on the authors' understanding of the external behaviors of each system as opposed to formal models – which are difficult to obtain for many real-world legacy systems. However, preliminary work with *a posteori* architectural models of GeoWorlds, defined in an architectural description language, shows promise. We are investigating avenues for automatic derivation of gauges, that is, Event Distiller complex event pattern rules, from those and other kinds of system and behavioral models. We are also considering a wider range of application domains. One particularly intriguing application area is autonomic service survivability in the face of insider and outsider security attacks, to coordinate what would otherwise be isolated, independently operating security mechanisms, as we propose in [20].

Acknowledgements

We would like to thank the other members of the Programming Systems Lab for their contributions to this effort; we would particularly like to thank Jinghua Yan for his theoretical work on the power and limitations of FleXML as a language formalism, and Gaurav Kc and Dan Phung for their efforts on Worklets. Our outside colleagues alternately provided criticism and encouragement – Bob Balzer, Dave Wile, David Garlan, Bradley Schmerl, David Wells, Nathan Combs, George Heineman, Bob Neches, Lee Osterweil, Alex Wolf and John Salasin. We would particularly like to thank Joe Hellerstein of IBM for his insightful discussions regarding control theory issues, which we anticipate will be reflected more fully in a future version of KX. During this effort, the Programming Systems Lab was 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-0203876, EIA-0202063, EIA-0071954 and CCR-9970790, and by Microsoft Research and IBM. The software described here can be downloaded for research and education purposes from http://www.psl.cs.columbia.edu/software.html.

References

[1] IBM Research. *Autonomic Computing.* http://www.research.ibm.com/autonomic.

[2] Smith D.J., Schuff, D., and St. Louis, R. “Managing your total IT cost of ownership”, *Communications of the ACM,* 45(1):101-106, January 2002.

[3] Gonsalves, A. “IBM Releases Blueprint For Automated Computing”, *TechWeb News*, April 4, 2003.

[4] IEEE. *Autonomic Computing Workshop: Fifth Annual International Workshop on Active Middleware Services*, June 2003.

[5] Feder, B.J. “On the Trailing Edge of the Arms Industry, by Choice.” *The New York Times*, March 30, 2003.

[6] DeBare, I. “Programmers in the Driver's Seat: Companies clamor for Year 2000 programmers.” *Dr. Dobb's Journal*, Spring 1998.

[7] Raijlich, V., Wilde, N., Buckellew, M., and Page, H. “Software cultures and evolution.” *IEEE Computer*, Vol. 34(9):24-28, Sep. 2001.

[8] Valetto, G. *Orchestrating the Dynamic Adaptation of Distributed Software with Process Technology*. PhD Thesis, Columbia University, April 2004.

[9] Bekker, S. “Microsoft Error Reporting Drives Bug Efforts”, *ENT News*, October 3, 2002.

[10] SANS, “What is Host-Based Intrusion Detection?” *Intrusion Detection FAQ*, http://www.sans.org/resources/idfaq/host_based.php.

[11] LANDesk Software, *LANDesk Management Software*, http://www.landesksoftware.com/.

[12] Hellerstein, J.L., Diao, Y., Parekh, S. And Tilbury, D. *Feedback Control of Computing Systems*. Wiley, 2004.

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

[14] Valetto, G. and Kaiser, G. “Using Process Technology to Control and Coordinate Software Adaptation.” *International Conference on Software Engineering*, May 2003.

[15] Kaiser, G., Parekh, J., Gross, P. and Valetto, G. “Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems.” *Fifth Annual International Active Middleware Workshop*, June 2003.

[16] Salasin, J. *DARPA DASADA Program*, http://www.rl.af.mil/tech/programs/dasada/program-overview.html.

[17] Luckham, D. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley, 2002.

[18] Valetto, G. *Orchestrating the Dynamic Adaptation of Distributed Software with Process Technology*. PhD Thesis, Columbia University, April 2004.

[19] Hagen, C. and Alonso, G. “Exception Handling in Workflow Management Systems.” *IEEE Transactions on Software Engineering*, 26(10):943-958, October 2000.

[20] Keromytis, A., Parekh, J., Gross, P.N., Kaiser, G., Misra, V., Nieh, J., Rubenstein, D. and Stolfo, S. A Holistic Approach to Service Survivability. *First ACM Workshop on Survivable and Self-Regenerative Systems*, October 2003, pp. 11-22.

[21] Garlan, G., Cheng, S.W., and Schmerl, B. “Increasing System Dependability through Architecture-based Self-repair.” In de Lemos, R., Gacek, C. and Romanovsky (eds.), *Architecting Dependable Systems*, Springer-Verlag, 2003.

[22] Wells, D.L. and Pazandak, P. Taming Cyber Incognito: Tools for Surveying Dynamic/Reconfigurable Software Landscapes. *Working Conference on Complex and Dynamic Systems Architectures*, December 2001.

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

[24] Balzer, B. *Probe Technology Adaptor Design.* February 2001. http://schafercorp-ballston.com/dasada/2001WinterPI/ProbeTechnologyAdaptorDesign.ppt.

[25] Schmerl, B. *A Proposal for a DASADA Gauge Infrastructure*. June 2001. http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/able/www/presentations/gauge.html.

[26] Rosenblum, D.S. and Wolf, A. L. “Survey: *Internet Scale Event Notification.”* Workshop on Internet Scale Event Notication, *July 1998. http://www.isr.uci.edu/events/twist/wisen98/presentations/Rosenblum/Rosenblum.PPT.*

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

[28] Segall, B., Arnold, D., Boot, J., Henderson, M., and Phelps, T. “Content-based routing with Elvin4.”*Australian UNIX and Open Systems User Group Winter Conference* (AUUG2K), June 2000.

[29] Gross, P., Parekh, J. and Kaiser, G. Secure ``Selecticast" for Collaborative Intrusion Detection Systems. *International Workshop on Distributed Event-Based Systems*, May 2004.

[30] Valetto, G., Kaiser, G.E. and Kc, G. “A Mobile Agent Approach to Process-based Dynamic Adaptation of Complex Software Systems.” *Eighth European Workshop on Software Process Technology*, LNCS 2077, June 2001.

[31] Kaiser, G., Stone, A. and Dossick, S. “A Mobile Agent Approach to Lightweight Process Workflow.” *International Process Technology Workshop*, September 1999.

[32] Sun. *Java Management Extensions (JMX).* http://java.sun.com/products/JavaManagement/

[33] Heineman, G., Calnan, P., Kurtz, B., et. al. *Active Interface Development Environment (AIDE).* http://www.cs.wpi.edu/~heineman/dasada/.

[34] Pazandak, P. and Wells, D. “ProbeMeister: Distributed Runtime Software Instrumentation.” *First International Workshop on Unanticipated Software Evolution*, June 2002.

[35] Balzer, R.M. and Goldman, N.M. “Mediating Connectors: A Non-Bypassable Process Wrapping Technology.” *DARPA Information Survivability Conference & Exposition*, Vol. 2, January 2000.

[36] Robertson, S., Siegel, E.V., Miller, M. and Stolfo, S.J. “Surveillance Detection in High Bandwidth Environments.” *DARPA DISCEX III Conference*, April, 2003.

[37] Mills, D.L. *Network Time Protocol*. RFC 958. 1985. http://www.faqs.org/rfcs/rfc958.html.

[38] Wise, S., Cass, A.G., Staudt Lerner, B., McCall, E.K, Osterweil, L.J., Sutton, S.M. Jr. “Using Little-JIL to Coordinate Agents in Software Engineering.” *Automated Software Engineering Conference*, September 2000.

[39] *The Workflow Management Coalition.* http://www.wfmc.org/.

[40] Cougaar: An Open Source Agent Architecture for Large-scale, Distributed Multi-Agent Systems. http://www.cougaar.org/.

[41] Phung, D., Valetto, G., Kaiser, G. and Gupta, S. *Optimizing Quality for Collaborative Video Viewing.* Columbia University Department of Computer Science, CUCS-009-04, April 2003. http://www.cs.columbia.edu/~library/TR-repository/reports/reports-2004/cucs-009-04.pdf.

[42] Erenkrantz, J.R. *Handling Hierarchical Events In An Internet-Scale Event Service,* March 2001. http://www.ucf.ics.uci.edu/~jerenk/siena-xml/SienaPaper.html.

[43] Gill, P.W. *Probing for a Continual Validation Prototype*. MS Thesis, Worcester Polytechnic Institute, May 2001. http://www.wpi.edu/Pubs/ETD/Available/etd-0826101-235008/.

[44] W3C. *SOAP Version 1.2 Part 1: Messaging Framework: W3C Recommendation 24 June 2003*. http://www.w3.org/TR/2003/REC-soap12-part1-20030624/.

[45] Carnegie Mellon University ABLE Group. *DASADA Gauge Infrastructure*. http://www-2.cs.cmu.edu/~able/rainbow/gaugeinf.html

[46] ISI. *GeoWorlds GIS System*. http://www.isi.edu/geoworlds/.

[47] Sun. *Jini Technology*. http://www.sun.com/software/jini/.

[48] Carnegie Mellon University ABLE Group. *Acme Architectural Description Language*. http://www-2.cs.cmu.edu/~acme/

[49] Carnegie Mellon University ABLE Group. *AcmeStudio Development Environment*. http://www-2.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio.html.

[50] Sendmail Inc. *Sendmail Mail Server*, http://www.sendmail.org/.

[51] Sendmail Inc., *Sendmail Mail Filter API,* http://www.sendmail.com/partner/resources/development/milter_api/.

[52] SpamAssasin. *Spam Filter*. http://www.spamassassin.org.

[53] Spitznagel, B. and Garlan, D. “A Compositional Formalization of Connector Wrappers.” *International Conference on Software Engineering*, May 2003.

[54] Geib, C., Vestal, S. and Binns, P. *Webpage for HTC's DASADA project.* http://www.htc.honeywell.com/projects/DASADA/.

[55] Combs, N. and Vagle, J. Adaptive mirroring of system of systems architectures. *Workshop on Self-Healing Systems*, November 2002.

[56] Cobleigh, J., Osterweil, L., Wise, A., and Lerner, B. “Containment Units: A Hierarchically Composable Architecture for Adaptive Systems.” *Tenth International Symposium on the Foundations of Software Engineering*, November 2002.

[57] Wolf, A., Heimbigner, D., Knight, J.C., Devanbu, P.T., Gertz, M., Carzaniga, A. “Bend, Don't Break: Using Reconfiguration to Achieve Survivability.” *Third Information Survivability Workshop - ISW-2000*, October 2000.

[58] Oreizy, P., Gorlick, M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. and Wolf, A. “An Architecture-Based Approach to Self-Adaptive Software.” *IEEE Intelligent Systems*, 14(2):54-62, May/June 1999.

[59] Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead E.J. Jr., Robbins, J.E., Nies, K.A., Oreizy, P. and Dubrow, D.L. “A Component- and Message-Based Architectural Style for GUI Software.” *IEEE Transactions on Software Engineering*, 22(6):390-406, June 1996.

[60] van Renesse, R., Birman, K. and Vogels, W. "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining." *ACM Transactions on Computer Systems*, 21(2):164-206, May 2003.

[61] Birman, K., van Renesse, R., and Vogels, W. “Navigating in the Storm: Using Astrolabe for Distributed Self-Configuration, Monitoring and Adaptation.” *Autonomic Computing Workshop, Fifth Annual International Workshop on Active Middleware Services,* June 2003.

[62] Candea, G., Kiciman, E., et. al. “JAGR: An Autonomous Self-Recovering Application Server.” *Autonomic Computing Workshop, Fifth Annual International Workshop on Active Middleware Services,* June 2003.

[63] OC Systems. *Aprobe: A New Approach for Testing Web Applications*. http://www.ocsystems.com/aprobe_web_testing.html

[64] OC Systems. *Improving Availability of Enterprise Applications with RootCause*. http://www.ocsystems.com/rootcause_white_paper.html.

[65] *System Management ARTS*. http://www.smarts.com.

[66] Konstantinou, A.V., Yemini, Y., and Florissi, D. “Towards Self-Configuring Networks.” *DARPA Active Networks Conference and Exposition (DANCE)*, May 2002.

[67] Konstantinou, A. and Yemini, Y. “Programming Systems for Autonomy.” *Autonomic Computing Workshop, Fifth Annual International Workshop on Active Middleware Services*, June 2003.

[68] Sterritt, R., Shapcott, C.M., Adamson, K., and Curran, E.P. “High Speed Network First-Stage Alarm Correlator.” *International Conference on Intelligent Systems and Control*, 2000.

[69] Steinder, M. and Sethi, A.S. “Probabilistic event-driven fault diagnosis through incremental hypothesis updating.” *IFIP/IEEE Symposium on Integrated Network Management*, 2003.

[70] Internet Security Systems. *RealSecure Network Protection*. http://www.iss.net/products_services/enterprise_protection/rsnetwork/.

[71] Cisco. *Cisco Intrusion Detection System*. http://www.cisco.com/univercd/cc/td/doc/pcat/nerg.htm.

[72] Lee, W., Stolfo, S.J. and Chan, P.K. “Learning Patterns from Unix Process Execution Traces for Intrusion Detection.” *AAAI-97 Workshop on AI Methods in Fraud and Risk Management*, 1997.

[73] Yemini, S.A., Kliger, S., Mozes, E., Yemini, Y. and Ohsie, D. “High speed and robust event correlation.” *IEEE Communications Magazine*, 34(5):82-90, May 1996.

[74] Luckham, D.C. and Vera, J. “An event-based architecture definition language.” *IEEE Transactions on Software Engineering*, 21(9):717-734, September 1995.

[75] Zhao, Y. and Strom, R. “Exploiting Event Stream Interpretation in Publish-Subscribe Systems.” *Principles of Distributed Computing*, 2001.

[76] Rose, M., ed. *RFC 1052: A Convention for Defining Traps for use with the SNMP*, 1991, http://www.ietf.org/rfc/rfc1215.txt.

[77] Kishimoto, H., Savva, A. and Snelling, D. “OGSA Fundamental Services: Requirements for Commercial GRID Systems.” *Global Grid Forum Draft*, October 3, 2002.

[78] Deen, G., Lehman, T., and Kaufman, J. “The Almaden OptimalGrid Project.” *Autonomic Computing Workshop, Fifth Annual International Workshop on Active Middleware Services,* June 2003.

[79] Agarwal, M., Bhat, V., et. al. “Automate: Enabling Autonomic Applications on the Grid.” *Autonomic Computing Workshop, Fifth Annual International Workshop on Active Middleware Services*, June 2003.

[80] Magee, J., Kramer, J. and Sloman, M. “Constructing Distributed Systems in Conic.” *IEEE Transactions on Software Engineering*, 15(6):663-675, June 1989

[81] Hofmeister, C.R, and Purtilo, J.M. “Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement.” *Thirteenth International Conference on Distributed Computing Systems*, May 1993.

[82] Kon, F., Campbell, R., Mickunas, M.D., Nahrstedt, K. and Ballesteros, F.J. “2K, A Distributed Operating System for Dynamic Heterogeneous Environments.” *Ninth IEEE International Symposium on High Performance Distributed Computing*, August 2000.

[83] Poellabauer, C., Schwan, K., et. al. “Service Morphing: Integrated System- and Application-Level Service Adaptation in Autonomic Systems.” *Autonomic Computing Workshop, Fifth Annual International Workshop on Active Middleware Services*, June 2003.

[84] Minsky, N.H. “On Conditions for Self-Healing in Distributed Software Systems.” *Autonomic Computing Workshop, Fifth Annual International Workshop on Active Middleware Services*, June 2003.

Appendix: Example Event Packager and Event Distiller Rulesets

A1. Event Packager

A1.1 Event Packager Rule Language

The full Event Packager documentation may be found at http://www.psl.cs.columbia.edu/xues/EventPackager.html.

A1.2 Event Packager Example

We give an example here of a configuration that subscribes to one event bus (Siena) for events which have the attribute-value pair (“TestAttribute”, “TestValue”), stores the results in a SQL database, and finally copies the results to another event bus. The references to the Console are needed if console-level control is desired of the Event Packager; it perceives the user console to be yet another source (and, potentially, a sink) for events.

```
<EventPackagerConfiguration>
  <Inputters>
    <Inputter Name="SienaInput1" Type="psl.xues.ep.input.SienaInput" SienaReceivePort="7890">
      <SienaFilter Name="TestFilter1">
        <SienaConstraint AttributeName="TestAttribute" Op="=" ValueType="String" Value="TestValue" />
      </SienaFilter>
    </Inputter>
    <Inputter Name="ConsoleInput1" Type="psl.xues.ep.input.ConsoleInput" />
  </Inputters>
  <Outputters>
    <Outputter Name="SienaOutput1" Type="psl.xues.ep.output.SienaOutput" SienaReceivePort="7891" />
    <Outputter Name="NullOutput1" Type="psl.xues.ep.output.NullOutput" />
  </Outputters>
  <Transforms>
    <Transform Name="Store1" Type="psl.xues.ep.transform.StoreTransform"  StoreName="HSQLDB1" />
  </Transforms>
  <Stores>
    <Store Name="HSQLDB1" Type="psl.xues.ep.store.JDBCStore" DBType="hsqldb"
            DBDriver="org.hsqldb.jdbcDriver" DBName="xues" DBTable="xues" Username="sa" Password="" />
  </Stores>
  <Rules>
    <Rule Name="TestRule1">
      <Inputs><Input Name="SienaInput1" /></Inputs>
      <Transforms><Transform Name="Store1" /></Transforms>
      <Outputs><Output Name="SienaOutput1" /></Outputs>
    </Rule>
    <Rule Name="ConsoleRule">
      <Inputs><Input Name="ConsoleInput1" /></Inputs>
      <Outputs><Output Name="NullOutput1" /></Outputs>
    </Rule>
  </Rules>
</EventPackagerConfiguration>
```

A2. Event Distiller

The full Event Distiller documentation may be found at http://www.psl.cs.columbia.edu/xues/EventDistiller.html.

A2.1 Event Distiller Rule Language

```
<rulebase xmlns="http://www.psl.cs.columbia.edu/2001/01/DistillerRule.xsd">
```

This line is the top-level XML rulebase declaration, and is required.

```
<rule name="rule name">
```

This is the top-level rule declarator, is declared within ruleBase, and is repeated at the beginning of each rule. The following parameters may be supplied:

- name (required): Rule names are used primarily for disambiguation at this time; they are not referred elsewhere.
- position (optional): the position where this rule is inserted, starting with 0. The position specifies the priority of this rule in receiving events: higher priority rules (smaller numbers) receive events first. This is useful in the case of event absorption on the part of a state (see the absorb attribute in state).
- instantiation (optional): the criterion for instantiating this rule. Legal values are: 0, 1, 2. '0' means the rule will only be instantiated once. '1' means there will always be only one instance at any given time, so a new instance is created as the previous succeeds or one times out. '2' means a new instance is created whenever a previous instance starts (i.e. receives its first event), so that there will always be one instance listening for the starting event(s).

```
<states>
```

This declarator, contained in a rule, signifies the beginning of state declarations (of which there may be many). Note: There can be only one "states" declaration in a rule.

```
<state name="state name" timebound="milliseconds" children="CSV-list
of children names" actions="CSV-list of actions" fail_actions="CSV-
list of actions" absorb="boolean value" count="integer value">
```

This is the beginning of declarator for a single state in the state-list (pattern) to be matched for this rule. A number of parameters are supplied alongside the rule declarator:

- name (required): specifies the state name. This name should not conflict with other state names within this rule.
- timebound (required): specifies the timebound in which this event can happen relative to the previous event. For the first event, the timebound is generally set to -1 (e.g. no time limit); for subsequent timebounds it is recommended, although not required, that the timebound be positive. If the first state of a rule has positive timebound, it is measured against the time at which the rule is created.(Note that timebound is not precise - a fudge factor is present in the Event Distiller to take care of race conditions.)
- children (optional): specifies (in a comma-delimited list, no spaces) states that can follow this particular state.
- actions (optional): specifies the notification(s) to be sent out when this state is matched. Usually, this is intended for the last state in a rule, which would signify as the "rule matching", but intermediate notifications can be sent out.
- fail_actions (optional): specifies the notification(s) to be sent out if this state times out while waiting for input. Unlike actions, this is intended for each state, so a different notification may be sent out for a failure at any given point in the state machine. Note that if a rule allows multiple paths between states (so that at one given time multiple states are subscribed), failure notifications for one of the states that timed out will be sent, only if all currently subscribed states time out.
- absorb (optional): Whether this state will absorb the events that match it. If this is set to true, when an event matches this state, the event will not be passes on to any other

state currently subscribed. If this field is not specified, a default value of 'false' will be used.

- count (optional): the number of times this event will need to be matched before it passes. A default value of '1' is used if this field is not specified. If the value specified is greater than '1' (say, n, children and actions will only apply to the nth time that the state is matched; while fail_actions, if specified, will apply at all times. The special value '-1' indicates that the event may occur any number of times (within the specified timebound), until one of the children is matched, thus terminating the loop.

<attribute name="attribute name" value="attribute value" op="comparison operator" type="value type" />
Declarator for an attribute-value pair required in this particular state. Note that there may be many attributes per state, and they are ANDed together in the filter that matches incoming notifications to this state. Since there are no embedded tags within this tag, you can use the compact end-tag notation (e.g. "/>"). The comparison operator is a string representation of the operator to be used for matching the value. For instance, you would specify op=">" to match all values greater than the one that the specified value. The default operator is the equality operator. Type needs to be specified since some operators only make sense for certain types; the default type is the string type. Note that the value field here is matched using a string-equals comparison, unless a special wildcard-binding notation is used (see below).

</state>

</states>

<actions>
This declarator, contained in a rule, signifies the beginning of action declarations (of which there may be many). Note: There can be only one "actions" declaration in a rule.

<notification name="notification name">
This is the beginning of the notification (action) declarator. There is one parameter, name, which corresponds to the actions and fail_actions references above. It is strongly recommended that this name be one contiguous phrase without whitespace or punctuation to avoid conflicts in the CSV-lists referenced above.

<attribute name="attribute name" value="attribute value" />
Declarator for an attribute-value pair to be included in this particular notification. Note that there may be many attributes per notification. Since there are no embedded tags within this tag, you can use the compact end-tag notation (e.g. "/>"). If you use wildcard-binding above, you may use the same wildcard-binding tag here - it will be substituted with the actual bound value (again, see below).

</notification>

</actions>

</rule>

</rulebase>

A2.2 Event Distiller Example

We provide here an extended view of the rules presented in Figure 6, including the corresponding notifications if the pattern is matched.

```xml
<rulebase xmlns="http://www.psl.cs.columbia.edu/2001/01/DistillerRule.xsd">

<rule name="ActiveEvent">
  <states>
    <state name="Start" timebound="-1" children="End" actions="" fail_actions="">
     <attribute name="Service" value="*service"/>
     <attribute name="Status" value="Started"/>
     <attribute name="ipAddr" value="*ipaddr"/>
     <attribute name="ipPort" value="*ipport"/>
     <attribute name="time" value="*time"/>
    </state>
    <state name="End" timebound="15000" children="" actions="Debug" fail_actions="Crash">
     <attribute name="Service" value="*service"/>
     <attribute name="State" value="FINISHED_STATE"/>
     <attribute name="ipAddr" value="*ipaddr"/>
     <attribute name="ipPort" value="*ipport"/>
     <attribute name="time" value="*time2"/>
    </state>
  </states>
  <actions>
    <notification name="Crash">
      <attribute name="Notification_Type" value="GW_Alarm"/>
      <attribute name="Message" value="Dead_Service"/>
      <attribute name="KX_Reaction_Type" value="Workflow"/>
      <attribute name="KX_Reaction_Spec" value="Disable_Service"/>
      <attribute name="Timestamp" value="*time"/>
      <attribute name="Service" value="*service"/>
      <attribute name="Name" value="gwHostAdapter"/>
      <attribute name="IPaddress" value="*ipaddr"/>
      <attribute name="port" value="*ipport"/>
      <attribute name="serviceURI" value="http://www.isi.edu/..."/>
      <attribute name="schemaURI" value="http://www.isi.edu/..."/>
    </notification>
    <notification name="Debug">
      <attribute name="GWFinish" value="Yes"/>
      <attribute name="Timestamp" value="*time2"/>
    </notification>
  </actions>
</rule>

</rulebase>
```

In this case, we allocated 15 seconds for the method to complete, and in the case of a crash, both static and dynamic (i.e., wildcard-bound) data were reported (note that we abbreviated a few URLs for readability).