

# The Living Systems<sup>®</sup>Technology Suite: An Autonomous Middleware for Autonomic Computing

Giovanni Rimassa

Dominic Greenwood

Martin E. Kernland

Whitestein Technologies AG  
Pestalozzistrasse, 24  
CH-8032 Zürich, Switzerland  
{gri, mek, dgr}@whitestein.com

## Abstract

*This paper presents the Living Systems<sup>®</sup>Technology Suite, LS/TS, a middleware based on autonomous software agents and autonomic computing principles. Specifically, the paper describes the autonomic principles built into LS/TS, and the features that follow from them. These features are described within the context of a general taxonomy of autonomic systems, elaborating on the relationship between middleware and application levels. Lastly, the paper addresses the problem of system self-representation, leveraging LS/TS support for ontology and semantic agent communication.*

## 1. Introduction

The tenets of Autonomic Computing are drawn from the phenomenon of homeostatic control that has evolved over millennia to maintain system equilibrium in biological organisms. The processes that manage this operate on the basis of feedback mechanisms which compensate for environmental fluctuations to regulate equilibrium within, or even between, organisms. Application of this metaphor leads to systems engineering approach that produces self-managing systems reliant on feedback loops and reviewing the *self*-\* qualities identified in the Autonomic Computing Manifesto<sup>1</sup> it is clear that each intrinsically employs feedback mechanisms, sometimes only for observation, other times also for control.

Accepted for the International Conference on Autonomic and Autonomous Systems ( ICAS'06), July 19-21, 2006, Silicon Valley, USA

<sup>1</sup>These qualities are sometimes called *self-CHOP* qualities, from their names: self-configuration, self-healing, self-optimization, self-protection. See also <http://www.research.ibm.com/autonomic/manifesto/>

Likewise, in the seminal paper [6], a conceptual architecture is sketched for autonomic systems that posits feedback loops both at the component and at the system level.

In this paper we identify a set of relationships between middleware and the principles of Autonomic Computing, grounded by a description of our autonomous-agent middleware. We discuss some of the specific features and tools provided which constitute a concrete means of creating autonomic applications, with a particular focus on the semantic tiers offered for effecting system self-description through support for ontologies and semantically-grounded agent communication.

Section 2 introduces the Living Systems<sup>®</sup>Technology Suite (LS/TS) middleware for creating autonomous and autonomic systems. Section 3 then identifies a taxonomy relating autonomic computing and middleware in terms of the dependencies required to engineer *self*-\* features into middleware of the kind that can be created with LS/TS. Section 4 discusses how LS/TS semantic communication and description can be used to support *self*-\* in autonomic applications. Finally, Section 5 presents the conclusions.

## 2. The Living Systems<sup>®</sup>Technology Suite

The *Living Systems<sup>®</sup>Technology Suite (LS/TS)* is an industry-grade foundation for the development, deployment and operation of software systems exhibiting the qualities defined by the goals of Autonomic Computing. LS/TS applies the concepts and techniques of software agent technology within an open infrastructure and process.

As the base for LS/TS, the Java programming language was chosen due to several reasons such as the wide acceptance and use of the language as well as its portability. Specific abstractions are implemented using standard language capabilities, thus LS/TS applications are pure Java applications. LS/TS provides a Java library called *Core Agent Layer*, CAL, with implementations of these new abstrac-

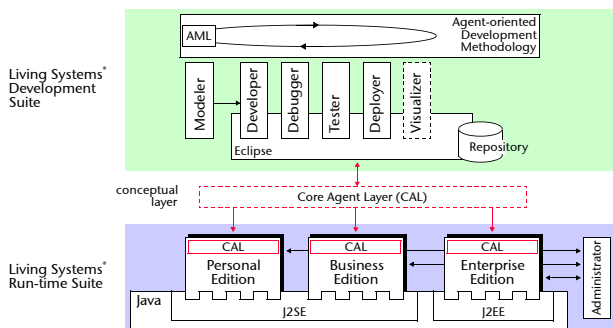
tions.

LS/TS provides a comprehensive development tool suite, the mentioned CAL library, and a sound methodology. The development suite is based on the *Eclipse* Java IDE. This allows agent technology relevant abstractions to be managed on the tool level by providing a new kind of view on the agent application. This approach enables developers to directly handle these new abstractions, while keeping the compiled Java application in line with the mainstream and open development tool suite for the Java platform.

Methodological support consists of the UML-based *Agent Modeling Language*, AML, and a comprehensive software development process called the *Agent-oriented Development Methodology*, ADEM. AML [8] is a semi-formal visual modeling language for specifying, modeling and documenting systems that incorporate concepts drawn from multi-agent systems theory<sup>2</sup>.

## 2.1. Concepts

LS/TS architecture consists of two separate parts according to when and by whom each component is used. On the one hand, the *Development Suite* comprises the tool chain used by software developers at system construction time. The *Run-Time Suite*, instead, is defined by all the LS/TS pieces working as an integrated whole to aid application users and administrators at system operation time. LS/TS is available in different editions: Personal and Business editions are based on J2SE, and the Enterprise Edition on J2EE.



**Figure 1. The overall structure of LS/TS**

Figure 1 shows that there is a *conceptual layer* that belongs neither to the development nor to the runtime domains. This is the *Core Agent Layer* which is open, extensible and independent of the LS/TS Edition in use.

The CAL contains two major component types, each providing a specific set of functionality.

1. The *Autonomous Agent* represents a control element of the application logic, able to sense its surroundings, act on them and also to coordinate with other agents.
2. The *Servant* offers a view on application services and reactive functionality. They are passive computational elements of the overall multi-agent system.

The Autonomous Agent is the locus of the autonomic management capabilities that distinguish an autonomic element, as defined in e.g., [6]. LS/TS Autonomous Agents have a composite structure, fostering separation of concerns and allowing different levels of functionality in different deployment environments. They have three parts:

- The *Agent Meta Information*.
- The *Agent Data Storage*.
- The *Agent Execution Engine*.

The Agent Meta Information holds the information necessary to maintain the properties of the Autonomous Agent programming abstraction. Such information includes, for example, the agent identity. The Agent Data Storage keeps the long term agent state, and can leverage persistent storage facilities when available.

The Agent Execution Engine contains all application relevant information and functionality. This includes the agent capabilities (i.e. what the agent can do) and the structure of the complex tasks it can perform (composite actions and perceptions, interaction protocols, and the like).

The *Message Dispatching Agent Logic (MDAL)* is the default execution engine provided by LS/TS, though developers can write their own Execution Engine (agents with different Execution Engines can still work together in a single application). The MDAL is internally based on *Petri Nets*: additional abstraction is added by modeling the task structure with a *Process Algebra* specification [1]. The Petri Net structure is then compiled from the algebraic expression representing the task [2].

Such an algebraic specification provides introspection into agent tasks, which is an essential enabler of self-knowledge: an agent can now reason on its own tasks without executing them. As an example, it is possible to attach application-specific attributes to algebraic sub-expressions and calculate execution costs or utility gain for a given task.

LS/TS also includes a *goal-oriented* execution engine, adopting a logic-based approach consistent with the *Belief-Desire-Intention (BDI)* notion [7]: information gathered from the world is represented as a set of predicates in logic (the agent *beliefs*), and the desired states are also expressed as predicates (the *goals* of the agent). An agent has also a

<sup>2</sup>For more detailed information on AML please refer to: <http://www.whitestein.com/pages/solutions/meth.html>

collection of *plans*, which have an executable *body* associated with logical predicates expressing when it is applicable, and what is the consequence of its successful completion.

The body of a plan is an MDAL task, along with its process-algebraic specification and Petri Net realization. The same activity can be used in a goal-oriented agent or in an agent with a more imperative control strategy. Moreover, LS/TS goal-oriented agents can profit by the symbolic introspection afforded by Process Algebra expressions.

The system-level properties of Autonomic Computing do not just arise due to the structure of each autonomic element considered in isolation. Rather, they also emerge from the interaction among the different components. In LS/TS, different Autonomous Agent instances can exchange messages among them, and a complete framework is provided, that enables *semantic communication* at the global level.

LS/TS defines a layered model for messages that partitions them according to their communicative function (e.g., whether they are requests, notifications, queries, and the like) according to the approach of an *Agent Communication Language* [4], [5] (*ACL* for short).

Moreover, LS/TS leverages the W3C *OWL*<sup>3</sup> Web Ontology Language to realize an expressive model of the application-specific knowledge that agents exchange. This higher level description of what the system is about enables deep self-awareness and cleanly separates the base system features from the ones that pertain to enforcing and granting autonomic qualities.

## 2.2. Development Suite

The development suite, as shown in Figure 1, currently consists of five tools, four of them are based on the Eclipse platform. The *Modeler* is based on Enterprise Architect, a UML modeling tool from Sparx Systems<sup>4</sup>. This tool uses the official AML Profile for UML 2.0 and a plugin library to allow direct AML modeling and code generation for LS/TS within the Enterprise Architect tool.

The *Developer* is the main design and implementation tool for the application programmer. It is an extension to the standard Eclipse Java programming environment, providing specific navigators, views, and manipulators of the Java code for the LS/TS agents. The Developer can import AML models from the Modeler and automatically create the appropriate code structure.

The *Debugger* is an extension to the standard Eclipse Java debugger, with direct support of the agent abstractions of the CAL and the logic of the agents. As an example, it is possible to add a breakpoint to an agent that is only triggered if a specific message sent from an agent arrives, or if

an agent reads or writes a specific value from its *Agent Data Storage*.

The *Tester* supports unit testing of single agents and their logics. It consists of helper classes and mock objects that ease the creation of unit tests for LS/TS specific agent logic.

The *Deployer* packages an application with the necessary third-party libraries, meta-information, and run-time environment specific information.

The *Administrator* is used to visualize and control application execution, as shown in Figure 2. It is possible to see a list of the messages throughout the platform, both before they are processed and after they have been delivered. An *Inspector* provides a view of agent state, life-cycle management of an agent (e.g. stopping an agent), etc. Both these tools offer a wide range of monitor and management features that greatly support both the debugging and the operation of LS/TS based applications.

## 2.3. Run-time Suite

LS/TS comprises three different run-time environments, the *Personal Edition*, the *Business Edition*, and the *Enterprise Edition*. These editions cover a wide range of application deployment needs. It is possible to execute agents in a persistent mode so that agent state and message queue updates are transactional. The Enterprise Edition can also run in a clustered mode to achieve application-transparent failover. Alternatively, both Business and Enterprise Edition allow *LS/TS Federation*, i.e., a set of loosely coupled LS/TS run-time nodes communicating with each other.

The Business and Enterprise Editions of LS/TS are equipped with resource management capabilities. The Enterprise Edition profits from the underlying application server whereas the Business run-time has its own resource management service. Because of this, a virtually unlimited number of agents and messages can be sustained by the run-time environment of both editions.

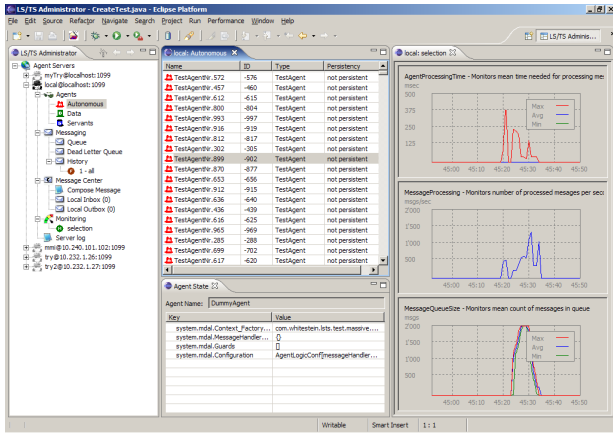
This resource usage monitor and control is made available through the *Administrator* that allows managing a running instance of LS/TS as shown in Figure 2.

The Administrator supports the operation of the run-time, which is valuable during debugging, but also important for monitoring a deployment. The monitoring allows the gathering of data from the run-time which concerns the application and the run-time itself (e.g. average processing time of each agent per incoming message, number of messages in the message queue, heap memory used, etc.). A special service collects this information during run-time and the individual statistics can be turned on and off at run-time.

LS/TS defines a suitable API through which the system can be monitored and configured. It is thus possible to have dedicated agents that leverage the administration API

<sup>3</sup>See <http://www.w3.org/2004/OWL/>

<sup>4</sup>See <http://www.sparxsystems.com.au/>



**Figure 2. Monitoring resource usage and inspecting agents with the Administrator**

to build self-configuration and self-healing (by relying on LS/TS clustering and federation capabilities) into the middleware or the LS/TS-based application.

### 3. Middleware Approach to Autonomic Computing

The purpose of middleware is to gather a set of application-independent features and subsystems and to reuse it across several applications. The resulting benefits range from improved system dependability due to stronger foundations up to shortened time-to-market due to reuse of generic subsystems.

In order to reap the above benefits, and many others, middleware-based systems need to adopt a *Layered* architectural pattern [3], where the middleware provides services to the application, and does not depend from it at all.

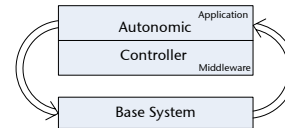
This layered approach seems at odds with the holistic view of Autonomic Computing: a layered structure has no dependency cycles, whereas the very concept of *self* is but a cycle between subject and object. It is then not obvious how to match the widespread presence of feedback loops needed by *self*-\* properties and the layered application/middleware separation.

A first case of interest is the design of an *autonomic compensator* for an existing *base system*: this is analogous to the problem of synthesizing a controller for a known system. If the autonomic compensation is built using a middleware such as LS/TS, the general schema of the overall system (made by the coupling of the base system and the compensator) is as shown in Figure 3.

In this situation, the feedback loop that takes data from the base system and operates control actions on it, only

involves the application level of the autonomic controller. Therefore, here the middleware/application layering and the autonomic holism are completely independent.

Though this first case is the simplest of the three, it already covers several interesting scenarios, and it's the blueprint followed by the first Autonomic Computing prototypes, such as driving the reconfiguration of networks and information systems with log file analysis.

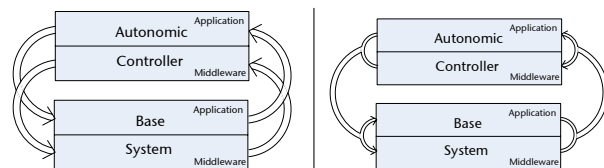


**Figure 3. Autonomic compensation of an existing system**

A second case considers the design of a new system, which is supposed to exhibit at least some of the characteristics of Autonomic Computing. Such a system will have both some components devoted to base functionality and some others that realize the feedback loop. In principle, then, this system can still be considered as the union of a *base system* and an *autonomic controller*.

Adopting a middleware results in both the base system and the autonomic controller being partitioned into middleware and application layers. Here the simplifying assumption is made, however, that the base system state representation is itself layered.

This means that there is a middleware-level, application independent part of the system state (e.g., the current memory consumption or network utilization belong there), and an application-specific part. Moreover, the control strategies themselves are separated so that the strategy controlling the middleware-level state only uses middleware-level system state values.



**Figure 4. Autonomic systems and middleware. Strict separation (left) and full coupling (right)**

In this case, the resulting overall system has two separate feedback loops at middleware and application level, as the left part of Figure 4 shows. This situation occurs whenever autonomic behaviour is achieved by the application without the need of it being aware of infrastructural data: most

often this happens due to an assumed relationship between entities in the infrastructure and others in the application.

Finally, a third relevant case is a more general one, and also more challenging. While it is still true that the system can be seen as an autonomic controller acting as the feedback chain of a base system, and both have an application and a middleware layer, it is no longer the case that the two are independent.

The whole base system state, regardless of the layer it belongs to, is fed back to the autonomic controller. Likewise, both layers of the controller effect a control strategy that manipulates both the middleware and the application layer of the base system.

This more general kind of architecture becomes necessary whenever a fixed (i.e., open-loop) relationship between the infrastructure-level and application-level parts of the system state cannot be modeled a priori, but still the only effective control strategies are application dependent.

The right part of Figure 4 shows the principled schema of this third case. With respect to the previous situation depicted in the left part of Figure 4, now the application layer of the autonomic controller needs to access data that belong to the middleware layer of the base system, and possibly to modify it while enacting its control strategy.

#### 4. Enabling Self-Describing Autonomic Systems with LS/TS

The three use cases described in Section 3 have shown how the middleware/application divide relates to Autonomic Computing. This can be used to gather a requirement list for middleware infrastructure so that it can be an effective support to build autonomic systems.

Considering the first case, since the middleware layering and the autonomic feedback loops are independent, the requirements for an effective middleware infrastructure do not change with respect to other kinds of systems. In this case the middleware adds value by increasing quality and productivity of the application development.

Considering now the second case, the simplifying assumption is made of a complete separation between middleware and application layers in both the base system and its autonomic controller. Moreover, it is argued that the middleware-level feedback loops would be provided *out of the box* by the middleware vendor.

Finally, while discussing the third case, which is the most general and challenging, it was suggested that a middleware infrastructure should monitor application execution, and should in turn provide applications with the means to query, monitor and control middleware-relevant state.

LS/TS provides a wide array of concepts and tools to support the development of autonomic applications in all

the cases mentioned above. The component model of autonomous systems helps already in the simplest case of compensating a given base system. When targeting the more complex scenarios outlined in Figure 4, the resource management and administration API form the basis to build the needed feedback loops.

A feature, which is particularly relevant to the above discussion, is LS/TS support for *Semantic Communication*. Semantic Communication provides a layered and structured communication model to autonomous agents. Each further layer is called a *tier* and adds a set of constraints that are applied to the agent interaction. These constraints specify the boundary of the interaction space within which all subsequent layers have to operate. This incremental constraining is meant to support various level of interoperability between agents; the more layers that two agents share, the deeper they can interoperate.

The *Linguistic Tier* defines a general taxonomy of messages, that can be applied in any application domain and classifies messages by considering them in isolation, according to the approach of Agent Communication Languages [4], [5]. The *Domain Tier* defines the conditions and constraints applying to the schemata for message content. The actual schemata are application specific and form an *Ontology*, but they follow a set of common rules with respect to how they are defined. The OWL language is used to define ontologies, as mentioned in Section 2.1.

The first case considered in Section 3, and depicted in Figure 3, deals with introducing autonomic properties into an existing system. A major milestone for such a task is to produce a model of how the base system operates (this is the very core of engineering the *self-link*).

An ontology would be the live realization of such a model, being both precisely structured and available at runtime. Within the approach of Agent Technology, the ontology would also serve as the schema for the message content agents exchange. The result would be an agent-based autonomic controller, where agents *converse about* the system in order to effect a cooperative control policy.

From a more concrete, software development perspective, the LS/TS API for Semantic Communication allows the connection of multiple OWL ontologies to a pre-existing object model (which does not need to be modified and stays completely independent from LS/TS classes). This results in a clean separation between *what the application is and does* and *how the agents converse about it*: different groups of agents can use different ontologies to describe the base system, according to their specific roles and goals in controlling it.

The second and third situations discussed in Section 3 deal with using middleware both in the base system and in the autonomic controller, and have been labeled *strict separation* and *full coupling* in Figure 4.

In the strict separation case, it is the middleware responsibility to provide a complete feedback loop: this means that it has to be possible to create middleware-level components that monitor and control the middleware itself. This, as in the previous case, requires those self-description capabilities an ontology can provide.

LS/TS internal architecture is modularized in a series of different *runtime services* (messaging, persistence, etc.), and the administration API mentioned in Section 2.3 can be used to build feedback loops in LS/TS runtime services. Here again Semantic Communication and ontologies are paramount: LS/TS service architecture and management API can be described in OWL and agents can be defined, that realize a sense/reason/act control loop at the middleware level. The ontology acts as an application-agnostic *system model* and can be shipped as part of LS/TS, along with the controlling agents.

The control loop at the application level is instead to be obtained in the same way discussed before: the base system (now just its application layer) model is captured by one or more application-specific OWL ontologies, and suitable messaging protocols among agents are created, whose message content is ruled by the ontology. The same software engineering benefits (separation between application model and agent view, runtime availability of the ontology definitions) can then be enjoyed as in the previous case.

Lastly, the full coupling case requires all the support of the previous two cases, and even has additional needs. The two ontologies of the strict separation case were independently defined, but in a full coupling scenario this is not possible anymore. The requirement for a middleware-level feedback loop that actually uses application-level perceptions and actions is that application operation is described by ontological entities, which are then manipulated by middleware-level agents.

Here the issue is that such middleware-level agents are to be shipped with LS/TS before any application is written, so they cannot depend on application-specific ontologies. The solution for this is to extend the system ontology to include an *ontological application framework*. Similarly to software application frameworks, who define an abstract layer that is made concrete by the application code, an ontological application framework will provide a set of abstract or meta-level concepts and relations (defined in OWL) that application ontologies will have to depend upon.

The middleware-level feedback components (mostly agents) can leverage the ontological application framework to, e.g., define resource control policies. An application whose ontology completes the framework will submit itself to those policies simply due to the subsumption link between its description and the general framework.

The self-description depth and structure, as shown, can vary and depends in general from the specific configura-

tion of the relationship between *self-\** feedback loops and middleware/application partitioning. The common trait is however that some self-describing facet is always needed to engineer Autonomic Computing properties in systems. A structured and flexible meta-level such as the one provided by LS/TS with Semantic Communication and OWL ontologies brings effective support to this task.

## 5. Conclusions and Future Work

This paper has introduced the *Living Systems<sup>®</sup>Technology Suite* as a means of engineering agent-based software systems that intrinsically exhibit the *self-\** principles of autonomic computing. We have described the most fundamental features of LS/TS and a selection of the tools provided.

To validate this position we argue that the dependency relationship between middleware and autonomic elements can take three primary forms, each classified and described each in terms of how LS/TS treats them. A particular conclusion is that there are a variety of means by which feedback mechanisms can be used to engineer autonomic middleware, each of which can be effectively realized by LS/TS especially when employing the ontological self-description and meaningful message classification and content features provided by the semantic communication tiers.

We are now developing a series of autonomic applications that demonstrate our arguments.

## References

- [1] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, 2001.
- [2] E. Best and M. Koutny. Process algebra: A petri-net-oriented tutorial. In *Lectures on Concurrency and Petri Nets*, pages 180–209, 2003.
- [3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
- [4] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML - a language and protocol for knowledge and information exchange. Technical Report CS-94-02, Univ. of Maryland and Valley Forge Engineering Center and Unisys Corp., 1994.
- [5] FIPA. Fipa communicative act library specification, 2001.
- [6] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
- [7] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.
- [8] I. Trencansky and R. Cervenka. Agent modelling language (AML): a comprehensive approach to modelling mas. *Informatika*, 8(5), 2002.