Dynamic Creation of Monitoring Infrastructures

Howard Foster and George Spanoudakis

1 Introduction

As a key part of monitoring and management, systems developed with a Service-Oriented Architecture (SOA) design pattern should utilise negotiated agreements between service providers and requesters. Typically, the results of these negotiations are specified in Service Level Agreements (SLAs), which are then used to monitor key levels of service provided, and to optionally specify preconditions and actions in case these levels are violated. Responsibility for monitoring SLAs (and often individual parts within them) must be dynamically allocated to different monitoring components, since SLAs — and the components available for monitoring them — may change during the operation of a service-based system [4]. The complexity of SLA terms, however, often means that several monitoring components may need to be selected for a single SLA-guaranteed term expression (e.g. availability $> 90\%$), since each part of the expression may be reasoned by a physically different provider. Existing work has shown examples of decomposition based upon simple decomposition of expressions [4], but there is also a need to consider variations between different monitors (e.g. trustworthiness or access constraints) in a dynamic monitoring configuration process.

In this chapter, we show how complex SLA terms specified in the SLA@SOI SLA (Chapter 'The SLA Model') can be decomposed into manageable monitoring configurations, and include a mechanism to support the selection of preferred monitoring components. Advanced configuration is supported by a *MonitoringManager* component which mechanically parses an SLA, generates a formal Abstract Syntax Tree (AST) and decomposes the terms of the AST into expressions for monitoring. Each expression is then used to select appropriate reasoning or service sensor monitoring components. The main contribution of this work is that both the monitoring

Howard Foster, George Spanoudakis,
Department of Computing, City University London, Northampton Square, EC1V 0HB, London.
e-mail: {howard.foster.1,g.e.spanoudakis}@city.ac.uk

P. Wieder et al. (eds.), *Service Level Agreements for Cloud Computing*,
DOI 10.1007/978-1-4614-1614-2_8,

configurator and the monitoring configuration specification are generic, reusable artifacts able to be incorporated into other frameworks where configuration of monitoring components is required. The monitoring configurator is already offered as a reusable service that uses standard web-service protocols to enable the use of replaceable selection criteria for candidate monitors; selection criteria can be driven from preferences for monitor provider and/or offered features.

The chapter is structured as follows: Section 2 illustrates the service monitoring architecture and components, whilst Section 3 describes the overall approach to monitoring configuration. In Section 4 we describe the parsing and decomposition of SLAs, and in Section 5, the monitoring component selection algorithms. In Section 6 we discuss implementation and testing of the approach, and in Section 7 we briefly discuss related work. Section 8 concludes the chapter with a discussion of present and future work.

2 Architecture

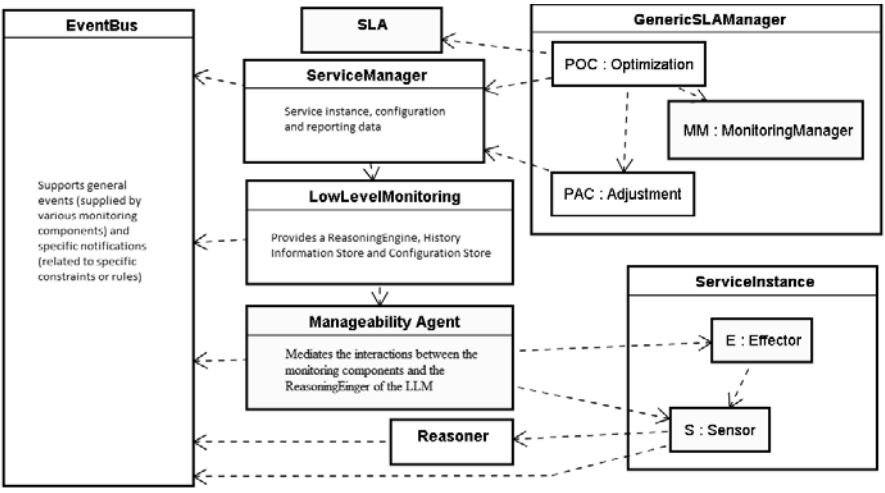

Fig. 1 A Service Monitoring Architecture

An overview architecture for service monitoring in the SLA@SOA project is illustrated in Figure 1. In this chapter, we focus on the GenericSLAManager (providing generic support for planning, optimisation, adjustment and configuration of monitoring) and monitoring component features (such as reasoners, sensors and effectors). The Planning and Optimization Component (POC) is a local executive controller for a service manager. It is responsible for assessing and customising SLA offers, evaluating available service implementations, and planning optimal service provisioning and monitoring strategies. The POC generates a suitable execution plan

for monitoring (based upon a configuration obtained from the MonitoringManager component) and passes this to the Provisioning and Adjustment Component (PAC). The PAC collects information from the Low Level Monitoring System, analyses the incoming events, decides if a problem has occurred or is about to occur, identifies the root cause, and then (if possible) decides on and triggers the best corrective or proactive action. If the problem cannot be solved at a local level, the PAC escalates the issue to a higher level component, namely the POC. In the case of an SLA violation, such an adjustment can trigger re-planning and reconfiguration, and/or alert higher-level SLA monitoring. These capabilities are considered important in order to assure preservation of service provision and resource quality.

The MonitoringManager (MM) coordinates the generation of a monitoring configuration for the system. The MM uses configurable selection criteria to determine which is the most appropriate monitoring configuration for each SLA specification instance it receives. Each monitoring configuration describes which components to configure and how their configurations can be used to best monitor guaranteed states. The Low Level Monitoring Manager is a central entity for storing and processing monitoring data. It collects raw observations, processes them, computes derived metrics, evaluates the rules, stores the history, and offers all this data to other components (accessible via the service manager). It also implements the monitoring part of a ProvisioningRequest, containing constraint-based rules (time- and data-driven evaluations) and ServiceInstance-specific sensor-related configurations. It is general by design, and thus capable of monitoring software services, infrastructure services and other resources. Since POC and PAC functionality is very closely related to domain-specific requirements, they are provided as extendible components. For SLA@SOI case studies, they are already extended for either software service monitoring or infrastructure service monitoring. The MM aims to be generic for all solutions and is provided as one solution.

There are three types of monitoring feature in the monitoring system: First, *sensors*, which collect information from a service instance. Their designs and implementations are domain-specific. Sensors can be injected into the service instance (e.g., service instrumentation), or can be outside the service instance (e.g. intercepting service operation invocations). A sensor can send the collected information to a communication infrastructure (e.g. an Event Bus), or other components can request (query) information from it. There can be many types of sensors, depending on the type of information they are designed to collect, but they all implement a common sensor interface. The interface provides methods for starting, stopping, and configuring a sensor. The second type of monitoring feature is an *effector*. Effectors are components for configuring service instance behaviour. Their designs and implementations are also domain-specific. Like sensors, effectors can be injected into a service instance or can interface with a service configuration. There can be many types of effectors, depending on the service instance to be controlled, but they all implement a common effector interface. The interface provides methods for configuring a service. The third type of monitoring feature is a *reasoner* (also known as a Reasoning Engine), which performs a computation based upon a series of inputs provided by events or messages sent from sensors or effectors. An example rea-

soner may provide a function to *compute the average completion time* of a service request. In this case, it accepts events from sensors detecting both requests for and responses to a service operation, and computes an average over a period of time. Reasoners also provide access to generic runtime monitoring frameworks, such as EVEREST [15].

2.1 Monitoring Features Specification

In addition to an SLA specification (Chapter 'The SLA Model'), the Monitoring-Manager requires a set of feature specifications for monitoring feature types (introduced at the beginning of this section). Component monitoring features are specified for a type of monitoring component and offered for a type of service (Chapter 'The Service Construction Meta-Model' for details of the Service Construction Meta-Model). A feature specification has two instance variables: The *type* variable holds the type of the component, and the permitted types are sensor, effector and reasoner. A sensor provides information about a service, an effector changes the properties of a service, and a reasoner processes information to produce a monitoring result (for example, it consumes information provided by sensors and reports whether or not an SLA is violated). The second instance variable is the *UUID* variable, which uniquely identifies the component with the monitoring features. This variable has the same value as the service UUID. Furthermore, each component feature contains a list of monitoring features. The example in Figure 2 illustrates the component features of an example service. In this example, the sensor component has two monitoring features: one for events reporting *cpu-load*, and another for reporting the number of *logged-users*. The example also illustrates a reasoner component with two monitoring features: one providing a *greater-than* comparison of two input parameter numbers, and the other providing an *MTTR* (Mean Time To Repair) computation output based upon request and response input events.

Basic monitoring features are used to distinguish between 'event' and 'primitive' monitoring features. There is a single parameter *type* for the type of basic monitoring feature. In the case of primitive monitoring features, allowed types correspond to the Java primitive types (e.g., Long, Boolean, String, etc). In the case of an event monitoring feature, allowed types are currently request, response and computation (as a result of a function). A basic monitoring feature with a sub-type of *primitive* is used to advertise an ability to report about primitive service information (e.g., cpu_load, logged_users, available_memory, etc). Sensors are the typical components with this kind of feature. A primitive feature has two instance variables: First, a *type* holds the variable type. This can be, for instance, one of the Java standard primitive types. It can also be any other type defined in an SLA vocabulary. The second instance variable is a *unit*, which holds the monitoring feature unit of measurement (e.g., mt, km, kg, etc.). *Event* monitoring features are used to advertise an ability to report about service interactions or service states (e.g., service operation requests and responses, service failures, etc.). Sensors and reasoners are the typical compo-

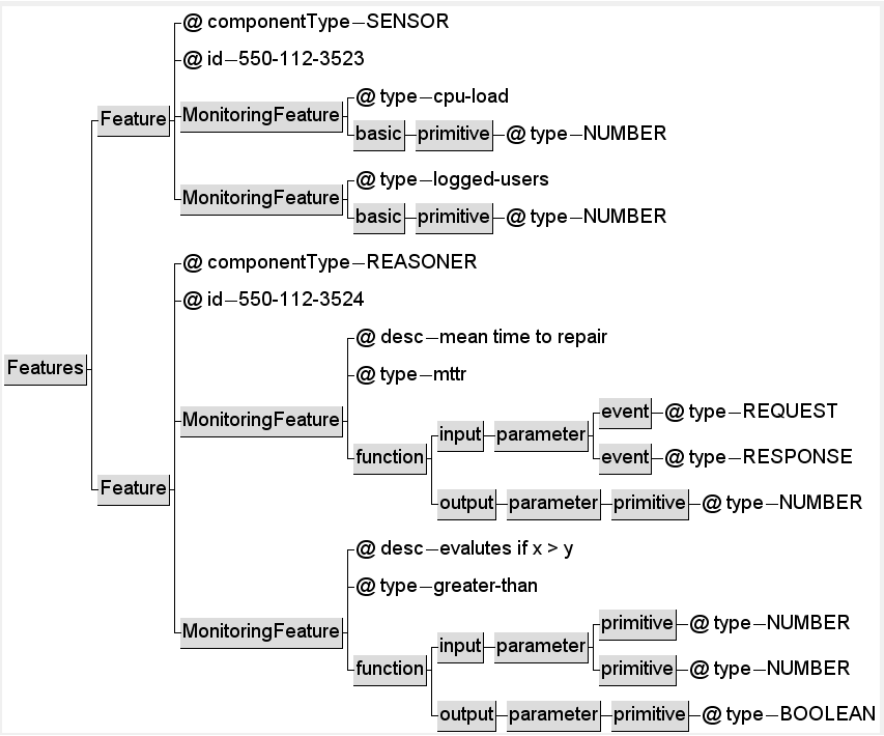

Fig. 2 Component Monitoring Features as XML Elements

nents with this kind of feature. An event basic sub-type has one instance variable: a *type*. This *type* holds the event type as either request, response or computation. Domain-specific event types can also be defined and used here.

Function monitoring features are used to advertise an ability to perform a computation and report its result (e.g., availability, throughput, response_time). Reasoners are the typical components with this kind of feature. The class *Function* has two instance variables: the first is *input*, which holds the list of function input parameters. The second is *output*, which holds the output parameters. Reasoner features are described by a type (the term or operator performed), one or more input parameters, and one output parameter.

3 Approach to Configuration

Given an SLA specification and a set of component monitoring features, our approach to dynamic configuration of monitoring infrastructures is based on the process illustrated in Figure 3.

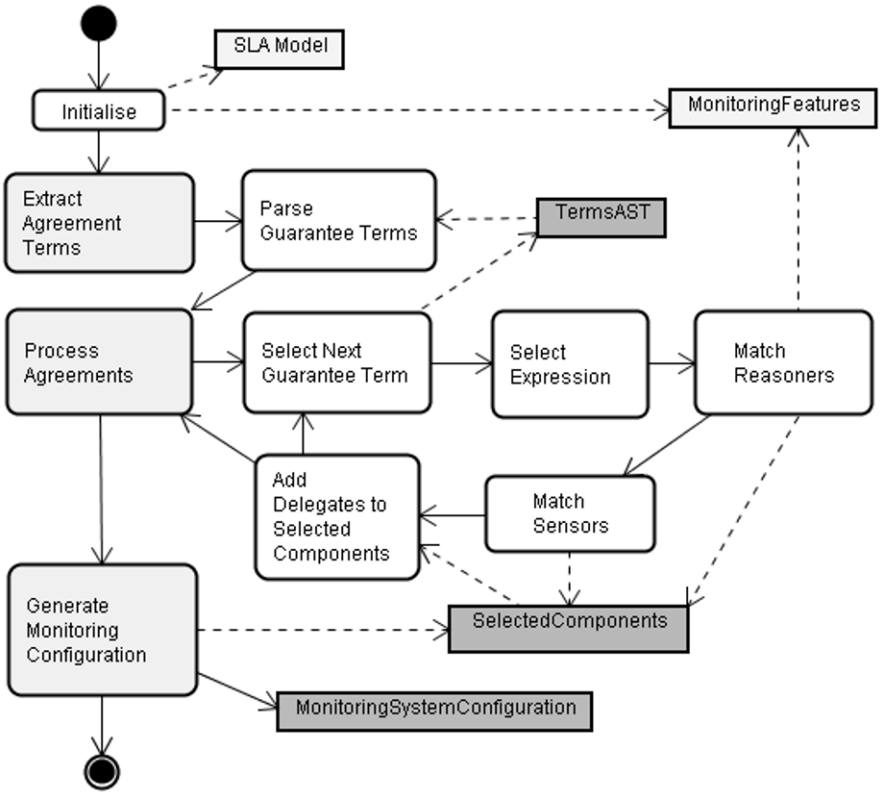

Fig. 3 SLA Monitoring Configuration Activities

The process starts by extracting the guaranteed states from AgreementTerms of the SLA specification. The terms are then parsed into a formal Abstract Syntax Tree (AST) for the expression of the states. The AST is then used as input to select each expression of each state (by traversal of the AST), and to match each left-hand-side (lhs), operator, and right-hand-side (rhs) of the expression with appropriate component monitoring features. The matching algorithms are discussed in Section 5. Following selection, the delegate components form a Selected Components list, which is used to generate a complete Monitoring System Configuration (MSC) for an SLA. If no suitable monitoring configuration can be formed (i.e. all monitoring requirements could not be matched), then an empty configuration is returned for a particular agreement term. This approach can be used in two types of situation: firstly, to configure the monitoring system when a new SLA needs to be monitored, and secondly, to perform adjustments to an existing configuration when requirements change or violations are detected. The main focus in this chapter is the first of these situations: we assume a new SLA is to monitored and therefore do not consider how this would affect the current state of monitoring.

4 SLA Term Decomposition

The MM abstracts the guaranteed states (guarantees made by any of the parties involved in the agreement) that certain states of affairs will hold for the service. We abstract these states from the agreement terms and parse the terms using a grammar based upon the Backus Normal Form (BNF) specification of the SLA specification [14]. The grammar for the parser is currently based only upon the agreement terms and guaranteed state expressions. A sample part of the grammar is listed in Figure 4.

```
1  /**********************************************************************
2  *SLA: The Specification of agreement terms
3  **********************************************************************/
4  void SLA() : {} {
5   AgreementTerm()* }
6  /**********************************************************************
7  *Agreement: AgreementTerms in SLA Model
8  **********************************************************************/
9  void AgreementTerm() : {} {
10 GuaranteeTerm()((TermOperator()) GuaranteeTerm())* }
11 /**********************************************************************
12 *GuaranteeTerm: A guaranteed state expression
13 **********************************************************************/
14 void GuaranteeTerm() : {} {
15 <QUOTED_STRING> (Term())(Comparator())(Term())}
16 /**********************************************************************
17 *Term: One or more TermFunctions or Identifier
18 **********************************************************************/
19 void Term() : {} {
20 LOOKAHEAD(TermFunction()) TermFunction() | <STRING> | <QUOTED_STRING> }
21 /**********************************************************************
22 *Comparator: Operators in Term expression
23 **********************************************************************/
24 void Comparator() : {} {
25 ( <EQUALS> | <NOTEQUAL> | <LTHAN> | <GTHAN> | <LEQUAL> | <GEQUAL> |
26 <ISEQUALTO> ) }
```

Fig. 4 Partial JavaCC Grammar for SLA Term Decomposition

The grammar is used as input to the Java Compiler Compiler (JavaCC) [16], which generates compiler source code to accept and parse source files specified in a defined grammar language. The resulting AST is built to represent the SLA specification terms and expressions. Beginning with the SLA declaration (lines 4–5), one or more AgreementTerms are parsed. Each AgreementTerm (lines 9–10) is parsed as one or more GuaranteeTerms, separated by a comparison operator. Each GuaranteeTerm (lines 14–15) is then parsed as an Identifier (which holds the ID label of the GuaranteeTerm), and a basic Term followed by a comparison operator and then followed by another basic Term. Each basic Term (lines 19–20) is represented by either one or more TermFunctions (similar to a normal function call syntax), a string Identifier (representing a variable of the SLA specification. The JavaCC function *LOOKAHEAD* informs the parser to check whether the next symbol to parse is

a function or string. Finally, the Comparator operators (lines 24–26) list the acceptable types of operators that can be used between GuaranteeTerms.

Since Term decomposition is based upon a generated parser, other SLA specification formats may generate their own parsers and transform their SLA specification to the AST input required by the MonitoringManager. In this way, the implementation of the configurator is generic and reusable. In addition, the generated AST compiler can be reused by monitorability agents (which accept the monitoring system configuration as a result of matching monitoring components). These Agents can translate the SLA terms into their own language specification. As an example, we have performed such a translation for the EVEREST monitoring language [15], which is based on Event Calculus and is used to analyse expressions in use cases of the SLA@SOI project SLA (Chapter 'Introduction to the SLA@SOI Industrial Use Cases').

5 Monitoring Configuration

5.1 Monitor Selection

A main configuration algorithm *MonitorConfig* (illustrated in Figure 5) is responsible for selecting all the term expressions from the prepared SLA term tree (Terms AST), obtaining a match for the expression terms with available monitoring component features, and then building a suitable monitoring system configuration. The algorithm begins by selecting the root of each AgreementTerm expression, which in turn holds one or more guaranteed state expressions (GuaranteeTerms). An AgreementTerm expression is predefined as a set of Boolean expressions (where all must result in *true* for the AgreementTerm to be upheld). Each GuaranteeTerm has a left-hand-side term, a right-hand-side term, and an operator. From these terms, a set of input types is determined. Two term monitors (M1 and M2) are set to analyse the terms, and a reasoner monitor is set to analyse the entire expression. If the left-hand-side of the expression is itself an expression, then the second monitor (M2) is recursively configured using the same algorithm (MonitorConfig). If this is not the case, then the value of the right-hand-side of the expression is used as the monitor. Furthermore, a reasoner monitor is assigned to select a monitor appropriate to the input types, operation, and required monitor features .

The MonitorConfig algorithm uses a *SelectMonitor* algorithm (Figure 5) to match the required types and operations (or term names) to the monitoring component features. The algorithm begins by iterating the monitoring component features available and building an appropriate feature list, (FeaturedMonitors), by selecting the monitors that match the type of term or operator. Each FeaturedMonitor is then selected and checked for appropriate input types. For example, the operator $<$ (less than) can be provided for numeric input types. If the feature and types match, the FeaturedMonitor is added to a list of selected monitors (SelectedMonitors). In the

	MonitorConfig. Given an agreement, select the most appropriate monitoring components.	**SelectMonitor**. Given a set of input types and a monitor term, select the first monitor that matches the term or event types required.
Input(s):	1) Terms AST: an AST of the Guaranteed Agreement Terms. 2) Features: a list of service monitoring features.	1) Input Types: a set of types (e.g. Number, Event, etc.). 2) Term: a term or operation to be monitored (e.g. completion-time or ¡ (operator)). 3) Features: a list of service monitoring features.
Output(s):	a set of monitoring components with configurations.	A monitoring component offering the types and operation/term.
Algorithm:	Given the Terms AST and a set of monitoring features 1) **select** root of AST and extract *expressions* 2) **extract** lhs, rhs, operation and **select** input-types 3) **set** M1 to **MonitorConfig(*lhs*)** 4) **if** *node.lhs* is **expression** then (a) **set** M2 to **MonitorConfig(*rhs*)** **otherwise set** M2 to *rhs.value* 5) **set** RM to **SelectMonitor(*input-types*,*operation*,*Features*)** 6) **store** delegate for expression	Given the input types, MonitoringFeatures and Term: 1) **for each** MonitoringFeature in Features **do** (a) **select** FeaturedMonitors where *type* **equals** the Term 2) **for each** Monitor in FeaturedMonitors **do** (a) **for each** *type* in input types **do** (i) **if** Monitor has *Type*, then (ia) **add** Monitor to SelectedMonitors 3) **select** the first Monitor in SelectedMonitors *(*replaceable selection criteria)* 4) **return** *SelectedMonitor*

Fig. 5 Algorithms for MonitorConfig (left) and SelectMonitor (right)

current implementation of the work, we simply select the first monitor matched. It is envisaged that an enhanced implementation will use some optimisation algorithm (at step 3. of the SelectMonitor algorithm), which will be based on criteria specified by the user (or indeed, specified as part of the overall SLA). This could also include assessing use of the same provider of features to group related monitors, reduce financial cost and optimise messaging.

5.2 System Configuration

As briefly discussed in Section 2, the MSC defines an entire configuration for monitoring an SLA within the monitoring system. An example MSC is illustrated in Figure 6, showing a reasoner component (for monitoring a guaranteed state), and a set monitoring feature components for each part of the guaranteed state expressions.

The MSC contains a list of components representing sensors, effectors or reasoners selected to support the GuaranteeTerms of agreements in an SLA.

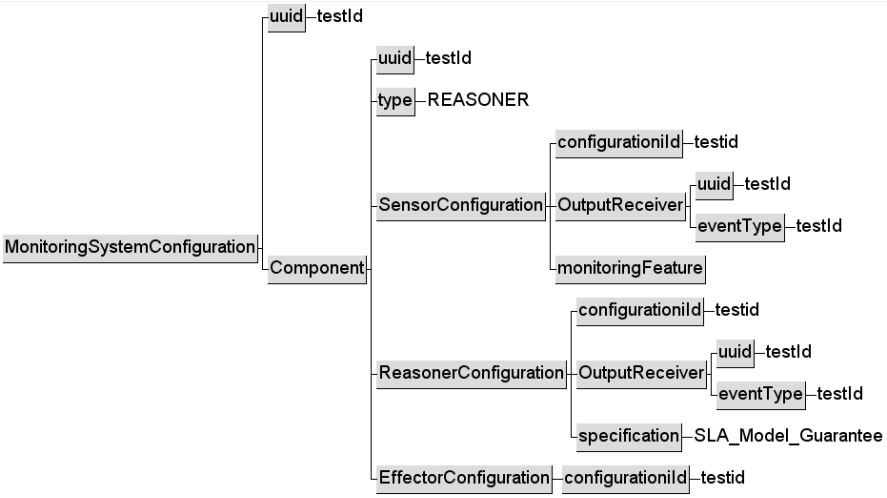

Fig. 6 A Monitoring System Configuration

Each component in an MSC contains one or more component configurations for each of the different components. For example, an MSC can contain a reasoner component that has component configurations for two sensor components and one additional reasoner component. The sensor component configuration contains a MonitoringFeature (that used to advertise features during selection of the sensor component) and one or more OutputReceiver(s). An OutputReceiver is another component which expects the result (as an event or value) to perform its own function. A reasoner component configuration also specifies one or more OutputReceivers, but a *specification* component replaces the MonitoringFeature component. The specification component lists the guaranteed states required by the component for reasoning.

5.3 Configuration Deployment

Here we briefly outline configuration deployment as an aid to the reader in understanding how the output is leveraged in the environment. As illustrated in Figure 1, a generated MSC is passed to a *service manager*, which links a service instance with a service *manageability agent*. The manageability agent exposes a method to accept a configuration and then, on behalf of the service under agreement, starts dependent components to monitor the service activities and to generate any notifications as part of that agreement. For example, each AgreementTerm has a reasoner (the

sum of evaluating all guaranteed states in the SLA). Each GuaranteeTerm also has a reasoner (to evaluate the expression of each guaranteed state). Once the service manageability agent is initialised, each reasoner is configured with the appropriate part of the MSC (e.g. for a cpu_load evaluation). The results generated by the reasoners and sensors in this configuration will be monitored by the manageability agent and appropriately routed from the Event Bus.

6 Implementation and Validation

6.1 The MonitoringManager Packages

The approach and algorithms discussed in this chapter are supported by a number of implementation packages. In particular the *MonitoringManager* component is available as an OSGI-enabled [10] JAVA package and can also be hosted as a web-service. In this section, we describe each of these packages with classes and their relationships (as depicted in Figure 7).

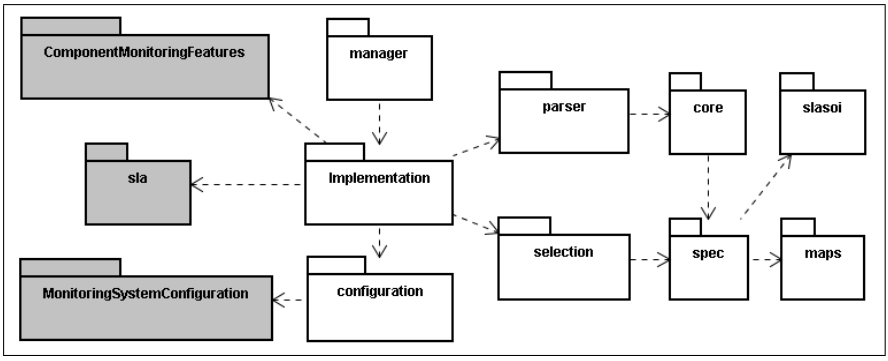

Fig. 7 Core Implementation Packages of the MonitoringManager

The MonitoringManager module is split into a number of packages: The core package *implementation* supports the MonitorConfig algorithm (as described in Section 5) provided by a *checkMonitorability* method, which accepts an SLA model (Chapter 'The SLA Model'), and a set of monitoring features (Chapter 'The Service Construction Meta-Model'). In turn, the implementation package depends initially on a *parser* package to support parsing of each AgreementTerm in the SLA model. The parser package provides an *AgreementTerm* class containing a *parse* method which accepts an AgreementTerm of the SLA and produces an expression AST (as described in Section 4). A sub-package of the parser package is the core parser itself, built from the compilation of a JavaCC grammar for the SLA agreements.

The implementation package also references the methods of a *SelectionManager* class contained within the *selection* package. This class provides methods and an overall framework for matching and selecting the most appropriate monitoring feature components with that of the expressions parsed previously (i.e. the SelectMonitor algorithm). To enable future dynamic configuration of selection algorithms, the SelectionManager refers to an extendable *ComponentSelector* module, offering a flexible *selectAppropriateComponent* method which may be redefined for preferred component selection strategies. Finally, the *configuration* package is used by the *checkMonitorability* method to configure the component selections into a required MonitoringSystemConfiguration format (Chapter 'The Service Construction MetaModel' for format specification).

6.2 Testing and Validation

To thoroughly test the implementation scope and suitability of configurations produced, we devised an SLA coverage test based upon each of the model elements described in the SLA@SOI SLA Model and the features available using a monitoring engine. Aligned with the work on translation and monitoring of SLAs (Chapter 'Translation of SLAs into Monitoring Specifications'), we listed: each element along with its specifications in a test SLA (SLA-ID), the events that required monitoring (Events), whether the model element expression in the SLA could be parsed by the MonitoringManager (Parsed), whether a suitable configuration was produced (MSC), whether the configuraton was accepted by a client monitoring component (Client), and whether any violation or service request and response events were successfully captured (Monitored). (Table 1 lists a sample of the results.) As we discussed in Section 4, the grammar for the SLA parser is currently based only upon the AgreementTerm and GuaranteeTerm expressions. Thus future work is required to enable guaranteed actions to be parsed and monitored. In addition, we also tested SLA model metrics (such as units of time) and primitive types (such as BOOL, CONST, TIME, etc.), mixing them and providing permutations for exhaustive testing.

The other main tests that have been carried out related to the use cases featured in the SLA@SOI project; The SLA specifications for both B4 (Chapter 'The Enterprise IT Use Case Scenario') and B6 (Chapter 'The eGovernment Use Case Scenario') have been fully covered in testing. We also expect to continue testing with other monitoring engines, for example, the ASTRO Project's [11] SLA monitoring tools can be tested with infrastructure monitoring components.[1]s

[1] The MonitoringManager implementation, EVEREST monitoring framework, and SLA@SOI test cases are an integrated part of the SLA@SOI project platform showcase and are available from: http://sourceforge.net/projects/sla-at-soi/

Table 1 Sample Test Cases for SLA Elements, Parsing, Configuration and Monitoring

Model	SLA-ID	Events	Parsed	MSC	Client	Monitored
InterfaceDeclrs	ID1	None	Yes	Yes	Yes	No
AgreementTerms	AT1	Violation	Yes	Yes	Yes	Yes
Guaranteed Actions[a]	GA1	Violation	No	No	No	No
Guaranteed States	GS1	Violation	Yes	Yes	Yes	Yes
VariableDeclrs	VD1	Computation	Yes	Yes	Yes	Yes
Terms	SLA-ID	Events	Parsed	MSC	Client	Monitored
core:and	GS1	Computation	Yes	Yes	Yes	Yes
core:equals	GS1	Computation	Yes	Yes	Yes	Yes
core:sum	GS1	Computation	Yes	Yes	Yes	Yes
core:series	GS2	Computation	Yes	Yes	Yes	Yes
core:availability	GS1	Request-Response	Yes	Yes	Yes	Yes

[a] The element is not currently supported

7 Related Work

Background and related work in this chapter falls within two areas: First, we consider the definition and translation of SLAs, and second, the runtime monitoring of service-based systems based upon monitoring features.

Several projects have focused on SLA definitions and provisioning in the context of both web and grid services. The Adaptive Services Grid (ASG) project, for example, has designed an architecture for establishing and monitoring SLAs in grid environments [8]. In this architecture, the monitoring rules and parameters as well as the architecture for SLA monitoring are statically defined and cannot be updated at runtime. The TrustCOM project has also produced a reference implementation for SLA establishment and monitoring [17]. This implementation, however, does not involve the dynamic setup of monitoring infrastructures. The SLA Monitoring and Evaluation architecture presented within the IT-Tude project [7] has several similarities with the approach presented in this chapter, such as the need to separate SLAs from service management. This work focuses, however, on statically binding services and monitors, whilst the SLA@SOI work focuses on dynamically allocating monitors to SLA parts, based upon matching the exact terms that need to be monitored and the monitoring capabilities available for different services. Further, in the IRMOS project architecture [19], service monitors are used to gather information about QoS levels. The SLA@SOI approach splits these monitors into three types, providing greater flexibility and catering for changing services (with effectors) as the need arises. With regards to SLA translation, in [18, 12], the authors describe decomposing an SLA of resource requirements (with the purpose of building a system that represents the SLA required). This approach is focused more on

building a system rather than monitoring existing services; however, it also employs techniques to optimise and arrange efficient configurations based upon the SLA expressions stated. In [13], the authors consider evaluating expressions for conditions of properties of services (e.g. response time), however, their SLA format appears to offer only single assertions rather than complex expressions.

Work on runtime monitoring of service-based systems has resulted in the development of different types of monitors. These monitors realise either intrusive or event-based monitoring. Intrusive monitoring relies on weaving the execution of monitoring activities at runtime using code that realises the service itself or the orchestration process. In the case of composite services, this can be done directly in a process engine, by interleaving monitoring code with the process executable code as in [2, 3, 1, 9]. The assessment of monitoring service properties required by SLAs cannot be easily achieved through this paradigm, since the properties to be monitored and the actions required for monitoring must be interleaved with service execution code, and therefore known *a priori* by the system designer.

The work described in this chapter extends existing approaches to dynamic generation of monitoring system configurations [4, 5, 6]. Specifically, we consider individual agreement terms within an SLA by decomposition of complex guarantee expressions, utilise a wider spectrum of monitoring components (e.g. sensors and effectors), and support complex monitoring configurations that can engage different monitoring components for checking the same SLA term if necessary.

8 Conclusions and Future Work

In this chapter we have described an approach to advanced configuration of service systems, in particular, systems in which an SLA agreement has been established and concerns services that require monitoring. The work aims to provide a generic module, applicable not only to the architecture illustrated, but also to other architectures (although still based upon SLAs and monitoring component features). This work will be extended to cover further elements of the SLA specification (such as guaranteed actions, which are not presently considered), and also to include preferential selection of monitoring components. Preferential selection of components is useful where there are multiple monitoring components offered for the same term. Preferences could be based upon monitoring cost (either in terms of computing resources or financially) or non-functional requirements. The existing implementation is already part of the wider SLA@SOI project monitoring platform, providing integration and validation testing, and we are keen to seek other environments in which to test it.

References

[1] Baresi, L., Bianculli, D., Ghezzi, C.: Validation of Web Service Compositions. IET Software **1**(6), 219–232 (2007)

[2] Baresi, L., Guinea, S.: Towards Dynamic Monitoring of WS-BPEL Processes. In: International Conference on Service-Oriented Computing (ICSOC) (2005)

[3] Bianculli, D., Ghezzi, C.: Monitoring Conversational Web Services. In: 2nd International Workshop on Service Oriented Software Engineering (IW-SOSWE) (2007)

[4] Comuzzi, M., Spanoudakis, G.: Dynamic Set-up of Monitoring Infrastructures for Service-Based Systems. In: 25th Annual ACM Symposium on Applied Computing, Track on Service Oriented Architectures and Programming (SAC 2010). ACM, Sierre, Switzerland (2010)

[5] Foster, H., Spanoudakis, G.: Model-Driven Service Configuration with Formal SLA Decomposition and Selection. In: The 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). Crete, Greece (2010)

[6] Foster, H., Spanoudakis, G.: Advanced Service Monitoring Configurations with SLA Decomposition and Selection. In: 26th Annual ACM Symposium on Applied Computing (SAC) Track on Service Oriented Architectures and Programming (SOAP). ACM, TaiChung, Taiwan (2011)

[7] IT-Tude: SLA Monitoring and Evaluation Technology Solution. Available from: http://www.it-tude.com/?id=gridipedia (2009)

[8] Jank, K.: Reference Architecture: Adaptive Services Grid Deliverable D6.V-1. Available from: http://asg-platform.org/twiki/pub/Public/ProjectInformation (2005)

[9] Lazovik, A., Aiello, M., Papazoglou, M.: Planning and Monitoring the Execution of Web Service Requests. International Journal of Digital Libraries (2006)

[10] OSGi Alliance: OSGi Service Platform Core Specification Version 4.2. Available from: http://www.osgi.org/Download/Release4V42 (2011)

[11] Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P.: Planning and Monitoring Web Service Composition. In: AIMSA, pp. 106–115 (2004)

[12] Richter, J., Baruwal, C., Kowalczyk, R., Quoc Vo, B., Adeel Talib, M., Colman, A.: Utility Decomposition and Surplus Redistribution in Composite SLA Negotiation. In: IEEE International Conference on Services Computing (2010)

[13] Sahai, A., Machiraju, V., Sayal, M., Jin, L.J., Casati, F.: Automated SLA Monitoring for Web Services. In: IEEE/IFIP DSOM, pp. 28–41. Springer-Verlag (2002)

[14] SLA@SOI: Deliverable D.A1a: Framework Architecture. Available from: http://sla-at-soi.eu/publications/deliverables (2009)

[15] Spanoudakis, G., Kloukinas, C., Mahbub, K.: The SERENITY Runtime Monitoring Framework. In: Security and Dependability for Ambient Intelligence,Information Security Series. Springer (2009)

[16] Sun Microsystems: The Java Compiler Compiler (JavaCC). Available from: https://javacc.dev.java.net/ (1999)
[17] TrustCOM: Deliverable 64: Final TrustCoM Reference Implementation and Associated Tools and User Manual. Available from: http://www.eu-trustcom.com/ (2007)
[18] Yuan, C., Iyer, S., Liu, X., Milojicic, D., Sahai, A.: SLA Decomposition: Translating Service Level Objectives to System Level Thresholds. In: Fourth International Conference on Autonomic Computing (ICAC) (2007)
[19] Menychtas, A., Gogouvitis, S., Katsaros, G., Konstanteli, K., Kousiouris, G., Kyriazis, D., Oliveros, E., Umanesan, G., Malcolm, M., Oberle, K., Voith, T., Boniface, M., Bassem, M., Berger, S.: Deliverable D3.1.3: Updated version of IRMOS Overall Architecture. Available from: http://www.irmosproject.eu/Deliverables/ (2010)