Towards Process-based Composition of Self-Managing Service-Oriented Systems

Min’an Tan  
University of New South Wales  
and NICTA  
NSW, Australia  
minant@cse.unsw.edu.au

Yan Liu, Liming Zhu  
NICTA  
NSW, Australia, 1430  
jenny.liu,liming.zhu@nicta.com.au

ABSTRACT

Loose coupling and preserving safe changes are two key criteria for composing self-managing services. Composition of adaptive control components with business services without interfering with the original service operations is complicated by the dynamic and highly distributed nature of service-oriented systems. Essentially, encapsulating control logic into abstract logical models enables a clear separation of concerns, with states and transitions indicating the logical control flow. The challenge is to seamlessly integrate these models with services and their host infrastructure as a unified self-managing environment. In this paper, we present an architectural solution towards process-based composition and coordination of self-managing services. This architecture framework leverages business process models to produce declarative and executable control models. We discuss the problem context and outline the research challenges of such an approach.

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures—Domain-specific architectures, Data abstraction; D.2.2 [Software Engineering]: Design Tools and Techniques—State diagrams

General Terms

Design, Management, Reliability

Keywords

SOA, software architecture, self-managing, business process

1. INTRODUCTION

As SOAs become more widely adopted in large software systems, the typical SOA environment has become more complex. Management of these increasingly complex environments is exacerbated by cross-application components and services as well as overlapping SOA environments with service providers beyond the administrator’s control. While some human inspection or administration tools can and should be provided, it is unrealistic to expect that all controls can be effectively handled manually. Being fully dependent on manual controls would void the improvements in timeliness and adaptivity gained with an increased level of automation. Consequently, incorporating adaptive and self-managing capabilities into services [3, 10] is attracting considerable attention as a means to respond to both the functional and environmental changes that occur after the service deployment.

The development of adaptive and self-managing service-oriented systems follows the paradigm of autonomic computing [5, 6]. The original focus of autonomic computing is on IT infrastructure, targeting problems in the administration and management of complex computing environments. A generic reference architecture has been proposed by IBM’s Autonomic Computing initiatives [5] to develop autonomic systems. In principle, a system exhibiting adaptive and self-managing capability consists of two parts: (1) a base system that implements the business logic and provides concrete functionalities; and (2) a set of control loops that comprise adaptive components for constantly monitoring the system, analyzing the situation and deciding on the actions to affect the system’s behavior.

The basic idea and primary concepts of autonomic computing apply to most systems including systems built on SOAs. When the base system is composed of services in an SOA, the addition of control loops results in adaptive and self-managing service-oriented systems.

Effectively managing complex SOAs in an intrinsically distributed and highly dynamic environment demands a balance between flexibility and dependability. This requires that services based on SOAs be easily adaptable to unforeseen changes during their operation. It is also critical to ensure that these changes are safe and do not incur any unfavorable consequences.

In order to address these two requirements, Kramer and Magee [7] outline research challenges and potential solutions of developing self-managing systems at the software architecture level. Of the many aspects of adaptive and self-managing architectural design, our research aims to achieve loose coupling and separation of concerns for service-oriented systems. We believe a loosely coupled architecture further enables supervision and governance of adaptive components to preserve safe changes and prevent misbehavior.

In this paper, we discuss the research challenges in design-
ing software architectures to achieve loosely coupled composition of self-managing service oriented systems. In particular, we focus on the discussion of the following issues:

1. Increasing the understandability of the adaptation and self-managing strategy.
2. Improving the monitoring and tracking of the adaptive and self-managing behavior.
3. Reducing maintenance costs. Updating or replacing existing adaptive components should be easily achieved without infringing the adaptive logic implementation.

We propose a research solution towards applying business process models to abstract the logical computation of adaptive components and automate their execution. As a result, the adaptive control logic is abstracted and declaratively modeled as business processes. Implementation of adaptive components are organized into processes by attaching them to the states or transitions in a process. The structure of adaptive components is composed by the roles and intentions of individual components rather than hard coded references.

The structure of this paper is as follows. We discuss the problems and issues to realize this solution in section 2. In section 3, we present a conceptual architecture framework and illustrate its utilization by an example application. We further discuss the research directions that will lead to the production of enabling technologies and mechanisms in section 4. The related work is summarized in section 5 followed by reflection and conclusion.

2. PROBLEM DESCRIPTION

Software adaptation can be achieved through three strategies, namely interface adaptation, architecture adaptation and behavior adaptation. Interface adaptation changes the type of an application component such as a method signature. Architecture adaptation monitors its software architecture model to create changes in the structure of components and their interactions. Lastly behavior adaptation changes the execution of components by changing the configuration of components or by monitoring and intercepting interactions between components.

A compositional approach enables architecture and behavior adaptation, which allows algorithms or components to be composed into control loops to improve a program’s fitness with respect to its current environment [9]. The advantage of this approach is that an application can adopt new algorithms to address unexpected changes that it was not originally designed to meet. An architecture that supports compositional adaptation satisfies the separation of concerns to construct a base system and complementing control loops.

Component-based development is one of the enabling technologies to compose adaptive systems [9]. While component-based development helps to modularize and encapsulate adaptive and self-managing computation, there is still tight logical coupling and interdependencies between adaptive components. Examples of such a tight coupling include systems where the monitoring, analysis and configuration control components explicitly invoke one another without an intervening layer of logical abstraction.

This tightly coupled implementation is simpler to implement, but this comes at the cost of flexibility and scalability. Since a system must record hard-coded references in order to change its implementation, tightly coupled adaptation is far from practical for large, complex systems.

Tight coupling is a particularly risky architectural style in self-managing systems. This is because certain types of tight coupling can make the self-managing mechanism itself vulnerable to misbehavior when system components fail, leading to situations where the self-managing system itself may need management.

A better design is for adaptive components to be identified by their logical or functional roles, so that connections between components are not hard coded, but established dynamically by their intent or roles. The resulting architecture will be much more extensible and flexible in maintaining changes made to self-managing controls. For example, misbehaving adaptive components can be more easily isolated and replaced than those with hard coded references.

3. CONCEPTUAL ARCHITECTURE FRAMEWORK

We propose a conceptual architecture framework that addresses the relevant software architecture issues raised in the previous sections. The proposed framework aims to provide a hierarchical, process driven approach towards the management of service-oriented systems. By utilizing business process models, the control logic is abstracted into executable models. As a result, the overall architecture is loosely coupled between business and management functions and is able to support the deployment of policies at various levels. In addition, it supports declarative design and deployment of adaptive control loops using process modeling languages to provide an administrator with a wide degree of flexibility in systems management. We exemplify this architecture using a representative enterprise integration example derived from industry best practices [4].

3.1 An Illustrating Example

Consider a loan brokering application, where a customer submits requests for a loan quote to a loan broker. The loan broker checks the credit worthiness of a customer using a credit agency. The request is then routed to appropriate banks who each give a quote, and the lowest quote is returned to the customer.

**Figure 1: Loan Broker application deployment**

The application has been deployed (see Fig 1) over an Enterprise Service Bus (ESB) with messaging capabilities provided by Java Messaging Services (JMS), bringing together Web Services, Plain Old Java Objects (POJOs) as well as remote Enterprise Java Beans (EJB). Event flow in
the application is specified by the ESB configuration and processing is driven by the arrival of events. In this application, there are two scenarios concerned with adaptation and self-managing controls: failover and overload controls [4].

3.1.1 Failover control
Suppose the responsiveness to requests of the credit agency is in question, and the administrator wants to allow a graceful failover to an alternative credit agency should the primary agency fail. One solution is to insert an additional switching component between the loan broker and the credit agency that can reroute traffic to an alternative credit agency (see Fig 2). In this solution, a heartbeat sensor constantly sends test messages to the credit agency to ensure its correct operation. A notification message is sent to the switch to reroute traffic to a backup credit agency if the test message fails. This forms a feedback control loop between the heartbeat sensor (feedback) and the switch (control).

![Figure 2: Failover control-feedback loop](image)

3.1.2 Overload control
In addition to ensuring the robustness of the credit agency, the administrator also wishes to prevent the loan broker from getting overloaded by requests. As shown in Fig 3, a throttling component can be used to regulate the flow of requests by limiting the number of concurrent requests being processed. A traffic flow sensor would also be used in this solution to detect the flow rate. Beyond the threshold of the system’s computing capacity, higher flow rates reduce the number of concurrent processes handling requests and vice versa.

![Figure 3: Overload control-feedback loop](image)

3.1.3 Complications
Both scenarios share the same characteristics of composing self-managing services: composition of adaptive control components with existing services should be transparent to business logic, i.e. without interfering with the original service operations.

In a component based implementation of the controls, flexibility is reduced as the logic for the control loop would be embedded in the components. For example, if the criteria to trigger the failover switch is changed, a rewrite of the basic switching and heartbeat components would be required to coordinate their logic. Another possible solution is to use a “coordinating” component to control the interaction of components in the feedback loop, but again, a change to the hard coded logic is required to this coordinating component if the loop structure changes. The high coupling mentioned in section 2 is still present.

Introducing control components/services also creates dependencies between the business and management flows. The loan broker needs to be aware of the switch in order to send messages correctly to the switch and not the credit agency directly. As more management controls are added, the introduced dependencies both obscure the original business flow as well as reduce the system’s flexibility to changes in both flows.

3.2 Architecture Overview
We propose the use of business process execution engines to address the high logical and structural level coupling. A conceptual architecture framework is depicted in Fig 4. The key entities of this architecture are the Executable Process Models (EPMs) and the Process Execution (PE) engines.

An EPM is a graphical description of an execution flow for adaptive control components. The basic structure of a process is made up of nodes and transitions. Each node in the process has a piece of code associated as its behavior. In contrast with static business process modeling approaches, where only the process itself is defined in the model, this executable process model can be complemented with complex behavior. For example, the above two scenarios can each be abstracted into an equivalent EPM as shown in the middle of Fig 4.

The control components are associated with nodes. For example, the throttling component, the switch and the heartbeat sensor are all attached to individual nodes. The interactions of components are not hard coded but managed by the workflow in the EPM. When a transition is triggered, and the workflow enters a specific node, the actions defined in the behavior components are performed.

Coherence between the actual control components deployed and their associated nodes in the EPM is provided by the sensor aggregators and effecting routers. The sensor aggregator receives status updates from the sensor components and sends it to the EPM. Execution of actions in the nodes associated with the sensors cause changes in state to other components. These changes are then propagated to the actual control components by the effecting router.

The EPM is deployed in a PE engine. The PE engine introduces an additional layer of indirection to process workflow management. It enables runtime redeployment of an EPM, allowing the logical management flow to be changed on-the-fly. This allows structural changes to be effected without change to the configuration of individual services and components.

An EPM, its participating control components and the associated PE engine forms the computational boundary for specifying a self-management concern. Multiple EPMs for
multiple self-managing concerns are coordinated in a hierarchy: individual PE engines are concerned with the specific control loop, while the coordinating PE engine is concerned with the interaction of concerns (shown in the center of Fig 4). This can be further extended into a hierarchical structure, where the coordinating PE engine may form the PE engine of a larger scale service system.

Note that communication between all control components, such as execution engines or sensors, is done through sending and receiving events. This is to ensure loose coupling between the control components for flexible deployment.

The following sections highlight key functional requirements for the PE engines and the middleware to realize the proposed architecture in service-oriented systems. The required architecting roles to support such an architecture are also discussed.

### 3.3 Process Execution (PE) Engine

The PE engine is the core of the system, and is responsible for the execution of process flows.

Functionally, the PE engine must be easily integrated into middleware infrastructures that provide the host environments for business services and the control components. It must also support on-the-fly deployment and updates of EPMs to allow changes to be made to the management logic without disrupting existing business flow. For the engine to be effective, an established tool chain will need to be in place for an EPM lifecycle, from the creation of the EPM up till its deployment and replacement. The tool chain includes major entities, such as:

- **Modeler** The development tool to model, design and edit business process models expressed with an enriched modeling language. This modeling language allows annotations for metrics that can be monitored and measured at runtime. For example, the response time taken or resource utilization of a process task. The created EPMs can also be saved as packages for distribution.

- **Management Console** The runtime monitoring tool which allows advanced users and administrators to supervise the operation of a running business process application and infrastructure.

- **Deployer** The deployment tool to deploy or update EPMs in an application. It is also responsible for injecting any application specific information into the model, such as environment variables. The deployer will be integrated with the management console to allow quick identification and resolution of deployment issues.

- **Simulator** The simulation tool to provide a virtual environment for the deployment of an EPM. Administrators can view the predicted effects of an EPM deployment and refine the EPM accordingly.

Besides the tools mentioned, techniques for the translation of process flows to actual logic will also be required. We foresee the best practices being a combination of process design expertise and domain knowledge of the actual process being modeled.

### 3.4 The Middleware Host

The middleware is the glue that binds the various components and services of both the management and business levels together. To support our proposed architecture effectively, a key feature of the middleware will be a need to transparently deploy and replace service and component endpoints, as well as runtime redirection of traffic between endpoints. This will ensure the loose coupling between management and business functions, and that changes to their underlying physical structure can be made without disrupting each other’s execution flow. Such on-the-fly deployment or replacement would also require that active servicing components be “paused” and “resumed” without any messages being lost, and should be built into the middleware.

### 3.5 Architecting Roles

To take advantage of the architecture’s flexibility, we foresee a number of specialized roles involved. In addition to the specific knowledge provided by domain experts such as network and database engineers, process designers will also play a key role in the effectiveness of an EPM. Key architecting roles include:

- **Domain Expert** The domain expert is an expert in the management concern being modeled. The expert provides the functional workflow and ensures the correctness of the mapping between the model design and its associated execution.

- **Process Designer** The process designer is responsible for decomposing the expert’s functional workflow into the nodes and transitions associated with the modeling language. The designer will draw upon his knowledge of the modeling language’s capabilities and workflow management theories to decide the organization of process information to in a maintainable manner.

- **System Architect** The system architect is the administrator of the system and is responsible for the deployment of EPMs. The architect has intimate knowledge of the application and resolves any conflicts arising from decisions made by EPMs. The architect will also be the expert in the creation of EPMs for coordinating process engines.

The actual flow of events might be as follows: a domain expert, such as a network engineer will determine the broad processes and states required for a network load balancing control loop. The expert will then work together with a process designer to model the task as a series of nodes and transitions in the execution EPM.

The expert will then translate the states into executable behavior components attached to the EPM. Relevant parameters affecting the process will be exposed to a coordinating EPM at the application level, building up the hierarchy of control. The exposure of provisioning requirements and process parameters will conceivably allow the coordinating EPM to refine its picture of the application’s management capabilities and make changes to available and/or deployed policy parameters.

The system architect oversees the creation of the EPM’s management parameters and ensures a smooth interaction with other EPMs in the system. The architect might also modify the coordinating EPM to reorganize the management coordination in an application, if needed.
4. RESEARCH DIRECTIONS

So far we have discussed the architectural solution to a flexible and loosely coupled self-managing architecture. Another key requirement of this architecture is to ensure that system safety properties are not violated during changes. Even though the adaptive and self-managing capabilities are guided by high level policies derived from business goals, adaptive controls are constrained by a number of factors such as resource scheduling, cost, and potential conflicts with cross-cutting controls. Preserving the system’s safety properties means continuous monitoring of the runtime status of both the business services as well as the adaptive components.

We believe this process-based architecture framework can accommodate mechanisms towards monitoring, supervising and managing control components to preserve safety. We summarize the emerging research challenges towards realizing these mechanisms.

4.1 Policy Management

Policy is a loaded word in SOA circles and extremely context specific. Research into policy specification has been diverse, including ontology, rule and rationale based approaches [1, 8]. Of particular interest to us, however, is to identify any specific policy approach that is suited for process management, or can be tailored to meet the specific needs of a process-based approach. Such a policy management system would need to be able to operate at multiple levels; at a generic process level to allow for extensibility of the logical functionality of components in an application, and at a specific, component based level to allow for human domain expertise.

The key to an effective policy approach is ensuring that sufficient enforcement mechanisms are in place to effect it. Some indication of the predicted effectiveness of the policy should be determined to aid the administrator in making policy deployment decisions. This can be taken a step further to only allow the implementation of enforceable policies, or policies with an effectiveness beyond a certain threshold. Such a system will require detailed dependency analysis to be carried out among related enforcement mechanisms and policies during the creation and/or deployment of new policies, analysis that seems to lend itself to the use of a hierarchical process management structure. A hierarchical, coordinated process approach will intuitively allow traceability of dependencies from high level concerns down to low level components to determine their role in the effectiveness of the overall policy picture.

What is perhaps more difficult than policy provisioning and effectiveness evaluation is the problem of policy conflict resolution. It is likely that policies in a system will involve overlapping components and may require conflicting configurations. Depending on the nature of the specified policy, conflicts can be determined at compile time or run time, and resolution effected. The potential for such conflicts suggests the need for centralized detection and resolution, to ensure appropriate dependencies have been considered at all levels for the conflicting policies. A centralized view would also provide avenues for the reuse of already deployed control components, though such a step would add an additional layer of complexity to the policy provisioning dimension.

4.2 Plug and Play Architecture

One of the benefits of SOAs is its dynamic composition, in which its underlying physical structure can be altered so long as its logical process flow is retained. Control mechanisms should respect and retain this loose coupling at the business level, and consequently, should be non-intrusive to the core business logic of an application. Consequently, the control mechanisms and core business logic should also be loosely coupled, to allow changes to the service/component structure at both management and business levels with minimal effects on each other.

More fundamental than the coupling between
and management functions is the coupling between components and services at these levels themselves. One complication in realizing a truly plug and play architecture in the management level is system policy. Ideally, applied policies should still be in effect even after changes to the underlying physical structure of a management process. The management functions should seamlessly handle dependency resolutions between other policies and processes, as well as changes to the configuration of other management services and components.

Depending on the granularity of management required, we do recognize that it may not be possible to provide out of the box management solutions. Monitoring of business process messages is highly specific to the business process itself, and can only be achieved through custom code. Nevertheless, a major goal of any management system is to provide a sufficiently comprehensive platform to minimize the amount of required customization.

5. RELATED WORK

Applying business process models to support self-managing software systems has recently been investigated in various domains [2, 12]. For example, Verma and Sheth envisioned autonomic web processes [12]. Similar to the core of this paper, they elevated autonomic computing concepts from infrastructure to a process level. Their paper discussed existing technologies and steps needed to shorten the gap from current process management to autonomic web processes.

Extensive research has been done in the translation of business process models to execution languages, and there has been an increasing adoption of business process modeling (BPM) tools to coordinate business process flows, as opposed to hard-coded application logic. A good summary of the relevant techniques and tools is covered in [11]. As our core model, the EPM, leverages business process modeling languages to represent the adaptive logic, integration of the semantics essential to adaptation and self-management with general business process modeling languages remains an open question. Our on going research in evaluating the expressiveness of business process definitions will contribute insights to this research space [13].

A comprehensive survey [9] discussed existing technologies that could enable dynamic composition of adaptive software. It also classified different approaches by how, when and where composition might occur. The core of all these approaches was intercepting and redirecting interactions among program entities. These mechanisms help to customize our architecture framework to a specific middleware platform.

6. CONCLUSION

In this position paper we envision a conceptual software architecture framework that enables separation of concerns and loose coupling when composing adaptive self-managing services. The flexibility and extensibility of this architecture are achieved by encapsulating the adaptive control logic into executable process models. The architecture framework supports the execution of these models. This loosely coupled architecture is also essential to enable supervision and governance of adaptive components to preserve safety and prevent misbehavior of adaptive controls. The discussions in our problem statement and research directions call for further research to ensure the safety of self-managing services.

Realizing this vision will require addressing open challenging issues including policy management and enabling plug-and-play architectures.

7. ACKNOWLEDGMENT

NICTA is funded through the Australia Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

8. REFERENCES