Towards Self-management in Service-Oriented Computing with Modes

Howard Foster, Sebastian Uchitel, Jeff Kramer, and Jeff Magee

Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2BZ, UK

Abstract. A self-managed system is both self-assembling and self-healing. Service-oriented Computing (SoC) architectures, such as a Web Services Architecture (WS-A)illustrate a highly distributed, potentially dynamic,domain for component configurations. We propose the use of component architecture "modes" to facilitate the self-management of services within a SoC environment. A mode abstracts a set of services that are composed to complete a given task. Our approach, named "SelfSoC" includes designing and implementing key parts of a self-managed system specifically aimed at supporting a dynamic services architecture. We extend Darwin component models, Alloy constraint models and distributed system management policies to specify the mode architectures. We also propose the generation of dynamic orchestrations for service compositions to coordinate different modes of an automotive services platform.

1 Introduction

Self-management of components in distributed system architectures is becoming a widely researched area, with the emerging paradigms of Service-oriented Computing (SoC) and Service-oriented Architectures (SoA), these have increased the interest of having less human intervention in the management of complex, distributed systems. With the use of a Web Services Architecture (WS-A) [3] to overcome technical interoperability difficulties using standard protocols and messaging formats, the issue of managing complex configurations of services is a leading requirement of progressing the use of this type of architecture. The availability of services is also more prominent in a dynamic service configuration whereby the provider of the service may not be known in advance. As services should be developed with much closer representation to clients requirements, the context and environment co-exist to support requests. Events that change context or conditions within the environment however, need self-managing mechanisms to support both initial configurations (self-assembly) and reconfigurations (self-healing) [10]. Further self-management techniques can be applied to support optimal configurations of a given set of components (self-optimisation or also known as self-tuning).

The main contribution of this paper is to present self-management techniques applied to a services architecture, using the concept of modes and the coordination of reconfigurable component architectures. We firstly discuss the mode

approach, where the design of composite component architectures (as services) provides a formal abstraction of elements that change within a self-managed environment. The relation to scenarios and events that can invoke changes in the architecture, such as services becoming unavailable is illustrated through behavioural models, which when combined with component mode architecture descriptions can be used to generate coordination processes which detect and switch architectures depending on the context. Secondly, we provide an example related to a case study defined in the SENSORIA EU project [12] for a highly dynamic coordination of services in Automotive Route Planning scenarios. Our role in this project explores the analysis and coordination of components in a self-managed services environment.

In Section 2, we provide a background to SoC, WS-A and self-management techniques for distributed component architectures. In Section 3, we describe our approach and a case study example providing a running example of a services architecture requiring self-management. In Section 4, we discuss our notion of modes, the design of architectures based upon components with modes and the self-assembly and self-healing techniques used. We also discuss specifying rules, constraints and analysis on reconfiguring a services architecture based upon modes. Section 5 discusses current assumptions and limitations of the approach,whilst section 6 concludes this paper with a work summary and an indication of our future work direction.

2   Background

Service-oriented Computing (SoC) [15] has emerged from a number of different directions in computing science. Technically, the most influential aspects have been from that of remote procedure calls (RPC), messaging standards and internet protocols, where agreement has been formed on standard ways to interface components in a distributed system. SoC however, also encompasses a design philosophy, bringing re-use and context support directly to the specification of components and composite components (composed of more than one component). As an example of a SoC architecture, a Web Services Architecture (WS-A) is a set of conceptual elements defining a common set of standards between interoperating components, running on different platforms or frameworks. There remains an ambitious task of building systems on such dynamic architectures and this is closely aligned with the capabilities of self-management and reconfiguration of services. Self-management of systems is not a new idea, with ideas from both the cybernetics [20] and system theory [18] worlds. As discussed in [17] however, one of the main problems that exists in self-management is to understand the relationship between the system and its subsystems: can we predict a system's behavior and can we design a system with a desired behavior in all situations? The issue is deemed the "closed control loop issue" [2] - to tailor a system with all possible states and external influences completely specified.

For these tasks, and in a web services perspective, there has been some work on providing dynamic orchestrations of web service compositions using dynamically

generated workflows for service orchestration engines[1]) and service matching techniques on service brokers. In [8], a workflow BPEL process is monitored and exceptions handled for failing services by creating new service proxies (i.e. re-binding to a set of pre-determined services). An alternative approach in [6], uses semantic web service specifications to match new services and generates compositions based upon their specification for use (e.g. in a correct sequence). Several other works [16,4,14] apply Artificial Intelligence (AI) Planning to composing service compositions from higher-level goals, typically also in a semantic web service specification. A characteristic across these works however, is that they aim to provide dynamism for the same type of service, focusing mainly on availability and the ability to switch providers. The wider issue of self-management must also consider a change in service process functionality and reconfiguring the architecture to support correctness in such functional dynamism.

One of the leading industrial efforts for self-management is part of IBM's Autonomic Computing programme. Their blueprint [5] suggests three goals for an architecture in autonomic computing. Firstly, it must describe the interfaces and behaviours required by individual system components. Secondly, it must describe how to compose these components so that the components can contribute toward the goals, and thirdly, it must describe how to compose systems from these components in such a way that a system as a whole is self-managing. As with most complex systems development, an iterative elaboration of requirements is key in capturing key points of change. To address the broader issues in service architecture self-management we propose the use of component architecture "modes" [13] to facilitate the self-management of services within a SoC environment. A mode abstracts a set of services that are composed to complete a given task. Our approach, named "SelfSoC" includes designing and implementing key parts of a self-managed system specifically aimed at supporting a dynamic services architecture.

3 Approach

SelfSoC is our approach to addressing some of the issues in dynamic reconfiguration of services. Figure 1 illustrates the approach, and the areas considered. In this paper we specifically discuss the areas of mode component architectures, behaviour and analysis whilst highlighting future work on implementation of coordination. SelfSoC consists of two parts. Firstly the specifications are analysed for correctness (e.g. that reconfiguration can be undertaken and that constraints are realised). Secondly, the verified source and models for service composition modes are used to derive some deployment artefacts to assist in the self-management runtime. These artefacts include coordination requirements and capabilities for service brokering, and coordination processes which may be executed to handle events and service architecture changes. Our work so far has concentrated on the specification and analysis of modes, and we report this in the following sections. To help us define the necessary elements of SelfSoC we utilised a case study from the European Union SENSORIA project, for which we are participating to support enhanced service deployment mechanisms.
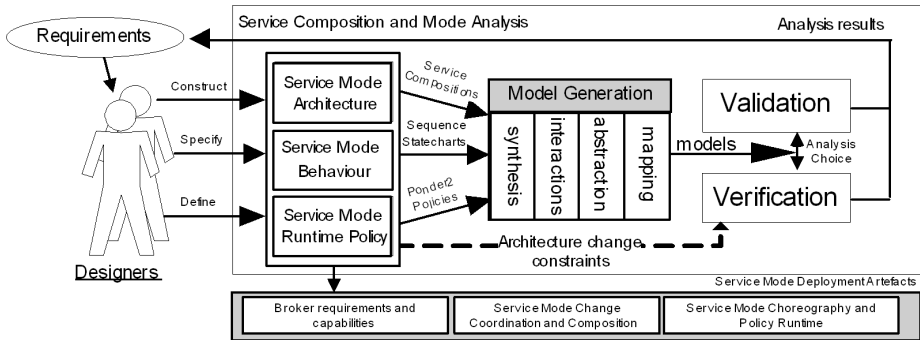
Fig. 1. Service Composition, Analysis and Deployment Artefacts in SelfSoC

3.1 Case Study: Driving Assistance

Our case study is based upon an In-Vehicle Services Platform and the interactions, events and constraints that are posed on this services architecture. One particular scenario focuses upon Driving Assistance, and a navigation system which undertakes route planning and user-interface assistance to a vehicle driver. Within this scenario are a number of events which change the operating mode of the navigation systems. For example, two vehicles are configured where one is a master and another is a slave. Events received by each vehicle service platform, for example an accident happens between vehicles, requires that the system adapts and changes mode to recover from the event. In a more complex example, the vehicles get separated on the highway (because, say, one of the drivers had to pull over), the master vehicle switches to planning mode and the slave vehicle to convoy. However, if an accident occurs behind the master and in front of the slave vehicle, meaning only the slave needs to detour it must somehow re-join the master vehicle route planning. The slave navigation system could firstly change to a detour mode (to avoid the accident), then switch to planning mode (to reach a point in range of the master vehicle), and finally switch to convoy mode when close enough the master vehicle. We now explore a formal way to specify these service mode changes.

4 Modes

4.1 Architecture

A mode, in the context of SoC, abstracts a specific set of services that must interact for the completion of a specific (sub)system task. Note that modes not only determine configuration but also coordination and communication mechanisms. We utilise the work reported in [13] which proposes modes for the Darwin component architecture notation, however, we could equally have used UML 2.0 as the Darwin form of component can now be satisfactorily encoded in UML2.0. A component

```
component AComponent {
    provide PrvServices;
    require ReqServices;
    mode ASetOfModes; }
```

Fig. 2. A Darwin Service Component Specification (graphical and textual)

describing a mode specification, is illustrated in figure 2. Note also that the mode label is a simple attribute of the component, however, related with the mode is a notion of behaviour, for example to describe the expected interaction behaviour with required and provided services whilst the component is in that mode. We discuss behavioural specification of modes in SoC in a later section.

Modes of a composite component depend on their constituent component modes defining a mode-state based service composition. This is useful to determine when a service changes mode, which other services in the composition must also change mode. An example of this mode-state composition is illustrated in two elaborated Darwin models, for a mode change of "planning" to "Convoy", in Figure 3.
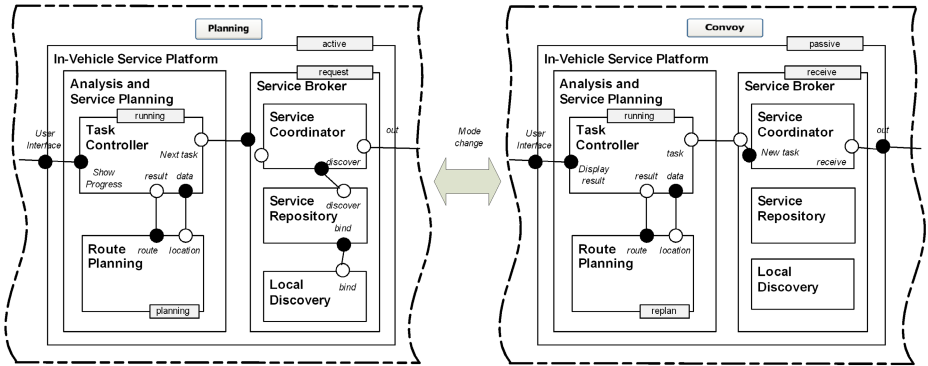

Fig. 3. Vehicle Service Component Architecture Composition and Modes

A service component composition is specified with multiple instances of components and a series of bindings between them. In Darwin this is declared using the **inst** and **bind** constructs respectively. A mode has a designated set of mode labels (e.g. active and disabled listed previously). These are used to identify the current state of mode for a component and derive the permissible behaviour.

4.2   Behaviour

We extend the Darwin component model with a description of the interaction behaviour between component services given a particular mode. Each behavioural
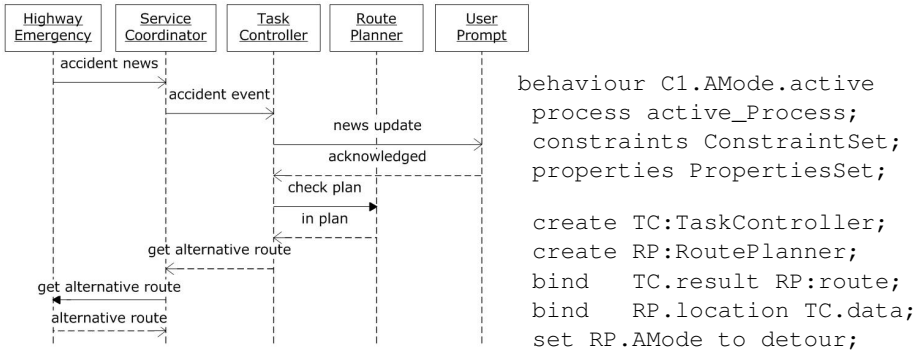
```
behaviour C1.AMode.active
  process active_Process;
  constraints ConstraintSet;
  properties PropertiesSet;

  create TC:TaskController;
  create RP:RoutePlanner;
  bind    TC.result RP:route;
  bind    RP.location TC.data;
  set RP.AMode to detour;
```

Fig. 4. A Scenario leading to Detour Mode Activation

type is determined by the component, mode and mode label. Each type consists of a definition of interaction process (a set of interactions), constraints and properties (Figure 4). Each behaviour type describes a process (in this case we assume a process specifies the interaction scenarios expected for this component and mode), a set of constraints forming rules for when the mode can be changed, which modes are applicable in sibling components when this mode is enabled and other architectural correctness characteristics, and a set of properties which must be upheld. The properties form a source for runtime verification analysis as mode transition occurs.

A process defines a set of scenarios in which the mode can operate. At this stage in our work we have assumed that this is a series of interaction sequences, which when combined, provides a composition of all possible interaction sequences in the given mode. An example in our case study is a switch from a "planning" mode to "detour" mode. From this scenario, a component process can be generated which provides a specification of the mode interactions and their sequence. For example, if the TaskController service is in "running" mode it can perform various things, such as receive events from "news service" or invoke an update of the UserPrompt service. Elaborating on the example given for detour, a client vehicle service can respond to an event (such as that as of an instruction that an accident has occurred in its current route) and the service architecture can change modes of participating services to react accordingly. In the case of normal behaviour, we could assume that the RoutePlanning component should be in "planning" mode unless it is instructed to change to enable a "detour" mode configuration. The permissible reactions must satisfy the architectural correctness, that for example all participating components are in a suitable mode that corresponds to the parent mode change. A series of constraints are specified as to when and how these modes changes can occur.

Self-Assembly. A requirement for adaptable software architectures is that it must easily accommodate the addition of new capabilities (self-assembly). For example, in addition to adding new tasks to activate and control the user

interface of a vehicle, it would be necessary to add new components that encapsulated the software drivers to the physical device and for it to be able to switch modes. Initially a service composition with a designated "mode" would be configured with an initial set of services. As events occur which trigger changes in the composition environment, these sets may expand or shrink in the number of services required and change the required behaviour between services in the architecture. Assembly of the services requires a set of instructions, these are listed below and include service component instance creation, removal, binding and mode changing as follows:

```
create C: T
    -- create component instance C from type T.
delete C
    -- delete component instance C.
bind C1.r to C2.p
    -- connect required port r of component C1
       to provided port p of component C2.
unbind C1.r
    -- disconnect required port r of component C1
set C1.m to val
    -- set mode m of component C1 to val.
```

Note that when an instruction to change mode is undertaken, or indeed other environmental events occur which change the state of the component instances and behaviour, a transition from one architecture configuration to another is made. Such transitions between architecture configurations when a disturbance occurs requires a form of self-healing.

Self-Healing. An extension to self-assembly is the notion of self-healing in which a system attempts to repair it's architecture in response to a disturbance, such as a service failure. A reactive layer is required in self-healing which can perform the transition from one activity plan to another through a coordinator function. This plan relates directly with a change in mode, switching the required or provided interfaces of a service, and the expected behaviour of interactions between services. When a failed service is no longer available, the system attempts to assemble itself into a configuration that satisfies the overall architectural constraints. One of the main considerations in self-healing is the explicit specification of constraints that express permissible combinations of component modes and behaviour (interactions) between these modes such that self-healing could cause a transition to a degraded mode, and the behaviour analysis of these situations.

4.3   Runtime Policy

To specify the architecture constraints on reconfigurations and the component mode behaviour, we utilise the work of Alloy and Ponder toolsets respectively. In particular, the Alloy language [9] can be used to describe the valid structural

and evolutionary constraints of component mode changes in the architecture. It is both a relational and declarative language with allows for partial models and incremental elaboration to be specified. One particularly interesting feature of Alloy is that it can be used to generate sample instances of reconfigured structures that conform to the mode changes specified previously. A set of configuration actions can be generated when the structure of the system is no longer valid with respect to its architectural description due to either a scheduled change (e.g. event or obligation) or a component failure. An Alloy specification consists of component and field declarations (signatures), constraints (facts and predicates), and properties (assertions). Ponder [7] is a language for specifying management and security policies for distributed systems management. In Ponder, a policy is a rule that can be used to change the behaviour of a system. Separating policies from the managers that interpret them allows the behaviour and strategy of the management system to be changed without re-coding the managers. The management system can then adapt to changing requirements by disabling policies or replacing old policies with new ones without shutting down the system. This is a key requirement for a self-managed system. More recently, a redesign of the Ponder framework has produced Ponder2, which enables an extensible management framework based upon the eXtensible Markup Language (XML).

Architecture Constraints. We utilise the work presented in [11], where Alloy models of Darwin are described and a series of constraints are defined for components and their interfaces. In particular, we are interested in modelling the structural constraints (which define a valid set of architectural instances) and evolution constraints (which define the properties of an architecture that can be changed in a transition between two architectural instances). The Darwin components can be described in Alloy with signatures. The basic elements of the Alloy model of Darwin+Modes are the components (Comp), a set of service ports (Port), a set of interface types (Type) and a set of Mode definitions (Modes). As an example, the In-Vehicle ServicePlatform for our case study can be expressed as the following:

```
sig ServicePlatform extends Component {
    ports : set Port, type  : set Type,
    modes : set ServicePlatformModes,
    AP    : one AnalysisAndPlanning,
    SB    : one ServiceBroker }
```

Reconfiguration constraints can be expressed as facts or predicates. Generally, facts always hold whilst predicates are only held when invoked. Predicates are more flexible as predicates can be included in facts, but not vice-versa. For example, if we wish to specify that if the ServicePlatform component is in Convoy mode, no UserPrompt component instance may exist in a trace of the architecture configuration instance we could state the following fact.

```
fact ServicePlatformTrace {
      init[Time:t]
      all t : Time | no UP: UserPrompt
            in SP.modes.convoy }
```

Distributed Management Policy. There are two types of runtime policy construct that we utilise in the mode work using the PONDER language. Firstly, specifying the event (or trigger) behaviour policy and secondly, the constraint policy on when mode changes can occur (i.e. by constraining when obligations maybe carried out). In the first case we are interested in which events cause an architectural change. This is specified in PONDER using the event construct. An event generally triggers an obligation policy. In the following example we specify that given an event raised by the service platform, in this case that an accident event has been received, then an obligation is to switch the related event targets to Convoy mode.

```
event accident(s) = sp.accident_event
type oblig switchToConvoyT (subject s,
        target<taskcontroller> t) {
    on accident(source);
    do t.switchToConvoy(t); }
inst oblig switchToConvoy(taskcontroller) {
    subject tc = taskcontroller;
    target rp = routeplanner;
    do rp.stopplanning;
    do rp.acceptroute;
    do rp.replan;  ... }
```

The event construct is also flexible enough to specify Quality of Service (QoS) characters for service brokering. An example maybe that a service composition requires that a particular interaction occurs within a specified timescale. This can be specified in PONDER using an event, with a when attribute attached to the do clause. e.g. when time.duration(5) - the duration of an action is only valid for 5 minutes. Furthermore, constraints may also be specified as part of an obligation to decide when a change of mode is permissible - given the current state of component instances within the architecture. For example, to specify that the switch to Convoy mode can only be undertaken when the routeplanner is active and is not planning, we could define a constraint as follows:

```
constraint rpConvoyMode(rp) =
    rp.isActive() and !rp.isPlanning()
```

Note that runtime policy constraints could also be derived from some of the architecture constraints specfied in section 4.3. These definitions provide us with a specification for coordination analysis, which we consider in section 4.4.

4.4   Analysis

At this point we have defined three key elements to the self-management of SoC with modes. Firstly, we have a representation of the system (in the form of a component architecture with specified instances and behavioural protocols). Secondly we have defined a set of constraints attributed to the architecture and more specifically, to specify how the architecture can evolve given different mode changes. Thirdly, we have also specified a policy when these mode changes can occur and what actions must be undertaken in the event a change is required. An interesting question in analysis is given different component and mode architecture models, rules for reconfiguration and given an arbitrary failure of a service, can the system reconfigure into a valid mode or can any mode convert in a finite number of steps into another? We split these types of analysis in to two sections, one focusing on the architectural correctness of a mode change, and the other on the ability to reconfigure in to another given a series of configuration tasks.

Considering the architecture analysis, we can link the policy description specified in section of PONDER with that of the archiectural constraints specified in Alloy to determine if such situations exists. This technique is similiar to that reported in [19], where assertions are used to check the correctness of the model, assuming the facts specified on that model.

```
assert validchangetomode_Convoy {
    all sp:ServicePlatform, rp:RoutePlanner |
    (rp in sp.components => (rp !in rp.mode.replan) }
check validchangetomode_Convoy
```

The above assertion states that a RoutePlanning (RP) component should not already be replanning if the component is already in a replan mode. If we found that a counterexample violates this assertion then a predicate can be formed to constrain our model. The following predicate states that preconditions for a mode change to "Convoy" are that both the ServicePlatform component is in mode "active" and the RoutePlanner component is in mode "planning". Additionally, a postcondition is that the UserPrompt component is in mode "disabled".

```
pred mode_convoy [t: time,
  SP : ServicePlatform, RP : RoutePlanner, UP : UserPrompt]{
    SP in Mode.active.t, RP in Mode.planning.t, (preconditions)
    RP.Mode = planning.t, UP.Mode = disabled.t }(postconditions)
```

There are two aspects of behavioural analysis, from the viewpoint of that specified for the service component composition and that of the policy defined in section 4.3. Firstly, safety and liveness analysis can be undertaken on the behavioural scenarios for the component composition. We synthesise the message sequence charts to Finite State Processes (FSP) to construct a Labelled Transition System (LTS) which can be compiled and analysed in the LTSA toolset. As a default function, the LTSA toolset can detect the presence of deadlocks but we
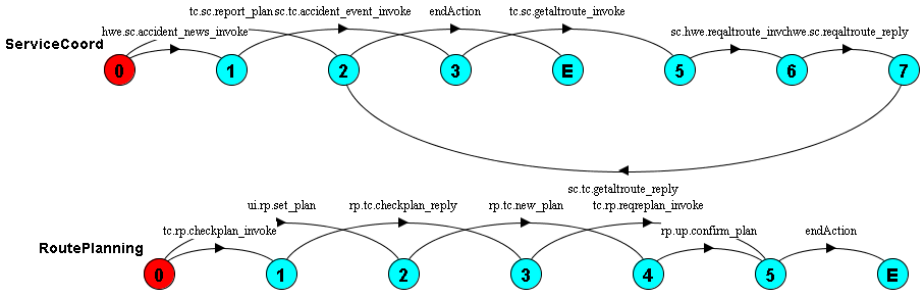
Fig. 5. LTS of ServiceCoordinator and RoutePlanner Process Model

may also specify other analysis properties (such as liveness or fluent assertions) in the FSP model. Figure 5 illustrates LTS models for ServiceCoordinator and Routeplanning services. For example, each state transition specified for the behaviour of the ServiceCoord component represents a message activity specified in Figure 4. Note that shortnames (e.g. tc for TaskController) have been used to represent the component names. Secondly, the tasks involved in mode change, as that of specified in the PONDER policy can be translated to FSP and modelled as a series of processes. In a similiar way to analysing the scenarios for component behaviour, a composition of events, obligations and constraints can be analysed by compiling the FSP and performing model checking on the policy rule set. This model can be used to detect whether there is any conflict between obligations, and also to check whether it is possible to apply the obligations given the architecture model of behaviour created previously.

4.5   Preparation for Deployment Artefacts

A combination of architecture, behaviour and policy specifications can be used to derive the deployment artefacts from the SelfSoC approach. More specifically, the architecture descriptions (section 4.1) can be used to extract a series of services required to be brokered and composed in sets, compositions of these services can be used to generate coordination processes for the behaviour specified in section 4.2 (potentially in an executable orchestration language such as BPEL4WS) and service brokering requirements and capabilities can be generated using the architecture and policy sets described in section 4.3. The process for this generation activities is part of our on-going work and we expect to provide a prototype as part of this.

5   Assumptions and Limitations

In our work, we have assumed that alternative mode scenarios are a choice of sequences for interactions within a service orchestration. To synthesise these to runtime coordination processes is complex, involving inspection of all possible

paths and refactoring conditional elements around these. We plan to explore the synthesis in much greater detail in our future work. We also assume that engineers can express sufficient modes to represent common changes in architecture, the concept of modes is not an exhaustive one, as such it handles those specified and not all changes in all cases (although some of these may be discovered through analysis or indeed, may be expressed as an "all other" rule).

6 Conclusions and Future Work

We believe that the notion of modes helps engineers abstract appropriate elements, behaviour and policy from the services domain, and can facilitate the specification of appropriate control over both architectural change and service behaviour. In this paper we have presented our approach to the modelling and analysis of service-oriented computing component architectures using an abstraction of modes to represent the changes in such an architecture. Our future work will explore how these artefacts are more accurately built and analysed, and also on how our approach can assist in the dynamic invocation of services given component requirements and capabilities. This work has been partially sponsored by the project SENSORIA (IST-2005-016004).

References

1. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business process execution language for web services version 1.1 (2004)
2. Andrzejak, A., Hermann, U., Sahai, A.: Feedbackow-an adaptive work- ow generator for systems management. In: ICAC, pp. 335–336. IEEE Computer Society, Los Alamitos (2005)
3. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web services architecture (ws-a) - w3c working group note (February 11, 2004)
4. Bova, R., Hassas, S., Benbernou, S.: An Immune System- Inspired Approach for Composite Web Services Reuse. In: Workshop AI for Service Composition (ECAI 2006) (July 2006)
5. IBM Corporation. An architecture for autonomic computing, fourth edition. Technical report (June 2006)
6. da Costa, L.A.G., Pires, P.F., Mattoso, M.: Automatic composition of web services with contingency plans. In: ICWS 2004: Proceedings of the IEEE International Conference on Web Services (ICWS 2004), Washington, DC, USA, p. 454. IEEE Computer Society, Los Alamitos (2004)
7. Damianou, N., Dulay, N., Lupu, E.C., Sloman, M.: The ponder policy specification language. In: Sloman, M., Lobo, J., Lupu, E.C. (eds.) POLICY 2001. LNCS, vol. 1995, p. 18. Springer, Heidelberg (2001)
8. Denaro, G., Pezz, M., Tosi, D.: Adaptive integration of third- party web services. In: DEAS 2005: Proceedings of the 2005 workshop on Design and evolution of autonomic application software, pp. 1–6. ACM Press, New York (2005)
9. Jackson, D.: Alloy: A lightweight object modeling notation (1999), http://sdg.lcs.mis.edu/alcoa

10. Foster, H., Magee, J., Kramer, J., Uchitel, S.: Adaptable software architectures and task synthesis for uavs. In: Systems Engineering for Autonomous Systems (SEAS) DTC Conference, Edinburgh, UK (2006)
11. Georgiadis, I., Magee, J., Kramer, J.: Self-organising software architectures for distributed systems. In: WOSS 2002: Proceedings of the first workshop on Self-healing systems, pp. 33–38. ACM Press, New York (2002)
12. Gnesi, S., ter Beek, M., Baumeister, H., Hoelzl, M., Moiso, C., Koch, N., Zobel, A., Alessandrini, M.: D8.0: Case studies scenario description. Technical report (August 2006)
13. Hirsch, D., Kramer, J., Magee, J., Uchitel, S.: Modes for software architectures. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, pp. 113–126. Springer, Heidelberg (2006)
14. McDermott, D.: Estimated-regression planning for interactions with web services (2002)
15. Singh Munindar, P., Huhns Michael, N.: Service-Oriented Computing - Semantics, Processes, Agents. John Wiley and Sons, Ltd., Chichester (2005)
16. Petrie, C.J., Genesereth, M.R., Bjornsson, H., Chirkova, R., Ekstrom, M., Gomi, H., Hinrichs, T., Hoskins, R., Kassoff, M., Kato, D., Kawazoe, K., Min, J.U., Mohsin, W.: Adding AI to Web Services, pp. 322–338. Springer, Germany (March 2004)
17. Van Roy, P.: Self management and the future of software design. In: Formal Aspects of Component Software (FACS 2006), Prague, Czech Republic (2006)
18. Von Bertalanffy, L.: General System Theory: Foundations, Development, Ap- plications. George braziller, New York (1969)
19. Warren, I., Sun, J., Krishnamohan, S., Weerasinghe, T.: An automated formal approach to managing dynamic reconfiguration. In: ASE 2006: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE 2006), Washington, DC, USA, pp. 37–46. IEEE Computer Society, Los Alamitos (2006)
20. Wiener, N.: Cybernetics, or Control and Communication in the Animal and the Machine. MIT press, Cambridge (1948)