Towards a Programming Model and Middleware Architecture for Self-configuring systems

Umesh Bellur  
KReSIT, IIT Bombay  
Mumbai, India  
umesh@it.iitb.ac.in

N.C. Narendra  
IBM Software Labs India, Airport Road,  
Bangalore, India  
narendra@in.ibm.com

Abstract

Today’s distributed computing solutions are mostly static in nature and variations in system behavior have to be compiled into the system in advance, because today’s technology does not permit self-configuring through dynamic rebinding of components. Self-configuring systems need a design approach based on behavioral specification as well as a middleware layer that can use these specifications to dynamically bind system components at run-time. In this paper, we present a service-oriented programming model and middleware for self-configuring systems. Our approach is based on semantic descriptions of components augmented with contextually dependent non-functional requirements for accomplishing the dynamic binding. For this purpose, we present a service-oriented programming model and middleware for self-configuring systems. Our approach is centered on the concepts of dynamic binding and the separation of an abstract description (both syntax and semantics) of a task known as a service from its underlying execution typically provided by software components (see Service-oriented Computing (SOC) [10]). A component is passive and becomes active only by the assignment of resources such as CPU and memory to it by the middleware on which the component executes. This also allows an additional dimension of reconfigurability done by allocating resources depending on non-functional needs which are constantly varying in adaptive environments.

Keywords: Self-Configuring Systems, Service-Oriented Computing, Reconfigurable Programming

1 Introduction & Background

The need for distributed computing systems that can adapt to changes dynamically without recompilation or redeployment of existing components, is well recognized [10,9]. Such a system is also referred to as a self-configuring system & we define it as a (distributed) computing system that can change its behavior based on changes in users’ requirements and/or environmental conditions, dynamically without the need for human intervention of any kind. Reconfiguration could cover an entire continuum - from the completely configurable, where every possible variation in system behavior is modeled, to the completely static and hardwired where there is no reconfiguration possible.

Techniques such as languages and language extensions [7,6], patterns [4], frameworks [3,8], and architectures [1,2] have been developed to facilitate reconfiguration. Most of these techniques however require that the system be designed with specific kinds of variability in mind rather than letting the system deal with it dynamically. In this paper we propose an approach towards adaptability that allows the middleware to make decisions about the components that need to be used dynamically, depending on the needs of the user and availability of the appropriate components. Our approach is centered on the concepts of dynamic binding and the separation of an abstract description (both syntax and semantics) of a task known as a service from its underlying execution typically provided by software components (see Service-oriented Computing (SOC) [10]). A component is passive and becomes active only by the assignment of resources such as CPU and memory to it by the middleware on which the component executes. This also allows an additional dimension of reconfigurability done by allocating resources depending on non-functional needs which are constantly varying in adaptive environments.

Middleware underlying distributed systems today is mostly an inter-object communication backbone supplemented by often used services such as transaction processing, asynchronous messaging and persistence. Our middleware needs to also provide for the discovery and dynamic binding capability along with dynamic resource allocation. Please note that although CORBA[CORBA Reference] does provide some level of dynamic binding, it suffers from limitations such as requiring exact syntactic matches for interfaces, which violate the loose coupling requirement necessary for dynamic binding. We term the points at which (re)binding occurs, variability points. This idea of using abstract descriptions and variability points forms the basis of our programming model, which we call reconfigurable...
programming.

The rest of this paper is organized as follows. Section 2 introduces the concepts underlying our self-configuring system design approach, and illustrates them via a simple yet realistic running example. We present our reconfigurable programming model in Section 3, which unifies our system design concepts introduced in Section 2. We conclude our running example in Section 4. Some related work is discussed in Section 5. The paper concludes in Section 6 with suggestions for future work.

2 Middleware Design for Self-Configuring Systems

The starting point is usually a user requirement which is abstractly represented as a service, and is to be realized via a sequence of actions, termed an execution sequence. An action can be the execution of a software component or a call to another service. A component is a passive implementation of a set of features with a specific interface and becomes “live” only when enabled with resources such as a thread (representing underlying physical resources such as a CPU and memory). Component execution may in turn spawn the need for additional “services” which will yet again be mapped to sequences of component actions. This will recurse till no more dependencies exist at which time the recursion unravels back to the starting execution. The key point to note here is that the realization of a service involves two steps: selection of an appropriate component that can meet the service’s functional needs; and appropriate resource allocation to satisfy the service’s non-functional needs such as performance, availability, etc.

2.1 Middleware Architecture and Operation

We motivate our concepts by showing how our system middleware satisfies a user requirement. The modules of the middleware are illustrated in Figure 1. In this diagram, the numbers indicate a sequencing of actions to satisfy a user requirement. The main operations involved are: discovering components, binding components and composing components in an execution sequence in order to satisfy the user requirement. The middleware therefore is composed of the following modules: the Component Repository, the Component Discovery and Publishing module, the Service Configurator and the Resource Allocator. Components are initially published into a distributed component repository. The Service Configurator responds to a requirement initially by mapping the (abstract) user requirement to a concrete component which can get the processing started. The component is then allocated appropriate resources such as threads, CPUs, database connections etc. to turn the component into a live execution engine. The component executes till such time a variability point is encountered which causes it to trap back to the Service Configurator with a variability point description and the loop is now repeated till the execution sequence is completed. During reconfiguration, the Service Configurator will either map to a different component based on the reconfiguration needs, and this could trigger a change in the execution sequence itself.

![Figure 1: Self-configuring System – Middleware Modules & Operation](image)

2.2 Realizing Self-Configuration

In order to model the execution dependency among the components, we model each of them in the well-known IOPE form, consisting of Inputs, Outputs, Preconditions and Effects. Component composition or chaining is driven off of the outputs of a component and the inputs of another along with the appropriate constraints. The execution sequence is represented as a dependency graph, where each component is represented as a node and each variability point is represented as an edge. The unique feature of the dependency graph that renders it useful for us is that it is built on the concept of interface automata [13], which is a temporal model for representing software component behavior, and is an extension of finite state automata. The key feature of interface automata is that they are equipped with a composition concept and a formalism by which the safety of component composition is
evaluated. In particular, interface automata are an optimistic approach to composition, wherein two components are compatible if there is some environmental context that can make them work together. In such a composition, the output of one component is the input to the other component. Such a property makes component composition highly amenable to reconfiguration, since changing environmental conditions are one of the primary reconfiguration triggers. Interface matching is based on type matches that include polymorphism along with the ontology equivalencies which allow disparate type systems to come together.

The environmental conditions play an important role in triggering architectural adaptation, and are represented as contextual information. We have identified the three levels of contextual information [16] - E-context (environment level), S-context (service level), R-context (resource level). The E-context for the preceding component in the execution sequence (i.e., consumer) would be its device and connectivity information (e.g., PDA with limited bandwidth & display); whereas, for the succeeding component (i.e., provider), it could be the execution infrastructure needed (e.g., J2EE component would require a JVM, etc.). The S-context for the consumer would be the QoS level required for execution; whereas, for the provider, it could be the QoS range supported. The R-context for the provider would be the resources needed by the provider to supply the supported QoS range and has no analog on the consumer side.

Each component in the execution sequence possesses its own contextual information at all three levels; in order for component composition in the execution sequence to be successful — in other words, for a preceding component to successfully “hand off” execution to its successor as per the execution sequence — their respective contextual information need to match. This matching becomes especially crucial during reconfiguration, due to the dynamic selection and binding property of our system. During the handoff or dynamic binding, contextual information can be thought of as originating from two sources — one from the component providing the service and one from the component requesting the service.

Figure 2 below shows a pictorial description of our Matchmaking Engine (which will be part of the Component Discovery & Publishing module of Figure 1), whose architecture is leveraged from IBM’s Web Service Matchmaking Engine (WSME) (see [23] for details).

3 Putting It All Together – Reconfigurable Programming

Our programming model, reconfigurable programming, will represent each component via its IOPEs. In a manner consistent with the interface automata model, the reconfigurable programming approach would also link the IOPEs for each interface with the environmental conditions under which the component would bind to other components for providing its respective functionality; these will be variability points.

Instead of defining new concepts, our model will reuse from existing formalisms wherever applicable, as will be clear below.

3.1 User Requirement Formalism

We assume that the user will specify the desired functionality via the inputs and outputs and the relation between them, in the following format: \( \text{Req} = (A_i^{req}, A_o^{req}, D_{req}) \), where \( A_i^{req} \) is the set of input data, \( A_o^{req} \) is the set of output data, and \( D_{req} \) is a set of tuples \( <A_i^j, A_o^k> \), expressing the dependency between the input and output data. This request will provide the input for the Service Configurator module to define the appropriate execution sequence meeting the input and output conditions of the user requirement.

3.2 Execution Sequence

The execution sequence is represented as a dependency graph \( G = (V, E) \) as in [11]. The set \( V \) of nodes represents a component to be executed, while the edge \( E \) represents a pairwise execution dependency between two components. The execution dependency can be of the following forms – sequential, conditional and parallel. A special type of sequential or
conditional dependency, called composition dependency, exists when the output of one component becomes the input to the other component.

3.3 Component Representation

We represent each component via an interface automaton, thus [11]: \( C = \langle V_p, V_p^{init}, A_I^p, A_O^p, A_E^p, T_p \rangle \), where \( V_p \) is the set of states of the automaton, and \( V_p^{init} \) is the set of initial states.

\( A_I^p, A_O^p, A_E^p \) are mutually disjoint sets of input, output and internal actions executed by the component. Together they form the set of all actions. \( T_p \) denotes the set of transitions \((v, a, v')\), where the state of the component changes from state \( v \) to a state \( v' \) via an action \( a \). In other words, the action \( a \) is said to be enabled at the state \( v \).

Two interface automata \( P \) and \( Q \) are composable if their execution actions are disjoint, except that an input action of one may coincide with an output action of the other. Hence the necessary condition for two components to possess composition dependency, is that their respective interface automata be composable.

We see that the necessary condition for composition dependency represents the Inputs and Outputs part of the IOPE-based component description. However, composability of interface automata is also context-dependent, and contextual information is needed in order to determine the Preconditions and Effects for the components to be composed. This contextual information is created by the Service Configurator and Resource Allocator modules of Figure 1 while generating the execution sequence from the user requirement.

3.4 IOPE Representation

The Inputs are the input data needed by the component for execution. Similarly, the Outputs are the output data created by the component and which could serve as potential inputs for succeeding components.

Representing the Preconditions and Effects is more complicated, since they have to model aspects of the semantics of the execution sequence. Preconditions need to model the following: initial states of the component, environmental conditions at the time of execution start, resource availability information for the component (represented in the R-context of the preceding component). Similarly, Effects need to model: the states of the component once it finishes execution, environmental conditions at the time of execution completion (E- and S-contexts), and resource availability information (R-context).

3.5 Variability Points

As already introduced above, a variability point is the connection or a dependency between the components in the execution sequence where reconfiguration in our system can be effected. We have identified two main reasons for this reconfiguration [15] – changing user requirements (functional adaptation) and variations in environmental and resource conditions as represented in contextual information (architectural adaptation).

Functional adaptation would result in changes in the execution sequence, which would change the Input and Outputs to be verified for the dynamic selection of the next component in the sequence. Architectural adaptation would be necessitated by changes in E/S/R-contextual information, which would be evaluated via the Preconditions and Effects in the components. This would also affect the selection of the next component in the execution sequence, and many even trigger functional adaptation also.

For example, let a component currently under execution use a database, with further database accesses not possible due to too many requests. If the succeeding component to be executed also needs to access that database as a resource for its execution, then this component cannot be executed. This should force the Service Configurator module to obtain an alternate database for the component; or modify the execution sequence itself so that some other component in the execution sequence can be executed. Once the R-context variable value for the former component changes, i.e., the database becomes accessible, the former component can then be executed. Hence such a reconfiguration would trigger functional adaptation, by changing the execution sequence itself.

4 Related Work

In [17], a method for augmenting code with specification information so as to facilitate runtime component composition is presented. This is complementary to our work; in the future we expect to leverage from the approach in [17] for completing our reconfigurable programming model. The issue of determining the “safety” of adaptation composed of middleware services is
discussed in [12]; “safety” here means the extent to which the adapted composition is able to meet the changed user requirements. In our paper, we subsume this notion of safety via checking component composability vis-à-vis user requirements via a combination of the interface automata and contextual concepts.

The Aura project [20] has also explored adaptation issues similar to ours; indeed, our context-oriented architecture for pervasive computing systems described in [15], closely resembles the Aura model. The significant difference is that the Aura model is so far focused on a single user implementing a workflow execution sequence, whereas we do not impose any such restrictions either in [15] or in this paper.

A service-oriented component model for OSGi (Open Service Gateway Initiative - http://www.osgi.org/) is presented in [19], where the problem of automating service dependency management is discussed. This model is based on an assumption similar to that of ours, viz. software components of the future will need to be context-aware; hence contextual information is to be used for component composition.

The “design by contract” approach of Eiffel [22] provides an assertion mechanism by which preconditions and effects of software components can be specified and verified. Our approach is similar, but more comprehensive, since it explicitly incorporates contextual information.

The design for a reconfigurable context-sensitive middleware for pervasive computing systems has been presented in [14]. This middleware facilitates context-sensitive ad-hoc communication among the components, whereas we are more focused on satisfying more precisely specified user requirements.

5 Conclusions and Future Work

In this paper, we have shown that the static binding and hardwiring properties of today’s middleware renders them incapable of supporting self-configuring systems, since extensive re-programming would be needed to accommodate constant changes. We have argued for the employment of service oriented computing (SOC) principles as a means of overcoming this issue. Via the usage of dependency graphs built on the interface automata concept, we have shown how reconfiguration can be modeled. We have also presented our ongoing work on our programming model – reconfigurable programming – for building and maintaining self-configuring systems.

Our work is still ongoing; hence future work would consist of completing and implementing our reconfigurable programming model, and demonstrating it on real-life examples. We will also be aligning our model with the Autonomic Computing vision [18], to incorporate aspects such as self-healing, self-optimizing and self-protecting.

Acknowledgements

The authors would like to thank Yogesh Murarka, Zakaria Maamar and Giovanna Di Marzo Serugendo for their feedback. The second author also wishes to thank his manager, K. Muralidharan, for supporting his work.

IBM and Autonomic Computing are trademarks of International Business Machines Corporation in the United States, other countries, or both. Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both. Other company (i.e., non-IBM), product and service names may be trademarks or service marks of others.

References

security, and dynamic configuration with the dynamic TAO reflective ORB. In *International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware’2000)*, number 1795 in LNCS, pages 121-143. Springer-Verlag, April 2000.


