An Intelligent Connector Based Framework for Dynamic Architecture*

Xin Peng, Wenyun Zhao, Liang Zhang, Yijian Wu
Computer Science and Engineering Department, Fudan University, Shanghai, China
{pengxin, wyzhao, 032021166, wuyijian}@fudan.edu.cn

Abstract

Component based software development provides an architectural way for dynamic reconfiguration. Interactions between components are explicitly represented by connectors, then reconfigurations can be performed on the macro-architecture and be non-intrusive on the components. In this paper, we introduce agents to act as intelligent connectors and the global coordinator. An intelligent connector based framework for dynamic architecture is proposed, in which the global agent and connector agents can perform auto-adaptation on the system-level and component-level respectively. Ontology is introduced to provide the common base for communication and auto-adaptation. The domain ontology also provides a knowledge base for event inference and service assembling. So the framework can offer more flexible and semantic ways for dynamic reconfigurations.

1. Introduction

Dynamic reconfiguration has been an essential feature for critical and long-running applications. Two classes of applications are stated in [1] to require dynamic reconfiguration as an enabling technology: adaptive systems and highly available systems. Ad hoc approaches to dynamic reconfiguration are used in practice but these tend to be system specific and therefore limited [1].

Component-based software development provides an architecture-based way for dynamic reconfiguration. In such systems interactions between components are explicitly represented by connectors, then the reconfiguration can be enforced on the macro-architecture and be non-intrusive on the components. Examples can be addition, removal or modification of existing connections and components [2].

Reflective architecture is a common concept in existing architectural approaches for dynamic reconfiguration (e.g. [1] [2]). The motivation for reflection is to promote openness and flexibility [1]. There are both requirement-driven and context-driven software evolutions [3]. Requirement and context are both semantics-rich factors. However, in most existing implementations, reconfigurations are realized in a non-semantic or user-driven way. To gain more flexible and semantic reconfiguration ability we introduce agents as connectors. Based on these intelligent connectors an implementation framework is proposed, in which each component is wrapped by a connector agent. Agents take on dual roles of communication delegates and reconfiguration aids. Communications between components are realized by agents through ACL (Agent Communication Language). In the process of reconfiguration, agents can decide and perform the adaptation.

There are both adaptable and self-adaptive systems [4]. Adaptable systems provide open ability of dynamic reconfiguration for external actors. Self-adaptive systems can adapt themselves to requirements or the environment, such as the context-driven evolution described in [3]. In our framework, both styles of reconfiguration are supported. Self-adaptation is performed by the agents (connector agents and the global agent), which can decide the reconfiguration actions based on the domain ontology and reconfiguration policies.

The remainder of this paper is organized as follows. In section 2, we introduce and comment on related works. The general structure of the intelligent connector based framework is presented in section 3. Details about the component/communication model and the implementation of dynamic reconfiguration are described in section 4 and 5. Section 6 provides information on implementation and evaluation of the framework. Finally, in section 7 we present our conclusions and discuss our future work.

* Supported by the National Natural Science Foundation of China under Grant No. 60473061; the National High Technology Development 863 Program of China under Grant No. 2004AA112070, 2004AA113030.

2. Related works

A layered architecture based on the separation between computation and coordination is proposed in [5] to achieve higher levels of auto-adaptability. The coordinating units are represented as first class entities, so components and coordinators can evolve independently. This concept is common in architecture-based adaptable systems.

DYVA, a virtual dynamic reconfiguration machine for component-based applications is presented in [2]. Three levels of kernel, meta-level and base-level are identified. However, only physical events, such as memory, method call, are concerned and only some primitive policies are supported.

A layered multi agent architecture [6] is proposed to enable dynamic reconfiguration for highly autonomous systems. However, in our framework the main units are components, and agents are only introduced to support component-based dynamic architecture. Furthermore, in that architecture, the reconfiguration is driven by commands and auto-adaptation is not supported.

3. Framework overview

A component is a unit of composition with contractually specified interfaces and explicit context dependencies only [7]. In our framework, a component is described by two kinds of ports: service ports and request ports. Ontology-based descriptions of all the ports should be given before the component can be integrated into the system. The system architecture will first be initialized on the original architecture specified by the user. Then the architecture can evolve continually, for example component instances can be created, removed or replaced. So the software architecture referred in our framework is an instance-level architecture, which means the basic unit in the architecture is not conceptual component but component instance.

3.1. The domain ontology

Domain ontology clarifies the domain's structure of knowledge and enables knowledge sharing [8]. In our framework, the domain ontology provides a common description and communication base for components. For example, messages between components can be encoded on the ontology, and a component can find required components from the directory facilitator by ontology-based descriptions of related ports.

There are three different kinds of concepts in our domain ontology, namely action, object and event. Action concepts are identified business actions in the domain, they are usually verbs. Object concepts represent business entities, which are identified to be the objects of specific actions. These two kinds of concepts can be used in semantic description of components and service matching. Event concepts include all the logical events involved in the event model. Each of the three kinds of concepts constructs a concept net from its own aspect.

An example of the domain ontology of education system is described in figure 1, in which only action concepts are involved for simplicity. A virtual action "Action Root" is introduced and all the action concepts are organized as a tree according to two relations of extension and constitution. For example, the action "login" is constructed by "input account" and "treat login". Action "validate user" and "refuse login" are two different manners of "treat login". The former represents a usual validation and the latter means to refuse the login due to certain causes (e.g. a high load). The notation "+" denotes a sequential construction of sub-actions. Other relations, such as parallel construction, are also supported.

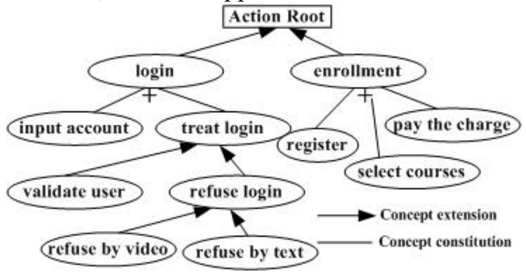

Figure 1. An example of the domain ontology

3.2. Architecture of the framework

Figure 2 presents the architecture of the agent-based framework and shows that each component is wrapped by an connector agent, which is identified by an unique AID (Agent ID). The connector acts as interaction and reconfiguration proxy for the corresponding component, operations of which can be:

- *Communication proxy*. The components do not interact directly but communicate through the connectors by ACLs constructed on the domain ontology. The connector is responsible for the conversion between ACLs and component messages. It also manages the input/output message queue for the component.
- *State management*. For stateful components, the internal state should be maintained in each session. In our framework, each connector contains a state space, which can store and manage multiple state schemas for the component.
- *Local event handling and auto-adaptation*. The component-level sensor is implemented in the

connector agent, which monitors the component/connector state and collects local events. Component-level auto-adaptation, called local reconfiguration is also enforced by the connector based on local policies.

System-level reconfiguration, such as addition, removal and replacement of components, are implemented by the global agent, which also takes three other responsibilities: management of component instances, naming and directory service, global event monitoring and aggregation. The service of "Yellow Pages" is provided by the DF (Directory Facilitator), which maintains a complete directory of all the running component instances and corresponding AIDs. Then runtime discovery of services can be supported by the DF. Other responsibilities are related with reconfigurations and will be discussed in section 5.

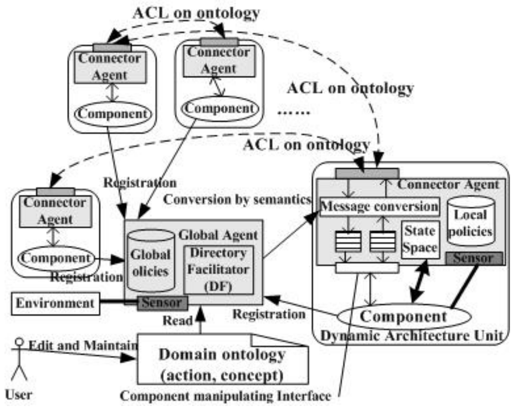

Figure 2. Framework Architecture

4. The component/connector model

In our framework, components and connector are relatively independent units. Components correspond to "core", "stable" entities in the sense that auto-adaptation should not be intrusive on their implementation [5]. Connector agents are responsible for communications between components. Then the architectural reconfigurations can be enforced on the coordination level in a way transparent for components.

4.1. Model overview

Structural reflection is concerned with exposing the structure of components and connectors, while behavioral reflection is concerned with system activity, such as component interaction [1]. In our framework, the structural reflection is provided by the component model, which makes it possible for the connector to inspect the internal structure of the component such as properties. Interactions between components are implemented by connector agents through ACLs, so the behavioral reflection can be provided by connector agents. Connections to service ports of other instances are recorded in the connection register with AIDs of corresponding connector agents. All the components should implement the basic interfaces of dynamic creation and removal.

Figure 3 presents the component/connector model. It can be seen from the figure that besides service ports and request ports the component model must provide mechanisms for event provision and state management.

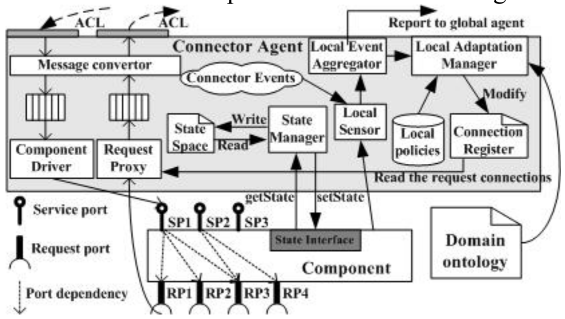

Figure 3. The component/connector model

4.2. Ontology-based description of component

The semantic description of a component consists of ontology-based specifications of all the service and request ports. Each port can be described as the 2-tuple of <*action*, *object*>, in which *action* is the action concept and *object* is the object concept in the domain ontology. They denote the operation and the operated object respectively. For example, <register, graduate> represents the service of registering for graduates.

The semantics of request ports can be exact when it corresponds to a concrete action or variable when it is represented by an abstract action. For example, if a request port is represented by <refuse login, user> according to figure 1, then the semantics is variable in that the real provided service can be "refuse by text" or "refuse by video". So request ports can have some variability according to the domain ontology and the real semantics needs to be fixed at runtime.

4.3. The request proxy

A component can have both service and request ports (figure 3) and the provided services may depend on services of other components through the request ports. This kind of relations is represented by port dependencies. For example, in figure 3, service port SP2 depends on request ports RP3 and RP4, while SP3 does not depend on any request ports. The request proxy implements the service requests for request ports according to the connection relations recorded in the connection register. In reconfiguration times the register can be modified to adapt the connections. Only those request ports involved in certain active service are recorded and used. For example, in figure 3, if services provided on SP2 are not requested by any other instances, request port RP4 will not appear in the connection register.

The runtime discovery of services is provided by the DF, where all the instances are registered with ontology-based descriptions of ports. So when the connector needs to find a new service-provider, it can query the DF according to description of the request port. AID of the connector agent corresponding to the found provider component will be returned and recorded in the connection register of the requesting connector. Then the component can communicate with the service-providing connector agent by AID. If there is no runtime instance of the requested component, the global agent will create an instance of it.

The connector can also provide more intelligent proxy services for the component by assembling multi-services for request ports. For example, if a request port needs the "enrollment" service described in figure 1 and there is no component providing the service, then the request proxy can assemble the three services of "register", "select courses" and "pay the charge" instead in a way transparent to the component. The precondition is that those three services can be provided by several other components. The dynamic assembling is based on the service definition in the domain ontology. In that example, "enrollment" is a sequential combination of the three services.

4.4. The communication process

Message handling is a main job for the connector, including message conversion, message queue management, and component proxy. The message convertor takes the responsibility of converting messages between ACLs and component messages. Then the component message will be put into the queue and be scheduled sometime. Finally the component driver will pass the message to certain service port and a service process of the component is started. In the service process, the component many need services of other components. Then the component can call the request proxy through corresponding request port and the request messages will be converted and sent by the connector.

It can be seen that the connector can enforce the coordination logic in a way transparent for the component. This feature provides the essential mechanism for reconfigurations.

4.5. State manager

A component instance may switch between several service sessions, and each of them may contain an independent state schema. In our framework, the state persistence and scheduling are implemented by the state manager. The component model provides two related meta-level interfaces of *getState* and *setState*, through which the state manager can get current state of the component then save it in the state space, and read certain state schema from the state space then set current state of the component according to it. All the effective state schemas can be stored in the state space and indexed by session ID. When a session switch happens, the state manager will save current state schema of the instance and restore corresponding state from the state space. Usually the meta-level state interfaces are supported by the language-level reflection, such as the Java reflection in our prototype.

When replacement of component instance happens, the connector instance will remain and change to serve the new instance. State transfer between component instances of different types is much more difficult in that their internal structures may be quite different. Currently, this problem can be partly resolved by the ontology-based mapping between stateful properties of different components types.

5. Dynamic reconfiguration

Both user-driven reconfigurations and self-adaptation are supported in our framework. The former usually corresponds to live upgrade (e.g. replacement of instances) and can be decided by users. The latter exhibits the adaptation of the system to the context (physical environment or business environment) and requirements. User-driven reconfiguration is relatively easy to achieve and can be directly supported by the dynamic component/connector model (see section 4).

A general process of self-adaptation is presented in figure 4. Event handing is the trigger of self-adaptation, then reconfiguration decisions can be made according to event analysis and the policies. Reconfiguration will first be performed on the meta-level and then be mapped to the base-level, which represents the concrete application. Reconfiguration policies specify the user requirements of reconfiguration, which are modeled by the user.

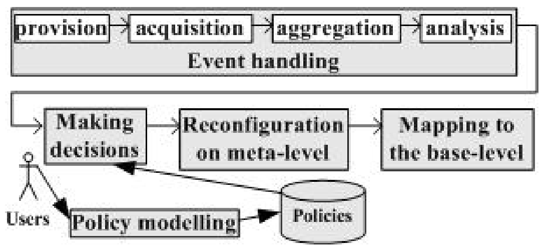

Figure 4. The general process of self-adaptation

In most of the existing frameworks or approaches for self-adaptation, a system-level reconfiguration manager serves as the kernel of self-adaptation. In our framework, agents are introduced to support both component-level and system-level reconfigurations. It can be seen from section 4.3 that connections between instances are established according to descriptions of ports. So two levels of auto-adaptation are distinguished based on semantics of request ports:

- Instance-level. In the case that semantics of request ports remains unchanged, reconfigurations can also happen on the instance-level. For example, a component instance may reconnect to another new instance for higher performance or the old instance is no longer available. This kind of reconfigurations is performed on the component-level by the connector agent.
- Semantics-level. There are also requirements for the runtime semantics of components to change. Supposing a component C has a request port RP with the semantics of "treat login" according to figure 1, then for an instance of C the runtime semantics of RP may change from "validate user" to "refuse login" when the length of the input queue exceeds certain number. This kind of reconfigurations is performed by both the connector agent and global agent. The former enforces reconfigurations according to local requirements, while the latter enforces reconfigurations according to global requirements.

5.1. The overall structure of reconfiguration

Events handling is the base of auto-adaptation, and system-level handling and component-level handling are often separated (e.g. the system monitoring and local monitoring discussed in [1]). In the conceptual model proposed in [3] five tasks are identified to be performed by context-aware entities: context provision, context acquisition, context aggregation, context analysis, and context notification. In our framework, the two levels of events handlings are taken by the global agent and connector agents respectively. And related tasks are performed by components, sensors, aggregator, adaptation manager respectively.

The overall structure of reconfiguration is presented in figure 5. In the component-level, the connector agent monitors local events and performs local adaptations (see also figure 3). Local sensor acquires physical events (e.g. property changes or method invocation of the component) from the component and the connector agent itself. These physical events are joined and converted by the local aggregator to logical events, which have certain basic business meaning. Then the local manager can decide and perform local adaptations according to local policies. Local logical events will also be reported to the global event aggregator. The global aggregator also acquires events from the global sensor, which monitors the system-level environment, such as load of the CPU. All the global events and local events are aggregated and sent to the global manager, which can decide and perform global adaptations according to global policies.

5.2. Realization mechanisms

In the component/connector model presented in section 4 several key mechanisms are integrated to support dynamic reconfigurations, including separation between computation and coordination, ontology-based communication, message queues and state space. Reconfiguration actions include addition, removal, replacement of component instances and connections. The dynamic creation of component instances is the task of the component/connector factory (figure 5). When a new instance created, instance of corresponding connector will also be created and recorded in the DF. After that, the new instance can run and serve for other instances. In the removal of instances, the instance will first be passivated to stop receiving new request messages and related instances will need to find instances to serve. However, it will still complete the current service process before being really removed. After the service process ends, the instance reaches a safety state and can be removed from the runtime platform and the DF. When replacing an instance, the connector will first stop dispatching messages to the component. At the same time, the new component instance can be created. It is different from separate instance creation in that only the component instance is created and the old connector will be reserved. After the old instance reaches the safety state it can be really replaced by the new one and the old connector will connect to the new component instance and resume the message handling. In the process of replacement, request messages can still be sent to the connector and be buffered in the queue. Connection-related reconfigurations are implemented by the connector agent. When local adaptation manager modifies the connection register, the actual connection relations of the component are changed along with the overall architecture.

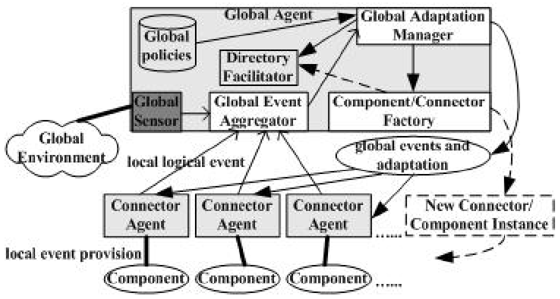

Figure 5. The overall structure of reconfiguration

5.3. Local reconfiguration

Local reconfiguration is performed by local sensor, local event aggregator and local adaptation manager (figure 3). The local sensor collects both connector events and component events. Data of connector events can be acquired from the internal structure of the connector agent, such as length of the queue, etc. Component events include property changes and method invocation of the component, which can be acquired from the state space and the component driver. The local sensor collects data and produces events after each invocation. For example, it will check values of the monitored properties and decide whether concerned changes have happened. Local aggregator receives physical events from local sensor and joins them to produce logical events. These logical events will be analyzed by local adaptation manager and corresponding adaptations may be enforced.

Local reconfiguration performs dynamic adaptation according to requirements of the single component. The basic responsibility of local reconfiguration is to satisfy the service requirements required on the request ports. For example, when a requested component instance is no longer available, current connector must find another instance for the request port or assemble several services. Another example can be requirement-driven reconfiguration. If current connector satisfies a request port by assembling several services according to the domain ontology, then when the definition of the requested service changes, the connector will adapt the assembling logic to be consistent with the definition, which is part of the user requirements.

There are also user-specific requirements of reconfiguration for a component. For example, a login component may need to ensure the response time within an acceptable range, so when the length of the requesting queue exceeds certain value the component may stop sending requests for user validation but change to request the service of playing a small video of prompting the user to try later. Then a reconnection will happen. This kind of reconfiguration should be specified as local policies by the user.

5.4. Global reconfiguration

The responsibility of global reconfiguration is to satisfy the system-level requirements (e.g. performance, reliability or specific business logic). This kind of requirements usually involves multiple instances and can not be satisfied by single component. For example, when the occupied bandwidth exceeds certain value the global agent may adapt all the request ports of "refuse by video" to "refuse by text". Then all the instances of certain components will be affected.

The global aggregator aggregates events from both component/connector instances and the global sensor, which monitors the overall environment. Then the global adaptation manager can decide and perform adaptations. Instructions of global adaptation are broadcasted to every related instances and new component/connector instances may also be created.

5.5. Specification of policies

In our framework, reconfiguration policies have the form of trigger/reaction like the mode in [5]. The triggers are logical events produced in local or global aggregation. Component-level aggregation transforms physical events to local logical events. The rules of local aggregation are defined as "PEC$\rightarrow$LE", where PEC represents conditional operations on physical events and LE is a logical event defined in the domain ontology. The global aggregation is performed on the event net modeled in the domain ontology to infer more complex events from basic logical events. All the logical events produced in aggregation can be triggers of policies. Reactions of the policies can be of both instance-level and semantics-level. Instance-level reactions include reconnection, instance creation, etc. Semantics-level reactions are adaptation of request semantics of certain components.

In this way, events can be aggregated and treated gradually and the whole structure of event handling and self-adaptation can be more clear and easy to use.

6. Implementation and evaluation

FDDyArc, a Java-based implementation of the framework is under development. Fruits of two open-source project, namely JADE and Jena, are used in our prototype. JADE (by Telecom Italia Lab) [9] is an open source platform for Java-based implementation of multi-agent systems and complies with the FIPA specifications. We have implemented the global agent and the connector model on the agent model of JADE. Jena [10] (by HP Labs Semantic Web Programme) is a Java framework for building semantic web applications. We implement access interfaces of the domain ontology on the base of APIs provided by Jena.

The component model is implemented as Java class or class cluster. At runtime, each connector agent runs in a separate thread and corresponding component instance runs in the same thread. JADE supports further parallelism within each agent by the concept of behaviors. So the component instance can execute in parallel with other behaviors, such as the message manager and local sensor, in the same thread. Currently, we have implement supports for user-driven reconfigurations and some basic policies. Support for user-defined policies is still underway. Figure 6 shows a snapshot of the monitoring tool, by which uses can view the runtime architecture and reconfigure it.

Knowledge share among distributed applications is still a difficulty. However, it is entirely possible to construct domain ontology for specific applications. Components can be developed and described according to the common component model and the domain ontology. Then these components can be integrated into the agent-based framework and support the architecture evolvement at runtime.

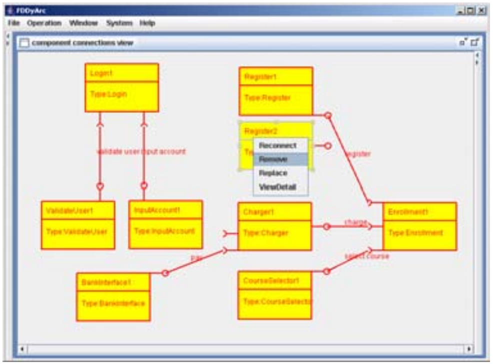

Figure 6. The monitoring tool

7. Conclusion and future work

In this paper, we have presented an intelligent connector based framework for dynamic architecture, which can implement auto-adaptation in a semantic way. The framework also presents several other novel features. First, both user-driven reconfiguration and auto-adaptation are supported. Second, ontology is introduced as the common base for communication and auto-adaptation. Third, the auto-adaptation can happen on both instance and semantics level, so adaptation requirements on various levels can be satisfied.

How to ensure the overall behaviors to be consistent in the complex reconfigurations remains unresolved for us. There are also needs for the global agent and connector agents to negotiate on policies. This negotiation can help to ensure the overall consistence. In the current implementation, ontology-based inference for event handling and dynamic assembling of services is still limited. All these issues will be further explored in our future work.

8. References

[1] Jamie Hillman and Ian Warren. An Open Framework for Dynamic Reconfiguration. In 26th *International Conference on Software Engineering (ICSE'04)*, 2004.

[2] Abdelmadjid Ketfi and Noureddine Belkhatir. A Metamodel-Based Approach for the Dynamic Reconfiguration of Component-Based Software. In 8th *International Conference on Software Reuse (ICSR'04)*, 2004.

[3] Javier Jaén Martínez and Isidro Ramos Salavert. A Conceptual Model for Context-Aware Dynamic Architectures. In 23rd *International Conference on Distributed Computing Systems Workshops* (ICDCSW'03), 2003.

[4] Jim Dowling and Vinny Cahill. The K-Component Architecture Meta-Model for Self-Adaptive Software. In Proceedings of *Reflection 2001*, LNCS 2192.

[5] L.Andrade and J.L.Fiadeiro. An Architectural Approach to Auto-Adaptive Systems. In 22rd *International Conference on Distributed Computing Systems Workshops* (ICDCSW'02), 2002.

[6] Eiichi Inohira, Atsushi Konno and Masaru Uchiyama. Layered Multi Agent Architecture with Dynamic Reconfigurability. In *2003 IEEE International Conference on Robotics & Automation*, 2003.

[7] C. Szyperski and Component *Software: Beyond Object-Oriented Programming*, Addison Wesley 1998.

[8] Chandrasekaran, B. and Josephson, J.R.; Benjamins, V.R.. What are ontologies, and why do we need them? *Intelligent Systems and Their Applications*, IEEE Volume 14, Issue 1, Jan.-Feb. 1999 Page(s):20 – 26.

[9] JADE home. http://jade.cselt.it.

[10] Jena home. http://jena.sourceforge.net.