Specifying Dynamic Software Architectures with Dynamic Description Logic

Zhikun Zhao
School of Computer & Information Engineering, Shandong University of Finance, Jinan, Shandong, China, 250014
zhaozk@sdfi.edu.cn

Wei Li
School of Information & Communication Technology, Central Queensland University, Rockhampton, Australia, 4702
w.li@cqu.edu.au

***Abstract*—Dynamic software architectures provide support for building long running and reconfigurable applications. Formal specification is useful to the design of correct and robust dynamic software architectures. In this paper, dynamic software architectures are specified with dynamic description logic. Dynamic description logic inherits the expressiveness and decidability of description logic and it has the ability to represent state changes. Reconfigurable dataflow model, which is an extension of the widely used dataflow model, is used as the architecture meta-model. Architectures, reconfiguration operations, and reconfiguration plans are represented in a unified framework from the view point of data flow. Three levels of constraints have been proposed to aid designers in predetermining the possible side effects of reconfiguration plans. The work can guide the development of dynamic software systems from component definition to reconfiguration plan design.**

***Index Terms*—dynamic software architecture, dynamic description logic, runtime reconfiguration**

I. Introduction

Dynamic software architectures support reconfigurations of their structures during execution [15]. They can be used to build long running applications that face changing requirements and/or execution environment [2]. Architecture description language (ADL) [17] is widely used in specifying dynamic software architectures. Most of the ADLs are based on some kinds of formal foundations [13]. For example, Dynamic Wright [2] and Darwin [8] have laid their foundations on process algebra. Approaches proposed in [12] [18] [19] are based on graph theory. And [1] [10] [16] [11] have presented logic-based specifications.

The reconfiguration of software architecture is usually expressed as reconfiguration plan, a sequence of steps to change the architecture. The validity of reconfiguration plan is crucial to many software systems, such as airport management systems, bank systems, and e-business systems. Execution of an improper reconfiguration plan may cause disastrous results. Logic-based specifications have solid foundations and sound reasoning algorithms. So they play important roles in the area of analyzing and validating reconfiguration plans. Expressiveness and reasoning services are two critical features of the logic-based specifications.

In this paper, dynamic description logic [3] is used to specify dynamic software architectures. The features of the approach include: 1. The Reconfigurable Data Flow (RDF) model is proposed as the architecture meta-model. It is an extension to the widely used Data Flow (DF) model and its architecture can be changed during execution. 2. Architectures, reconfiguration actions, and reconfiguration plans are represented in a unified framework from the viewpoint of data flow based on dynamic description logic. 3. Three levels of constraints have been proposed to aid designers in predetermining the side effects of the reconfiguration plans.

II. Related Works

Logic-based specifications of dynamic software architectures represent the architectural elements, structures, and reconfiguration plans based on some kinds of logics. So far as we know, the logics used as the foundations of the specifications include first order logic, temporal logic, predicate logic and set theory, and spatial logic.

Generic Reconfiguration Language (Gerel) [10] specifies the reconfigurations of components as change scripts. The script language is based on first order logic. It also can be used to describe the properties of components. It uses the precondition and selection mechanisms to check if the current configuration has the required properties and to apply the reconfiguration commands only to the components satisfying these properties.

ZCL [16] framework uses the denotations of Z and the semantics of CL, a language based on predicate logic and set theory. ZCL models concepts in two schemas: state and operation. A state schema consists of a variable-definition part and a predicate part in which relations and constraints are described. An operation schema has a before state, an after state, inputs, outputs, and a set of pre-conditions, which models an operation as a transition between two states. Thus the properties of the reconfiguration can be validated through reasoning.

Aguirre-Maibaum's approach [1] presents a formal specification language for component-based systems. Based on temporal logic, the language aims at specifying the dynamical behaviors of architectures. Re-configurable systems can be built hierarchically and their behaviors can be reasoned based on the semantics of the language.

Han's approach [11] tailors spatial logic as a specification for structures, which is used to verify whether the system evolution satisfies some structure constraints. Spatial logic is a kind of formal language representing the geometrical entities and relations over a class of structures. It employs the syntax and semantics of first-order logic.

Although these approaches specify dynamic software architectures with different fragments of first order logic, they all use the reasoning algorithm of first order logic. The expressiveness of first order logic is sufficient to formalize dynamic software architectures. But as Brachman and Levesque pointed out, there is a tradeoff between the expressiveness of a language and the difficulty of reasoning using the language [7]. The specifications of dynamic software architectures do not require all the machinery of first-order logic. So the machinery of first order logic is too general for the specifications of dynamic software architectures to reason efficiently.

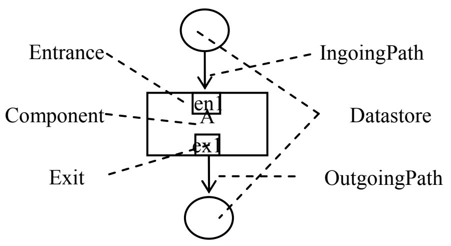
Compared with these approaches, the approach presented in this paper is based on dynamic description logic [3]. Dynamic description logic extends description logic [4] with the representation of state changes. Description logic is a decidable fragment of first order logic. Compared with first order logic, its notable feature is that it provides efficient reasoning services, although it has less expressiveness. Dynamic description logic inherits the features of description logic on the one hand; it has the ability to represent state changes on the other hand. Its expressiveness is sufficient to represent dynamic software architectures and it supports efficient reasoning services at the same time. Therefore it is an appropriate formal foundation for the specifications of dynamic software architectures.

III. Specification of Dynamic Software Architectures

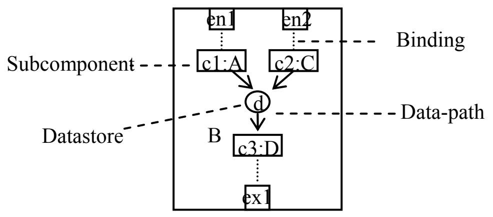
A. Architecture Meta-model

We find it difficult to handle the existing computations when dynamic reconfiguring a control flow based system. So we propose an architecture meta-model, RDF model [20], based on the DF model semantics. DF model is one of the most popular models for structured analysis and design [6][9]. It focuses on representation of the flow of data through an information system.

The basic elements of a RDF model are components, data-stores, and data-paths. A component is a software module that could consume data through entrances and produce data through exits. A data-store is a random-access data container with infinite capacity. A data-path is a route by which data can flow.



(a) A simple component



(b) A composite component

Figure 1. Graphical representation of components.

Components are divided into simple components and composite components. A simple component is a black-box (Fig.1-a), while a composite component is composed of other components (Fig.1-b). A simple component works in a block-read and non-block-write mode. Block-read means that a process does not consume data until its fire rule is satisfied. Non-block-write means that a process does not wait when trying to write data to an exit without outgoing data-path. If the exit is connected with a data-path, the data flows through the data-path. Otherwise, the data is thrown away. The internal structure of a composite component is a data flow system that consists of other components and data-stores. An internal component could bind its entrances/exits to the composite component's entrances/exits so that it could use the entrances/exits as its own to interchange data with the environment outside the composite component. Thus a software system could be modeled as a composite component. And a complex system could be constructed hierarchically from small parts.

A data-path does not queue any data, thereby a datum always pass through a data-path instantaneously. And because components all work in block-read and non-block-write mode, the data transmitted from an exit to an entrance must pass through a data-store.

A set of operations could be applied to change the structure during runtime. These operations include addition and removal of a component, a data-store, or a data-path. Using these operations, the RDF model supports dynamic reconfigurations without influence on the data flow rate. For example, to replace a component, a replacement pattern could be used: 1)start the new component; 2)set up the outgoing data-path for the new component; 3)set up the incoming data-path for the new component; 4)remove the incoming data-path of the old component; 5)wait the old component to finish the current processing; 6)remove the outgoing data-path of the old component; 7)remove the old component.

B. Dynamic Description Logic

Description logic is considered as a structured subset of first order logic and one of the most effective formulizations of knowledge representations [4]. It provides several kinds of useful services, such as terminology consistency detection and ABox query. As description logic was originally designed for representing static knowledge, some researchers present dynamic description logic [3], which is an integration of description logic and situation calculus [14]. Actions are generally defined by pre- and post-conditions and they cause changes of the system state.

Based on dynamic description logic, we represent architectures, reconfiguration actions, reconfiguration plans, and architectural constraints in a unified framework. Table I lists the basic concepts and roles used in the representation.

TABLE I.
ATOMIC CONCEPTS AND ROLES

Concepts/Roles	**Explanation**
Component(x)	x is a component.
SimpleComponent(x)	x is a simple component. SimpleComponent⊑Component
CompositeComponent (x)	x is a composite component. CompositeComponent⊑Component
Datastore(x)	x is a datastore.
Entrance(x)	x is an entrance.
Exit(x)	x is an exit.
hasEntrance(x,y)	Component x has an entrance y.
hasExit(x,y)	Component x has an exit y.
canFlowInData(x,y)	Entrance x allows type y data to flow in.
canFlowOutData(x,y)	Exit x allows type y data to flow out.
canContainData(x,y)	Data-store x can contain type y data.
hasIncomingPath(x,y)	Entrance x has an incoming data-path from datastore y.
hasOutgoingPath(x,y)	Exit x has an outgoing data-path to datastore y.
bindTo(x,y)	Entrance/exit x is bound to entrance/exit y.
hasInflow(x,y)	Datastore x has an inflow from exit y.
hasOutflow(x,y)	Datastore x has an outflow to entrance y.
hasSubComponent(x,y)	Composite component x has a subcomponent y.
hasDatastore(x,y)	Composite component x has a datastore y.
hasRoute(x,y)	Composite component x has a data route y.
SimpleActive(x)	Simple component x is active.
Empty(x)	Datastore x is empty.

hasOutflow is the inverse of *hasIncomingPath* and *hasInflow* is the inverse of *hasOutgoingPath*. *hasIncomingPath* and *hasOutgoingPath* are defined to describe the properties of Components, while *hasOutflow* and *hasInflow* are defined to describe the properties of *Datastores*. The following axioms always hold:

$\forall x \forall y(\text{hasIncomingPath}(x,y) \leftrightarrow \text{hasOutflow}(y,x))$

$\forall x \forall y(\text{hasOutgoingPath}(x,y) \leftrightarrow \text{hasInflow}(y,x))$

bindTo means that two Entrances/Exits are connect to the same data-path. So the following axioms always hold:

$\forall x \forall y \forall z(\text{bindTo}(x,y) \wedge \text{hasIncomingPath}(y,z) \rightarrow \text{hasIncomingPath}(x,z))$

$\forall x \forall y \forall z(\text{bindTo}(x,y) \wedge \text{hasOutgoingPath}(y,z) \rightarrow \text{hasOutgoingPath}(x,z))$

C. Architecture Description

Components

A simple component is described from its interface, which includes the entrances and the exits. In an entrance or exit declaration, the data it can consume or produce is also defined. For example, the component A in Fig.1(a) can be represented as

```
Component A
  hasEntrance(en1);            // an entrance
    canFlowInData(datatype1); // data type
  hasExit(ex1);                // an exit declaration
    canFlowOutData(datatype2);        // data type
end of component;
```

A composite component is described from its interface, structure, and route map. The interface, similar to that of simple component, includes the entrances and the exits. The structure defines its subcomponents, data-stores, data-paths, and bindings. The route map defines the routes that data elements could pass through the component. A data route is a sequence of components that a data element might pass through. It could be viewed as a description of the logic processes of the component from a data flow viewpoint. For example, the component B in Fig.1(b) can be defined as

```
Component B
  hasEntrance(en1);            // interface
    canFlowInData(datatype1);
  hasEntrance(en2);
    canFlowInData(datatype3);
  hasExit(ex1);
    canFlowOutData(datatype4);
  hasDatastore(d);             // structure
    canContainData(datatype2);
  hasSubComponent(c1);
    hasComponentType(A);
    hasEntrance(en1_c1);
      bindTo(en1);
    hasExit(ex1_c1);
      hasOutgoingPath(d);
  hasSubComponent(c2);
    hasComponentType(C);
    hasEntrance(en1_c2);
      bindTo(en2);
    hasExit(ex1_c2);
      hasOutgoingPath(d);
  hasSubComponent(c3);
    hasComponentType(D);
    hasEntrance(en1_c3);
      hasIncomingPath(d);
    hasExit(ex1_c3);
      bindTo(ex1);
  hasRoute([c1, c3]);          // route map
  hasRoute([c2, c3]);
end of component;
```

Reconfiguration

A composite component can change its structure and route map during runtime. A reconfiguration transfers the composite component from one configuration to another, where configuration is a snapshot of the structure and route map of a running composite component. A reconfiguration is achieved by a reconfiguration plan, which is a program that has a sequence of reconfiguration actions. A reconfiguration action is an instance of one of the reconfiguration operations, which can cause a type of changes on the configuration. Formally, configuration, operation, action, plan are defined as follows.

A **configuration** is a set of facts, which represent the interface, structure, and route map of the component.

An **operation** is in the form of

$$OP(x_1,\ldots,x_n) \equiv \langle C, N, E\rangle$$

where *OP* is the operation name; $x_1,\ldots,x_n$ are variables, which denote the individuals the operation operates on; *C* is the constraint on the operation; *N* is the negative effects and *E* the positive effects of the operation.

An **action** is an instance of an operation by binding the variables to individuals. Suppose action *a* change the system from configuration *F* to *F'*. *C(a)* must be satisfied in *F*. And the execution of *a* will remove all the facts in *N(a)* from the configuration and add all the facts in *E(a)* into the configuration, i.e. $F'=(F-N(a))\cup E(a)$.

A *reconfiguration plan* is a sequence of actions $a_1,a_2,\ldots,a_n$. The execution of a plan will cause the component to reach a new configuration after experiencing a sequence of interim configurations. $C_{before} \xrightarrow{a_1} C_1 \xrightarrow{a_2} C_2 \xrightarrow{\cdots} \ldots \xrightarrow{\cdots} C_{n-1} \xrightarrow{a_n} C_{after}$.

D. Constraints

The running of a system requires that the configuration satisfies several constraints, including route connectivity and data consistency. Route connectivity means all the routes are connective so that data elements could pass through. Data consistency means that the data elements produced by the predecessor component are exactly what the consequent component needs. The following constraints 1 and 2 are for route connectivity, and constraints 3 and 4 are for data consistency.

Constraint 1. A data route should be connective. Or in other words, there should be a data-path between a component and its subsequence. Suppose a data route is $[c_1,c_2,\ldots c_n]$, for any $1\le i\le n-1$,

$\exists x\exists y\exists z$ (Exit(x)$\wedge$Datastore(y)$\wedge$Entrance(z)$\wedge$hasExit(ci,x)$\wedge$
  hasPathTo(x,y)$\wedge$hasEntrance(c_{i+1},z)$\wedge$hasPathTo(z,y))

Constraint 2. An entrance could be connected to at most one data-store. On the contrary, an exit could be connected to multiple datastores for representation of broadcasting.

$\forall x$(Entrance(x) $\rightarrow$ ($\le$1 IngoingPath)(x))

Constraint 3. The data that could flow in an entrance or flow out an exit must be consistent with the data that the connected data-store could contain.

$\forall x\forall y\forall z$(Entrance(x)$\wedge$DataType(y)$\wedge$Datastore(z)$\wedge$
  canFlowInData(x,y)$\wedge$hasPathTo(x,z)$\rightarrow$canContainData(z,y))

$\forall x\forall y\forall z$(Exit(x)$\wedge$DataType(y)$\wedge$Datastore(z)$\wedge$
  canFlowOutData(x,y)$\wedge$hasPathTo(x,z)$\rightarrow$canContainData(z,y))

Constraint 4. The data that can flow through two entrances/exits that are bound together by a binding must be consistent.

$\forall x\forall y\forall z$ (Entrance(x)$\wedge$DataType(y)$\wedge$Entrance(z)$\wedge$
  canFlowInData(x,y) $\wedge$Binding(x,y)$\rightarrow$canFlowInData(z,y))

$\forall x\forall y\forall z$ (Exit(x)$\wedge$DataType(y)$\wedge$Exit(z)$\wedge$
  canFlowOutData(x,y) $\wedge$Binding(x,y)$\rightarrow$canFlowOutData(z,y))

Besides configurations, a reconfiguration must satisfy several constraints to ensure the system correctness during the reconfiguration progress. Because the route map defines the logic processes of the component, the most important change in a reconfiguration is the change on the route map, e.g. add a route, remove a route, or replace a route.

A new route should be established from the end point to the start point. Thus the building process of the route has already finished before data elements could flow into the route. Otherwise, if data elements were allowed to flow into a route that is under construction, they might encounter a dead end, a component or data-store without outflow. Then unexpected side effects would appear. A dead end component might cause data lost if data elements flow into because the data elements produced by the component are thrown away. A dead end data-store might cause flow rate decline if data elements flow into because the data elements stop flowing until an outflow is set up.

An old route should be removed from the start point to the end point. The route is closed first so that no data element could flow into the route any more. Then along the route, the components are removed one by one after the existing data elements have all flowed through. Otherwise, there would be side effects of data lost or flow rate decline because removing a working component or causing dead end.

In a route replacement, the new route should be established before the removal of the old route. Data elements could flow through the new route during the removal process of the old route. Thereby no flow rate decline during the process. Also the process of new route establishment and old route removal should satisfy the corresponding constraints.

These constraints are formally represented as follows.

Constraint 5. The data produced by a component should be able to find a path to flow into a datastore.

$\forall x$((Activable$\sqcup$Active)(x)$\rightarrow$DataExportable(x))

Activable $\equiv$ $\forall$HasEntrance.(=1 IngoingPath)

Active$\equiv$(SimpleComponent$\sqcup$SimpleActive)
    $\sqcup$ (CompositeComponent$\sqcap$
     (($\exists$ContainComponent.Active)
      $\sqcup$ ($\exists$ContainDatastore.$\neg$Empty)))

DataExportable $\equiv$ $\forall$HasExit.(=1 OutgoingPath)

This constraint is to prevent a possible condition of data losses. A component might produce data if it is *Activable* or *Active*. *Activable* means that the components may become active, e.g. each of its entrances is connected to a datastore. *Active* is a recursive attribute. For a simple component, *Active* equals to *SimpleActive*. For a composite component, *Active* means there is an internal *Active* component or a non *Empty* datastore. The

Active property can be checked by a recursive procedure. *DataExportable* represents the component that each of its exits is connected to a datastore.

Constraint 6. The data already in a datastore or the data that may flow into a datastore should have a path to flow out.

$\forall$x ((Datastore$\sqcap$ (($\exists$Inflow)$\sqcup\neg$Empty))(x) $\rightarrow$ ($\exists$OutFlow)(x))

If a datastore has inflow(s) but has no outflow, data that flow into the datastore cannot flow out. Although the data is not lost, the flow of data is blocked. The successor components have to wait for data. Therefore the QoS declines.

Constraint 7. Any component should be *Activable*. Otherwise it is useless to the system.

$\forall$x (Component(x)$\rightarrow$Activable(x))

Constraint 8. Any datastore should be connected with some component(s). Otherwise it is useless.

$\forall$x (Datastore(x)$\rightarrow$($\exists$Outflow)(x))

These constraints could aid the designers in predetermining the side effects of a system configuration, According to the side effects that they could detect, the constraints could be classified into three levels, fatal, flow-rate-decline, and sleeping-node. They are listed in Table II.

TABLE II.
THREE LEVELS OF CONSTRAINTS

Contraints	Level	Side effects detected
1~5	fatal	The system is unable to work because of data lose.
6	flow-rate-decline	The system could work functionally but its performance declines.
7,8	sleeping-node	There are useless components or datastores in the system.

The side effects of a reconfiguration plan can be predetermined by the following rules. Suppose the reconfiguration causes the system experiencing a sequence of configurations C_1, ..., C_n.

i) If one or more configurations of C_1, ..., C_n have error side effects, the reconfiguration plan may cause the system unable to work or losing data.

ii) If one or more states of C_1, ..., C_n have QoS-decline side effects, the reconfiguration plan may cause decline of the QoS.

iii) If C_n has sleeping-node side effects, there are useless components or datastores after the execution of the reconfiguration plan.

IV. A CASE STUDY

In this section, we illustrate how to use our approach to specify an Upgradeable Client-Server (CCS) system. There are one server and multiple clients in the system. The server provides services to the clients in a request-reply mode. The server can be upgraded during runtime.

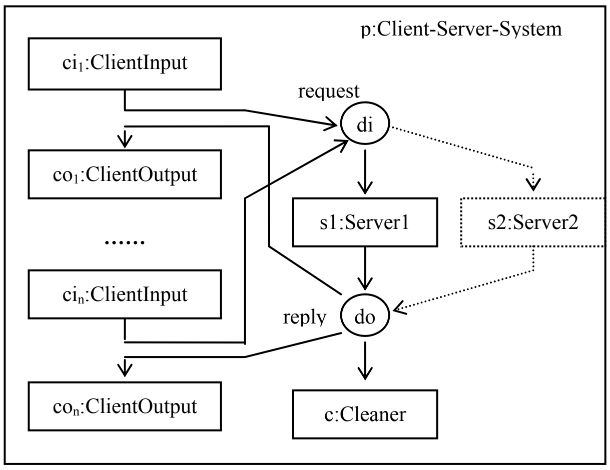


Figure 2. The upgradeable client-server system.

The architecture of the system is shown in Fig.2. A client is composed of two components, *ClientInput* and *ClientOutput*. *ClientInput* generates a request when it needs the service of the server. The request has a tag marked with the client's address. It flows into datastore di and then is consumed by the Server component. After processing the request, the Server component generates a reply, which has an address tag and a timestamp tag. The reply flows into datastore do, and then is retrieved by a *ClientOutput* according to the address tag. A client can freely connect to the server and disconnect from the server. If a client submits a request and then quits before retrieving the reply, the Cleaner component will gather the outdated replies based on the timestamp tag.

The component definitions are as follows:

```
component Client-Server-System
  hasDatastore(di);
    canContainData(request);
  hasDatastore(do)
    canContainData(reply);
  hasSubComponent(s1);
    hasComponentType(Server1);
    hasEntrance(en1);
      canFlowInData(request);
      hasIncomingPath(di);
    hasExit(ex1);
      canFlowOutData(reply);
      hasOutgoingPath(do);
  hasSubComponent(c);
    hasComponentType(Cleaner);
    hasEntrance(en1);
      canFlowInData(reply);
      hasIncomingPath(do);
  hasSubComponent(ci);
    hasComponentType(ClientInput);
    hasExit(ex1);
      canFlowOutData(request);
      hasOutgoingPath(di);
  hasSubComponent(co);
    hasComponentType(ClientOutput);
    hasEntrance(en1);
      canFlowInData(reply);
      hasIncomingPath(do);
  hasRoute([ci,s1,co]);
  hasRoute([ci,s1,c]);
end of component;
```

In Fig.3, we show four reconfiguration plans for server upgrade. The unsatisfied constraints are listed after each action. Plan-A (see Fig.3(a)) has no side effect because only the *InactivableComponent* constraint is not satisfied during the procedure and all the constraints are satisfied after the execution. Plan-B (see Fig.3(b)) has sleeping-node side effects because s1 is an *InactivableComponent* after the execution. Plan-C (see Fig.3(c)) has QoS-decline side effects because *di* breaks the *DeadEndDatastore* constraint during the procedure. Plan-D (see Fig.3(d)) has error side effects because there are two temporary *InvalidComponents* during the procedure.

```
PLAN upgrade-plan-A{                    // Unsatisfied Constraints
    start-component(s2,Server2);                // IAC(s2)
    establish-outgoing-path(ex1_s2, do);        // IAC(s2)
    establish-ingoing-path(en1_s2, di);         // Nil
    destroy-ingoing-path(en1_s1, di);           // IAC(s1)
    wait-component-inactive(s1);                // IAC(s1)
    destroy-outgoing-path(ex1_s1, do);          // IAC(s1)
    stop-component(s1);                         // Nil
}
```

(a) Plan A

```
PLAN upgrade-plan-B{                    // Unsatisfied Constraints
    start-component(s2,Server2);                // IAC(s2)
    establish-outgoing-path(ex1_s2, do);        // IAC(s2)
    establish-ingoing-path(en1_s2, di);         // Nil
    destroy-ingoing-path(en1_s1, di);           // IAC(s1)
    wait-component-inactive(s1);                // IAC(s1)
    destroy-outgoing-path(ex1_s1, do);          // IAC(s1)
}
```

(b) Plan B

```
PLAN upgrade-plan-C{                    // Unsatisfied Constraints
    destroy-ingoing-path(en1_s1, di);   // DED(di), IAC(s1)
    wait-component-inactive(s1);        // DED(di), IAC(s1)
    destroy-outgoing-path(ex1_s1, do);  // DED(di), IAC(s1)
    stop-component(s1);                 // DED(di)
    start-component(s2,Server2);        // DED(di), IAC(s2)
    establish-outgoing-path(ex1_s2, do);// DED(di), IAC(s2)
    establish-ingoing-path(en1_s2, di); // Nil
}
```

(c) Plan C

```
PLAN upgrade-plan-D{                    // Unsatisfied Constraints
    start-component(s2,Server2); // IAC(s2)
    establish-ingoing-path(en1_s2, di);         // IVC(s2)
    establish-outgoing-path(ex1_s2, do);        // Nil
    destroy-outgoing-path(ex1_s1, do);          // IVC(s1)
    destroy-ingoing-path(en1_s1, di);           // IAC(s1)
    wait-component-inactive(s1);                // IAC(s1)
    stop-component(s1);                         // Nil
}
```

(d) Plan D

IAC=InactivableComponent, DED=DeadEndDatastore, IVC=InvalidComponent

Figure 3. Reconfiguration plans.

Therefore, Plan-A is the best reconfiguration plan among them. In plan-A, component s_2 goes into operation first. Then the ingoing data path to s_1 is cut off so that there will be no data flowing into s_1 any more. And s_1 is stopped after it finishes processing the data that has already flowed into it. The flow of the data has not been interrupted or blocked during the substitution procedure, so the reconfiguration plan has little side effects on the system running.

The upgradeable client-server case is a quite simple example. But it shows that our approach works well in modeling and verifying dynamic software architectures. Based on the formal specification of the architecture, reconfiguration plan, and architectural constraints, the side effects can be predetermined before bringing a reconfiguration plan into effect.

V. Conclusion and Future Works

The logic based formal specification plays an important role in analyzing, planning, and validating dynamic software architectures. In this paper, we present a configurable extension of the widely used dataflow model as the architecture meta-model. Then we propose a formal specification for the configurable dataflow model based on dynamic description logic. Architectures, reconfiguration actions and reconfiguration plans are represented in a unified framework. Three levels of architectural constraints are defined to predetermine the side effects of the reconfiguration plans. Our work can guide the development of software systems that have dynamic architectures from component definition to reconfiguration plan design. For the systems built under our framework, the side effects of the reconfiguration plans are predictable and disastrous results can be avoided.

Further work focuses on the automatic generation of reconfiguration plans. Given the initial architecture, goal architecture, reconfiguration actions, and architectural constraints, a planner should be able to generate the reconfiguration plan that has the minimal side effects automatically.

Acknowledgment

This work was supported in part by the Scientific Research Foundation for the Returned Overseas Chinese Scholars from State Education Ministry and Scientific Research Foundation for Doctors from Shandong University of Finance.

References

[1] N. Aguirre and T. Maibaum, “A temporal logic approach to the specification of reconfigurable component-based systems”, Proc. of the 17th Int. Conf. on Automated Software Engineering (ASE 2002), Edinburgh, Scotland, UK, 2002, pp. 271-274.

[2] R.J. Allen, R. Douence, and D. Garlan, “Specifying and Analyzing Dynamic Software Architectres”, Proc. of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98), Lisbon, Portugal, March 1998.

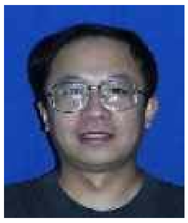
[3] A. Artale and E. Franconi, “A survey of temporal extensions of description logics”, Annals of Mathematics and Artificial Intelligence, 30(1-4), 2001.

[4] F.Baader and B.Hollunder, “A terminological knowledge representation system with complete inference algorithms”, Proc. of the workshop on Processing Declarative Knowledge (PDK-91), Kaiserslautern, Germany, 1991, pp. 67-86.

[5] F. Baader, et al. The Description Logic Handbook: Theory, Implementation and Applications, Cambridge University Press, 2002.
[6] J. Bacus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", Communications of the ACM (CACM), 21(8), 1978, pp. 613-641.
[7] R. J. Brachman and H. J. Levesque, "The tractability of subsumption in frame-based description languages", Proc. of the Fourth National Conference on Artificial Intelligence (AAAI-84), Austin, USA, 1984, pp. 34--37.
[8] C. Canal, E. Pimentel, and J. M. Troya, "Specification and refinement of dynamic software architectures". Proc. of the Working IFIP Conf. on Software Architecture (WICSA'99), Kluwer, Belgium, 1999, pp. 107-126.
[9] Gang Cheng, A Dataflow-Based Software Integration Model in Parallel and Distributed Computing and Applications. Ph.D. Dissertation. Syracuse University, Italy, 1997.
[10] M. Endler and J. Wei, "Programming generic dynamic reconfigurations for distributed applications", Proc. of the International Workshop on Configurable Distributed Systems, IEE, 1992, pp. 68-79.
[11] T. Han, T. Chen, J. Lu, "Structure Analysis for Dynamic Software Architecture Based on Spatial Logic", Proc. of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05), Edinburgh, Scotland, 2005, pp. 71-76.
[12] D. Hirsch, P. Inverardi, and U. Montanari, "Graph grammars and constraint solving for software architecture styles", Proc. of the 3rd International Software Architecture Workshop (ISAW-3), ACM Press, 1998, pp. 69-72.
[13] J.S. Bradburya, J.R. Cordyay, J. Dingela, M. Wermelinger, "A Survey of Self-Management in Dynamic Software Architecture Specifications", Proc. of the international workshop on self-managed systems(WOSS'04), California, USA. 2004.
[14] H. Levesque, F. Pirri, and R. Reiter, "Foundations for the situation calculus", Electronic Transactions on Artificial Intelligence, 2(3-4), 1998, pp. 159-178.
[15] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages", IEEE Trans. on Software Engineering, 26(1), 2000, pp. 70-93.
[16] V. C. C. de Paula. ZCL: A Formal Framework for Specifying Dynamic Software Architectures. PhD thesis, Federal University of Pernambuco, 1999.
[17] M.Shaw and D.Garlan, Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Englewood Cliffs, New Jersey, 1996.
[18] G. Taentzer, M. Goedicke, and T. Meyer, "Dynamic change management by distributed graph transformation: Towards configurable distributed systems", Proc. of the 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98). Paderborn, Germany, 1998.
[19] M. Wermelinger, A. Lopes, and J. L. Fiadeiro, "A graph based architectural (re)configuration language", Proc. of the 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2001), Vienna, Austria, 2001, pp. 21-32.
[20] Wei Li, Zhikun Zhao, Influence Control for Dynamic Reconfiguration of Dataflow Systems, Journal of Software, 2007(6)

Zhikun Zhao was born in Qingzhou, Shandong province, China in 1975. He received his Ph.D. degree on computer software theory from the Graduate University of Chinese Academy of Sciences, Beijing, China in 2003.

He was an Associate Professor of the Graduate University of Chinese Academy of Sciences from 2003 to 2005. He worked as a Postdoctoral Research Fellow of Central Queensland University from 2006 to 2008. Currently he is an Associate Professor of Shandong University of Finance in Jinan, Shandong province, China. His research interests include dynamic software reconfiguration and multi-agent systems.



Wei Li was born in Haerbin, Heilongjiang province, China in 1964. He received his PhD degree on computer science from the Institute of Computing Technology, Chinese Academy of Sciences in July 1998.

He is currently a Senior Lecturer in School of Information & Communication Technology at the Central Queensland University, Rockhampton, Australia. His research interests include dynamic software architecture and multi-agent systems.