Influence Control for Dynamic Reconfiguration

Zhikun Zhao and Wei Li
School of Computer Science, Central Queensland University
{z.zhao, w.li}@cqu.edu.au

Abstract

Dynamic reconfiguration is a useful technique for software update because it can achieve an architectural change without shutdown of a system. However, so far in the state-of-arts, there has not been an approach that can evaluate and control both the functional influence and performance influence of reconfiguration in a unified framework. In this paper, we present an approach that addresses the above drawback. In our approach, we use a reconfiguration algorithm and a reconfiguration scheduler to control these two types of influence. The algorithm reduces the logical performance influence by allowing old and new components coexisting and uses a version management mechanism to avoid functional side effect in the coexisting period. The scheduler controls the physical performance influence through restricting the processor time spent on the reconfiguration procedure. We implement the algorithm and the scheduler in our Reconfiguration Data Flow model.

1. Introduction

Dynamic reconfiguration aims to implement runtime architectural changes of long-running software [12]. Many research efforts have been made on component models that support dynamic reconfiguration, such as SOFA2 [15], ArchJava [1], Fractal [2], Darwin [11], Rapide [10], and K-Component [5]. However, these models are all weak in controlling the influence of dynamic reconfiguration. As dynamic reconfiguration is performed during system runtime, its influence should be evaluated systematically. It may bring severe threats to the system if its influence is not determinable or controllable [14].

The existing studies on the influence of dynamic reconfiguration can be divided into two main branches. The first branch focuses on the influence on functionality. Zhang has presented the term 'safe adaptation' to mean the program maintains its integrity during dynamic adaptation [17]. Desmet has considered the correctness of a dynamic system during and after a reconfiguration [4]. Feiler has introduced the concept of configuration consistency in terms of syntax, type, resource utilization, and semantics [6]. The second branch concentrates on the influence on performance. Gorinsek has laid out a base to support QoS (Quality of Service) management during reconfiguration [7]. Mitchell has demonstrated a generic algorithm for scheduling dynamic reconfigurations that maintains QoS guarantees for multimedia stream [13]. Hillman has designed OpenRec, an open framework for managing dynamic reconfiguration and measuring the impact of reconfiguration in terms of time and disturbance [8]. However, there has not been a method that can evaluate and control these two types of influence in a unified framework.

In this paper, we propose an approach that addresses the above drawback. Based on the representation of the data flows of a system, we propose a reconfiguration algorithm and a reconfiguration scheduler to control the influence of reconfiguration. The reconfiguration algorithm can change the architecture of a system from one configuration to another. The reconfiguration scheduler can control the execution of reconfiguration. Under the co-work of the algorithm and the scheduler, the influence of dynamic reconfiguration on functionality and performance is controllable.

2. Reconfigurable system and influence of reconfiguration

2.1. Reconfigurable system

For a reconfigurable system, we have the following assumptions.

A reconfigurable system is a component-based system. A component-based system is composed of components and connectors [16]. A reconfiguration is a change of the system architecture, i.e.

1530-0803/07 $25.00 © 2007 IEEE

addition/removal of components/connectors. A component is a system element that implements certain functions and is able to communicate with other components through its input and output ports. A connector provides a communication link between two components. Such a system could change its architecture using component as the basic operational unit.

To support reconfiguration, the reconfiguration operations must be able to be executed during runtime and will not cause incorrect status of the system. For example, a system will undertake an incorrect status if removing a connector between two components when they are in communicating. Therefore, some system procedures, such as communication, must be defined as atomic. And a synchronization mechanism is necessary to prevent these atomic processes from being interrupted by reconfiguration operations.

A reconfigurable system needs a representation of architectural configurations to specify the current and target configurations of the architecture. The most popular way for this purpose is Architecture Description Language (ADL). It usually uses a form of predicates or scripts to describe the components and the connectors between them. For a reconfiguration, the current architectural configuration can be detected by the system itself; the target architectural configuration is specified by the administrator.

There needs a representation of reconfiguration plan, which specifies how to change the architecture step by step. A reconfiguration plan could be represented as a piece of program, a script, a partial order plan, or vice verse. Usually, a reconfiguration plan is written by an administrator manually or generated by a planner automatically. Then an execution engine executes it.

2.2. Influence of reconfiguration

We classify the influence of reconfiguration into two types, *functional influence* and *performance influence*. Functional influence is the change on the system functionality caused by reconfiguration. Performance influence is the change on the system running speed caused by reconfiguration.

At most cases, a reconfiguration has functional influence on the system because changing functionality is one of the commonest purposes of reconfiguration. But a reconfiguration should not have *functional side effect*, which means incorrect outputs for some inputs. Formally, the functionality of a system could be represented as a function F. For any input e, its output is $F(e)$. Suppose a reconfiguration r change the functionality of a system from F to F', r has functional side effect if r causes the system output neither $F(e)$ nor $F'(e)$ for an input e.

Under the above constraint, there are two practicing modes for transferring system functionality in reconfiguration.

The first mode is *switching* transformation. The functionality of the system is switched from $F(e)$ to $F'(e)$ at a time stamp t in the reconfiguration progress. Therefore, the system output is as follows (Figure 1-a):

1) the system outputs $F(e)$ for any input e appearing before t, whether the computing progress covers t or not (C_1, C_2 in Figure 1-a); and
2) the system outputs $F'(e)$ for any input e appearing after t (C_3 in Figure 1-a).

The second mode is *coexisting* transformation. The old functionality and new functionality are coexisting in reconfiguration. Therefore, the system output is as follows (Figure 1-b):

1) The system always outputs $F(e)$ if the processing of input e completes before r starts (C_1 in Figure.1-b);
2) The system always outputs $F'(e)$ if the processing of input e completes after r ends (C_4 in Figure.1-b);
3) The system outputs either $F(e)$ or $F'(e)$ if the processing of input e completes after r starts and before r ends, i.e. during the reconfiguration period (C_2, C_3 in Figure.1-b).

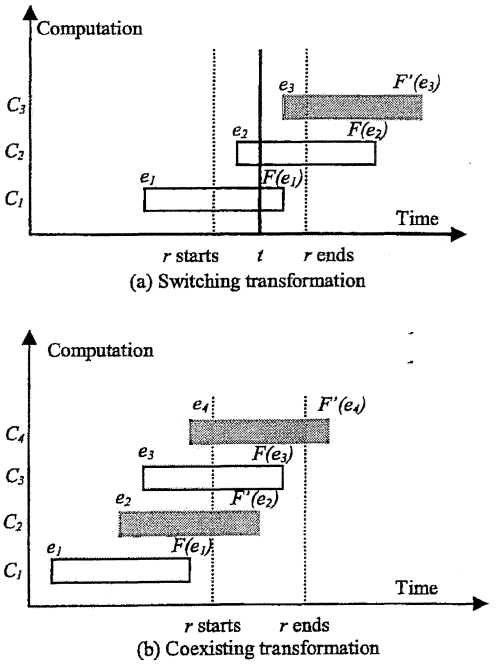


Figure 1. Functional influence

The performance influence of a reconfiguration is expected to be as small as possible. The performance influence can be reflected by the change of the system output rate, which is the amount of outputs the system produces in a time slice. In a time slice, suppose the actual output rate is o' and the output rate should be o if there was no reconfiguration, then the performance influence is $d=|o'-o|/o$. We name o theoretical output rate.

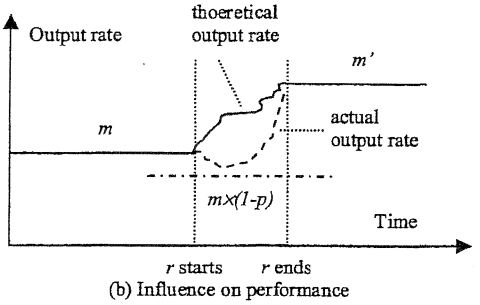
If a reconfiguration is performed at the system's different workload, its performance influence will be various. Due to the limitation of computing capability, every system has a maximum workload and therefore a maximum output rate, which is the output rate when the system is fed with enough inputs. The performance influence is the largest if the reconfiguration is performed at this time.

To evaluate the performance influence of a reconfiguration, the worst condition should be considered. Therefore we use the *Possible Maximum Performance Influence (PMPI)*, the performance influence of a reconfiguration performed at the maximum system workload, to represent the performance influence of a reconfiguration.

The theoretical maximum output rate is not fixed during reconfiguration and it is not easy to calculate because the system architecture is in changing. Suppose the theoretical maximum output rate of a system is m before reconfiguration r and m' after reconfiguration r. We expire the theoretical maximum output rate ranging between m and m' during reconfiguration period. Thus if the PMPI is p, the actual maximum output rate is:

1) m before r starts;
2) m' after r ends;
3) $\geq min(m\times(1-p),m'\times(1-p))$ during reconfiguration r, where *min* returns the smaller value of the two parameters.

Therefore the actual maximum output rate of the system is higher than $min(m\times(1-p),m'\times(1-p))$ (Figure.2).



(b) Influence on performance

Figure 2. Performance influence

3. Influence control for reconfiguration

3.1. Representing application computation

To design a reconfiguration approach that can not only achieve an architectural change but also restrict the influence in a user-acceptable scope, the two types of influence have to be considered in a unified framework. Application computation is a suitable object to integrate these two types of influence. An application computation is a working progress of an application, which starts when receiving an input and ends after generating output for the input. A reconfiguration will cause functional side effect if it interrupts a running application computation or results in a new application computation not pre-designed by the user. A reconfiguration will cause performance influence if it pause or slow down an application computation. Therefore, a representation of the application computations of a system is fundamental to the influence control of reconfiguration.

From data flow viewpoint, we represent an application computation as a route, a sequence of components that a data element passes through one by one. A route could be a *designed route* or a *data route*. A designed route is a route pre-designed by designers. A data route is a route that a data element has actually passed through. In a system, all the designed routes compose a *route map*, which is part of the system design. As a designed route defines a progress of data processing, it corresponds to one of the system functions. And the route map is a system design to describe how components cooperate to implement system functions.

Route map and architecture are different levels of system design. For a system, the architecture should support the route map. It means that the architecture can control data elements not to pass through a route that does not belong to the route map but able to pass through every route in the route map. If the architecture does not support the route map, it means implementation can not guarantee design. The relations between architecture, route map, application computations, and system functions are shown in Figure.3.

A reconfiguration will change the architecture of a system from one configuration to another. Correspondingly, the system route map supported by the architecture will be changed from one to another too.

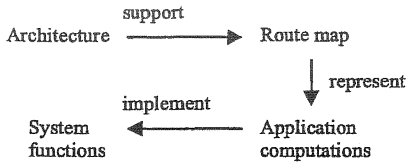


Figure 3. Architecture, route map, computations, and functions

Definition 1. An **architectural configuration** (or configuration in short) of a system is a tuple $<C, N>$, where C is the set of components in the system, N is the set of connectors in the system. A connector is a tuple $<c_1,p_1,c_2,p_2>$, which means the connector links component c_1's output port p_1 and component c_2's input port p_2.

Definition 2. A **route** r is a sequence $[c_1, c_2, ..., c_n]$, $n \geq 1$, where $c_1, c_2, ..., c_n \in C$ are components. On the route, c_1 is an input component that imports data from outside environment. c_n is an output component that exports data to outside environment. $[c_{i+1}, ..., c_n]$ is the subsequence of c_i, $1 \leq i \leq n-1$. A **designed route** is a component sequence pre-designed by designers. A **data route** is a component sequence that a data element has actually passed.

Definition 3. A **route map** R is a set $\{r_1, r_2, ..., r_k\}$, $k \geq 1$, where $r_1, r_2, ..., r_k$ are designed routes. A configuration A **supports** a route map R if $p(A)=R$, where $p(A)$ is the set of data routes possible appearing under A.

3.2. Eliminating functional side effect

Functional side effect results from incorrect data route.

Definition 4. A reconfiguration has **functional side effect** if it may cause a data route $r \notin (R \cup R')$, where R is the route map before reconfiguration and R' is the route map after reconfiguration.

Such a data route r could appear under three conditions:

1) *Datum loss.* A datum loss means a data element is removed from the system incorrectly.
2) *Dead datum,* A dead datum means a data element is still in the system but not able to flow any more.
3) *Wrong route.* A wrong route means a data element has passed through the system but the data route it passed does not equal to any designed route predefined in the route map.

A datum loss or dead datum results in a data route that is an incomplete designed route. Under such a condition, the data route does not equal to a designed route of old or new route map, although it is a prefix of a designed route.

To implement the switching transformation, a reconfiguration should theoretically support an operation that can switch the architecture from the old configuration to the new configuration 'instantly'. The architecture supports the old route map before the switching and the new route map after the switching. Therefore, in Figure 1-a, the result is $F(e)$ if e appears before t and $F'(e)$ if e appears after t.

However, in practice, a real instant switching is impossible because any reconfiguration operation needs a time period to execute. But a reconfiguration operation can be viewed as 'instant' if no inter-component communication is performed during the execution of the operation. It is because a reconfiguration operation has no influence on the running of inner-component processing. Therefore, a practical 'instant switching' is a sequence of reconfiguration operations that satisfy the following constraints (Figure 4):

1) before the execution of these operations, the system architecture supports the old route map and the system is running normally;
2) during the execution of these operations, any inter-component communication are buffered;
3) after the execution of these operations, the communication is resumed and the system architecture supports the new route map. And the old part of the system keeps running until all existing computations complete.

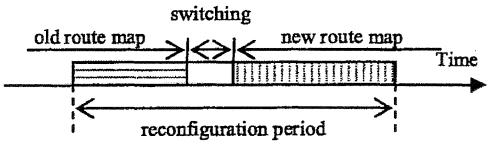


Figure 4. Switching transformation

To implement the coexisting transformation, a reconfiguration should establish the new routes before removing the old routes. As a result, the new routes and old routes will coexist for a period. During this period, an input data element can follow either an old route or a new route. Therefore, in Figure 1-b, the result could be either $F(e)$ or $F'(e)$ if a computation completes during reconfiguration period.

A reconfiguration for coexisting transformation could be divided into three stages (Figure 5):

1) The first stage is the establishment of new route map. During this stage, new routes are established one by one. Once a new route is established, it allows data elements to pass through.

2) The second stage is a short period after the establishment of new route map ends and before the removal of old routes starts.
3) The third stage is the removal of old route map. In this stage, old routes are removed one by one. Before removing a route, it should be guaranteed that no data element is on the route.

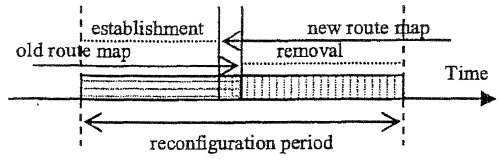


Figure 5. Coexisting transformation

3.3. Managing performance influence

The influence of reconfiguration on system output rate may be caused by two reasons. First, the system has different maximum output rate under different architectural configurations. During reconfiguration progress, the system experiences a series of interim configurations before reaching the target configuration. Under these configurations, the system is likely to have different maximum output rate. Second, the execution of a reconfiguration procedure itself requires some resources, especially the processor time. The reconfiguration procedure is executed along with the functional processes of components on the same node. If the node has a fixed computing capability and it spends some time on the reconfiguration procedure, it spends less time on the functional processes than normal. Therefore there is a practical decline of the system output rate during a reconfiguration even though the reconfiguration procedure does not change the maximum output rate theoretically. We name the first kind of influence as *logical performance influence* and the second as *physical performance influence.*

To control the performance influence of a reconfiguration, both design time and runtime supports need to be considered. The logical performance influence of a reconfiguration is decided by the reconfiguration plan. Thereby, in design time, the operations in a reconfiguration plan should be arranged carefully to ensure the theoretical maximum output rate of every interim configuration. The physical performance influence of a reconfiguration can be controlled through reconfiguration scheduling during plan execution. The reconfiguration procedure should run alternatively with other procedures and the processor time spent on it in a time slice should be restricted in a small percentage.

Although both switching transformation and coexisting transformation have no functional side effect, their logical performance influence is quite different. In the switching period of switching transformation, the theoretical maximum output rate is down to zero since inter-component communications must be paused. Since the switching period is necessary for a switching transformation, this kind of impact on performance is inevitable. However, in a coexisting transformation, such a switching period is not necessary. Both inter-component communication and inner-component processing can run normally during reconfiguration. Therefore, the logical performance influence of coexisting transformation is smaller than that of switching transformation.

To reduce logical performance influence, we choose coexisting transformation as the reconfiguration method. Suppose the theoretical maximum output rate of the old configuration and new configuration are t and t' respectively, the theoretical maximum output rate of the system during a reconfiguration is changed from t to t' since old routes are stopped and new routes are started one by one.

To restrict physical performance influence, we use a reconfiguration scheduler, which can control the processor time spent on the reconfiguration procedure in a time slice. Suppose the theoretical maximum output rate of a system is m and the processor time spent on the reconfiguration procedure is p in time slice s, the actual max output rate is $m\times(1-p/s)$.

3.4. Reconfiguration algorithm

We propose a reconfiguration algorithm, which can implement a coexisting transformation. Suppose the architectural configuration and route map are A, R before reconfiguration and A', R' after reconfiguration. A supports R and A' supports R'. A coexisting transformation includes two main steps:

1) *Establishment* - establishing the new routes.
2) *Removal* – removing the old routes.

But the challenge problem in the reconfiguration is how to ensure version compatibility during the coexisting period, i.e. old version data elements should be distributed to old version components and new version data elements should be distributed to new version components. An example is shown in Figure 6. Component c could process both old version and new version data elements. After being processed by c, old version data elements must be distributed to component d and new version data elements to component d'. The old route is $[a,b,c,d,e]$; the new route is $[a,b',c,d',e]$. Routes $[a,b,c,d',e]$ and $[a,b',c,d,e]$ should not be permitted.

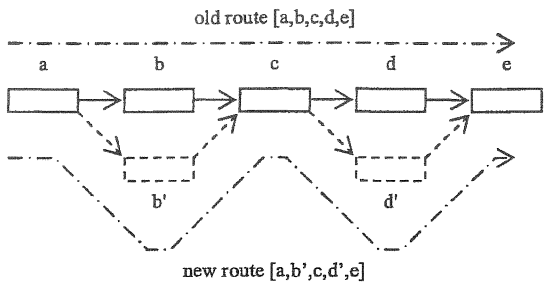


Figure 6. A sample of version compatibility

The requirement of version compatibility is caused by the dependency relationship between components. Component *A* depends on component *B* means that data elements must be processed by *B* before they can be processed by *A*, whether *A* is *B*'s direct descendant or not. In a data transfer system, for example, the decryption component depends on the encryption component.

The dependency relationships in a system belong to the domain of application requirements. Therefore, for a system, the route map must satisfy all the dependency relationships. For a reconfiguration, the old route map should satisfy the old dependency relationships and the new route map should satisfy the new dependency relationships.

Definition 5. A **dependency relationship** *d* is a tuple $<O,Q>$, where O,Q are component classes and Q depends on O. For a route $r=[c_1, c_2, ..., c_n]$ and a dependency relationship $d=<O,Q>$, r **satisfies** d if 1) there is no instance of O or Q in r; or 2) for each instance of O in r there is an instance of Q in its subsequence. For a system, the route map R **satisfies** the dependency relationships if $(\forall r \in R, \forall d \in D)(r$ *satisfies* $d)$, where D is the set of all the dependency relationships of the system.

To ensure version compatibility, different versions of data elements should be distinguishable. Therefore a version tag should be attached to each data element. The tag could be a real one or a virtual one. A real tag is an item in the data structure of data element. A virtual tag is a parameter that indicates the version of data elements in a communication between components.

Components should be able to recognize the version tag and control which version of data elements it can process. A component should support three modes:

1) *Strict(x,y).* The component can only process version x data elements; the results are all of version y.
2) *Transparent.* The component can process any version data elements; the results are of the same version as the data elements processed.
3) *Filter(x).* The component can process any version data elements; but the results are all of x version.

A component can only process the same versions of data elements in a processing in despite of which mode it is. The mode of a component can be changed at any time, but the new mode is brought into effect after the current processing and before next processing.

Using the version control mechanism, the version compatibility can be ensured, even if there are intersections between old routes and new routes. Figure 7 shows a configuration of the components' modes that can support the route map shown in Figure 6. Thus, by arranging components' modes, version compatibility can be ensured in reconfiguration.

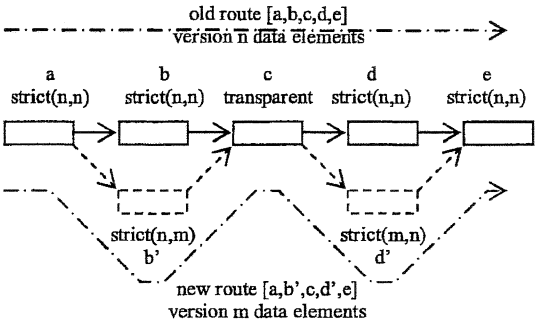


Figure 7. A sample of configuration that supports version compatibility

In the reconfiguration algorithm, all the components are divided into three groups, *removed components*, *added components*, and *reserved components*. The reserved components are further divided into *involved components* and *uninvolved components*. Their definitions are as follows:

The set of **removed components** is $RC = A.C\text{-}A'.C$, where $A.C$ is the set of all the components of configuration A (see definition 1), $A'.C$ is the set of all the components of configuration A'.

The set of **added components** is $AC = A'.C\text{-}A.C$.

The set of **reserved components** is $VC = A.C \cap A'.C$.

The set of **involved components** is $IC = \{c \mid c \in VC$ *and* $((\exists c_i,c_j)(c_i,c_j \in RC \wedge [...,c_i,...,c,...,c_j,...] \in R)$ *or* $(\exists c_i,c_j)(c_i,c_j \in AC \wedge [...,c_i,...,c,...,c_j,...] \in R'))\}$.

The set of **uninvolved components** is $UC = VC\text{-}IC$.

The components in $RC \cup IC$ and the connectors between them form an old subsystem, which can process old version data elements. The components in $AC \cup IC$ and the connectors between them form a new subsystem, which can process new version data elements. A coexisting transformation can be achieved through arranging these two subsystems running in

parallel. To control data elements flow into and out the new subsystem, the algorithm also needs to distinguish the *entrance components* and *exits components* of the new subsystem from other components.

The set of **entrance components** of the new subsystem is $ENC = \{c \mid c \in AC \text{ and } ((\exists b)(b \in VC\text{-}IC \wedge \langle b,*, c,* \rangle \in A'.N) \text{ or } \neg(\exists b)(b \in A'.C \wedge \langle b,*,c,* \rangle \in A'.N))\}$. Here '*' represents any variable.

The set of **exit components** of the new subsystem is $ENC = \{c \mid c \in AC \text{ and } ((\exists b)(b \in VC\text{-}IC \wedge \langle c,*, b,* \rangle \in A'.N) \text{ or } \neg(\exists b)(b \in A'.C \wedge \langle c,*, b,* \rangle \in A'.N))\}$.

We give the reconfiguration algorithm in pseudo code.

```
procedure reconfiguration
var
  A: Configuration; // current configuration
  R: RouteMap; // current route map, A supports R
  A', Configuration; // target configuration
  R', RouteMap; // target route map, A' supports R'
begin
  //1. Set all involved components to transparent mode.
  for each c∈IC do  set c to transparent mode;
  //2. Establish new routes.
  //2.1. start new components and set their modes.
  for each c∈AC do begin
    start c;
    if c∈ENC then  set c to strict(n,m) mode; //m≠n
    else if c∈EXC then  set c to strict(m,n) mode;
    else  set c to strict(m,m) mode;
  endif
  //2.2. connect new components to the system.
  F=ϕ;
  while F≠AC do begin
    // find a added component whose descendants all
    // have been connected to the system
    for each c∈AC-F do
      if ¬(∃b)(b∈AC-F∧[...,c,...,b,...]∈R') then
    break;
    // set up connectors for the component
    for each l=<c,*,*,*>∈A'.N do  set up l;
    for each l=<*,*,c,*>∈A'.N do  set up l;
    F=F∪{c};
  endwhile
  //3. Remove old routes.
  //3.1. wait all old data to pass the old routes.
  for each r=[c1,c2,...ck]∈R-R' do begin
    if ¬(∃<*,*,c1,*>∈A'.N) then
      wait c1 to finish processing version n data;
    for i=1 to k-1 do begin
      for each l=<ci,*,ci+1,*>∈A.N-A'.N  do
        remove l;
      wait ci+1 to finish processing version n data;
    endif
  endif
  //3.2. remove all old components.
  for each c∈RC do  remove c;
  //4. Rearrange modes of component.
  for each c∈EXC do  set c to filter(n) mode;
  for each c∈AC-EXC-ENC do
    set c to transparent mode;
  for each r=[c1,c2,...ck]∈R'-R do begin
    for i=1 to k do begin
      if ci∈ACUIC do begin
        set ci to strict(n,n) mode;
        wait ci to finish processing version m data;
      endif
    endif
  endif
end procedure
```

The algorithm requires all components working in strict(n,n) mode before reconfiguration. And all components are set back to strict(n,n) mode after reconfiguration. Therefore, the system is ready for another reconfiguration.

To use the algorithm, several preconditions must be satisfied. First, *cycle* should not appear in old or new route map. Second, *flow synchronization* is not allowed in old or new route map.

Definition 6. There is a **cycle** on a route $[c_1, c_2, ..., c_n]$, $n \geq 2$, if $\exists i,j,\ 1 \leq i < j \leq n$, such that $c_i = c_j$.

Definition 7. There is a **flow synchronization** if there exist two routes $[c_{11}, c_{12}, ..., c_{1n}]$, $n \geq 2$, and $[c_{21}, c_{22}, ..., c_{2m}]$, $m \geq 2$, in a route map and $\exists i,j,k,l,\ 1 \leq i < j \leq n,\ 1 \leq k < l \leq m$, such that $c_{1i} \neq c_{2k}$ and $c_{1j} = c_{2l}$.

A cycle in the new route map will cause the algorithm to fail to connect all new components to the system in step 2.2. Once a component is connected to the system, the data it produces should be guaranteed to be able to flow continuously. Therefore the algorithm set up the connectors for a component after all the descendants of the component have been connected to the system. But in a cycle, a component is one of the descendants of itself; thereby the algorithm will fall into dead circulation.

A cycle in the old route map will cause the algorithm to be not able to wait all old data to pass the old routes correctly in step 3.1. When the algorithm detects that all old data have flow through, there may be some data still flowing in the cycle.

In a route map, when two routes have flow synchronization, the two routes must transmit equal numbers of data elements to the intersection part; otherwise the redundant data elements could not be processed. The algorithm cannot guarantee exactly the same number of data elements have passed the two routes when the two routes are removed.

3.5. Reconfiguration scheduler

We propose to use a reconfiguration scheduler to control the execution of reconfiguration procedure. In a time slice, the scheduler can suspend the reconfiguration procedure after it has run for a period. And the scheduler should resume the reconfiguration procedure in next time slice.

The requirement on the PMPI on output rate can be represented as a tuple *(t,p)* where *t* is the length of the time slice for output rate statistic, and *p* is a percentage that refers to the actual maximum output rate in relation to the theoretical maximum output rate. Then the time that can be spent on the reconfiguration procedure is *t×(1-p)* per time slice *t*.

The scheduler procedure runs in a high priority; the reconfiguration procedure runs in a middle priority; and other functional procedures run in a low priority. The scheduler procedure wakes up two times in every slice. The first time takes place when a new time slice begins. It resumes the reconfiguration procedure, which is put into running after the scheduler procedure falls into sleep since its priority is higher than other functional procedures. The second time takes place when after *t×(1-p)* time passes. It suspends the reconfiguration procedure so that other functional procedures can possess the rest time in the time slice. Thus, the time spent on the reconfiguration procedure in a time slice t is *t×(1-p)*. The scheduling algorithm is as follows.

```
procedure scheduling
var
   r: Procedure; // the reconfiguration procedure
   t: TimeLength; // the length of the time slice
   p: float;          // the percentage
begin
    while r is alive do begin
       if r is running begin
          suspend r;
          sleep t×p;
       end else begin
          resume r;
          sleep t×(1-p);
       endif
    endwhile
end procedure
```

4. A case study

We use a data broadcasting system to demonstrate how the algorithm and the scheduler work. In a data broadcasting system (Figure 8), data elements are designed to be encrypted by *encrypter* first, and then broadcasted by *broadcaster*. Next, the data elements can be received by all *receivers* and decrypted by *decrypters*. To update the encryption/decryption algorithm, a reconfiguration is needed to replace the *encrypter* and all the *decrypters* with new version ones.

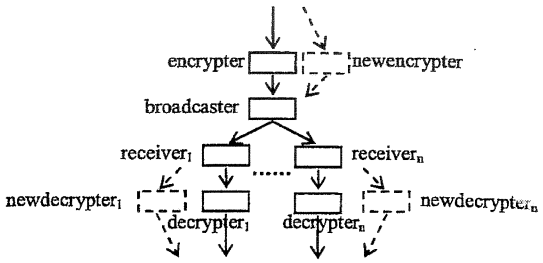


Figure 8. Data broadcasting system

The original system is specified as follows.

Dependency relationship:
<Encrypter, Decrypter>
Architectural configuration:
{{encrypter, broadcaster, receiver$_1$, ..., receiver$_n$,
decrypter$_1$, ..., decrypter$_n$},
{ <encrypter,1,broadcaster,1>,
<broadcaster,1,receiver$_1$,1>,...,<broadcaster,1,receiver$_n$,1>,
<receiver$_1$,1,decrypter$_1$,1>,...,<receiver$_n$,1,decrypter$_n$,1>}}
Route map:
{[encrypter, broadcaster, receiver$_1$, decrypter$_1$], ...,
[encrypter, broadcaster, receiver$_n$, decrypter$_n$]}

The target system is specified as follows.

Dependency relationship:
<NewEncrypter, NewDecrypter>
Architectural configuration:
{{newencrypter, broadcaster, receiver$_1$, ..., receiver$_n$,
newdecrypter$_1$, ..., newdecrypter$_n$},
{ <newencrypter,1,broadcaster,1>,
<broadcaster,1,receiver$_1$,1>,...,<broadcaster,1,receiver$_n$,1>,
<receiver$_1$,1,newdecrypter$_1$,1>,...,
<receiver$_n$,1,newdecrypter$_n$,1>}}
Route map:
{[newencrypter, broadcaster, receiver$_1$, newdecrypter$_1$], ...,
[newencrypter, broadcaster, receiver$_n$, newdecrypter$_n$]}

In the reconfiguration algorithm, some variables are as follows.

RC = {encrypter, decrypter$_1$, ..., decrypter$_n$}
AC = {newencrypter, newdecrypter$_1$, ..., newdecrypter$_n$}
VC = IC = {broadcaster, receiver$_1$, ..., receiver$_n$}
ENC = {newencrypter}
EXC = {newdecrypter$_1$, ..., newdecrypter$_n$}

The parameters for the scheduler are as follows.

t = 1 second
p = 80%

The system is implemented on the Reconfigurable Data Flow (RDF) model, which is a component model on Java platform. We have developed the RDF model as an implementation of our influence control approach. The RDF model is an extension to the

widely used Data Flow (DF) model [3], which focuses on representation of the flow of data through a system. DF model explores data flow diagram as graphical representation and data flow programming [9] as programming framework. For detail description of the RDF model, please reference [18].

In order to highlight our concern about the influence of reconfiguration on system performance, we make the following assumptions in our experiment.

1) Enough data elements are provided for the system to consume, thereby the system works under its maximum workload and the influence of reconfiguration is maximized.
2) The *newdecrypter* and the *decrypter* are exactly the same, thereby the influence on output rate is only caused by structure reconfiguration.

In the experiment, we provide an sequence of natural numbers (1,2,3,...) as the input of the system and record the output and the output rate of every branch through the runtime. We run the system two times. In the first run, the reconfiguration is not executed; and in the second run, the reconfiguration is executed. Comparing the results, we can find the influence of the reconfiguration.

The output:

In the first run

$decrypter_1$: 1,2,3,...

$decrypter_2$: 1,2,3,...

.....

In the second run

$decrypter_1$ & $newdecrypter_1$: 1,2,3,...

$decrypter_2$ & $newdecrypter_2$: 1,2,3,...

...

The output rate: (in dps - data elements per second)

Time slice	1	2	3	4	5	6
First run	413	425	421	432	405	428
Second run	409	373	348	389	417	408

There is no difference between the outputs of the two runs. Every branch outputs a sequence of natural numbers as same as the input, whether the reconfiguration takes place or not. Therefore we can draw the conclusion that the reconfiguration has no functional side effect.

Because of the uncertainty of the thread scheduling, the output rate is not fixed during runtime. We choose the average output rate in first run as the maximum output rate. It is 421dps. And in second run, since *newdecrypter* and *decrypter* are exactly the same, the theoretical maximum output rates before reconfiguration and after reconfiguration should be same to that of the first run. Thus, the max performance influence is $|348-421|/421=17\%<(1-p)$. Therefore, we can say the performance influence of the reconfiguration is in the expected range.

5. Conclusion and future work

In this paper, we have shown how to control the influence of dynamic reconfiguration. First, we analyze the influence of dynamic reconfiguration on system functionality and performance. Then, we use two models, switching transformation and coexisting transformation, to character reconfiguration. They all have no functional side effect, but only the coexisting transformation is suitable for performance influence control. Next, we present a reconfiguration algorithm and a reconfiguration scheduler, which are based on the coexisting transformation reconfiguration model. And finally, we give an implementation of the algorithm and scheduler on the RDF model. The experimental results of have confirmed that our influence control approach is practical for dynamic reconfiguration.

Future work focuses on extension of the reconfiguration algorithm. Because at the current stage, the algorithm requires that there is no ring or flow synchronization on any route of the old or new route maps, further efforts should be made to extend the plan to adapt to these two conditions.

Acknowledgements

This work was supported by Central Queensland University, Australia under Research Advancement Awards Scheme (RAAS) grants, 2006~2007.

References

[1] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation", Proc. 24th International Conference on Software Engineering, Orlando, Florida, USA, 2002, pp.187-197.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. B. Stefani, "An open component model and its support in Java", Proc. 7th International Symposium on Component-Based Software Engineering, Edinburgh, UK, May 2004, pp.7-22.

[3] G. Cheng, A Dataflow-Based Software Integration Model in Parallel and Distributed Computing and Applications, Ph.D. Dissertation, Syracuse University, Italy, 1997.

[Demarco1978] T. Demarco, Structured Analysis and System Specification, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[4] L. Desmet, N. Janssens, S. Michiels, F. Piessens, W. Joosen and P. Verbaeten, "Towards Preserving Correctness in Self-Managed Software Systems", Proc. the Workshop on Self-Managing Systems (WOSS'04), 2004, pp.34-38.

[5] J. Dowling and V. Cahill, "The K-Component architecture meta-model for self-adaptive software". Proc. 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, 2001, pp.81-88.
[6] P. Feiler, J. Li, "Consistency in Dynamic Reconfiguration", Proc. the International Conference on Configurable Distributed Systems, 1998, pp.189-196.
[7] J. Gorinsek, S. Van Baelen, Y. Berbers, and K. De Vlaminck, "Managing quality of service during evolution using component contracts", Proc. 2nd international workshop on unanticipated software evolution, Warsaw, Poland, 2003, pp.57-62.
[8] J. Hillman, and I. Warren, "An Open Framework for Dynamic Reconfiguration", Proc. the 2004 International Conference on software Engineering (ICSE), Edinburgh, UK, 2004, pp.23-28.
[9] W. M. Johnston, J. R. Paul Hanna, R. J. Millar, "Advances in dataflow programming languages", ACM Computing Surveys (CSUR), 36(11), 2004, pp.1-34.
[10] D.C. Luckham et al, "Specification and analysis of software architecture using Rapide", IEEE Transactions on Software Engineering, 21(4), April 1995, pp.336-355.
[11] J. Magee, J. Kramer, "Dynamic structure in software architectures", Proc. 4th ACM SIGSOFT Symposium on Foundations of Software Engineering, San Francisco, USA, Oct 1996, pp.3-14.
[12] 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.
[13] S.R.Mitchell, Dynamic Configuration of Distributed Multimedia Components. Ph.D. thesis, University of London, 2000.
[14] N. De Palma, P. Laumay, and L. Bellissard, "Ensuring Dynamic Reconfiguration Consistency", Proc. 6th International Workshop on Component-Oriented Programming (WCOP 2001), 2001.
[15] F. Plasil, D. Balek, and R. Janecek, "SOFA/DCUP: Architecture for component trading and dynamic updating", Proc. International Conference on Configurable Distributed Systems, Annapolis, Maryland, USA, 1998, pp.43–52.
[16] C. Szyperski, Component software: beyond object oriented programming, 2nd edition, Addison-Wesley, 2002.
[17] J. Zhang, Z. Yang, B. H. Cheng, and P. K. McKinley, "Adding safeness to dynamic adaptation techniques", Proc. ICSE 2004 Workshop on Architecting Dependable Systems, 2004, pp.17-21.
[18] Zhikun Zhao, Wei Li, "Dynamic Reconfiguration with QoS Management", Proc. International Conference on Software Engineering and Applications (SEA2006). Dallas, USA. 2006, pp.387-392.