LogicFence: Architecture for an interactive systems execution monitor

Abstract

LogicFence is a framework that is embedded in the runtime environment of applications to enable seamless and automatic global constraints implementation. LogicFence does not require the applications to follow a specific messaging abstraction. The applications may not be aware themselves of being monitored by LogicFence. LogicFence monitors the application objects directly and controls their activity. Using LogicFence global constraints are specified as a plug-in in the environment. Applications may be executing in their own ways, but LogicFence takes care of their execution by enforcing global constraints and preventing them from entering into states which makes the overall system state unsafe. This paper carries forward the concept of LogicFence, first discussed in [1], describes some modifications, and then goes on to discuss possible implementations of this automatic seamless execution monitoring mechanism for interactive systems.

Introduction:

Any large information system (IS) can be viewed as having two broad concerns: *services* that deals with application logic providing services to users and *coordination* that maintains system wide integrity. There exists a degree of autonomy between services and coordination. Application or service logic may be written in several different forms while bound by the same set of coordination rules. Similarly the coordination logic may be changed due to changes in policy without affecting the service logic. Coordination requirements of an IS are usually modeled using coordination language. Some of the

early approaches to coordination were actually communication abstractions. Later approaches like CSDL[4] and Manifold[2,9] separate communication and coordination logic. The framework Logic Fence discussed in [1] acts at a layer below the application. It is embedded in the virtual machine or at the runtime environment of application process. It enforces coordination constraints by reasoning on the states of application objects. Applications need not be aware that they are being monitored and coordinated by an underlying layer. Constraints are enforced using states among set of applications. A change in state occurs when the application affects the external world by performing an output which is like a commit operation. Along with that, the deployer may use the names of some variables in the application program which determine the state of the application. Inputs can also cause a change in state to denote that the application has accepted it from an external entity. Logic Fence is presently implemented as a prototype in Java which implies that constraint implementation is currently possible only on Java codes. Compiled class files have to be first run through Logic Fence and then deployed and executed under the JVM.

Initial approaches

One particular approach for runtime monitoring is insertion of codes in the application program to check state variables and their states. For Java applications, As the source codes for the application programs are not available, the codes to be inserted for monitoring purpose should also be written in the form of byte codes. However, writing codes in the byte code format directly is quite cumbersome as they are almost illegible resembling machine codes. Hence the procedure needs some round about way to write the monitoring codes in byte code format.

Initially we tried to make changes in the Java compiler itself such that while compiling the compiler will insert the necessary codes for the purpose of monitoring the execution. The compiler will read the constraint file given in the form of a plug-in during the process of compilation and insert monitoring codes accordingly. At the time Sun JDK being not open source software, we started experimenting with IBM Research Virtual Machine (RVM). However, the sheer size of the code, accompanied by lack of enough documentation, made any modification of the virtual machine less feasible. Another compiler called Kafé also had similar kind of problems.

A secondary significant approach is to use a disassembler to convert the byte codes into readable assembly form, insert the monitoring codes therein and assemble it back to byte code format. javap –c which comes with JDK though disassembles java byte codes, yet it does not have a corresponding assembler with it that can reconvert the assembly code back to byte code format. In fact the assembly format produced by javap –c has some limitations that do not allow it to be assembled into byte code format. Hence the idea of disassembling the byte codes using javap –c was not of any use. Jasper was found to be another disassembler which converts the byte codes into assembly format almost similar to that produced by javap –c with the exception that one assembler called Jasmin can read assembly files produced by Jasper and can convert them back to byte code format. Thus using Jasper and Jasmin the modification of the code to enable it to receive execution monitoring commands from LogicFence was made simple.

Overall Architecture

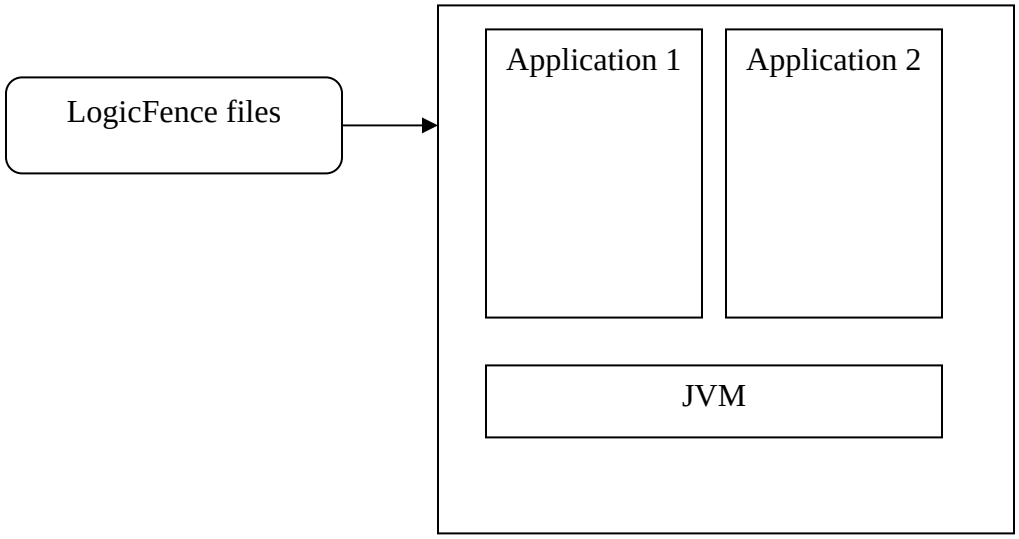

Fig 1

Fig 1 depicts the overall architecture of LogicFence. The compiled Java class files are modified to receive execution monitoring commands from LogicFence, and the main body of the execution monitor is generated from the **interaction schema**[1] specified by the LogicFence deployer. Applications then run in an environment under the constraints specified by the interaction schema, and enforced by LogicFence.

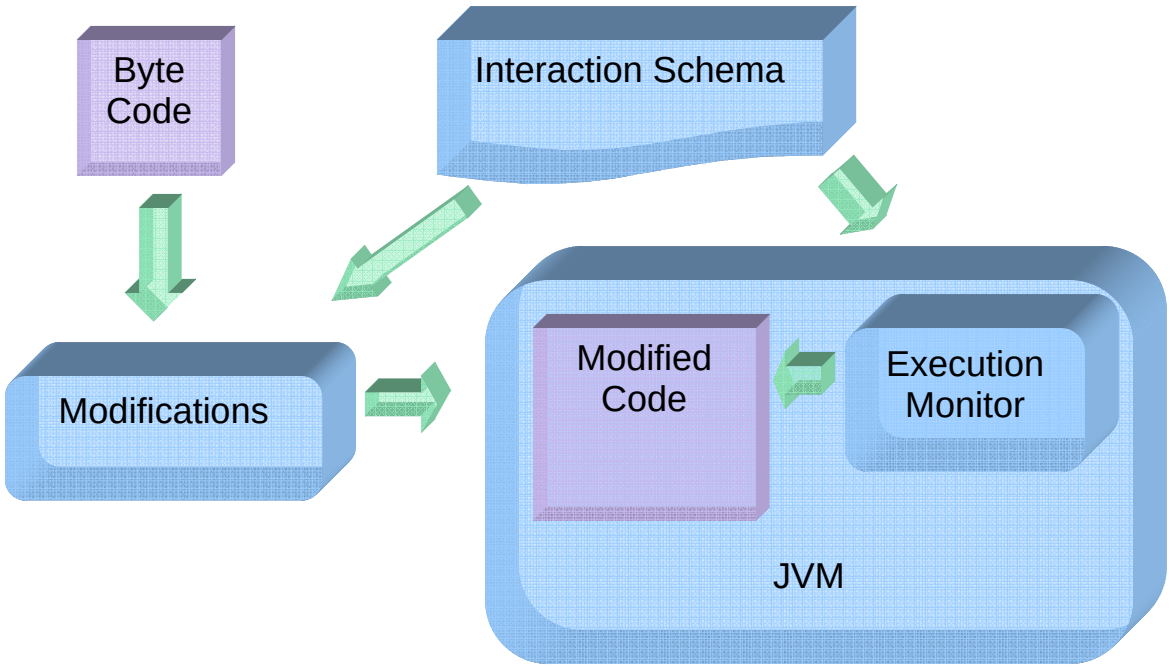

Fig 2

Fig 2 describes the process of deployment of an application using LogicFence. The compiled class files, (the byte code) is first modified to be able to receive execution monitoring directives from LogicFence. These modifications enable the execution monitor to keep a watch on the applications execution states. Then a main body of execution monitor is generated from constraints specified in the interaction schema which tracks the applications execution states and prevents it from entering into an invalid one. The modified application then runs on JVM, while its activities are constantly monitored by the execution monitor thereby assuring that the application does not violate any constraints, as specified by the interaction schema.

Interaction Schema

As discussed in [1] an interaction space defines the dependencies across applications. The interaction space is characterized by coordination across applications. Applications can move in their state space as long as the coordination constraints among them are not violated. The interaction schema(first proposed in [12]) describes this coordination paradigm in a formal way. The schema is specified through building blocks called contracts[1]. A contract can be formally defined as $C = (S, \psi)$. S denotes the states of the shared interaction space across applications. ψ denotes the set of coordination constraints across the interaction state space. The transitions across the shared interaction space states are defined by the application logic. The coordination constraints can be

expressed in the form of State Modality Rules(SMR). Though the application logic drives the application through shared state space, the application itself is unaware of the global rules or coordination constraints across the interaction state space. The application has only partial control over its environment. It is unaware of the coordination and interaction rules it must follow during its transitions in the global state space. LogicFence monitors the applications and does not allow them to enter an invalid state. It is worthy of mention that the applications need not be aware of the presence of LogicFence at all.
An application instance executing a contract is called a channel[1] and the modality rules take the following form: conf $_a$ -> M(conf $_b$), where M may be O, P or F, i.e. obligated, permitted or forbidden. A configuration is a state or a conjunction of states with associated predicates.

Rule	Interpretation
s1 → P[s2]	If a channel or an application is in state s1, then another application or channel can enter state s2.
s1 → F[s2]	A channel is forbidden to enter state s2 as long as there is a channel in state s1.
s1 → O[s2]	The presence of a channel in state s2 is obligated(required) for a channel in state s1 if it wants to leave its current state.

A rule like s1 → F[s2] applies to new channels trying to come to enter s2 but does not affect existing channels in state s2.

The negation of modalities are as: $\neg$ O => P, $\neg$ P => F, $\neg$ F => P. Also all the modalities are idempotent, i.e. O $\wedge$ O => O, P $\wedge$ P => P and F $\wedge$ F => F. For two or more incoming modalities on a particular configuration, the resultant modality becomes a conjunction of all the incoming modalities. The priority of the modalities is as follows: P < F and P < O. This is because in LogicFence safety is considered as more important than liveness. Hence P $\wedge$ F => F and P $\wedge$ O => O. There cannot be two incoming modalities with O and F on a particular configuration since it makes the interaction schema invalid.

In the current implementation of LogicFence, an application can by default move to any interaction space state unless specific constraints prevent it from doing so. Such kinds of systems are termed maximal systems.

Predicate support in LogicFence

In the implementation, the constraints are provided in a more generalized manner with associated predicates. A forbidden constraint is associated with some predicate that depicts a condition which forbids some application from entering a specific state. Similarly a permitted constraint is associated with some condition(s) in the form of predicates that allow an application to enter a state only if the conditions specified by the predicates are satisfied. The predicates have been implemented in the form of auxiliary

functions and hence arbitrary predicates can be added and used to specify constraints. The set of auxiliary functions in the latest version of LogicFence are as follows:

1. *boolean* filled("state_name"): This function checks the presence of applications other than the calling application in the state *state_name*. The calling application is not considered since the validity of the P, F or O constraints, as defined in the previous section, depends on the presence of applications other than the one making the transition.

2. *int* count("state_name"): This function returns the number of applications in the specified state including the calling application, i.e. the application trying to make the transition.

3. *int* numberOf("state_name", "property_conditions"): This function returns the number of applications in the state *state_name* with the specified conditions of the properties. Here is an example: `numberOf("b1", "test_color>=4")`. Here the function returns the number of applications in state b1 with value of property *color* being greater than or equal to 4.

4. *boolean* getVal("property_conditions"): This function checks if the specified property conditions hold for the application trying to make a transition.

Below are some example configurations:

Constraint statement	**Representation in configuration.xml**
An application with a property *color*=red in class *car* is forbidden from entering state *s2* if some other application is in state *s1*. (It implicitly means that all applications with non-red color are always permitted to enter *s2*). Stated otherwise, a configuration with an application in state s2 with property *color*=red is forbidden if some other application is in s1.	`(filled("s1") && getval("car_color=red")) -> F -> s2`
An application with a property *color*=red in class *car* is forbidden from entering state *s2* if some other application is in state *s1*. However, applications with non-red color are always forbidden from entering *s2*.	`(!filled("s1") && getval("car_color=red")) -> P -> s2`
A configuration with an application in state s2 with property *color*=red is permitted if some other application is in s1. (It implicitly means that all applications with non-red color are always forbidden from entering *s2*.).	`(filled("s1") && getval("car_color=red")) -> P -> s2`

An application with a property *color*=red in class *car* is permitted to enter state *s2* if some other application is in state *s1*. However, applications with non-red color can always enter *s2*.	`(!filled(“s1”) && getval(“car_color=red”)) -> F -> s2`
An application with a property *color*=red in class *car* can leave state *s2* only if some other application is in state *s1*. (It implicitly means that all applications with non-red color are never allowed to leave *s2*).	`s2 -> O -> (filled(“s1”) && getval(“car_color=red”))`
An application with a property *color*=red in class *car* can leave state *s2* if some other application is in state *s1*. However, applications with non-red color can always leave *s2*.	`s2 -> O -> ((filled(“s1”) && getval(“car_color=red”)) || (getval(“car_color<>red”)))`
Entry in a narrow bridge (state s1) is forbidden if (property) *direction* of the entering car (application) is different from the cars (applications) already present on the bridge.	`(filled(“s1”) && appl_s1[0].direction != car_direction) -> F -> s1`
Two applications, one in state *s1* and another in state *s3* are required to forbid an application from entering state *s2*.	`(filled(“s1”) && filled(“s3”)) -> F -> s2`
An application is allowed to leave state *b1* only if **i)** there are more than 1 application in state *b1* (the one that is trying to leave *b1* can also be considered) with property *number*, defined in state *test*, greater than or equal to 4 and **ii)** the *test* value of the application trying to make the transition should be greater than 2.	`b1 -> O -> (numberOf("b1", "test_number>=4")>1 && getval("test_number>2"))`

Here is an example of configuration.xml file that contains all the configurations for the system running in LogicFence framework.

```
<db_host_ip>localhost</db_host_ip>
<state_properties>int test_color;boolean test_wheels</state_properties>
<config><state_used>b1</state_used>b1 -> O -> (numberOf("b1",
"test_color>=4")>1 && getval("test_color>2"))</config>
<config><state_used>b1</state_used>(filled("b2")) -> F -> b2</config>
<config><state_used>b1</state_used>(filled("b1") || filled("b3")) -> P
-> b2</config>
<config><state_used>b1</state_used>b3 -> O -> (filled("b2"))</config>
```

Here *color* and *wheels* are properties defined in test class of the application program. These properties also need to be defined in the file properties.xml.

LogicFence Constraint Enforcement

Coordination constraints are enforced by preventing applications from entering an illegal state. An application's path of execution (i.e. which all states it transits through) is determined at runtime by various factors, primarily the inputs from other applications and the environment. During the lifetime of an application, it performs various outputs affecting the external environment and hence causes a change in state as an output is equivalent to a commit operation. Also various inputs and computations cause changes in the internal variables of the application. The change of state of an application is reflected in the changes of values of some of these internal variables. Change in state can also be specified in terms of inputs accepted by the application. Thus when an external entity like an execution monitor wants to keep track of the various states of an application, it can do so by watching the outputs and inputs performed by the application and those specific variables which reflect changes in the applications state.

Consequently, for deploying an application in a monitored environment, it is necessary to know the names of those variables in the application, which reflect the state of the application's execution.

In LogicFence, the interaction schema designer is expected to know the inputs and outputs performed by the application. Thus the schema designer specifies the current state of the application and the inputs and the outputs that cause a transition to the future state. The exact input and output need not be required to be specified by the schema designer. The schema designer can specify the input or the output in the form of regular expressions as recognized by Java. The transitions caused by outputs are specified by the schema designer in the outputStateDefs file. In the current implementation, this is an XML file. LogicFence keeps track of the current state and the next state is defined based on the current state and the specific input. An example outputStateDefs file looks as follows:

```
<states>
      <State>
            <current>s0</current>
            <output type=yes>0</output>
            <future>b2</future>
      </State>
</states>
```

Similarly the schema designer should also specify the state transitions corresponding to inputs. The transitions caused by inputs are specified by the schema designer in the inputStateDefs file. An example inputStateDefs file looks as follows:

```
<states>
      <State>
            <current>b2</current>
            <input type=yes>1</input>
            <future>b1</future>
      </State>
      <State>
```

```
            <current>b1</current>
            <input type=yes>hello</input>
            <future>s4</future>
      </State>
</states>
```

The attribute ‘type’ may be of the following types: yes, no, yes_db and no_db. The description of each of them is as follows:
i) yes: This denotes the presence of the regex in the output or input.
ii) no: This denotes the absence of the regex in the output or input.
iii) yes_db: This denotes the presence of the regex in the output or input provided the output is a database output like create, insert, update, delete statements etc.
iii) no_db: This denotes the absence of the regex in the output or input provided the output is a database output like create, insert, update, delete statements etc.
The database IO is different from other types of IO as the IO strings are matched in a case insensitive manner and hence requires different attributes. Between the <State> and </State> tags one can specify multiple input tags. A conjunction will be formed among the different inputs or outputs. Let us illustrate with the following example of the inputStateDefs.xml file.

```
      <State>
            <current>b1</current>
            <input type=yes>.{5}</input>
            <input type=yes>.*[a-c]{2}.*</input>
            <input type=no>.b</input>
            <future>b2</future>
      </State>
```

It denotes that if the current state of the application is ‘b1’ and if the input has a length of 5 characters with at least two contiguous characters from ‘a’, ’b’, ’c’ with ‘b’ ***not*** being followed by any character, then the future state is ‘b2’.

The interaction schema designer is also expected to know the names of the variables to be watched. In this literature, we refer to those variables (which reflect changes in an application’s state) as State variables. Thus the schema designer specifies the names of the state variables in the StateVariablesDefinitions file. In the current implementation, this is an XML file, which lists down all the state variables in the application which will be deployed under LogicFence. LogicFence watches these variables to determine current state, and state changes in the application.

An example StateVariablesDefinitions file is as follows:

```
<StateVarDef>
<State>
                  <Variable>
                        <class> Account </class>
                        <Type>long</Type>
                        <Name>balance</Name>
```

```
                    </Variable>
                    <Variable>
                        <class>Account</class>
                        <Type>long</Type>
                        <Name>Amount_withdrawn</Name>
                    </Variable>
</State>
</StateVarDef>
```

The schema designer also defines a StateDefinitions file. In the current implementation, this also, is an XML file, which lists the various states of the application in terms of the values of the state variables.

```
<states>

<State>
<name> Offer_premium_services </name>
<varused> Account_balance </varused>
<condition> Account_balance > 50000 </condition>
</State>

<State>
<name> impose_fine </name>
<varused> Account_balance </varused>
<condition> Account_balance < 500 &&
        Account_balance > 0
</condition>
</State>

<State>
<name> Overdraft </name>
<varused> Account_balance </varused>
<condition> Account_balance < 0 &&  </condition>
</State>

</states>
```

LogicFence keeps track of the application by looking at the state variables and the outputs and inputs performed by the application and determines the states as described in these state definition files. The steps involved in tracking the execution steps of the applications are as follows:

The third party compiled Java file is received along with the state variable definition files. Below is an example of the output of a simple disassembled Java file.

The compiled Java file is disassembled using Java disassembler ‘Jasper’.

The original Java file:

```
public class HelloWorld{
        int x;
        int y;

        public static void main(String args[])
        {
                HelloWorld h1 = new HelloWorld();
                try{
                BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
                System.out.println("Enter x: ");
                String ip=br.readLine();
                h1.x=Integer.parseInt(ip);
                System.out.println("Enter y: ");
                ip=br.readLine();
                h1.y=Integer.parseInt(ip);

        } catch(Exception e){
        }

        }
}
```

The disassembled form using Jasper:

```
.source              HelloWorld.java
.class               public HelloWorld
.super               java/lang/Object

.field               x I
.field               y I

.method              public <init>()V
  .limit stack       1
  .limit locals      1
  .line              15
  aload_0
  invokespecial      java/lang/Object/<init>()V
  return
.end method
```

```
.method              public static main([Ljava/lang/String;)V
  .limit stack       5
  .limit locals      4
  .line              21
  new                HelloWorld
  dup
  invokespecial      HelloWorld/<init>()V
  astore_1
  .line              23
LABEL0x8:
  new                java/io/BufferedReader
  dup
  new                java/io/InputStreamReader
  dup
  getstatic          java/lang/System/in Ljava/io/InputStream;
  invokespecial      java/io/InputStreamReader/<init>(Ljava/io/InputStream;)V
  invokespecial      java/io/BufferedReader/<init>(Ljava/io/Reader;)V
  astore_2
  .line              24
  getstatic          java/lang/System/out Ljava/io/PrintStream;
  ldc                "Enter x: "
  invokevirtual      java/io/PrintStream/println(Ljava/lang/String;)V
  .line              25
  aload_2
  invokevirtual      java/io/BufferedReader/readLine()Ljava/lang/String;
  astore_3
  .line              26
  aload_1
  aload_3
  invokestatic       java/lang/Integer/parseInt(Ljava/lang/String;)I
  putfield           HelloWorld/x I
  .line              27
  getstatic          java/lang/System/out Ljava/io/PrintStream;
  ldc                "Enter y: "
  invokevirtual      java/io/PrintStream/println(Ljava/lang/String;)V
  .line              28
  aload_2
  invokevirtual      java/io/BufferedReader/readLine()Ljava/lang/String;
  astore_3
  .line              29
  aload_1
  aload_3
  invokestatic       java/lang/Integer/parseInt(Ljava/lang/String;)I
  putfield           HelloWorld/y I
  .line              32
LABEL0x44:
  goto               LABEL0x48
```

```
  .line          31
LABEL0x47:
  astore_2
  .line          34
LABEL0x48:
  return
  .catch                java/lang/Exception from LABEL0x8 to LABEL0x44 using
LABEL0x47
.end method
```

Portions of the disassembled code, where there are output statements, call to the execution monitor function checkTransitionOnOutput() is inserted before the output statement. The call to the execution monitor function is as follows:

```
  .....
  .....

  .line          24
  getstatic        java/lang/System/out Ljava/io/PrintStream;
  ldc            "Enter x: "
LABEL24:
  dup
  invokevirtual      java/lang/Object/toString()Ljava/lang/String;
  ldc                                                    "          invokevirtual
java/io/PrintStream/println(Ljava/lang/String;)V"
  getstatic        ldList/appId I
  invokestatic
ChkStSpace/checkTransitionOnOutput(Ljava/lang/String;Ljava/lang/String;I)I
  iconst_1
  if_icmpeq         LABEL30
LABEL26:
  ldc2_w           1500
  invokestatic       java/lang/Thread/sleep(J)V
LABEL28:
  goto             LABEL24
LABEL30:
invokevirtual      java/io/PrintStream/println(Ljava/lang/String;)V

  .....
  .....
```

Similarly for portions of the disassembled code, where there are input statements, call to the execution monitor function checkTransitionOnInput() is inserted after the input statement. The call to the execution monitor function is as follows:

```
  .....
```

```
  .....

  .line          28
  aload_2
  invokevirtual      java/io/BufferedReader/readLine()Ljava/lang/String;
LABEL32:
  dup
  invokevirtual      java/lang/Object/toString()Ljava/lang/String;
  getstatic        IdList/appId I
  invokestatic       ChkStSpace/checkTransitionOnInput(Ljava/lang/String;I)I
  iconst_1
  if_icmpeq        LABEL38
LABEL34:
  ldc2_w           1500
  invokestatic       java/lang/Thread/sleep(J)V
LABEL36:
  goto           LABEL32
LABEL38:
  astore_3

  .....
  .....
```

Also portions of the disassembled code, where the state variables change values, are noted and a call to the execution monitor function checkTransitionValidity() is inserted before the assignment of the new value. The call to the execution monitor function is as follows:

```
  .....
  .....

  .line          26
  aload_1
  aload_3
  invokestatic       java/lang/Integer/parseInt(Ljava/lang/String;)I
LABEL8:
  dup
  invokestatic       java/lang/Integer/toString(I)Ljava/lang/String;
  ldc            "HelloWorld_x"
  getstatic        IdList/appId I
  invokestatic
Chk_stspace/checkTransitionValidity(Ljava/lang/String;Ljava/lang/String;I)I
  bipush           1
  if_icmpeq        LABEL14
LABEL10:
  ldc2_w           1500
  invokestatic       java/lang/Thread/sleep(J)V
```

```
LABEL12:
  goto          LABEL8:
LABEL14:
  putfield      HelloWorld/x I

  ......
  .......
```

The modified file is assembled back to a valid Java class file using the Java assembler *Jasmin*. The modified byte code takes care of the integrity constraints by giving call to the checkState() function that does not allow the application to enter an invalid state.

The execution monitor functions, *checkTransitionOnOutput()*, *checkTransitionOnInput()*, and *checkTransitionValidity()* are generated from the State Definitions files, which include the different states the application can move through, and the Constraints Specification file which includes the constraints incident on those states.

Some important details regarding these functions are as discussed below:

1. This function is used to check the validity of a state transition. If a transition is found to be illegal at any point in time, the application waits there until the transition is found to be valid. The application is made to sleep for a time interval after which it wakes up and checks for the validity of the next state transition. The outputs are performed only when the transition is found to be valid and after the application enters the new state.

The functions described above reside in a class called ChkStSpace which gives call to other functions as mentioned below:

determineCurrState() //the function has a copy of the values of all the state variables in the application at the current instant. Using these it determines the current state of the application. Returns the state name.

determineFutureState() // this function has all the constraints as generated from the constraints file. Using the value passed in the transition validity functions, i.e the requested new value of a state variable or the specific outputs or inputs, it finds out the state the application is trying to transit to and returns this future state.

want_to_leave() // this function checks the exit status of the application from the current state depending upon the obligatory constraint specified in the constraints file. Using the

rules in it, it checks whether this transition is valid or not. This procedure is described in detail below. If valid, the function returns TRUE, else returns FALSE.

want_to_enter() // this function checks the entry status of the application in the future state depending upon the permitted and forbidden constraint specified in the constraints file. Using the rules in it, it checks whether this transition is valid or not. This procedure is described in detail below. If valid, the function returns TRUE, else returns FALSE.

If the transition is allowed, i.e if TRUE is returned then ,

- The output is performed if the state transition is caused due to an output.
- The current state of the application is updated so as to make it visible to the outside world. The updating is done with the help of a database as described later.
- The control is returned to the application with the new value assigned to the state variable if the state transition occurs due to change in the value of a state variable.

If the transition is not allowed, i.e. if FALSE is returned,

- The application is made to sleep for a specific time period as provided by the deployer in the file sleepTime.txt.
- The application is again woken up after the specified time period and it checks for the validity of the state transition.
- The above steps continue until the transition to next state is found valid.

Limitations of the current implementation

The framework currently causes state changes of the applications because of inputs and outputs caused by standard methods. For example, in file, console and socket IO, the methods from the reader and writer classes are only tracked. Similarly, for database IO, the mostly used methods in Statement interface like execute(), executeQuery() and executeUpdate() will cause the state change.

Determination of validity of state transition from a global snapshot of state space

The constraints in LogicFence can be global constraints across states. Hence to determine the validity of a state transition, it is necessary to have a snapshot of at least a subset of the global state space.
In the current implementation this is achieved with the help of a database. The database maintains a single table called state_tabl which among other attributes has the following attributes: statename, and no_of_applications. One tuple is created in the database corresponding to every state. When an application wants to make a transition from the current state:

- It checks the obligatory constraints from the current state. The presence of some other applications in one or more of the states holding an obligatory

constraint from the current state will allow the application to leave the current state.

- Once the application finds out the next state where the transition takes place, it checks the validity of the transition. The validity of the transition is determined from the state_tabl table looking at the states holding permitted and forbidden constraints on the next state. If the transition is found to be valid, it locks all the states which hold a permitted or a forbidden constraint on the next state. Theoretically an exclusive lock is required for the state from which the application exits and the state in which the application enters. Shared lock is required for the states which have an obligatory constraint on the current state and those having permitted and forbidden constraints on the future state. In the current implementation however a MySQL database is used where the complete table is locked exclusively for the transactions to take place.
- With the state transition, the records corresponding to the old and new states are updated and the locks are released.

Future work

1. In the version 2.0 of LogicFence one needs to provide all the constraints or configurations before the applications are registered with the framework. If one needs to add or delete constraints then all the applications are to be registered with the framework having the new set of constraints. The applications will be restarted once again from their initial state and they will follow the new set of constraints. However in real world restarting all the applications once again from the start state is not always desirable. Hence considering practical systems, one should be allowed to add or delete constraints without stopping any of the applications. The framework will gradually adapt itself to the new set of constraints and the new set of constraints will be enforced across all applications. The next version of LogicFence aims at including this feature.

References:

1. S. Srinivasa, S. Mukherjee, A. Hegde. *LogicFence: A Framework for Automatic Enforcement of Coordination Constraints*

2. F. Arbab, I. Herman. Manifold. *Future Generation Computer Systems.* Vol. 10 , pp 273-277, June 1994

3. F. Depaoli , F. Tisato . Development of a Collaborative Application in CSDL . proc. of 13th Int'l conf. on Distributed Computing Systems, 1993, Pittsburg, USA, pp .210-217

4. G. A Papadopoulos , F . Arbab, IWIM Model for Coordination of Concurrent Activities. Proc. First Int. Conf. on Corordination Models and Languages, Lecture notes in Computer Science , Vol. 1061, pp. 34-56, April 1996.

5. Srinath Srinivasa. An Algebra of Fix Points for Characterizing Interactive Behavior of Information Systems, PhD Thesis Brandenburg Technical University at Cottbus, Germany, April 2001.

6. Jasper reference: http://www.angelfire.com/tx4/cus/jasper/
7. Jasmin reference: http://jasmin.sourceforge.net/
8. I. Stoica, R. Morris, D. L. Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, H. Balakrishnan, Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications
9. Kaffe reference: http://www.kaffe.org/
10. RVM reference: http://jikesrvm.sourceforge.net/
11. http://linuxgazette.net/issue81/sandeep.html (ptrace reference)
12. http://en.wikipedia.org/wiki/Emergence (Emergent properties/behaviour)

Ptrace() is highly dependent on the architecture of the underlying hardware. Applications using ptrace are not easily portable across different architectures and implementations(11).