LogicFence: A Framework for Enforcing Global Integrity Constraints at Runtime

Shibashis Guha, Srinath Srinivasa, Saikat Mukherjee, Ranajoy Malakar
International Institute of Information Technology
Bangalore 560100, India
{shibashis.guha,sri,saikat.mukherjee,ranajoy.malakar}@iiitb.ac.in

Abstract

Large information systems (IS) comprise of several independent applications that share a common set of resources and data. Usually, there are implicit and subtle dependencies across these applications that are not specifically captured. This is especially so if the applications are bought off the shelf or are developed by independent third parties. Dependencies or global semantic constraints are difficult to discern and incorporate into the design of individual software components. Global constraints may change over time and it is usually expensive or infeasible to change individual application logic in every such situation. In order to address such an issue, we propose LogicFence, a framework that accepts a definition of global constraints and translates these constraints into primitives that are embedded into the run-time environments of application programs (currently, into the JVM of Java applications). Once here, LogicFence monitors the state of application programs and prevents the disparate instances to collectively form a globally inconsistent state.

Keywords*: Coordination, Constraint enforcement, Interaction schema, Contracts, Reference monitor*

1. 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 and dependencies across applications.

Large scale information systems are made of independent applications that collectively provide services to end users. In doing so, they share a common set of resources and data. While each application is individually concerned about its service logic, the more subtle issues of coordination across disparate application instances often go unnoticed. Even when dependencies across applications are known, distributing these dependencies and embedding them into the application logic is often an error-prone process.

From the end-user perspective, code that goes into managing coordination, amounts to wasted code since it does not add anything towards the services themselves. Moreover, with increase in number of applications in the system, the computations required for enforcing coordination constraints increase dramatically. In addition, subtle changes in coordination requirements occur much more frequently than changes to service logic. Among other things, coordination requirements may change due to re-organization of the information system, changes in business processes, or due to change in policies of the organization or government. Exception handling is another situation where spurious coordination requirements suddenly manifest. If every such change in coordination requirements necessitates a change in application logic, the process becomes cumbersome and error-prone. This problem is aggravated when information systems are built by integrating application components that are bought off the shelf. In such cases, changing application logic may even be infeasible. In order to address this issue, we propose a tool called "LogicFence" that acts as a safeguard against semantic violations of global integrity constraints. While local constraints (or service constraints) are handled by application logic, LogicFence takes a declarative formulation of global integrity and enforces these global integrity constraints (or coordination constraints) in a transparent fashion. LogicFence is embedded into the run-time of application programs, where it monitors the application state and interacts with other LogicFence instances across the information system. A separate LogicFence instance is created corresponding to every application. The disparate instances of LogicFence collectively prevent the system from entering into a globally inconsistent state.

Figure 1 depicts the overall architecture of LogicFence. LogicFence is embedded in the virtual machine or run-time

10th International Database Engineering and Applications Symposium (IDEAS'06)
0-7695-2577-6/06 $20.00 © 2006 IEEE

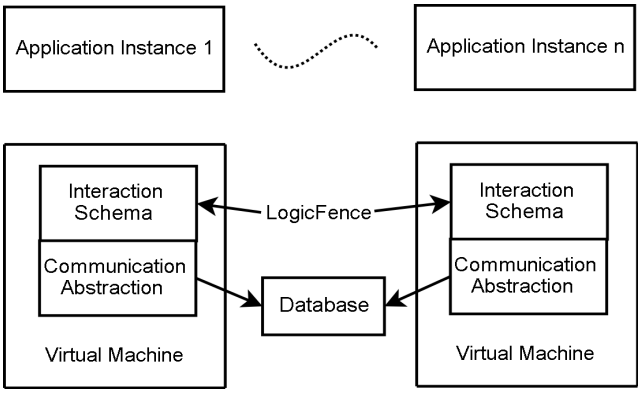

Figure 1. Overall Architecture of LogicFence

environment within which application programs run. LogicFence comprises of two layers: an interaction schema and a communication abstraction. The interaction schema is a declarative framework for specifying coordination constraints in the form of contractual norms. It is explained in more detail in section 3. The communication abstraction layer provides a uniform abstraction using which disparate instances of LogicFence can communicate. Note that communication abstraction across LogicFence instances need not have any bearing over the communication abstraction (if any) used by application instances. In addition to acting as a security barrier, LogicFence also simplifies the construction of information systems, since application developers can concentrate on service logic without having to unduly worry about coordination with other instances.

2. Literature Survey and Positioning

Coordination languages have been extensively studied for specifying coordination primitives. Coordination languages are broadly divided into two types: Data driven and Control driven. Though the data driven category is mostly used for parallelizing computational problems and control driven languages are primarily used for modeling systems [6], this form of separation is not very rigid. Examples of data driven coordination languages are Linda [1], Law governed Linda [5], Laura [13] etc. The communication across processes takes place through a shared data space called tuple space (Linda) [1] or service space (Laura) [13]. However these coordination languages do not separate services from coordination. The coordination logic has to be embedded into the application logic by the applications themselves. In control driven coordination languages like Manifold [2], a set of "manager" processes contain coordination logic that can manipulate connectivity among the ports of "worker" processes. Application logic at the "workers" should be able to read manager commands from its input ports and comply with them. This is unlike the LogicFence model, where applications are not aware of the presence of LogicFence.

LogicFence can also be seen as a framework that enforces security features in a system. Conventionally, security framework techniques addressed the issue of secure executions of individual programs. This is achieved through sandboxing [11] and employing execution monitoring techniques.

Model carrying code [10] provides a framework that secures the execution of mobile code in the consumer's machine. It allows the consumer to check if the actions of the mobile code adhere to his or her security policy. The consumers have the opportunity to refine their security policies according to the functional requirements of the mobile code. The mobile code comes with a model of its behavior that can be checked manually. At run time, execution monitoring techniques are used to ensure that the code does not violate the consumer's policies. However the model carrying code does not capture dependencies across applications and also the security constraints cannot be changed while the applications are already running.

Security Automata [8] employs execution monitoring for secure execution of applications. It is a non-deterministic finite automaton that enforces safety properties [8] as follows. A 'target' is monitored at run time. At each step the target execution generates an input for the security automaton. If the automaton can make a transition on the input symbol, then the target is allowed to perform that step and the automaton state is changed according to its transition function. If the automaton cannot make a transition on that input symbol, then the target is terminated or halted, for attempting a security violation. Security automata can be regarded as defining reference monitors [3]. A reference monitor observes a target and halts the target whenever it is about to violate some security concern. The state of the automaton is encoded with the help of state variables. The execution of the target governs the state transition of the automaton. In other words, the state of the automaton reflects the execution state of the target. Though theoretically the security automaton is expressive enough to define any security policy that is enforceable using execution monitoring, the inputs for a security automaton are mostly defined in the form of system calls of the target execution. So the system calls are traced for possible state transitions of the automaton. The program counter is also treated as a state component. Hence at a functional level, the behavior of the target program may not always be captured; rather, state transitions are defined on the structural constructs. For example, a loop may cause the same value for a program counter and hence the same state for the security automaton but from a functional viewpoint each execution of the loop may cause inputs and outputs or change of state variables in the tar-

get and may lead to different states. Besides an automaton based on program counter cannot be defined properly for dynamically linked programs. The Security Automata SFI Implementation (SASI) [3] is based on Software Fault Isolation (SFI), which enforces security policies that prevents reads, writes, or branches to memory locations outside of certain predefined memory regions associated with a target system. The prototype implementation merges security policy enforcement code into the object code of the target system.

LogicFence shares a number of features with security automata, but has certain significant differences as well. A system with security automata forms a *minimalist* system, where any behavior that is not explicitly permitted by a transition in the automaton is forbidden. LogicFence on the other hand, is a *maximalist* system. LogicFence only monitors application states and does not dictate how applications should traverse across states. Application logic can be as rich and expressive as the developer sees fit; however, disparate applications will not be allowed to collectively form a globally inconsistent state when LogicFence is monitoring them. Hence every behavior that is not explicitly forbidden by LogicFence is permitted.

At an implementation level in SASI, while there is no provision for changing the constraints at run time, in LogicFence, constraints can be changed at run time without necessarily stopping the applications. The applications will follow the new set of constraints even when they are in the middle of their execution. Finally, the SASI framework monitors individual application states and does not provide a way to express global dependencies spanning across different application instances.

There are learning automata as well for detecting anomalous program behavior. A finite state automaton (FSA) is built by observing several executions to model program behavior. Learning the program behavior and building the corresponding FSA can be done at run time [9]. Execution monitoring based on the FSA can be done at run time by intercepting system calls. However it may not always be possible to build the entire automata corresponding to the program. Also the FSA based method cannot capture constraints at an application level abstraction since security measures are enforced by tracing the system calls. Moreover, it does not take care of dependencies across multiple applications in a system. Our approach also has significant differences from Petri Net based process models. Petri Net is an imperative model where the process operation is described. In our approach, we define a schema with a set of constraints required to prevent the applications from forming a globally inconsistent system state. The novelty of LogicFence can be summarized as follows:

- The coordination constraints are enforced in a completely transparent manner. During design phase, the applications need not be aware of the fact that they will be monitored by LogicFence. Hence, in principle, even legacy applications can also be instrumented with LogicFence.

- The application logic does not need to capture system-wide dependencies and integrity constraints. Hence the development and maintenance of application code will be easier compared to the case where global dependencies are coded in the application logic.

- Any change in coordination logic requires changes in the constraints model and not in application code. However, it must be mentioned that LogicFence can only act as a safeguard against constraint violation but it does not suggest the application any change in its execution strategy once it gets halted for any attempted constraint violation.

- LogicFence is a maximalist system and does not unduly restrict applications from providing richer behavioral semantics as long as they don't violate constraints.

- It is easier to incorporate new constraints or applications in the system since the incorporation will not require any change in the existing components.

3 Interaction Schema

Coordination constraints in LogicFence are defined with an "interaction schema" (first proposed in [12]). The interaction schema is a characterization of the "interaction space", which is the global shared state space of application dynamics. Applications can move in the interaction space as long as coordination constraints among them are not violated. The schema is specified through building blocks called coordination contracts. A coordination contract is formally defined as: $C = (S, \psi, \delta)$,where S denotes the shared state space of all applications using the contract C, and ψ denotes a set of coordination constraints across states in S. Applications traverse the shared state space S and in doing so, may affect other applications due to the coordination constraints. The contract itself does not specify state transitions; they are defined by application logic.

However, the contract specifies a small number of "I/O transitions", represented by the term δ. Every input or output operation by an application necessitates a state change and needs to be specified in the contract. This is because, interaction with the external world is irreversible and can be performed only when it is safe to do so. All other state transitions made by the application, should not involve any input/output activity. The term I/O refers to any interaction made by the application or a component under observation,

Table 1. Constraint Definition

Rule	Interpretation
$s_1 \rightarrow P(s_2)$	If an application is in state s_1, only then another application can enter state s_2.
$s_1 \rightarrow F(s_2)$	An application is forbidden from entering state s_2 as long as there is an application in state s_1.
$s_1 \rightarrow O(s_2)$	The presence of an application in state s_2 is obligated (required) for an application in state s_1 if it wants to leave its current state.

with some other component outside of its control. An I/O transition is specified in the form: $cs \xrightarrow{i/o} fs$.

This specifies that an application in state cs should move to state fs on reading an input i and give an output o. Every input necessitates a state change, and every output can be provided only on reaching the target state. In any I/O transition specification like the above, either *i* or *o* can be omitted.

Coordination constraints in ψ are expressed in the form of State Modality Rules (SMR). The modality rules take the following form: $head \rightarrow M(body)$, where $head$ and $body$ are "configurations" and M is either O, P or F, standing for *obligated*, *permitted* or *forbidden*, respectively. A rule of the above form mentions that when configuration $head$ holds then configuration $body$ gets a modality M.

A configuration is a *conjunction* or *disjunction* of terms or their *negations*. A term here can be one of the following:

1. state name.
2. *predicates* associated with a state.
3. *conjunction* or *disjunction* of smaller *terms* or their *negations*.

The permitted (P) and forbidden (F) modalities specify constraints on the *formation* of configurations whereas the obligated modality (O) specifies constraints on the *dissolution* of a configuration. For a constraint of the form $head \rightarrow P(body)$, if *head* holds, only then *body* is allowed to form. Similarly, for a constraint of the form $head \rightarrow F(body)$, if *head* holds, then *body* is prevented from being formed.

On the other hand, for a constraint of the form $head \rightarrow O(body)$, *body* is obligated to exist when *head* holds. In other words, *head* cannot cease to hold until *body* is true.

In their simplest form, configurations appear as described in 1 where *head* and *body* are state names.

A rule like $s_1 \rightarrow F(s_2)$ or $s_1 \rightarrow P(s_2)$ applies to new applications trying to enter s_2 but does not affect existing applications in state s_2.

In an active constraint of the form $head \rightarrow M(body)$, if the configuration *head* ceases to hold, then *body* gets a modality $\neg M$. The negations of modalities are given as follows:

- $\neg O \Rightarrow P$
- $\neg P \Rightarrow F$
- $\neg F \Rightarrow P$

For two or more incoming modalities on a particular configuration, the resultant modality becomes a *conjunction* of all the incoming modalities. All the modalities are idempotent, i.e. $O \wedge\ O \Rightarrow O$, $P \wedge\ P \Rightarrow P$ and $F \wedge\ F \Rightarrow F$. When there are constraints of different modalities acting on a configuration, then F is given a higher priority than P. This is because in LogicFence safety is considered to be more important than liveness. Hence $P \wedge\ F \Rightarrow F$. Obligation is given more priority over permission. Hence $O \wedge\ P \Rightarrow O$ and any application waiting due to the obligation will continue to wait. (Note that obligation implies permission. Obligation is stricter in that the configuration is not only allowed to form, but the dependant configuration is also required to wait till the obligation is fulfilled).

Since obligation acts in the reverse direction, the conjunction effect is also reversed. That is, if $conf_a \rightarrow O(conf_b)$ and $conf_a \rightarrow O(conf_c)$ hold, then $conf_a$ can be dissolved only when both $conf_b$ and $conf_c$ hold together.

There cannot be two incoming modalities with O and F on a given configuration $conf_b$ from another configuration $conf_a$ since it makes the interaction schema invalid. Also, a constraint of the form $conf_a \rightarrow P(conf_a)$ is not allowed semantically. However the interaction schema by itself, does not specify any rule to check such semantic errors. There are also possibilities of deadlocks, which are not specifically prevented by the schema. Separate schema checking tools are required to validate a given interaction schema.

4. Constraint Enforcement in LogicFence

LogicFence needs to *prevent* applications from forming a globally inconsistent system state. The main challenge here is that of prevention. It is necessary to predict what global state is going to be formed and prevent it from being formed if it is an illegal state.

In order to do this, we can first observe that global states change only when applications give an output to the external world or to some other part of the system outside their control. Global states also change when applications receive inputs from the external world and start processing them. In order to prevent the formation of globally inconsistent states, outputs may need to be blocked or inputs may

need to be held up from being processed. For LogicFence, it is hence mandatory that every I/O operation necessitates a change of state in the interaction schema.

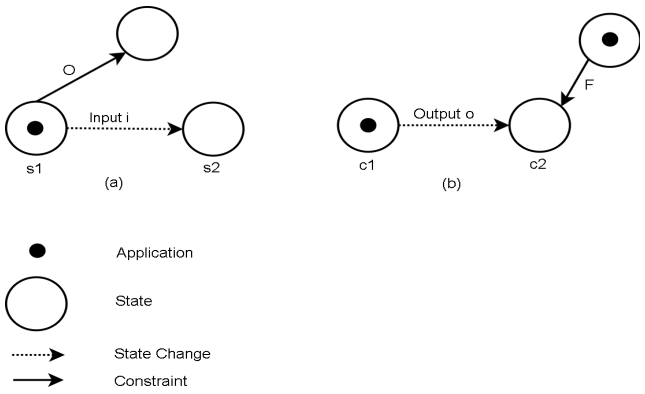

Figure 2. Preventing applications from forming globally inconsistent state

Figure 2(a) shows a case where an application tries to process an input *i*, by making a transition defined as $s_1 \xrightarrow{i} s_2$. However, since state s_1 is waiting for an obligated constraint to be satisfied, LogicFence acts as a safeguard by preventing the transition. Hence the application is halted and is not able to process the input *i*.

Similarly, an application with current state c_1 tries to provide an output *o* to some part of the system outside of itself. However, as mentioned before, outputs necessitate a state transition and can be given out only on successfully entering the target state. As shown in figure 2(b), the transition is defined as $c_1 \xrightarrow{o} c_2$, where *o* is the output that is performed once the application can enter state c_2. However, LogicFence will prevent the application from entering state c_2 since a forbidden constraint holds active on state c_2. An output is not performed until the transition is found to be valid and the application moves to the new state.

LogicFence will disallow even a transition of the form $c_1 \xrightarrow{o} c_1$, i.e. a transition to the same state, if a forbidden constraint on the specified state holds at the time of the transition. This is because, an output signifies a change in the global state and the target forms a globally inconsistent state due to the active forbidden constraint. Similarly, a transition of the form $s_1 \xrightarrow{i} s_1$ will also be prevented if an obligated constraint from state $s1$ holds, thus disallowing the exit from s_1.

States in the shared state space S can also be defined based on the value of some internal variables of an application. Such variables are termed *state variables*. Hence the assignment to such a state variable may cause a change in state. Thus calls to LogicFence functions are also inserted in the object code of the application before an assignment to a state variable to check whether the state change of the application is valid or not.

The constraints in LogicFence are global constraints across states and may involve multiple applications. 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, the snapshot is maintained by a central database. In order to prevent race conditions, all states on either sides of a constraint needs to be locked before allowing a state transition by an application.

4.1. Embedding LogicFence in the run-time environment

LogicFence accepts a definition of global constraints written in a constraint specification language based on XML. It then translates these constraints into primitives that are embedded into the run-time of application programs (currently, into the JVM of Java applications). This is done as follows:

i) The application class files are converted into an assembly format with the help of a disassembler called *'Jasper'* [7]. Jasper reads the java class files and converts them into corresponding assembly format.

ii) After every input, a call to a function *'checkTransitionOnInput'* of the execution monitor is inserted in the assembly file. The input forms an argument to the function. Similarly before every output a call to a function *'checkTransitionOnOutput'* of the execution monitor is also inserted in the assembly file. Here the output forms an argument to the function. Also a call to the function *'checkTransitionValidity'* is inserted before an assignment to a state variable. The function takes as arguments the name and the immediate future value of the state variable.

iii) The instrumented java assembly code is assembled back into class files with the help of an assembler called *'Jasmin'* [4].

Figure 3 schematically depicts the LogicFence instrumentation process.

5 Predicate support in LogicFence

Predicate support in LogicFence increases the expressiveness of configuration specifications. Configurations along with predicates are implemented in the form of auxiliary functions and hence arbitrary predicates can be defined and used to specify constraints. The set of auxiliary functions in the current version of LogicFence are as follows:

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

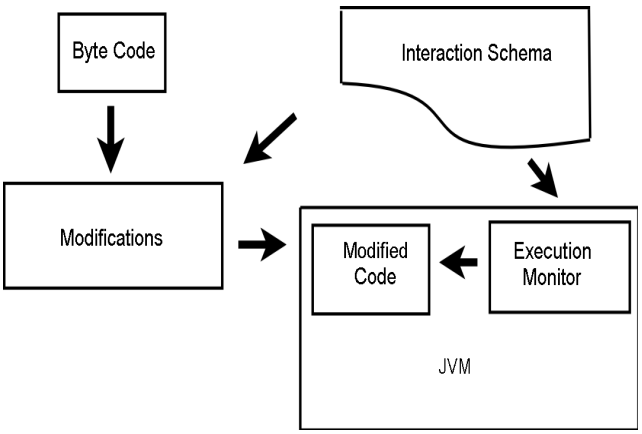

Figure 3. Embedding LogicFence in the runtime of application

the presence of applications other than the one making the transition.

2. *count(state_name, comparison_op, num)*: This function checks if the number of applications in the specified state including the application corresponding to the calling LogicFence instance satisfies the condition specified by the comparison operator '*comparison_op*' with respect to num.

3. *numberOf(state_name, property_conditions, comparison_op, num)*: This function is similar to 'count' but has an additional parameter '*property_conditions*' to ensure if the predicates specified by *property_conditions* are satisfied for the *state state_name*.

   Here is an example: `numberOf(b1, test/color>=4, <, 5)`. Here the function checks if the number of applications in state b1 with value of property color being greater than or equal to 4 is less than 5. 'test' is the name of the class which contains the property 'color'.

4. *checkProperty(state_name, property_conditions)*: This function checks if the specified property conditions hold for the application trying to make a transition to the state *state_name*.

Some example configurations along with their LogicFence representations are shown in table 2.

LogicFence supports *dynamic reorganization* of coordination constraints. Dynamic reorganization of coordination constraints are quite significant when the rules of interaction or the dependencies across applications in a system are likely to be changed frequently, and it would be costly to restart applications every time there is a change. The set of constraints or configurations may change while the applications are running. This does not require an application to resume from its initial state. LogicFence also allows *dynamic state definitions*. New states can be added and existing states can be deleted or modified when the applications are running.

Table 2. Example Configurations

Let us consider a parking area with two adjacent slots *b1* and *b2*. There is constraint that if a Sport Utility Vehicle (SUV) is parked on *b1*, then no SUV can fit into *b2* (although a smaller car can). Note that here SUV is a property variable of class Car. A similar inverse constraint can also be specified, which is not mentioned here.	`[numberOf(b1, car/SUV=true,>,0)] -> F[checkProperty(b2 ,Car/SUV=true)]`
A non-police car is allowed to cross a check post if there is one or more police car at the check post. State *s1* denotes that a car is located at the check post and *s2* is the state denoting crossing over the checkpost.	`numberOf(s1, Car/police=true, >,0) -> P[checkProperty(s2, Car/police=false)]`

6 Experimental Evaluation

We performed experiments considering a simple set of coordination constraints involving a traffic system to find out how many constraint violations can be caught.
We used three different types of applications for performing the experiments: i) Car application ii) Police car application, and iii) Tollgate application. The states used to specify the configurations are as follows:

1. t0: Tollgate is closed
2. t1: Tollgate is open
3. c1: A car is on the bridge
4. c6: A car has paid toll amount
5. c9: A car is at the check post

6. c10: A car is after the check post

Figure 4 gives a pictorial representation of the environment in which the system runs. The constraints specified in the system are:

1. A tollgate can leave the closed state (and go to open state) only if a car has paid the toll amount.
2. A car can leave state 'c6' (and cross the tollgate) only if the tollgate is open.
3. Two cars cannot be there on the bridge at the same time.
4. A non-police car application is not allowed to cross the check post if there is no police car at the check post.
5. Two police cars cannot be there at the check post at the same time.

This toy experiment model is indicative of larger classes of applications where coordination constraints are quite removed from the service logic. The car applications would be written for the purpose of driving. However, the cars enter several traffic scenarios, each with their own coordination requirements. It is extremely difficult for the car applications to have taken care of all the different coordination requirements that they encounter in their lifetime.

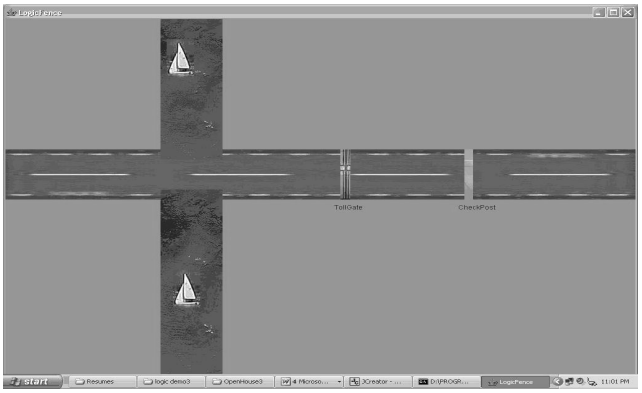

Figure 4. Traffic system in Experiment

The experiment was initially conducted with 5 non-police car applications, 4 police car applications and 1 tollgate application running simultaneously. The non-police car applications were created at different random intervals and moved at different random speeds. In the experiment, we noticed how many times the non-police car application instances violate the constraints when they are not running LogicFence. A non-police car application is supposed to follow constraints (ii), (iii) and (iv) mentioned above. In the experiment, each non-police car application instance was driven 1000 times across the path shown in figure 4. So the 5 instances of the application can violate the constraints for a maximum number of 1000*5*3 = 15000 times. For different set of experiments, we found the number of times the constraints were violated varied between 942 and 1078, i.e. for this system, the constraints are violated for 6% to 7% of the cases. Though the result is highly dependent on the type of the application and the system, it certainly shows the importance of LogicFence since a single constraint violation may turn out to be devastating for a system. LogicFence is yet to be tested in a substantially large real world system. However, the experiment conducted here serves as a baseline for any large constraint enforcement model.

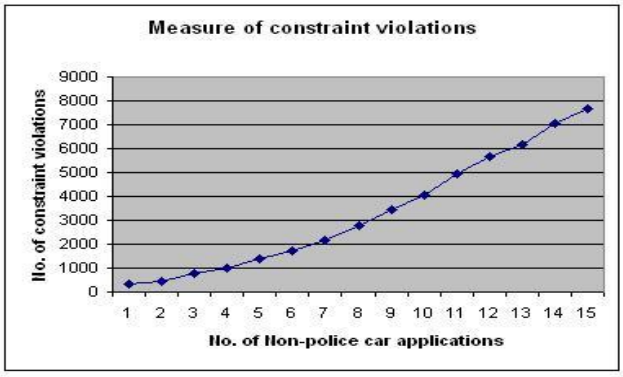

Figure 5. Measure of constraint violations caught by LogicFence

We also conducted experiments to measure the amount of computation "wasted" if coordination requirements are incorporated in the application logic. A large number of constraints violations signify that more amount of computation needs to be involved at the application logic to enforce the coordination requirements. Figure 5 plots the number of constraints violations versus the number of non-police cars running in the system. Note that a non-police car application can violate a maximum number of three linearly-placed constraints in the path shown in figure 4. Each non-police car was driven 1000 times across the path in a single direction. One tollgate application and one police car application were also executed simultaneously for all the cases.

The general trend in the plot shows a super-linear increase in the number of constraint violations with an increase in number of applications in the system. This is the case when the system had only three independent constraints arranged linearly. Incorporating coordination logic into the applications is hence not desirable, since its complexity grows faster than the number of application instances in the system.

7 Conclusions

LogicFence provides a seamless mechanism for enforcing system-wide integrity constraints across applications. LogicFence is targeted for large information systems involving multiple applications where coordination policies among the applications are very likely to change over time. LogicFence separates the coordination policies from the service logic or computation of the individual applications. The framework is itself independent of any application logic and enforces the global constraints from a layer below the application logic.

In the current implementation, the execution monitor maintains the state of an application in a database that serves as a communication medium for enforcing dependencies across applications. While the current architecture of LogicFence is suitable for clusters or LAN environments, it is not suitable for wide-area distributed systems. Also, the issue of fairness among different applications waiting for the same transition has not been taken care of in the current implementation. LogicFence can also be enhanced to support aggregate queries involving multiple applications.

8 Acknowledgments

The authors express their gratitude to the people working in Open Systems Laboratory at IIIT Bangalore for their help; especially Ambar Hegde, Satish Chandra D and Manjunath A. Sindagi. The authors also express their gratitude to Intel and the PlanetLab Consortium. This work was supported by a research grant from Intel and the PlanetLab Consortium that helped establish the first PlanetLab node in India.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, 1986.

[2] F. Arbab, I.Herman, and P. Spilling. An overview of manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.

[3] U. Erlingsson and F. B. Schneider. Sasi enforcement of security policies: A retrospective. *Proceedings of the New Security Paradigm Workshop*, 1999.

[4] J. Meyer and D. Reynaud. Jasmin. *http://jasmin.sourceforge.net/*, 2005.

[5] N. H. Minsky and J. Leichter. Law-governed linda as a coordination model. *LNCS 924*, pages 125–145, 1994.

[6] G. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46, 1998.

[7] C. Rathman. Jasper. *http://www.angelfire.com/tx4/cus/jasper/*.

[8] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1), 2000.

[9] R. Sekar, M. Bendre, D. Dhurjati, and P. Bolineni. A fast automaton-based method for detecting anomalous program behaviors. *Proceedings of IEEE Symposium on Security and Privacy*, pages 144–155, 2001.

[10] R. Sekar, C. R. Ramakrishnan, I. Ramakrishnan, and S. A. Smolka. Model carrying code (mcc): A new paradigm for mobile code security. *New Security Paradigms Workshop (NSPW'01)*, 2001.

[11] A. Singh. A test of computer security. *http://www.kernelthread.com/publications/security/ sandboxing.html*, 1994.

[12] S. Srinivasa. *An Algebra of Fix Points for Characterizing Interactive Behavior of Information Systems*. PhD thesis, Brandenburg Technical University at Cottbus, Germany, 2001.

[13] R. Tolksdorf. Coordinating services in open distributed systems with laura. *First International Conference on Coordination Models, Languages and Applications (Coordination'96)*, pages 386–402, 1996.