38th Annual Simulation Symposium, San Diego, CA, April 2005, pp. 91 - 98.

Simulation Verification and Validation by Dynamic Policy Enforcement

W.T. Tsai, X. Liu, Y. Chen, R. Paul*
Computer Science and Engineering Department, Arizona State University,
**Department of Defense, Washington DC, U.S.A*

Abstract

This paper presents a new verification and validation (V&V) technique for simulation using dynamic policy enforcement. Constraints are formally specified as policies, and they will be used to check whether simulation satisfies these policies at runtime. This paper also proposes a development framework where policies are developed along with system development and V&V. Once policies are extracted from requirements and specified in a policy specification language, the rest of the development work is automatically performed by the tools in the framework. Both security requirements and functional requirements can be specified as policies and dynamically enforced during the simulation. An automated tool is available for policy specification and enforcement, and it is fully integrated with the simulation infrastructure. This paper also presents a sample system that is modeled and simulated, and policies are used to verify and validate the system model. The paper also discusses the overhead imposed to perform this kind of automated policy-based V&V compared to the hard-coded implementation of the same approach.

Keywords: Simulation, Verification and Validation, Automated Policy Specification and Enforcement.

1. Introduction

Simulation Verification and Validation (V&V) need to check 1) the model of problems, 2) the program that simulates the problem, 3) the simulation engine that interprets the simulation program, and 4) the interactions between the simulation program and the simulation engine. Traditional simulation V&V techniques, such as code inspection, model checking and software testing, can check the simulation program and the simulation engine respectively, but cannot check the dynamic behaviors and the interactions between the program and the engine at runtime. Because the outputs of a complex system may depend not only on the inputs, but also on the internal states, testing has to initialize the environment to a given state and then run the program to see if the program behaves as intended, which is time consuming and difficult.

This paper presents a new technique that applies policy enforcement to detect the manifestations of faults at runtime caused by either the simulation program or the simulation engine. This V&V technique focuses on the behaviors instead of the causes of faults and the structure of the program, and thus is independent of the complexity of the system under test. Policies are traditionally used to enforce security requirements. This paper also suggests using policies to enforce functionality requirements and thus performing dynamic V&V.

A *policy* is a statement of the intent of policy makers or administrators of a computing system, specifying how the system should be used [1][2]. A policy-based system has the advantage of flexibility. Traditionally, policies are considered to be requirements and are hard-coded into the system implementation. For instance, if a policy states that "passwords must be at least 6-character long", there must exist a snippet of code in the system implementation that checks the length of passwords. Hard-coded policies can cause major problems such as:

- It is difficult and expensive to change. Whenever a policy needs to be changed (e.g. the system administrator wants to increase the minimum length of valid passwords from 6-character long to 8-character long), the whole system has to be shut down. The code needs to be inspected, modified, recompiled, and redeployed. The process is lengthy and error prone. It significantly increases an organization's operating expenses and risk. Shutting down a mission-critical system, in most cases, is prohibitive and may cause disastrous consequences to the mission.
- It is difficult to manage. Hard-coded policies do not separate the policy specification from the system implementation. Policies spread over everywhere in the system implementation. If a policy maker wants to know "How many policies are there in the system" or "What are the policies that are defined on the role of Supporting Arms Coordinator", there is no easy way to find out the answers.

In the past decade, a number of *Policy Specification Languages (PSLs)* have been proposed [2][4][5][6][7][8] to solve the problems caused by hard-coded policies. Many PSLs provide a set of simple and easy-to-use syntax to specify policies which is separate from the system implementation. A user interface can provide how policy

makers can specify and manage policies. A policy engine can load, interpret, and enforce policies, which allows policies to be dynamically added, removed, updated, and enabled or disabled.

Figure 1 shows the modeling and simulation framework that we developed for developing highly dependable systems. When developing highly dependable systems, verification and validation (V&V) need to be performed in each step of the development. Our framework supports both traditional V&V (left branch) and dynamic V&V (right branch). In traditional development methodology, the system requirements are first translated into the formal specification. The specification is verified statically by model checking [9] or Completeness and Consistency (C&C) analysis [10]. After several iterations of V&V, the final specification is obtained. Test cases are generated from the specification. An automated code generation tool generates the executable for simulation. This process has been reported in [11].

This paper focuses on the right branch of the modeling and simulation framework showed in Figure 1: performing dynamic V&V using policy enforcement. Independent of the development processes on the left branch, the policies are extracted from the requirements and then specified in a PSL. Similar to the specification, the policies are verified by C&C analysis to detect any incomplete and inconsistent policies. After the policies pass the verification, they will be stored in a policy database. Test cases that dynamically check C&C can be generated based on the policies. During the simulation, a policy engine dynamically loads the policies from the policy database, interprets them, and enforces them at runtime, which allows policies to be easily changed (added, removed, updated, and enabled / disenabled) on-the-fly at any time.

The advantage of using simulation to enforce policy is that simulation can run extensive cases to ensure extensive coverage. Thus, we can have both static coverage and dynamic coverage, e.g., how many times a specific set of scenarios have run, and how many times a specific scenarios will happen, and how many of these scenarios are performed correctly.

The remainder of the paper is organized as follows: Section 2 introduces the requirements of a sample system. Section 3 explains the system modeling process. Section 4 discusses policy specification and our Policy Specification and Enforcement Language (PSEL). Section 5 illustrates how policies are enforced during the simulation. Section 6 presents the experiment data collected in the simulation. Section 7 concludes the paper.

2. A Sample System: Battle of Tanks

To explain the concepts behind the modeling process, we present a sample system (Battle of Tanks) in this section. The system descriptions below serve as the system requirements that are the inputs to the modeling process.

System Descriptions: An observer (*an actor*) in the SOF (Special Operations Forces) Team locates an enemy SSM (Surface to Surface Missile) launcher (*an actor & data*) being ready for movement. The observer inputs the target information (*data*) and sends it to the Brigade TOC (Tactical Operations Center). On receiving the target information, the FSO (Fire Support Officer, *an actor*) issues a CFF (Call for Fire, *data*) and sends it to the SAC

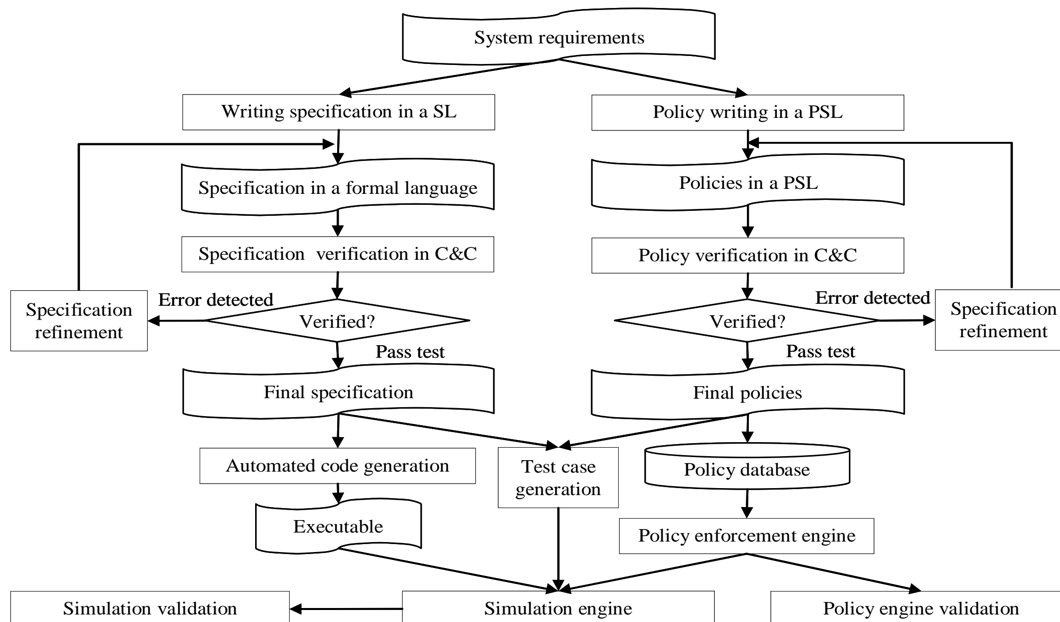

Figure 1. The modeling and simulation framework

(Supporting Arms Coordinator, *an actor*) in the Commanding Tank. On receiving the CFF, the SAC (*an actor*) issues a Fire Order (*data*) and sends it to the shooter (*an actor*) in the Main Battle Tank A (*an actor*). Issuing a Fire Order can be broken down into a sequence of sub-actions: determine the best weapon (*data*) to attack, check if the SOF team is out of the target area, check if the SOF Team has lain down, and input the Fire Order. If Main Battle Tank A can not shoot for some reasons, the Fire Order Rejection (*data*) is sent back to the Commanding Tank, where the SAC resends the Fire Order to the shooter (*an actor*) in the Main Battle Tank B (*an actor*). On receiving the Fire Order, the Main Battle Tank B fires towards the SSM launcher, if there are two bullets or more. After firing, the shooter transmits the MFR (Mission Fire Report, *data*) to the Commanding Tank, where the SAC then sends the Fire Advisory (*data*) to the FSO at Brigade TOC. Meanwhile, the observer in the SOF team observes and inputs the target status information (*data*) and sends it to the Brigade TOC. After analyzing the reports, the FSO at Brigade TOC sends the TDR (Target Destroyed Report, *data*) to the Commanding Tank where the SAC reads the TDR.

3. System Modeling Process

The IASM (Integrated ACDATE & Scenario Model) is a formal model for system modeling. The system structure is modeled by *ACDATE* (Actor, Condition, Data, Action, Timing and Event) elements. System behaviors and system constraints are modeled through scenarios and policies, respectively. Scenarios and policies are defined on the top of ACDATE elements. In other words, ACDATE elements are the building blocks of scenarios and policies. The three steps of IASM modeling process are illustrated in Figure 2.

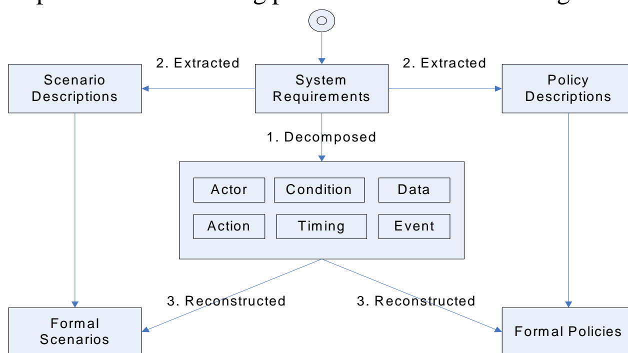

Figure 2. IASM Modeling Process

In Step 1, system requirements are carefully analyzed and decomposed into ACDATE elements. In Step 2, both system scenarios and policies are extracted from system requirements. In Step 3, system scenarios and policies are formally reconstructed by ACDATE elements.

3.1 Step 1: Decompose System Requirements to ACDATE Elements

After analyzing the system requirements, ACDATE elements can be identified. Table 1 lists several ACDATE elements extracted from the system requirements of Battle of Tank. To simplify the system model, timing is not defined for this system.

3.2 Step 2: Extract Scenarios & Policies

From the user's point of view, a system scenario consists of a sequence of atomic scenarios. In each atomic scenario, actions are performed, conditions are evaluated, data are accessed, and events are emitted to trigger other atomic scenarios (as shown in Figure 3). Through event triggering, atomic scenarios are brought together to represent a complete system scenario. Table 2 and Table 3 list several scenarios and policies.

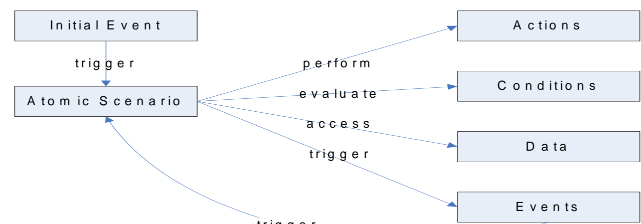

Figure 3. Atomic Scenarios

3.3 Step 3: Reconstruct Scenarios & Policies

System scenarios and policies are usually informally stated in the system requirements as we have seen in Step 2. The purpose of system modeling is to create a formal model to represent the system structure (ACDATE elements), behaviors (system scenarios), and various constraints (policies). After ACDATE elements are identified and system scenarios and policies are extracted, system scenarios and policies are formally reconstructed based on ACDATE elements as well as constructs that control the flow of scenario execution. Figure 4 shows an example of reconstructed scenarios.

```
Tanks.e2e   Scenario03
using ACTOR:SOF_Observer
using ACTOR:Supporting_Arms_Coordinator

do ACTION:Supporting_Arms_Coordinator.A_DetermineBestWeapon
DATA:Supporting_Arms_Coordinator.BestWeapon = "Gun"

DATA:SOF_Observer.SOFTeamDistanceFromSSM = "6000"
DATA:SOF_Observer.SOFTeamStatus = "LieDown"

do ACTION:Supporting_Arms_Coordinator.A_CheckSOFTeamOutOfTargetArea
do ACTION:Supporting_Arms_Coordinator.A_CheckSOFTeamLiedown

do ACTION:Supporting_Arms_Coordinator.A_InputFireOrder
DATA:Supporting_Arms_Coordinator.FireOrder = "Issued"

emit EVENT:Supporting_Arms_Coordinator.E09_SendFireOrderToMBTA
```

Figure 4. Reconstructed Scenarios

Table 1. ACDATE Elements	
Actors	SOF_Observer, SSM_Launcher, Fire_Support_Officer, Supporting_Arms_Coordinator, …
Conditions	SSMFound, CFFIssued, SOFTeamNotLainDown, SOFTeamWithinFieldOfFire, FireOrderIssued, …
Data	TargetInfo, CallForFire, FireOrder, SOFTeamDistanceFromSSM, SOFTeamStatus, …
Actions	InputTargetInfo, IssueCFF, DetermineBestWeapon, CheckSOFTeamLainDown, InputFireOrder, …
Timing	N/A
Events	LocateSSMLauncher, ReceiveTargetInfo, ReceiveCFF, …

Table 2. Atomic Scenarios	
Scenario01	On locating SSM launcher, observer inputs and sends target information
Scenario02	On receiving target information, Fire Support Officer issues CFF command
Scenario03	On receiving CFF command, Supporting Arms Coordinator determines best weapon, checks if SOF is out of target area, checks if SOF is lain down, and inputs and sends fire order
Scenario04	On receiving fire order, Main Battle Tank A fires to SSM launcher if it can shoot; otherwise, it rejects the fire order

Table 3. Policies	
Policy01	On receiving Call For Fire, Supporting Arms Coordinator must issue a fire order
Policy02	If a main battle tank can shoot, it must not reject the fire order
Policy03	The distance between the SOF Team and SSM launcher must always be >=3000 feet
Policy04	Action DetermineBestWeapon must occur before action InputFireOrder
Policy05	the CFF (Call for Fire) command can be issued only once
Policy06	On receiving the Fire Order, Main Battle Tanks must fire to SSM if the number of remaining bullets >= 2

4. Policy Specification

We designed the Policy Specification & Enforcement Language (PSEL) to specify system constraints (policies) in the IASM model. PSEL covers obligation policies, authorization policies and system constraints. A policy editor and a graphical policy management interface have been developed for policy input. Obligation policies and authorization policies are specified on roles rather than on individual actors. A role represents a management position with its associated responsibilities and rights. Actors are assigned to roles according to their management positions. Since actors take particular roles in an organization, policies specified on a role will in turn apply to actors who take this role.

4.1 Obligation Policies

Obligation policies define a role's responsibilities, specifying what actions a role must or must not perform under a condition. *Positive obligation policies* are ECA (Event-Condition-Action) rules with the semantics that "on receiving a triggering event E, a role R must perform the action A if condition C is true". For instance, Policy01 should be defined as a positive obligation policy, because it specifies the responsibility of actors who take the Supporting Arms Coordinator role.

Negative obligation policies forbid a role to perform action A if condition C is true. For instance, the Policy02 is a negative obligation policy. If violated, the policy engine will inform the simulator of the detection of the policy violation. The simulator will perform the compensation action that is intended to minimize the consequences caused by the policy violation.

4.2 Authorization Policies

Authorization policies define a role's rights of performing actions, specifying what actions a role is allowed or denied to perform under a certain circumstance. In PSEL, authorization policies are specified and enforced through access control models. Currently, two access control models are supported in PSEL: Bell LaPadula (BLP) model and Role-Based access control model.

Bell LaPadula (BLP) model [12] is a mandatory access control model widely used in military and government systems. It controls information flow and prevents information from releasing to unauthorized persons. To specify BLP in PSEL, actors and date are associated with a security level attribute. Actions are also associated read / write attribute. When an action is performed, the policy engine can check these attributes to allow or deny the action through the two BLP access rules (no read-up and no write-down).

Role-Based Access Control (RBAC) model [13][14][15] has been increasingly applied in various systems, due to its

Table 4. Obligation Policy & System Constraints

Positive Obligation Policy	Negative Obligation Policy	System Constraints on Data	System Constraints on Actions

MUSTDO {
definedOn ROLE
triggeredBy EVENT
do ACTION
on CONDITION
}
	MUSTNOTDO {
	definedOn ROLE
	do ACTION
	on CONDITION
	perform COMPENSATION
	}
		MUSTBE / MUSTNOTBE {
		appliedTo DATA
		status EXPRESSION
		on CONDITION
		perform COMPENSATION
		}
			MUSTBE / MUSTNOTBE {
			do ACTION:Operator
			(action parameters)
			on CONDITION:TRUE
			perform COMPENSATION
			}

nature of policy neutral. To specify RBAC in PSEL, a group of roles are defined and actors are then assigned to these roles based on their management positions. A set of actions are assigned to a role, giving it the permissions to perform these actions. PSEL also supports role hierarchy. *A role hierarchy* represents the superior & subordinate relationship among roles, allowing a superior role to automatically obtain all permissions of its subordinate roles. *Role delegation* is also supported, which enables a role temporarily transfer its permissions to other roles and can be revoked at a later time.

4.3 System Constraints

System constraints define constraints on system status and behaviors that must hold in the system execution or simulation. *System constraints on data* specify that data must or must not be within a certain range. For example, Policy03 is a system constraint on data. *System constraints on actions* are temporal logic on actions. Examples could be Policy04 and Policy05. Currently, the following temporal logic operators are supported by PSEL:

- Concurrency (A, B): A, B occurs concurrently
- Sequence (A, B, C): A, B, C occurs in this sequence
- Order (A, B, C): A, B, C occurs consecutively
- Either (A, B): Either A or B occurs, but not both
- Exist (A): A must occur
- Once (A): A must occur once, and only once

Conditions are associated with system constraints, specifying when these policies are to be enforced. In addition, compensation actions are defined in a system constraint. When policy violation is detected, associated compensation actions will be performed by the simulator to minimize the consequences brought by policy violation.

5. Policy Enforcement

Policies are enforced during the simulation. In the initialization phase of simulation, the policy engine loads policies from the policy database and register them with the simulator according to their semantics. Policies are registered so that simulator knows when to trigger the policy enforcement. Simulator triggers the policy enforcement when a registered event occurs, a registered action is performed, or a registered datum is modified. Policy engine will enforce relevant policies registered to this event, action, or datum, and return the results of policy enforcement back to the simulator.

Policy enforcement can be classified into three categories: policy checking, policy execution, and policy compensation. *Policy checking* verifies if policy violations are detected when actions are performed or data are changed. *Policy execution* executes the action defined in the policy when receiving the triggering events. *Policy compensation* executes the compensation action defined in the policy when policy checking detects a policy violation. All checkable policies come with a compensation action.

Only positive obligation policies are executable policies. The rest types of policies (e.g. negative obligation policies, all authorization policies and all system constraints) are all checkable policies. *Executable policies* influence the paths of the simulation through the actions defined in them. When the triggering event occurs, executable policies registered to this event will be enforced, and the action specified in policy specification will be executed by the simulator. *Checkable policies* can also influence the paths of the simulation through compensation actions. When data are changed or actions are performed, checkable policies registered to the data or actions will be enforced. If there are policy violations detected, the compensation action specify in policy specification will be executed by the simulator.

5.1 Policy Enforcement Framework

Figure 5 illustrates the policy enforcement framework. After policies are extracted and specified in PSEL, the policy editor parses policies for correctness and stores them into the policy database. During the initialization phase of simulation, the policy engine loads policies, interprets them, and registers them to the simulator. While the simulator executes system scenarios, it triggers the policy enforcement when the registered events occur, registered actions are performed, or registered data get changed. The policy engine checks or executes policies and returns the results back to the simulator. Based on the returned results, the simulator determines the next system scenarios.

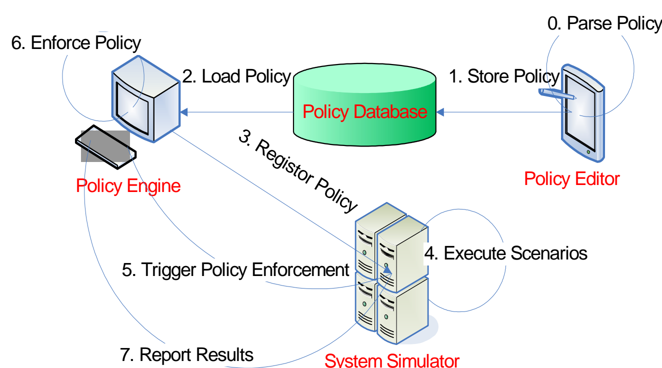

Figure 5. The Policy Enforcement Framework

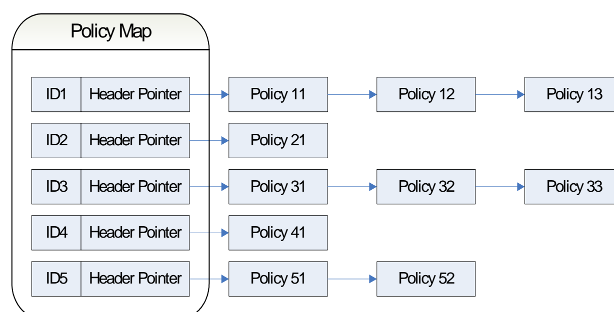

Figure 6. Policy Map

5.2 Policy Parsing and Storage

The policy editor parses policies for correctness. On successful parsing, policy editor translates policies into XML and stores them into the policy database. The policy parser is implemented by ANTLR (ANother Tool for Language Recognition). According to the policy syntax, ANTLR creates an abstract syntax tree (AST) for each policy. Policy elements (roles, actions, condition, etc) are extracted by traversing the tree, and translated to the XML representation. The XML representation of a policy is then stored into the policy database as a string.

5.3 Policy Registration & Policy Map

Policy registration lets the simulator know when policy enforcement should be triggered. In IASM, three out of the six ACDATE elements can trigger the policy enforcement: *event, action, and data*. Events occurrences will trigger the enforcement of positive obligation policies; action performances will trigger the enforcement of negative obligation polices, authorization policies, and system constraints on actions; data changes will trigger the enforcement of system constraints on data.

To improve performance, a policy map is created, mapping a particular event, action or datum to a list of relevant policies to which it has been registered with. All policies registered to a particular action, event or datum forms a linked list. The linked list and the ID of action, event or datum are then organized as a policy map. When policy enforcement is triggered, the policy engine locates the policy linked list of a particular action, event or datum, and enforces all policies in the list.

5.4 Policy Enforcement Triggering

After policies have been registered, the simulator initializes the data and starts running the system scenarios. The simulator keeps an eye on the system scenarios, and triggers the policy enforcement by invoking the EnforcePolicy() method in the policy engine whenever an event is triggered, an action is performed or a datum is changed. The policy engine enforces all relevant policies, records all violations in the policy log, and returns the policy log back to the simulator.

5.5 Policy Enforcement

When policy enforcement is triggered, the simulator invokes the EnforcePolicy() method in the policy engine, passing the ID of the event, action or datum that triggers the policy enforcement. The policy engine looks up its policy map, maps the ID to a list of policies to which it has registered, and enforces them one by one. Authorization policies are not registered in the policy map, and they are enforced before obligation policies and system constraints are enforced. When policies are enforced, all violations are recorded into a policy log that is returned to the simulator. The EnforcePolicy() method returns a Boolean value indicating whether policy violations are detected.

6 Experiment Data and Analyses

6.1 Policy Enforcement Analyses

Policy enforcement is used as a dynamic mechanism to verify and validate the system model, the simulation program, the simulation engine, and their interactions. If

Table 5. Policy Registration

Policy Type	Authorization	Positive Obligation	Negative Obligation	System Constraints (Data)	System Constraints (Action)
Event		X			
Action	X		X		X
Data				X	

the system does not behave as the policies state, the manifestation can be caused by the system model, the simulation program, the simulation engine, or the interactions among them. For example, Policy06 states that "on receiving the Fire Order, Main Battle Tanks must fire to SSM if the number of remaining bullets >= 2". In the system model, this scenario is incorrectly implemented as:

```
if (DATA:bullet>2) {do ACTION:MainBattleTank.FireToSSM }
```

policy enforcement will detect the violation given the number of the remaining bullets equals to 2, which indicates a fault in the system model. However, policy enforcement is based on the simulation and simulation may not execute a particular system path. A fault in the system model may not manifest as an error during a particular simulation. For example, the problem of system modeling can not be detected if the number of the remaining bullets equals 3 or more. In other words, performing simulation V&V using dynamic policy enforcement can not guarantee all faults to be detected or to be detected immediately. We deliberately injected the fault mentioned above into the system model that could violate Policy06, and run the simulation 20 times with policy enforcement to test if or how many times the fault is detected. In the 20 tests, only one violation occurred and detected by Policy06.

6.2 Overhead Analyses

Police-based computing enables dynamic V&V and brings flexibility of governing constraints on system status and system behaviors. By specifying and enforcing policies on the fly rather than hard-coding policies into system implementation, policy changes can take effect immediately. However, flexibility comes with a cost. Dynamic policy specification and enforcement inevitably increase the system overhead in terms of space complexity and time complexity. We conducted a series of experiments on the Battle of Tanks system, where 25 policies are extracted from the system requirements and enforced in the experiments, to explore the cost of dynamic policy specification and enforcement. The experiments are conducted in three setups:

- Policy enforcement is disabled.
- Dynamic policy enforcement is enabled.
- Policies are hard-coded into system scenarios.

Space Complexity Analysis: During the simulation, the simulator and system scenarios are loaded into the memory. When the policy enforcement is disabled, the memory size (5323k) is the combined size of the simulator and system scenarios. When policy enforcement is enabled, the policy engine is also loaded to the memory. The difference between the memory sizes (595k = 5918k – 5323k) is the size for the policy engine. When policies are hard-coded into system scenarios, the increase of memory size (276k = 5599k – 5323k) is brought by the increase of code of policies that is hard-coded into system scenarios. From the collected data, we can see the space complexity increases by 11% (595k / 5323k) when dynamic policy enforcement is employed. In contrast, the hard-coded policy enforcement only increases the space complexity by 5% (276k / 5323k).

Time Complexity Analysis: System scenarios are translated into LUA code when the simulation is being initialized. The simulator then interprets LUA code and feed it to the LUA Virtual Machine for execution. When policy enforcement is disabled, the simulation time (2722ms) is totally dedicated to LUA code execution. When policy enforcement is enabled, the simulator triggers the policy engine to enforce policies. Since policy engine is written in C++, which is about 20 times faster than LUA code, we converted the time spent on policy enforcement (C++ time) to LUA time, in order to make a fair comparison. After converted, the difference between simulation time (2280ms = 5002ms – 2722ms) is the time spent on policy enforcement. When policies are hard-coded into system scenarios, more LUA code is executed and the increase of simulation time (240ms = 2962ms – 2722ms) is brought by the LUA code of hard-coded policies. From the collected data, we can see the time complexity is increased by 84% (2280ms / 2722ms) when dynamic policy enforcement is employed. In contrast, the hard-coded policy enforcement only increases the time complexity by 9% (240ms / 2722ms).

Table 6. Sumamry of the experiment results

Battle of Tanks	Criteria	Policy disabled	Policy enabled	Hard-coded Policies
Space Complexity	Memory Size (KB)	5323	5918	5599
	Compared to No Policies	100%	111%	105%
	Memory Composition	SIM + SC	SIM + SC + PE	SIM + SC + HCP
Time Complexity	Simulation Time (ms)	2722	5002	2962
	Compared to No Policies	100%	184%	109%
	Time Composition	SC	SC + PE	SC + HCP
Comments	SIM = Simulator SC = System Scenarios PE = Policy Engine HCP = Hard-Coded Policies			

Dynamic policy enforcement trades the efficiency for flexibility and dependability. In following applications, policy-based dynamic V&V are appropriate:

- Flexibility of policy enforcement is a necessity;
- Highly dependability requirement makes dynamic V&V necessary;
- The execution speed of policy enforcement can meet the timing requirements;

The policy enforcement can be disabled once the system is proven to be correct.

7 Summary

In this paper, we presented a framework that verifies and validates system models and simulation by dynamic policy enforcement. In our framework, system structure is modeled by ACDATE elements. System behaviors and constraints are modeled by scenarios and policies, respectively. PSEL is designed to specify obligation policies, authorization policies, as well as a variety of system constraints. A graphic user interface and a policy editor have been developed for policy input. During the simulation, a policy engine dynamically loads, interprets and enforces policies. Policy violations are recorded into a log and indicate potential problems in the system model. Experiments were conducted and showed dynamic policy enforcement incurs system overhead in terms of time and space complexity.

References

[1] C. Pleeger, *Security in Computing*, 3rd Edition, Prentice Hall PTR, 2003

[2] J. Burns and D.M. Martin, "Automatic Management of Network Security Policy", Proceedings of DARPA Information Survivability Conference and Exposition, June 2001

[3] N. Damianou, A. Bandara, M. Sloman and E. Lupu, "A Survey of Policy Specification Approaches", Technical Report, Department of Computing at Imperial College of Science Technology and Medicine, 2002

[4] M. Kangasluoma, "Policy Specification Languages", Technical Report, Department of Computer Science at Helsinki University of Technology, November 1999

[5] N. Damianou, N. Dulay, E. Lupu, M. Sloman, "The Ponder Policy Specification Language", Proceedings of Workshop on Policies for Distributed Systems and Networks, 2001

[6] L. Kagal, "Rei: A Policy Language for the Me-Centric Project", Technical Report, HP Laboratories, 2002.

[7] J. Hoagland, R. Pandey and K. Levitt, "Security Policy Specification Using a Graphical Approach", Technical Report, Department of Computer Science at University of California, July1998.

[8] M. Locasto, "SPCL: Structured Policy Command Language", Technical Report, Department of Computer Science at Columbia University, May 2003.

[9] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 2002.

[10] W. T. Tsai, X. Wei, L. Yu, R. Paul, and H. Huang, "Condition-Event Combination Covering Analysis for High-Assurance System Requirements", Technical Report, Department of Computer Science and Engineering at Arizona State University, September 2004

[11] W. T. Tsai, X. Wei, Y. Chen, B. Xiao, R. Paul, and H. Huang, "Developing and Assuring Trustworthy Web Services", to appear The 7th International Symposium on Autonomous Decentralized Systems (ISADS), April 2005.

[12] D. Bell and L LaPadula, "Secure Computer System: Unified Exposition and Multics Interpretation", Technical Report, MITRE Corporation, March 1976.

[13] R. Sandhu, E. Coyne, H. Feinstein and C. Youman, "Role-Based Access Control Models", IEEE Computer, Volume 29, Issue 2, February 1996.

[14] E. Bertino, "RBAC Models – Concepts and Trends", Computers & Security, Volume 22, Issue 6, August 2003, pp. 511-514.

[15] S. Osborn, R. Sandhu and Q. Nunawer, "Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies", ACM Transactions on Information and System Security, Volume 3, Issue 2, May 2000, pp. 85-106.