Specification-Enhanced Policies for Automated Management of Changes in IT Systems

Chetan Shankar – University of Illinois at Urbana-Champaign
Vanish Talwar, Subu Iyer, Yuan Chen, and Dejan Milojicić – Hewlett-Packard Laboratories
Roy Campbell – University of Illinois at Urbana-Champaign

ABSTRACT

Enterprise and grid computing systems are complex and subject to a broad range of changes such as configuration updates, failures, and performance degradations. These changes affect infrastructure elements such as computation and storage nodes, applications, and system management elements such as monitoring infrastructures. Today's best practices in use by system administrators to manage these changes are manual and ad-hoc. In large complex installations, this would lead to high operational costs, broken closed loop automation, and reduced agility. Providing tools and mechanisms to administrators that automate the reaction to these changes is highly desirable and is an active research area.

Policy-based management using Event-Condition-Action (ECA) rules is a well-known approach for such automated change management where management actions are executed when specified event-conditions are observed. In complex systems, the interdependence of components generates multiple events when a single change happens causing multiple rules to be triggered. The order of execution of rule actions determines the system behavior necessitating reasoning about execution order. ECA rules do not contain explicit action specifications needed for reasoning and are therefore unsuited for specifying management rules.

In this paper, we propose a specification-enhanced ECA model called Event-Condition-Precondition-Action-Postcondition (ECPAP) for designing adaptation rules. ECPAP rules contain action specifications in first order predicate logic enabling us to develop reasoning algorithms to determine enforcement order of multiple rules. The enforcement order is represented as a Boolean Interpreted Petri Net workflow. We introduce a new notion called enforcement semantics that provides guarantees about rule ordering. We have built an adaptation framework using ECPAP model and have demonstrated it for automated change management of Ganglia and HP OpenView monitoring systems. The evaluation of the framework illustrates the significance of the ECPAP model and demonstrates its applicability for managing complex IT environments.

Introduction

Modern IT systems, such as enterprise data centers and grids, are paradigms of distributed computing where computation and data are distributed across diverse computational and storage elements. These systems are characterized by growing complexity, scale, and heterogeneity of infrastructure and applications. Further, these systems are highly dynamic, and subject to frequent changes such as service plug-in/plug-out, workload variations, failures, configuration updates, and application migration. Such changes affect the runtime operation of the system, and the service contracts offered to customers. Therefore, in reaction to these changes, infrastructure elements, applications, as well as system management components need to be adapted. For example, compute and storage resources may have to be re-allocated, applications may need to be restarted, and monitoring infrastructures may require re-configuration.

Current approaches used by administrators to manage these changes are manual and/or a combination of ad-hoc tools and scripts. The process typically requires special expertise and detailed actions by administrators. While these approaches may work fine in small scale installations, they do not scale in larger scale installations typical of modern IT systems and utility systems of tomorrow. In such environments, there is a need to provide administrators with tools that can capture the expert domain knowledge in machine readable format and thereafter react to changes in an automated manner.

The use of Event-Condition-Action (ECA) rules [1] is a well-known approach for enabling system administrators to specify the desired actions to be invoked on changes in policy rules [5, 6, 7, 10, 20]. When a change event is received, the rules matching the event are determined. If the conditions in these rules are true, the corresponding actions are executed.

An example of an ECA rule is "When checkpoint store is full (*event*), if backup store is running (*condition*), assign backup store as new checkpoint store (*action*)." When the checkpoint store becomes full, an event is sent that triggers the rule. The management system verifies if the backup store is running and if so assigns it as the new checkpoint store.

Policy-based management systems have been effectively used to manage network switches [5], content distribution networks [6], and general distributed systems [7]. The applicability of policy-based systems for reacting to changes in today's IT systems, such as data centers and utility infrastructures, presents numerous challenges due to highly interdependent components. Changes in these environments propagate rapidly causing several related changes.

Consider, for example, an infrastructure for monitoring performance of a data center as depicted in Figure 1. The infrastructure consists of a set of monitored nodes on which monitoring agents continuously collect performance data and send them to nodes called *aggregators*. These aggregators perform statistical functions on the collected data and send them to interested clients, such as archival stores and performance visualizers. An aggregator's output typically represents the performance data of a cluster and is useful for viewing consolidated information.

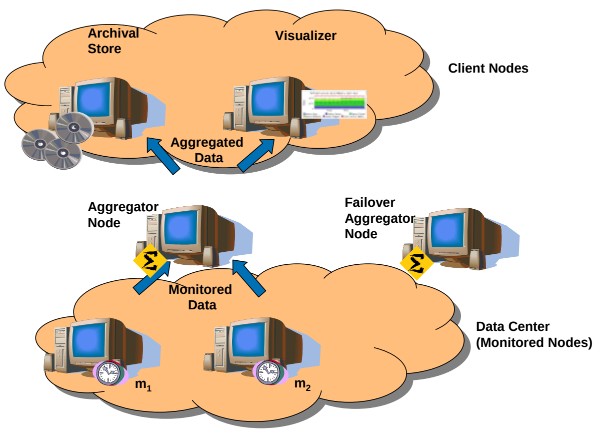


Figure 1: Infrastructure for monitoring performance of a data center.

If the aggregator node fails, monitored nodes cannot send data and may report failures. Similarly, clients using the performance data may report failures as they do not receive any data from the aggregator. Other clients such as load balancers and job schedulers using this data may unnecessarily rebalance the load or schedule jobs incorrectly thus affecting the overall utilization of the system.

Change detectors monitoring different aspects of the system sense several changes due to aggregator failure and generate various events. This causes several policy rules to get triggered in the adaptation system.

The order of enforcement of rules determines the final system state. As illustrated in Figure 2, the failure of aggregator node generates multiple events. These events trigger two rules – R_1 and R_2. R_1 states: "If aggregator node fails, assign failover node as aggregator" and rule R_2 states: "If data-send from monitoring agent fails, reconnect to aggregator." When the aggregator node fails, both rules are triggered. If R_1 is enforced before R_2, the failover node is assigned as aggregator and the monitoring agent is reconnected to it. But if R_2 is enforced before R_1, the action of R_2 fails since the monitoring agent tries to reconnect to the stopped aggregator.

Therefore, the order of enforcement of rules determines the final system state when multiple rules are simultaneously triggered. An adaptation system should reason about the enforcement order of rules and provide guarantees for system behavior to be deterministic.

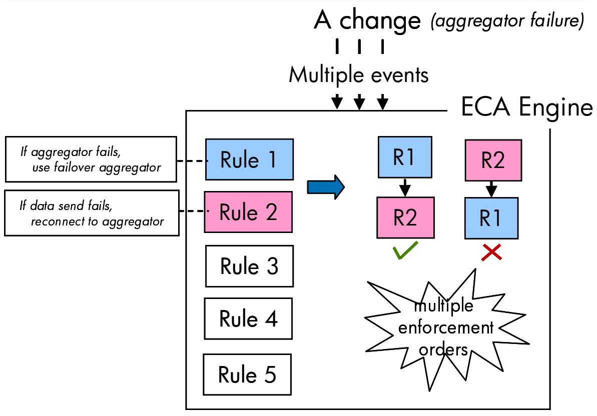

A change (aggregator failure)
Multiple events
ECA Engine
If aggregator fails, use failover aggregator
If data send fails, reconnect to aggregator
Rule 1
Rule 2
Rule 3
Rule 4
Rule 5
R1
R2
R2
R1
✓
✗
multiple enforcement orders

Figure 2: Problem with ECA based system.

ECA rules do not contain explicit action specifications needed for reasoning and are therefore unsuited for specifying management rules in such environments. Therefore, we propose an extended model of ECA called Event-Condition-Precondition-Action-Postcondition (ECPAP) for designing management rules. These rules contain the axiomatic specification of rule actions in first-order predicate logic as pre- and post-conditions. The pre-condition specifies the partial system state before execution of rule action while post-condition specifies the partial system state once the action has successfully executed.

Note that the rule condition is different from precondition because the rule condition is specified by the policy designer while the pre-condition is specified by the action developer (programmer). We have used the ECPAP rule framework for conflict detection and resolution and monitoring of rule enforcement in [11] and analyzing policy cycles in [10]. In this paper, we show how the pre- and post-conditions can be used to determine the dependencies among triggered rule actions and reason about their enforcement order.

Given a set of triggered rules, the adaptation system dynamically generates a Petri net workflow representing dependencies among rule actions. An action depends on another action if the pre-condition of the former is satisfied by the post-condition of the latter. Since specifications are in first-order predicate logic, which are undecidable in the general case, dependencies can only be determined at runtime when the specifications are instantiated to form propositional expressions. Therefore, workflows of rule actions cannot be constructed statically and must be done at runtime.

Certain actions may be independent of other triggered actions and therefore may not be present in the workflow. In such cases, the system could take one of several possible decisions, for, e.g., it may abort the workflow, or it may execute the maximum possible actions and so on. We identify the need for policy-based systems to provide semantic guarantees for rule enforcement in such circumstances.

We introduce a notion called *rule enforcement semantics* for policy-based systems that provides certain guarantees when multiple rules are concurrently triggered. We have identified three enforcement semantics – random, all-or-none and maximum rule – that have been found useful in different circumstances. We discuss this in detail in section "Ordering Rule Enforcement."

We make the following contributions in this paper:

- We propose a specification-enhanced rule framework called Event-Condition-Precondition-Action-Postcondition (ECPAP) for specifying adaptation rules.
- We present algorithms to determine enforcement order of multiple rules using the ECPAP model. The enforcement order is represented as a Boolean Interpreted Petri Net (BIPN) workflow. We also introduce a new notion called enforcement semantics that provides guarantees about rule enforcement.
- We describe an adaptation framework built using the ECPAP model and demonstrate its application for automated change management of Ganglia and HP OpenView monitoring systems.
- We present evaluation results that illustrate the need for enforcement guarantees and the feasibility of the ECPAP model.

Our extension of the ECA framework with action specifications follows naturally from current research efforts in autonomic computing. There has been widespread interest lately on using planning techniques from AI for programming and managing distributed systems with encouraging results [10, 11, 12, 13, 24]. In [10, 11] we showed how extending actions with specifications enabled advanced conflict and termination analysis for policy-based management systems. Andrzejak, et al. [12] have used actions with pre- and post-conditions for planning complex workflows from simple actions for system management.

The ABLE project [13] uses axiomatic specifications of actions for goal-based autonomic computing. Anand, et al. [24] use specification-enhanced actions, expressed as pre-conditions and effects, for programming pervasive computing environments. These research works have shown that annotating actions with simple pre- and post-condition specifications provides numerous benefits such as raising the programming abstraction level and automating system management. Based on the success of these efforts we have extended management policies with action specifications and introduced the ECPAP rule framework.

The rest of the paper is organized as follows. In the next section, we present in detail the ECPAP rule framework. We then describe the workflow generation algorithms and enforcement semantics. Subsequently, we present the ECPAP-based adaptation framework and its application for managing monitoring systems. We then discuss evaluation results and some lessons learned. Finally, we present related work and a conclusion.

Specification-Enhanced Rule Framework

The ECA rule framework is used in different paradigms such as active databases, access control and system management to react to different situations. Active databases use the ECA framework for designing triggers that specify actions to be executed when certain database operations such as record insertion or deletion are made. Access control systems use ECA rules to authorize or deny access when an access request is made. Management systems use the ECA framework for designing *obligation rules* [7] to specify management actions to be executed when system changes are observed. Rule actions in active databases and access control are normally well-defined and hence their effects on the system are implicitly known.

For example, active database trigger rules normally use insert, delete and update actions [21] while access control actions are normally authorize, deny and delegate. This enables complex reasoning such as confluence [21], rights-amplification and conflict analyses to be performed over these rules. But rule actions in system management are not well-defined and can range from simple atomic actions to complex scripts and so their effects on the system are not implicitly known. Therefore, explicitly specifying the action effects using pre- and post-conditions enables complex reasoning to be performed over management rules. This motivated us to design the ECPAP rule framework.

The ECPAP framework extends the ECA framework by using the Hoare triple [23]. A Hoare triple represented as *{P}C{Q}* describes how an action *C* changes the state of computation from a state where *P* is true to a state where *Q* is true. *P* and *Q*, expressed as first-order predicate logic expressions, are pre-and post-conditions of *C*, respectively and are called axiomatic specifications. The pre-condition specifies the system state that should exist before *C* can be executed.

Our adaptation policies are formulated as sets of ECPAP rules of the form

on *event* **if** *condition* **do** *action*

A policy rule is read as: "When event occurs in a situation where condition is true, then execute action." The action is a call to a method in a library of actions where each action is annotated with a pre-condition and a post-condition by the action developer (programmer). Note that pre- and post-conditions are not specified as part of the rules since an action may be invoked by multiple rules in the policy and this format avoids listing the specifications at multiple places.

We represent an ECPAP rule as $(e, c, p) \rightarrow (a, s)$ where e denotes the rule event, c denotes the condition of the rule, p is the pre-condition of the action, a is the action to be executed and s is the action post-condition. Our policy rule framework extends that of Policy Description Language (PDL) [1] by adding axiomatic specifications as "extension"s to the rule.

Policy Syntax and Semantics

There are three basic classes of symbols: primitive event symbols, action symbols and constant symbols. Primitive event symbols represent basic events that can be subscribed to in the system. For example, *MonNodeFail* and *ServiceFail* are primitive event symbols that are generated when a monitored node or service fails. An event is a primitive event symbol or a term of the form $e(T_1 t_1, \ldots, T_n t_n)$, where e is a primitive event symbol of n arguments and each t_i is a constant or a variable of type T_i. $e(T_1 t_1, \ldots, T_n t_n)$ represents a parameterized event where the parameters are bound to the data contained in the event.

The condition part of an ECPAP rule is a boolean expression containing constants and variables that appear in the event part of the rule.

Each action symbol denotes the name of a procedure that can be invoked in the system. An action is of the form $proc(t_1, \ldots, t_n)$ where $proc$ is an action symbol and t_is are parameters. For example, $startService(S)$ is an action. Actions are defined in an action library that also contains pre- and post-conditions of actions.

Pre- and post-conditions of an action are first-order predicate logic formulas of the form

$$p_1(\wedge \mid \vee\, p_k)^{k=2..m} \;,$$

p_i is a first-order predicate of the form $Q_1 t_1 \in X_1, \ldots, Q_n t_n \in X_n\ pred(t_1, \ldots, t_n)$: Q_i is a quantifier, X_i is a constant symbol and each t_i is a constant or a variable.

A policy, P is a finite set of ECPAP rules. The adaptation system enforcing the policy expects as input an event e, and its occurrence is represented by $occ(e)$. The semantics of each rule, $(e, c, p) \rightarrow (a, s)$ in the policy is specified by the implication,

$$occ(e) \wedge c \wedge p \rightarrow exec(a)$$
$$exec(a) \rightarrow s$$

where $exec(a)$ represents the initiation of the execution of action a. The evaluation of the rule and execution of the action is treated as an atomic operation, i.e., if the system state changes after the rule evaluation and before the action execution, the change is ignored.

Pre- and Post-condition Expressions

The pre- and post-conditions use pre-defined keywords for specifying first-order expressions. For example, *MonitoredNodes* and *ClientNodes* are keywords that represent sets of monitored nodes and client nodes in our system. In our example scenario these sets are: *MonitoredNodes* = $\{m_1, m_2\}$ and *ClientNodes* = *{store, visual}*. A quantifier over these sets enumerates all the elements of the set. First-order expressions are undecidable in the general case and therefore we convert them to propositional logic expressions, which are decidable, during evaluation.

R$_1$: on(AggregatorFail(Node n))
if(n.id != "FailOver")
{statusNode("FailOver", running)}
do (UseNodeAsAggregator("FailOver"));
{statusAggregator(running)}

R$_2$: on(DataSendFail(Node n))
if(n.id == "MonitoredNode")
{statusAggregator(running)}
do(ReconnectToAggregator(n));
{connectionStatusToAgg(n, connected)}

R$_3$: on(DataReceptionFail(Node n))
if(true)
{statusAggregator(running)}
do(ReconnectToAggregatorAsClient(n));
{connectionStatusToAgg(n, connected)}

R$_4$: on(AggregationAgentStopped(Node n))
if(true)
$\{statusAggregator(running) \wedge \forall\, x \in MonitoredNodes, connectionStatusToAgg(x, connected) \wedge$
$\forall\, x \in ClientNodes, connectionStatusToAgg(x, connected)\}$
do(RestartAggregationAgent());
{statusService("AggregationAgent", running)}

Policy 1: Adaptation Policy for Performance Monitoring Scenario.

Consider, for example, $\forall\, x \in MonitoredNodes$, $statusNode(x, running)$ is a first-order expression that is converted to $statusNode(m_1, running) \wedge statusNode(m_2, running)$ before analysis. These expressions are evaluated by invoking corresponding methods in the system and verifying the return values. For example, $statusNode(m_1, running)$ is evaluated by comparing the return value of the call $statusNode(m_1)$ with the string "*running*."

Example Adaptation Policy

The adaptation policy for the scenario described in the first section is shown in Policy 1. The pre- and post-conditions of actions are shown italicized in braces for convenience and are not specified as part of the rules.

Rule R_1 gets triggered when the aggregator node fails and if the failed node is not the failover node, it assigns the failover node as the new aggregator. Rule R_2 is triggered when any monitored node fails to send data to the aggregator. The rule tries to reconnect the monitored node to the aggregator. Rule R_3 is triggered when a node fails to receive data from the aggregator node. The rule reconnects the node to the aggregator node. Rule R_4 is triggered when the aggregation agent stops. The aggregation agent needs to be started everytime new clients or monitored nodes are connected to the aggregator node. This rule restarts the aggregation agent.

When the aggregator node fails, the monitored nodes are unable to send data and the archival store and visualizer nodes do not receive any data. Therefore, one *AggregatorFail* event, two *DataSendFail* events (one each from two monitored nodes), two *DataReceptionFail* events (one each from the archival store and visualizer nodes) and one *AggregationAgentStopped* events are generated. These events trigger multiple instances of the above rules.

If both instances of rules R_2 and R_3 are enforced before R_1, the nodes try to connect to the failed aggregator and therefore do not succeed. If R_4 is enforced before the other rules, the aggregation agent restart fails and so the nodes can neither send nor receive data. But if R_1 is enforced before R_2 and R_3 and R_4 is enforced in the end, the nodes get connected to the failover aggregator and the monitoring activity is restored. Therefore, the order of enforcement of rules determines system behavior. While "correct" order of rule enforcement is hard to define and requires experimental justification, other simpler guarantees about ordering can be provided as will be discussed shortly.

Ordering Rule Enforcement

An adaptation policy is subject to numerous changes such as addition and deletion of rules, rule modifications and policy composition. Each rule is generally evaluated and enforced independent of other rules in the policy. When multiple rules are triggered the order of enforcement of rules determines the system behavior, as demonstrated in the previous section. Therefore, we define a new notion called *enforcement semantics* that provides certain guarantees about rule enforcement. Enforcement semantics of a policy-based adaptation system dictates the way rules are to be enforced when multiple rules are simultaneously triggered.

V	:	set of trivially-enabled actions
A	:	set of actions of triggered rules
Enable(a)	:	set of actions enabled by action a
P = {Start}	:	set of Petri net Places – initialized to Place called 'Start'
T = {}	:	set of Petri net Transitions
place(a)	:	Place for action a
adj(x)	:	adjacency list of x represented as a set, where $x \in P \cup T$
trans(p, f)	:	Transition with Boolean function f connected by edges from places in set p

Figure 3: Notations used in the workflow generation algorithms.

When a set of rules is triggered, the execution order of the rule actions is determined by constructing a workflow that expresses dependencies between different actions. The pre- and post-conditions of actions determine which action enables which other actions. An action is said to *enable* another action if the post-condition of the former satisfies the pre- condition of the latter. In our example scenario, the post-condition in rule R_1 satisfies the pre-condition in R_2. Therefore, action of R_1 is said to enable that of R_2.

The workflow of rule actions is represented as a Boolean Interpreted Petri net (BIPN) [4], which is useful to model and reason about concurrent action execution. A Boolean Interpreted Petri net is a Petri net [3] whose transitions are assigned Boolean functions. A transition can fire only when all of its input places are marked and its Boolean function is *true*. We assign a place to each action and each transition leading to the place is assigned the pre-condition of the action as the Boolean function. We formally define our workflow BIPN in Appendix I. We will describe the algorithms that construct the workflow in the rest of this section. The various notations used in the algorithms in this section are catalogued in Figure 3.

Workflow Construction

The workflow is constructed by analyzing each pair of actions to determine if one enables the other. The current system state can be represented as a set of propositions and pre-conditions of certain actions may be satisfied by it. These actions are independent of other triggered rules and can be executed as the first set of actions in the workflow. These actions are called trivially-enabled actions.

Definition 1: An action a is said to be *trivially-enabled* if the current state of the system, I, satisfies its pre-condition. Formally, it is represented as $I \models pre(a)$, where $\models$ is the *satisfies* symbol.

In our example scenario, the pre-condition in R_1 : *statusNode*(''*FailOver*'', *running*) is satisfied by the current system state since the failover system is running when the aggregator node fails. Therefore, the action of R_1 can be executed independent of actions in other triggered rules. Algorithm 1 to determine trivially-enabled actions is shown in Figure 4a.

The algorithm initializes the Petri net by assigning a place to each action and creating a transition with the Boolean function *true*. This transition is connected to the *Start* place. The algorithm evaluates the pre- condition of each action to determine if it is true and marks the action as trivially-enabled if so. These trivially-enabled actions are connected by edges from the *true* transition. In our adaptation scenario, only action A_1 is trivially-enabled (action of rule R_i is represented as A_i). The Petri net workflow that results from the algorithm for our adaptation scenario is shown in Figure 4b.

Once trivially-enabled actions have been identified, we check to see which action enables which other actions through *enablement analysis*.

Definition 2: An action a_1 is said to *enable* action a_2 if $post(a_1) \models pre(a_2)$ where $post(a_1)$ represents the post-condition of action a_1 and a_2 is not trivially-enabled.

This implies that execution of a_1 would satisfy the pre-condition of a_2 and so a_2 can be executed after a_1. Since any proposition satisfies the true proposition, we do not check if post-condition of an action satisfies pre-condition of a trivially-enabled action.

Algorithm 2 for enablement analysis is shown in Figure 5a. This algorithm verifies for each triggered action if its post-condition satisfies the pre-condition of a non-trivially-enabled action. It does a pair-wise satisfiability check of actions to determine enablement. The set *Enable*(*a*) contains all actions that are enabled by action *a*. The algorithm iterates through each action *a* and if *a* enables other actions, it connects them to *a* through transitions labeled with their pre-conditions.

In our example scenario, both instances of action A_2 (corresponding to two monitored nodes) and both instances of action A_3 (corresponding to archival store and visualizer) are enabled by A_1. The Petri net resulting from Algorithm 2 is shown in Figure 5b. A_k^i represents the *i*th instance of action A_k. Since two instances of rules R_2 and R_3 are triggered, the Petri net contains two instances of their actions represented as A_2^i and A_3^i, ($i = 1, 2$).

Post-conditions of some actions may satisfy part of the pre-condition of another action. For example, post-pcondition of A_1: *statusAggregator*(*running*) satisfies a part of the pre-condition of A_4. Similarly, post-conditions of A_2^1, A_2^2, A_3^1 and A_3^2 satisfy the other parts of the pre-condition of A_4. Therefore, A_1, A_2^i and A_3^i must be executed to enable A_4. We say that each

```
for each action a ∈ A   //initialization
   P = P ∪ {place(a)}
t = trans({start},true)
adj(Start) = adj(Start) ∪ {t}
T = T ∪ {t}
V = {}                        //trivially-enabled action analysis
for each action a in A
      if pre(a) is true
            V = V ∪ a
for each action a ∈ V   //adding transitions to workflow
   adj(t) = adj(t) ∪ {place(a)}
```

Figure 4a: Algorithm 1: Workflow Initialization and Trivially-enabled action analysis.

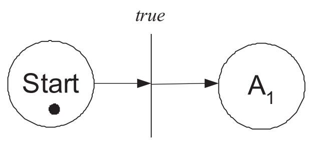


Figure 4b: Petri net workflow after trivially-enabled action analysis.

```
Enable(a) = {}, ∀ a∈A      //enablement analysis
for each action a ∈ A
   for each action b ∈ A-V
      if post(a) ⊨ pre(b)
         Enable(a) = Enable(a) ∪ {b}
for each action a ∈ A        //adding transitions to workflow
   for each action b ∈ Enable(a)
      t = trans({place(a)}, pre(b))
      if t ∉ T
         T = T ∪ {t}
         adj(place(a)) = adj(place(a)) ∪ {t}
      end if
      adj(t) = adj(t) ∪ {place(b)}
   end for
end for
```

Figure 5a: Algorithm 2: Enablement Analysis.

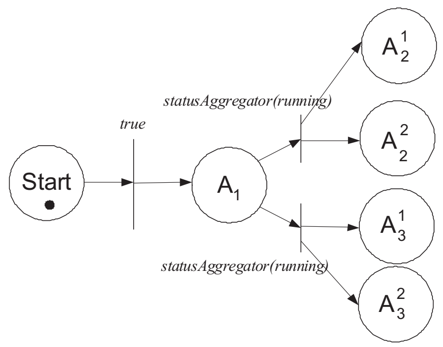


Figure 5b: Petri net workflow after enablement analysis.

action A_1, A_2^i and A_3^i partially-enables A_4. Note that the variables *x* and *n* in predicates *connectionStatusToAggregator*(*x*, *connected*) and *connectionStatusToAggregator*(*n*, *connected*) are bound to values of the nodes during evaluation.

Definition 3: An action a_1 is said to partially-enable action a_2 if post(a_1) $\models$ partial-pre(a_2), where partial-pre(a_2) is a conjunction of some proper subset of conjuncts of pre(a_2). A set of partially-enabling actions of an action a that together enable a is called a *partial-set* of a. An action may have multiple partial-sets and therefore, the set of all partial-sets of a is denoted by *partial-sets(a)* . In the above example,
partial-sets(A_3) = { {$A_1, A_2^1, A_2^2, A_3^1, A_3^2$} }.

Algorithm 3 in Figure 6a determines for every action a that is not trivially-enabled, which set of actions collectively enable a. If the set contains only one action, then it implies that a single action enables a and therefore is already determined by Algorithm 2. Therefore, Algorithm 3 only considers sets having more than one element. In addition, the algorithm does not test an action with itself for partial-enablement as this might lead to a deadlock.

Though the algorithm for partial-enablement analysis can replace enablement analysis of Algorithm 2, we separate the two algorithms since partial-enablement analysis has a much higher complexity. We will discuss this in more detail below when we evaluate the algorithmic complexities.

Once we determine the partial-sets, we complete the workflow construction by adding transitions. The Petri net generated from Algorithm 3 is shown in Figure 6b.

```
Partial-sets(a) = {}
S       : set that temporarily contains partially-enabling
          actions of an action
for each action a ∈ A-V        //partial sets determination
   S = {}
   for each action b ∈ A-{a}
      if b partially-enables a
         S = S ∪ {b}

   for each subset s of S
      if (cardinality(s) > 1)
         p = true
         for each action a ∈ s
            p = p ∧ post(a)
         if p satisfies pre(a)
            Partial-sets(a) = Partial-sets(a) ∪ {s}
      endif
   end for
end for

for each action a ∈ A-V        //adding transitions to workflow
   for each set s ∈ Partial-sets(a)
      t = trans(s, pre(a))
      T = T ∪ {t}
      adj(t) = adj(t) ∪ {place(a)}
      for each action b ∈ s
         adj(place(b)) = adj(place(b)) ∪ {t}
   end for
end for
```

Figure 6a: Algorithm 3: Partial-sets Determination.

Enforcement Semantics

Once dependencies among triggered rule actions have been determined, the enforcement semantics of the adaptation system specifies the execution order of actions. We have identified three different enforcement semantics for policy-based adaptation systems.

Random

This semantics executes rule actions in a random order. The pure ECA policy system without the specification enhancements follows this semantics, implicitly, since it does not provide guarantees about enforcement of multiple triggered rules. This is the weakest of all three semantics and does not require the action workflow to be constructed. This semantics can be used when dependency among rule actions is low and very few rules are triggered by a single change.

All-or-None

The all-or-none semantics specifies that the rule actions in the workflow must be executed only if all actions can eventually execute. This implies that even if one action in the workflow cannot be enabled then the entire workflow should be discarded. In order to

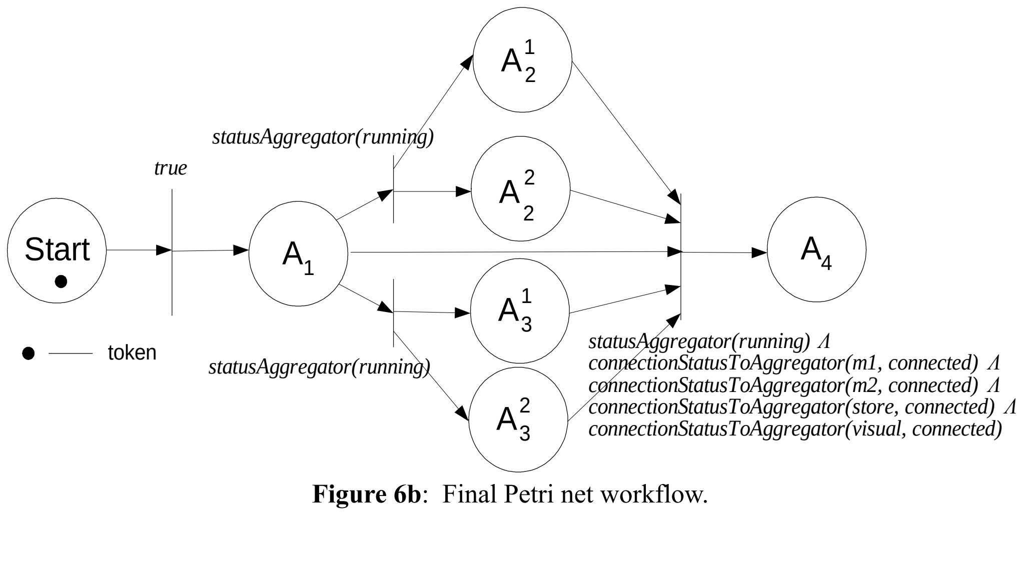

Start
token
true
statusAggregator(running)
statusAggregator(running)
A_1
A_2^1
A_2^2
A_3^1
A_3^2
A_4
statusAggregator(running) Λ
connectionStatusToAggregator(m1, connected) Λ
connectionStatusToAggregator(m2, connected) Λ
connectionStatusToAggregator(store, connected) Λ
connectionStatusToAggregator(visual, connected)

Figure 6b: Final Petri net workflow.

enforce this semantics, the BIPN workflow is analyzed to see if all places can be reached using a reachability algorithm [3]. The all-or-none semantics provides the strongest guarantee and is useful in policies that have high dependency among rule actions.

Maximum Rule

The maximum rule semantics guarantees that the management system enforces rules in an order that ensures as many rules are successfully enforced as possible, provided no other errors cause rule enforcement to fail. The difference between all-or-none and maximum rule enforcement semantics is that in the latter if any place in the workflow can be reached from the *Start* place it will be executed. If a place cannot be reached, the workflow is not discarded as in the all-or-none semantics. Our adaptation framework discussed in the "Framework" section uses the maximum rule enforcement semantics.

We prove formally in Appendix I that the workflow algorithms described above guarantee the above semantics by showing that ordering actions according to the workflow enables maximum number of rules to be successfully enforced.

Action Execution

The order of execution of rule action depends on the enforcement semantics used in the system. If random enforcement is used, the workflow construction is skipped and actions are executed in an arbitrary order. The all-or-none and maximum rule enforcement semantics use Petri net based traversal algorithms to traverse the workflow and execute actions. If the system guarantees all-or-none semantics a reachability analysis [3] is performed to determine if all places are reachable from the Start place prior to execution.

A workflow execution engine analyzes the Petri net for any deadlocks using the deadlock detection algorithm described in [3]. If a deadlock is found the execution engine does not execute any action in the workflow. Currently, we do not resolve deadlocks and abandon the workflow. If the Petri net is deadlock-free, the engine uses a simple Petri net traversal algorithm based on Breadth-First Search (BFS) to traverse the net and execute actions.

The transition states of the Petri net act as synchronization points in the workflow. When multiple places lead to a single transition, the engine waits for the completion of all actions in the places before executing actions of places leading out of the transition. At each transition, the engine verifies the Boolean function for satisfaction before executing the following action. For our adaptation scenario, action A_1 is executed, followed by concurrent execution of actions A_2^i and A_3^i and then action A_4 is executed.

ECPAP-based Adaptation Framework

We have designed a framework based on ECPAP rules for adapting Ganglia and HP OpenView monitoring systems to various changes. The framework is external to the monitoring systems and does not modify either system. It uses the reconfiguration support provided by the systems for adaptation. In this section, we discuss the details of the framework and its applications.

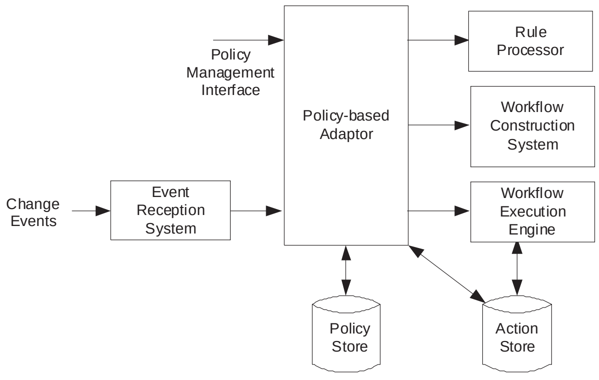


Figure 7a: Adaptation Framework.

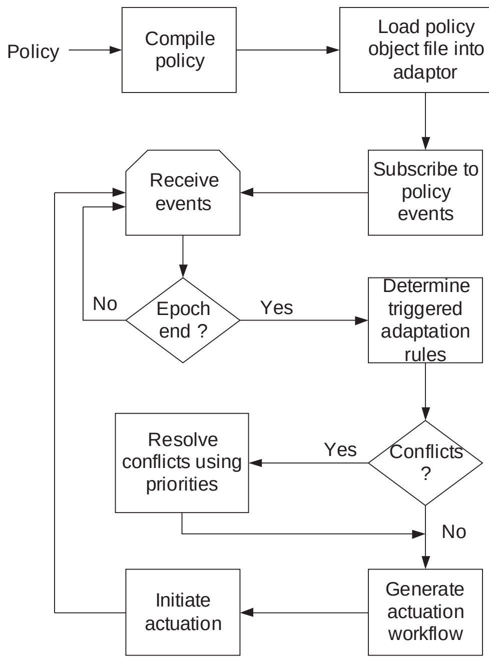

Policy
Compile policy
Load policy object file into adaptor
Subscribe to policy events
Receive events
No
Epoch end ?
Yes
Determine triggered adaptation rules
Resolve conflicts using priorities
Yes
Conflicts ?
No
Initiate actuation
Generate actuation workflow

Figure 7b: Adaptation flowchart.

System Architecture

Adaptation policy, containing ECPAP rules, is compiled and loaded into the adaptation system by the system administrator. The adaptation system subscribes to events specified in the policy and initiates corresponding actions when those events are fired. Figure 7a shows the adaptation framework and Figure 7b illustrates the steps involved in loading a policy, planning adaptation and initiating actuations.

A policy is compiled into a policy object file. The policy object file contains rules in a format suitable for loading into the enforcement system. The enforcement system subscribes to policy events and waits for the occurrence of events

Since a single change to the monitoring system may trigger more than one event occurrence, we define time intervals called *epochs* and consider all events received within an epoch to correspond to a single change. The epoch model for event reception was proposed in [2] and was found suitable for defining policy rules with composed events.

Since a composed event normally contains events that have occurred ''simultaneously'' and event reception system receives events sequentially, the epoch model provides a good approximation to simultaneity. At the end of each epoch, the adaptor evaluates the policy and determines the set of rules that are triggered. The triggered rules are checked for conflicts and resolved using a priority-based resolution technique [10]. The adaptor reasons about the enforcement order of rules using pre- and post-conditions of actions and generates the Petri net workflow. The workflow is executed by a workflow execution engine and the adaptor waits for further events.

The policy-based adaptor supports interfaces to load policies and conflict resolution rules, query the policy store for the loaded policies and retrieve the set of actions in the action store. In addition, the system also supports user interfaces to list available events and actions. These interfaces are useful for designing policies.

Event Reception Model

Our monitoring framework views a *change* as a set of correlated events and evaluates the policy based on the events in the set. Event correlation is a well-researched problem and numerous models have been proposed to group events corresponding to a change [25, 26, 27]. Since the focus of our work is on policy evaluation and enforcement, we use a simple event correlation model based on *epochs* proposed by Chomicki, et al. [2] for policy evaluation. Figure 8 illustrates the epoch model. The input to our adaptation framework is a set of events and therefore, the epoch model can be replaced by more appropriate correlation models without affecting policy evaluation and enforcement.

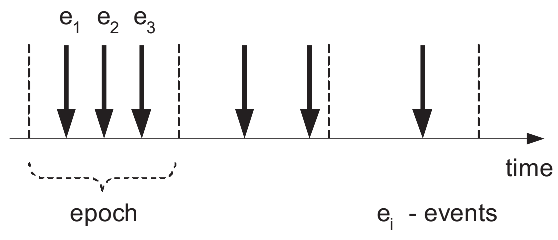
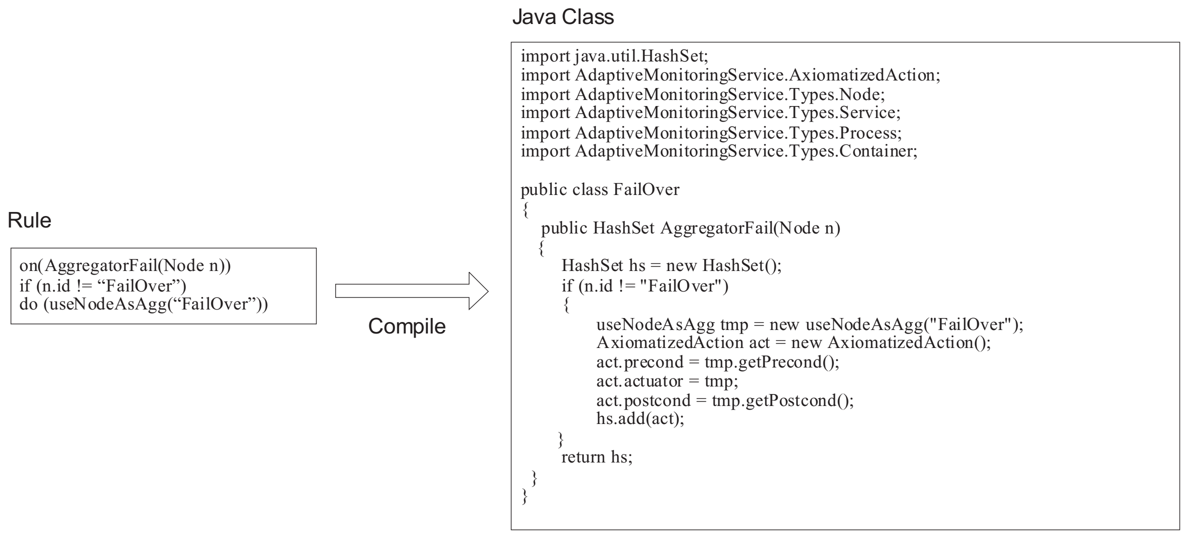


Figure 8: Epoch model.

Policy Tools and Adaptation Implementation

The adaptation framework consists of policy tools and the adaptation system. The policy compiler generates Java class files from policy rules. We used this approach to leverage the language features of Java. An ECPAP rule is compiled into a method that has the same signature as the event. The condition part of the rule is translated into an 'if' block in the method. The conditional expression of the 'if' statement is the same as that in the rule. The rule action is translated into a set of statements that create a Java object containing the action object along with its pre- and post-conditions. If multiple rules have the same event signatures, the compiler consolidates the rules into a single method with multiple 'if' blocks. A typical Java class, for a rule, generated by the policy compiler is shown in Figure 9.

These classes are compiled by a Java compiler into class files and loaded into the adaptation system by the policy loader. When an event is received by the adaptation system, an equivalent method signature is created with the event name as the method name and the event parameters as method parameters. This method is invoked on the rule class files and if the rule contains the event, the method invocation succeeds.



```
on(AggregatorFail(Node n))
if (n.id != “FailOver”)
do (useNodeAsAgg(“FailOver”))
```

```
import java.util.HashSet;
import AdaptiveMonitoringService.AxiomatizedAction;
import AdaptiveMonitoringService.Types.Node;
import AdaptiveMonitoringService.Types.Service;
import AdaptiveMonitoringService.Types.Process;
import AdaptiveMonitoringService.Types.Container;

public class FailOver
{
   public HashSet AggregatorFail(Node n)
   {
      HashSet hs = new HashSet();
      if (n.id != "FailOver")
      {
           useNodeAsAgg tmp = new useNodeAsAgg("FailOver");
           AxiomatizedAction act = new AxiomatizedAction();
           act.precond = tmp.getPrecond();
           act.actuator = tmp;
           act.postcond = tmp.getPostcond();
           hs.add(act);
      }
      return hs;
   }
}
```

Figure 9: Policy compilation.

The condition is checked and if it is satisfied, an object containing the action object along with its pre- and post-conditions is returned. This approach provides the adaptation system with the action specifications necessary for reasoning.

The adaptation system has been implemented in Java. Figure 10 shows the main components of the framework. It consists of a policy store that stores a set of Java objects instantiated from rule classes. An action library acts as an actuator store and contains a set of actions that can be invoked from policy rules. The action library in addition contains specification of actions in first order predicate logic.

Application to Monitoring Systems

Our data center and enterprise systems use HP OpenView and Ganglia systems for performance monitoring. Since monitoring systems run independent of the core system services, we tested the applicability of our adaptation framework to manage changes to these systems.

The configuration of our monitored environment consists of a set of nodes monitored by the Ganglia monitoring system [14]. Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and grids. It supports a hierarchical organization of monitoring agents called Ganglia Monitoring Daemons (gmond) that collect monitoring information from individual nodes.

Aggregation agents called Ganglia Meta Daemons (gmetad) collect data from gmonds and apply aggregation functions to provide consolidated information about cluster of machines. The focus of Ganglia is on monitoring nodes and provides simple replication-based approaches for tolerating node failures. It does not provide support for aggregation agent failures, application migration or other infrastructure changes.

We used the adaptation framework to enhance the resilience of Ganglia monitoring system to changes as proof-of-concept. We developed change detectors to detect aggregator failures and data-send and data-receive failures from the various nodes of the monitored environment. The change detection system generates parameterized events that contain relevant state information. The reasoning system uses XSB Prolog [19] to verify satisfiability of propositions.

We also used the adaptation framework for adapting OpenView monitoring system. The framework enables adaptation when services are plugged-in, plugged-out or migrated, alarm events are generated and so on. For example, the adaptive system dynamically configures the OpenView components to collect performance data from a service when it is plugged-in. Similarly, the framework reconfigures the components to collect additional metrics, through deep-diving (monitoring specific components), when alarms are generated due to high CPU utilization, low memory and so on. Our initial evaluation demonstrates that the ECPAP-based adaptation framework can dramatically reduce the administration cost of OpenView monitoring system.

Evaluation

In this section, we will discuss the algorithmic complexities of the various algorithms presented in the paper and use them to explain the system performance that we have empirically measured.

Algorithmic Complexity

Trivially-enabled action analysis (Algorithm 1) has a linear complexity of $O(n)$ pre-condition checks for n actions. Enablement analysis (Algorithm 2) does a pair-wise satisfiability check of actions and therefore has a quadratic complexity of $O(n^2)$. Partial-enablement analysis (Algorithm 3) analyzes for each action if it is enabled by a set of actions.

Each action subset must be determined and this has an exponential complexity of $O(2^n)$. Since each subset is tested to see if it enables the action for all actions the final complexity is $O(n^2 2^n)$. Currently, Algorithm 3 has a very high complexity but there are various optimizations that can be performed to reduce the value of n.

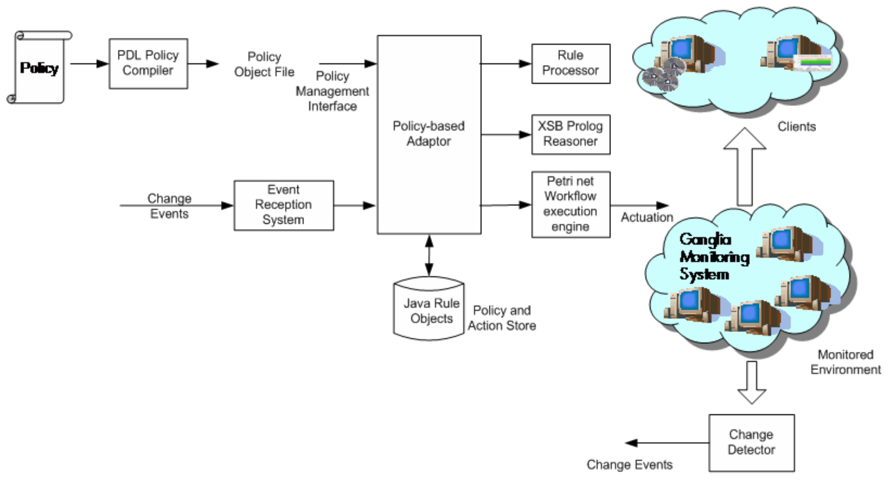


Figure 10: Adaptation framework implementation.

For example, the enablement analysis algorithm reduces the number of rules to be verified during partial-enablement analysis. Since enablement analysis has a quadratic complexity the overall performance overhead is greatly reduced. In addition, the number of rules that are normally triggered on a single event is quite less (fewer than five rules per event in our adaptation policy) and so the overhead is tolerable. We are currently looking at static analysis techniques to determine dependencies between different rules at policy compilation time. Finally, the overall complexity of the workflow generation is $O(n^2 2^n)$, bounded by the complexity of the partial-enablement analysis.

Experimental Validation

The performance overhead of Petri net workflow generation is shown in Figure 11. The adaptation system was executed on a Pentium III 1GHz dual processor SMP machine with 2 GB memory. Figure 11a shows the overhead with varying number of triggered rules. Our test policy had multiple instances of the same rule since the focus was on testing the overhead of the system. As predicted from the algorithmic complexity described above the overhead is exponential with the number of triggered rules. For 15 triggered rules the overhead was found to be around three seconds. Normally, for a typical policy, the number of rules triggered on a single change can be expected to be much less than 15 and so the approach is feasible. Several optimization using configuration and state models can be performed to reduce the overhead. We do not discuss these optimizations since they are out of the scope of the paper.

The number of predicates in pre- and post-conditions of actions influences the Petri net generation overhead. Therefore, we measured the overhead with varying number of predicates in action specifications. Figure 11b illustrates the performance overhead of the system. The x-axis indicates the average number of predicates for each pre- and post-condition. The y-axis shows the overhead in seconds. The overhead is less than one second for about 144 predicates.

Figure 12 shows the graphs of times required for policy compilation and rule evaluation. Policy compilation has a reasonable overhead of about one second for a policy containing 400 rules. Since policies are compiled only when they are modified, this is an acceptable overhead. Rule evaluation is an important component of the adaptation process and therefore its overhead contributes to the overall performance of the system. We measured the performance of the adaptation system with varying number of rules in a policy. We found that rule evaluation takes about 100 ms for a policy with 2000 rules which is a reasonable overhead for an adaptation system.

Experience in a Real Scenario

We have used our adaptation system for managing changes to monitoring systems as was described earlier. Figure 13 shows the Ganglia visualizer output

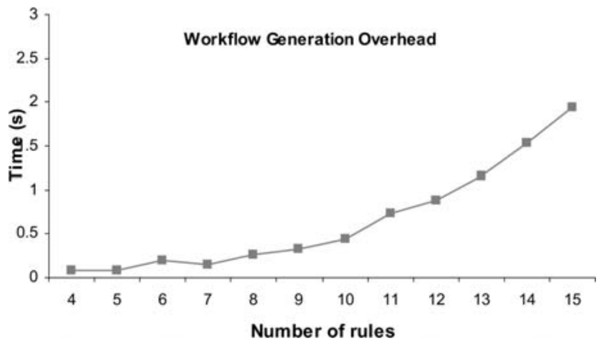


Figure 11a: Petri net Workflow Generation vs Number of Triggered Rules.

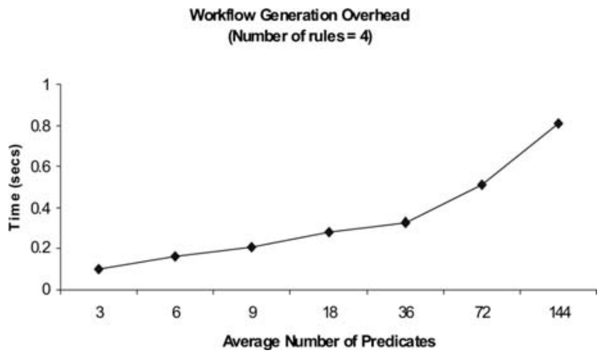


Figure 11b: Petri net Workflow Generation vs Average Number of Predicates.

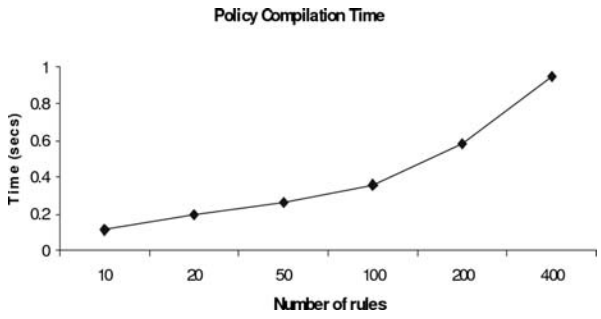


Figure 12a: Policy Compilation Times.

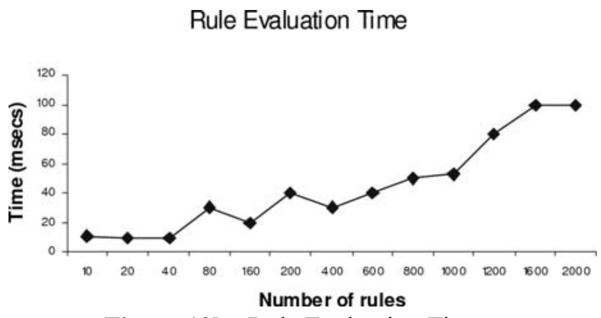


Figure 12b: Rule Evaluation Times.

on aggregator failure for our adaptation scenario. The policy consisted of four rules shown in the "Example Adaptation Policy" section. On aggregator failure, six events – one AggregatorFail, two DataSendFail, two

DataReceptionFail and one AggregationAgentStopped event – are generated. In the first experiment (Figure 13a), an ECPAP system with maximum rule enforcement enabled was used and the aggregator was manually stopped. The temporary disruption during adaptation is illustrated in the figure. The adaptation used the policy from the "Specification-based Rule Framework" section.

Once the rules are enforced, monitoring resumes using the failover aggregator. In the second experiment (Figure 13b), a pure ECA based system was used and the adaptation framework enforced rules as soon as they were triggered. As illustrated in the figure, random enforcement of rules failed to connect the nodes to the failover aggregator and so the visualizer node did not receive data.

Before performing the second experiment, the original aggregator was restarted and the failover aggregator was disconnected. This caused no disruption in data reception since atleast one aggregator was active during the switch. Thus, we see that with an ECPAP based approach, even though there is overhead in workflow generation, we guarantee recovery, whereas with a pure ECA based system (random enforcement order), complete disruption could happen.

Lessons Learned

Policy Design: Designing adaptation policies is an onerous task and requires significant knowledge of the system configuration and functioning. Policy designers should foresee various changes that may affect IT systems and specify rules. This demands significant system knowledge and expertise from the administrator.

Defining Correctness: In this paper, we introduced the notion of enforcement semantics and identified three different semantics for rule enforcement. In order to prescribe a specific semantics we need to define correctness for concurrent rule enforcement. This requires empirical validation to determine if application of a specific semantics provides appropriate guarantees.

Optimization using Models: As discussed in the previous section, the workflow generation algorithms have high complexity. In a cluster of nodes, a single change may cause similar events from multiple nodes to be generated triggering multiple instances of the same rule. The workflow generation complexity can be greatly reduced if it can be inferred that the triggered rules are instances of a smaller set of rules by using configuration models. Similarly, using information models, such as Common Information Model (CIM) for representing system states enables faster evaluation of specifications. These models represent state information of entities and provide a central service that can be queried. Therefore, extending the adaptation framework with models enables optimization and we plan to explore that as extensions to our work.

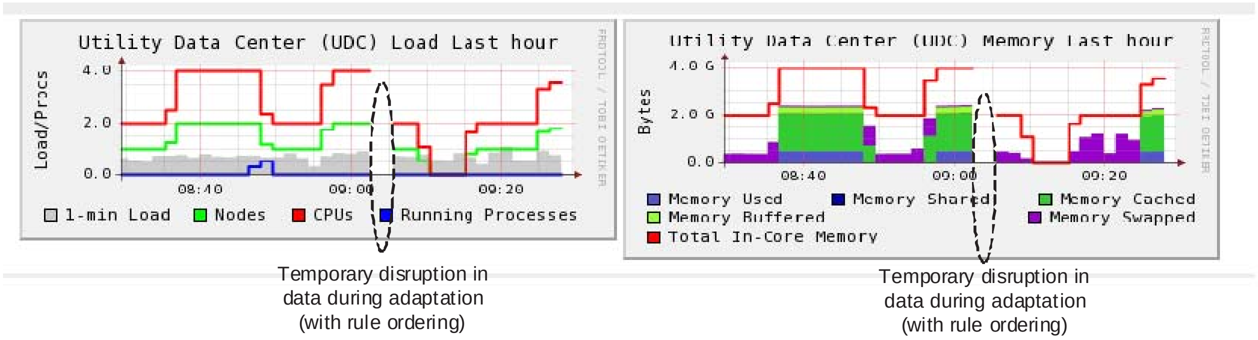


Figure 13a: Disruption in monitored data during adaptation (as perceived by the visualizer node) – ECPAP based system with Maximum Rule Enforcement (with reasoning).

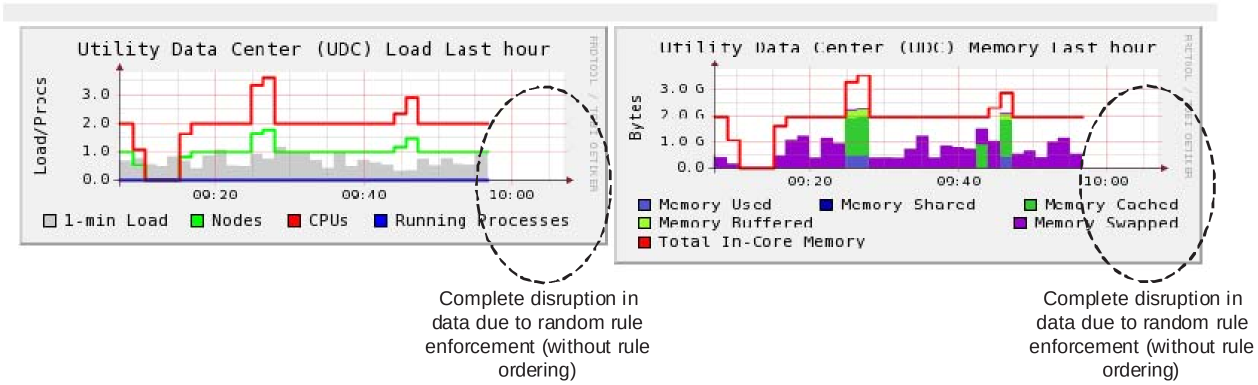


Figure 13b: Pure ECA based system (Random Rule Enforcement).

Event Correlation Models: Finally, the epoch model that we have used approximates a system change in our prototype and in order to develop a fully functioning adaptation system an event correlator is necessary. The correlator would require system information such as configuration, event delivery latency and so on, which can be represented as system models.

Related Work

Automated change management has gained attention in the past few years as a necessary technology to address the adaptation needs of rapidly growing enterprise and grid computing markets. Several research projects are focusing on reducing the administrator efforts in managing different aspects of large distributed systems through policy and model-based approaches.

The CHAMPS project [18] aims to reduce the complexity of IT change and configuration management in distributed environments through planning and scheduling approaches. The project uses model-based approaches to build task graphs to adapt to system changes. Our adaptation framework uses policies and therefore differs significantly from the CHAMPS project. Policy-based management has been used for network switch management [5], managing content distribution networks [6] and distributed systems [7].

The focus of our work is on managing complex IT systems. As we have described, the complexity of these systems causes simultaneous activation of multiple policy rules, which have to be enforced in proper order. None of the projects on policy-based management seem to address this problem, to the best of our knowledge, as we have addressed.

There have been several research efforts in designing policy languages [1, 15], detecting and resolving policy conflicts [2, 9, 11, 22], and various other analyses [8, 20]. To the best of our knowledge, no research in this area has addressed the problem of ordering management rules and providing enforcement guarantees as we have addressed in this paper.

Dunlop, et al. [22] use temporal characteristics of policies to dynamically reason about policy consistency. Their approach detects a large class of conflicts that cannot be detected statically. The focus of their work is on conflict analysis and not on ordering rule enforcement as we have presented in this paper. Our previous work [11] proposes an ECA-P framework to detect and resolve dynamic conflicts that occur due to side-effects of actions. The focus of the work was on conflict analysis and not on enforcement order determination.

Sloman, et al. [7, 8, 9] have developed the Ponder policy specification language and defined techniques for conflict analysis and role-based management. To the best of our knowledge, their work does not address the problem of ordering concurrently triggered rules.

Several research projects in autonomic computing reason about action ordering [12, 13]. These projects are based on AI planning techniques where users specify high-level goals and the planning system determines the ordered set of actions to be executed to reach the desired goal state. The main difference between these projects and our work is that in goal-based approaches the final system state that needs to be reached is known and the system has to determine the actions to be executed to reach that state. In the problem that we have addressed, the final system state is unknown. When an event occurs, a set of rules get triggered and we need to reason about the execution order of the rule actions based on some enforcement semantics.

Finally, our application of policy-based techniques for adapting monitoring systems is a novel research effort. System monitoring is a well-researched field and several monitoring systems such as HP Openview [16], IRISLOG [25], Ganglia [14] and MonALISA [17] are currently being used. These systems focus mainly on data collection, delivery, scalability and fault tolerance. None of these systems support a generic framework that adapts to a spectrum of changes such as application migration, service plug-ins and so on, which is required for automated change management. Our framework uses the configurability features of these systems to adapt to changes based on administrator-specified policies.

Conclusion

IT systems are dynamic and subject to various changes. Management of such changes needs to be automated to reduce administration cost. Policy-based adaptation is a suitable approach where management actions are specified by an administrator, as Event-Condition-Action rules, for different changes in the system. The interdependence of components in modern IT systems causes several change events to be generated when a single change occurs triggering multiple rules. Since the order of enforcement of rules determines the system behavior, adaptation systems should reason about the enforcement order of the rules before initiating corrective actions. We found the ECA rule framework to be poorly suited for reasoning since it does not contain specifications of rule actions.

In this work, we introduced a new rule framework, called Event-Condition-Precondition-Action-Postcondition (ECPAP) that contains action specification for designing adaptation policies for IT systems. When multiple rules are simultaneously triggered on a change, the adaptation system uses the specifications to analyze dependencies between rule actions and generate a Petri net workflow that is executed by an execution engine. We introduce a new notion called enforcement semantics that provides guarantees about rule enforcement. We have used this framework for adapting monitoring systems to changes and presented its performance and advantages in this paper.

Acknowledgements

We would like to thank Martin Arlitt, Keith Farkas, and our shepherd John "Rowan" Littell, for their comments, which have helped improve the content and presentation of the paper. We also thank Rob Kolstad for his extra typesetting efforts that have helped to significantly improve the look and feel of the paper.

Author Biographies

Chetan Shankar is a doctoral candidate at the University of Illinois at Urbana-Champaign (UIUC). He is one of the main contributors to the Active Spaces pervasive computing project at UIUC and has published several papers on pervasive computing and policy-based management. He is broadly interested in services and systems management, programming and management frameworks for dynamic systems and pervasive computing.

Vanish Talwar is a researcher in the Enterprise Systems and Software Lab at Hewlett-Packard Laboratories. His technical interests include distributed systems, operating systems, and computer networks, with a focus on management technologies. He received his M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana Champaign (UIUC) in 2001 and 2006 respectively. He is the recipient of the David J. Kuck Best Masters Thesis award in the Dept. of Computer Science, UIUC, and is an elected member of Phi Kappa Phi and Sigma Xi.

Subu Iyer is a Systems Software Engineer and researcher at HP Labs, Palo Alto. He joined DEC Network Systems Lab in 1997 where he worked on collecting and analyzing performance data from a large cluster of machines on DEC's Palo Alto Research Gateway. Over the years, Subu has worked on projects in the areas of distributed computing, performance monitoring and telepresence. His current work is on scalable adaptive performance monitoring.

Yuan Chen is a post-doctoral researcher in Enterprise Systems and Software Laboratory at HP Labs. He received a B.S degree from University of Science and Technology of China in 1994, and M.S. and Ph.D. degrees from the Georgia Institute of Technology in 2001 and 2005, respectively, all in Computer Science. His current research focuses on performance and systems management in complex and large-scale enterprise computing systems.

Dejan Milojicic is a senior researcher and a project manager at HP Labs. He has worked in the area of operating systems and distributed systems for more than 20 years. He has been the program chair of the IEEE Agent Systems and Applications Symposium (ASA/MA'99) and of the first USENIX Workshop on Industrial Experiences with System Software (WIESS'2000). Dr. Milojicic published in many journals and at various events. He is currently on the editorial board of IEEE Distributed Systems Online. He has been engaged in various standardization bodies, such as OMG and Global Grid Forum. He is a member of the ACM, IEEE, and USENIX. He received his B.Sc. and M.Sc. from University of Belgrade and his Ph.D. from University of Kaiserslautern. Prior to HP Labs, Dejan worked at Institute "Mihajlo Pupin," Belgrade and at OSF Research Institute, Cambridge, MA.

Roy Campbell is the Sohaib and Sara Abbasi Professor of Computer Science at the Siebel Center for Computer Science at the University of Illinois, Urbana-Champaign. He has supervised the completion of forty Ph.D. dissertations and the author of over two hundred and forty four research papers on security, programming languages, software engineering, operating systems, distributed systems, and networking. His research includes the Gaia project on Active Spaces, the security of the power grid, and mobile computer operating systems. Professor Campbell is Director of the University Of Illinois Center Of Academic Excellence in Information Assurance Education, a member of the Information Trust Institute and, with Guy Garnett, directs the Cultural Computing Program. He is a member of the ACM and an IEEE Fellow.

Bibliography

[1] Lobo, J., et al., "A Policy Description Language," *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pp. 291-298, July, 1999.

[2] Chomicki, J., J. Lobo, and S. Naqvi, "Conflict Resolution Using Logic Programming," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15, Num. 1, pp. 244-249, January/February, 2003.

[3] Reisig, W., *Petri Nets: An Introduction*, Springer-Verlag, New York, 1985.

[4] Roussev, B. N., "Self-checking Implementation of Boolean Interpreted Petri Nets," *Proceedings of IEEE Symposium on Emerging Technologies and Factory Automation*, 1994.

[5] Bhatia, R., et al., "Policy Evaluation for Network Management," *Proceedings of 19th Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM 2000)*, pp. 1107-1116, March, 2000.

[6] Amiri, K., et al., "Policy Based Management of Content Distribution Networks," *IEEE Network Magazine*, 2002.

[7] Sloman, M., "Policy Driven Management For Distributed Systems," *Plenum Press Journal of Network and Systems Management*, Vol 2, Num. 4, Dec., 1994, pp. 333-360.

[8] Lupu, E. C., *A Role-Based Framework for Distributed Systems Management*, Ph.D. Thesis, Imperial College, London, 1998.

[9] Lupu, E. C., et al., "Conflicts in Policy-Based Distributed Systems Management," *IEEE Transactions on Software Engineering*, Vol. 25, pp. 852-869, Nov., 1999.

[10] Shankar, C., et al., "A Policy-based Management Framework for Pervasive Systems using Axiomatized Rule Actions," *Proceedings of Fourth*

IEEE International Symposium on Network Computing and Applications (IEEE NCA05), MA, 2005.

[11] Shankar, C., et al., "An ECA-P Policy-based Framework for Managing Ubiquitous Computing Environments," *Proceedings of The Third Annual International Conference on Mobile and Ubiquitous Systems: Networks and Services (Mobiquitous 2005)*, San Diego, July, 2005.

[12] Andrzejak, A., et al., "FeedbackFlow – An Adaptive Workflow Generator for System Management," *Proceedings of The Second IEEE International Conference on Autonomic Computing (ICAC-05)*, June, 2005.

[13] Srivastava, B., et al., "The Case for Automated Planning in Autonomic Computing," *Proceedings of The 2nd IEEE International Conference on Autonomic Computing (ICAC-05)*, June, 2005.

[14] Massie, M., B. Chun, and D. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience," *Parallel Computing*, Vol. 30, Issue 7, July 2004.

[15] Damianou, N., et al., "The Ponder Specification Language," *Proceedings of Workshop on Policies for Distributed Systems and Networks (Policy2001)*, HP Labs Bristol, pp. 29-31, Jan., 2001.

[16] *HP OpenView Management Solutions*, http://www.managementsoftware.hp.com/ .

[17] Newman, H. B., et al., "MonALISA: A Distributed Monitoring Service Architecture," *Proceedings of 2003 Conference for Computing in High Energy and Nuclear Physics (CHEP03)*, La Jolla, California, March, 2003.

[18] Brown, A., et al., "A Model of Configuration Complexity and its Application to a Change Management System," *Proceedings of the 9th International IFIP/IEEE Symposium on Integrated Management (IM 2005)*, May, 2005.

[19] *XSB Logic Programming and Deductive Database system for UNIX and Windows*, http://xsb.sourceforge.net/ .

[20] Verma, D., "Simplifying Network Administration using Policy based Management," *IEEE Network Magazine*, 2002.

[21] Baralis, E. and J. Widom, "Better Static Rule Analysis for Active Database Systems," *ACM Transactions on Database Systems*, Vol. 25, Num. 3, pp. 269-332, September, 2000.

[22] Dunlop, N., et al., "Dynamic Conflict Detection in Policy-Based Management Systems," *Proceedings of Enterprise Distributed Object Computing Conference (EDOC '02)*, September, 2002.

[23] Hoare, C. A. R., "An axiomatic Basis for Computer Programming," *Communications of the ACM*, Vol. 12, Num. 10, 1969.

[24] Ranganathan, A., et al., "Pervasive Autonomic Computing Based on Planning," *Proceedings of IEEE International Conference on Autonomic Computing (ICAC-04)*, May, 2004.

[25] Nath, S., et al., "Tolerating Correlated Failures in Wide-Area Monitoring Services," Intel Research TR, May, 2004.

[26] Kliger, S., et al., "A Coding Approach to Event Correlation," *Proceedings of the 4th International IFIP/IEEE Symposium on Integrated Management (IM 1997)*, 1997.

[27] OpenView Event Correlation Service, http://www.www.managementsoftware.hp.com/products/ecs/ .

Appendix I

Definition 4: Formally, the BIPN of a set of actions $A = \{a_1, \ldots, a_n\}$ is a 1-safe marked Petri net [4] represented as a triple $B = (P, T, F)$ where

$P = \{ \text{place}(a) \mid \forall\ a \in A \} \cup \{\text{Start}\}$, where $\text{place}(a)$ is the place representation of action a.

$T = \{ t_{K,\text{pre}(a)} \mid \forall\ x \in K,\ t_{K,\text{pre}(a)} \in x^\bullet \wedge \text{place}(a) \in t_{K,\text{pre}(a)}{}^\bullet \}$, where K is a set of places and for $x \in P \cup T$, ${}^\bullet x = \{y \mid yFx\}$ is called the *input set* of x and $x^\bullet = \{y \mid xFy\}$ is called the *output set* of x and the flow relation, $F \subseteq (PxT) \cup (TxP)$ such that $\text{dom}(F) \cup \text{codom}(F) = P \cup T$. $\text{pre}(a)$ represents the pre-condition of action a.

The Petri net generated from Algorithms 1-3 for actoin set A is represented as $B = (P, T, F)$ where

$P = \{ \text{place}(a) \mid \forall\ a \in A \} \cup \{ \text{Start} \}$

$T = \{ t_{i,j} \mid (i = \{ \text{Start} \}, j = \text{true}) \wedge$

$\quad (i = \{\text{place}(a)\}, j = \text{pre}(b) \mid \forall\ a, b \in A, \text{post}(a) \models \text{pre}(b)) \wedge (\text{pre}(b) \not\models \text{true}) \wedge$

$\quad (i = s, j = \text{pre}(b) \mid \forall b \in A, (\forall s \in 2^{P-\{\text{Start}\}},\ \Lambda_{\forall k \in s}\ \text{post}(\text{action}(k))) \models \text{pre}(b)) \}$,

$\quad$ $action(k)$ represents the action in set A assigned to place k.

$F = \{ (x, y) \mid \forall\ t_{i,j} \in T, \forall\ x \in i, y = t_{i,j} \} \cup$

$\quad \{ (x, y) \mid \forall\ t_{i,j} \in T, (x = t_{i,j} \wedge y = \text{place}(k), \forall\ k \in A \mid j = \text{pre}(k)) \}$

The three conjuncts in the definition of T correspond to the transitions resulting from Algorithms 1-3. The transitions are labeled $t_{i,j}$ where $i = {}^\bullet t_{i,j}$ and j is the assigned Boolean function. The flow relation, F, represents the various edges of the Petri net.

Theorem 1: For a set of actions $A = \{a_1, \cdots, a_n\}$, the Petri net generated by Algorithms 1-3 enables maximum number of actions starting from the current system state I.

Proof. To prove the above theorem, it is sufficient to prove that for every action $a \in A$, if $I \Rightarrow_k a$, then there is a reachable path [3] in the Petri net from the $Start$ place to $place(a)$, where $I \Rightarrow_k a$ means that starting from the current system state I, successful execution of k actions of A enables a. $X \rightarrow a_1$ implies execution of all actions of set X enables a_1. We prove this by structural induction on the Petri net.

Basis: $I \Rightarrow_0 a$

$pre(a)$ is satisfied by current system state and so a is trivially-enabled by Algorithm 1. Therefore, $t_{\{Start\},true} \in T$ and $\{ (Start,\ t_{\{Start\},true}), (t_{\{Start\},true}, place(a)) \} \subseteq F$. Therefore, there is a reachable path from S to $place(a)$ through the transition labeled $t_{\{Start\},true}$.

Hypothesis: Assume if $I \Rightarrow_k a$ there is a reachable path from $Start$ to $place(a)$. We need to prove that if $I \Rightarrow_{k+1} a_1$ there exists a reachable path from Start to $place(a_1)$.

Since $I \Rightarrow_k a$ from our inductive hypothesis, there is a set of actions $A' \subset A$ such that $\forall\ x \in A', I \Rightarrow_{l \leq k} x$ and $A' \rightarrow a_1$. Therefore, there is a reachable path from $Start$ to $place(x)$ for all $x \in A'$. There are two cases to consider.

Case 1: $A' = \{a\}$ Since a is found to enable a_1 from enablement analysis in Algorithm 2, $t_{\{place(a)\},pre(a_1)} \in T$ and $\{ (place(a), t_{\{place(a)\},pre(a_1)}), (t_{\{place(a),pre(a_1)\}}, place(a_1)) \} \subset F$. Therefore, there is a reachable path from $place(a)$ to $place(a_1)$ and since by hypothesis there exists a reachable path from $Start$ to $place(a)$, by transitivity, there is a reachable path from $Start$ to $place(a_1)$.

Case 2: $Cardinality(A') > 1$ Actions in A' are found to enable a_1 from partial-enablement analysis in Algorithm 3. Therefore, $t_{\{place(x) \mid \forall\ x \in A'\},pre(a_1)} \in T$ and

$$\{ (place(a) \mid \forall\ a \in A', t_{\{place(x) \mid \forall\ x \in A'\},pre(a_1)}), (t_{\{place(x) \mid \forall\ x \in A'\},pre(a_1)}, place(a_1))\} \subset F.$$

Therefore, there is a reachable path from $place(x), \forall\ x \in A'$ to $place(a_1)$ through the transition $t_{\{place(x) \mid \forall\ x \in A'\},pre(a_1)}$. Since there is a reachable path from $Start$ to $place(x), \forall\ x \in A'$ from our hypothesis, by transitivity, there is a reachable path from $Start$ to $place(a_1)$.