Chapter 3

A Simplified Approach for the Design of Symptom Catalog in Autonomic Computing System

3.1 Introduction

With an increase in the complexity of business and business organizations, the need to maintain and monitor various real-time applications is also growing. Autonomic Computing presents a new approach to build systems that have an ability to sustain and optimize themselves, so that the working IT environment is stable and reliable. In this IT environment, the IT professionals are free from system maintenance and can focus on high value tasks by making technology work smarter, with business rules guiding systems to be self-configuring, self-healing, self-optimizing, and self-protecting (Self-CHOP) [37] [38] In an autonomic environment, components work together, communicating with each other and with high-level management tools. They can manage or control themselves with each other. Components can manage themselves to some extent, but from an overall system standpoint, some decisions need to be made by higher level components that can make the appropriate trade-offs based on policies that are in place.
IBM’s vision of Autonomic Computing [6] [8] [16] [39] has been a buzzword from last few years and has laid a foundation for autonomic systems by taking initiatives towards autonomic computing for relieving humans from the burden and tedium of managing the modern complex system. Autonomic computing helps to address the complexity by using technology to manage technology. It works independently on predefined policies and rules without human interaction and manages and configures itself on its own, based on general policies and knowledge that is gained over the time [4] [38] [40].

By enabling computer systems to have self-CHOP features, autonomic computing is expected to have many benefits for the business systems, such as reduced operating costs, reduced downtime, increased security, and the ability to have systems that can respond more quickly to the needs of the business within the market in which they operate. Also, the productivity of the computing system would be improved. Autonomic Computing Toolkit (ACTK) [13] [26] [41] which assists software developers in enabling components, products, and solutions to become autonomic. The toolkit also provides the base for developers of management software to be able to manage entities that are properly enabled. The Toolkit comprises of Log Trace Analyzer, Generic Log Adapter, Symptom Catalog, and Correlation Engine etc, which can together implement autonomic solutions. Symptom Catalog is a repository of the rules which are used for monitoring the application. It maintains the rules and associated steps to be taken when that rule is breached. IBM provides such a Symptom catalog in its Autonomic Computing Toolkit [42].

3.2 A framework for self-healing Autonomic System

The domain of this research orients around current day applications which maintain log files for maintenance purposes. The log-files ensure easier analysis and maintenance of the application by creating a record for every event occurring during the working of the application. The applications, although maintaining logs, can neither take any steps to curb the errors occurring nor can they prevent similar errors from occurring in future runs. The consequences of such types of errors may lead to a system-crash or loss of data and information. Such crashes or data-losses cannot be tolerated in today’s scenario where every millisecond lost by the crashed system can lead to enormous losses to the business.

The focus is around such systems in which applications create log-files of their executions but do not have capability to protect themselves from crashes. A computing system has been
created that can sustain itself and prevent errors from occurring by taking appropriate actions before they can occur.

3.2.1 Proposed System architecture

Figure 3.1 shows the proposed architecture of the Autonomic System Manager (ASM) software that attempts to implement the computing system based on autonomic concepts which can allow the administrator to create and maintain rules and policies as per the requirement using the symptom editor and presents the self-monitoring and self-healing property of an autonomous system by monitoring a log-file for breach of policies (set by administrator) and alerting the administrator in the case of any breach (an act healing) of policies.

![Figure 3.1 Autonomic System Manager (ASM) Architecture](image)

The proposed Autonomic System Manager (ASM) focuses on creating and representing the symptoms in human readable form that uses variable length record field format. The structure of the symptom catalog is simplified by representing the symptoms in variable record-field length format and the records are stored in hash table organization that has a constant symptom access time of $O(1)+|k|$. The ASM can monitor applications by creating policies and suitable actions in the Symptom catalog, using the user friendly symptom editor for smooth working of the application. Any breach of policy is undesired the ASM takes appropriate actions and alerts the administrator via SMS or Email. As the symptoms are organized in the hash table, symptom access time is enhanced as compared to traditional method.

The proposed architecture of the ASM software as shown in figure 3.1 consists of four main sub-components they are:
1. **Monitoring**: Monitors the application and fetches its log-records. After fetching, the log records are matched against policies. If the match is successful, action-sub-component is invoked.

2. **Action**: Once a breach of policy is detected, this action sub-component sends an alert to the administrator (SMS or EMAIL or both as preset by the administrator). The administrator can reply back with specific action-command. The action-sub-component extracts this command and executes it as remedial step. If no reply is received, a default action set for that policy is executed.

3. **Symptom Catalog**: The monitoring sub-component fetches the policies for matching the symptoms from the Symptom Catalog. The Symptom Catalog is a file organized as a hash table containing the symptoms/policies in variable record field length format, stored by the system administrator. The figure 3.2a and figure 3.2b show the symptom format and their representation using hash table organization.

![Figure 3.2.a Structure of Symptom format](image)

The creation and manipulation of symptoms in the symptom catalog would be carried out with the help of a tool called as the Symptom Editor for any application being monitored. The system administrator can create simple rule for monitoring a single event or can create complex rules containing sub rules and operators between them to form expressions. These complex rules can be added, modified or deleted any time by using the symptom editor. Figure 3.3 illustrate the snapshot of symptom editor screen. The symptom catalog contains a set of rules and their associated actions. The rules are
identified by a unique rule-id and this rule-id is also helpful in creation of complex rules.

![Figure 3.2 b Hash table organization of Symptom catalog](image)

Every rule created by the user is validated and written into the symptom catalog. Each symptom in the symptom catalog is variable length record with three fields namely Rule_ID, Rule description and Action. The details are as under.

1. **Rule-ID:** This is a unique identifier for every rule which is automatically generated from R1 to R999. This is the first field of the Symptom that is terminated with delimiter “#”.

2. **Rule Description:** This specifies the description of the rule being stored in the symptom catalog. It is a symptom which might occur in the log files enclosed in a parenthesis. It can be a simple event consisting of the log variables with value or can be a rule by itself. A simple event is indicated by the field name and its value separated by a relational operator (== or !=). Such events are used as operands in the expression and operators like AND, OR, EXCEPT, BETWEEN etc are used for the operations.
Syntax: Rule → (simple_event)  
| (simple_event) Logical_OP (simple_event)  

Simple_event → Log_variable == value  
| Log_variable != value  

Logical_OP → AND  | OR  | BETWEEN | EXCEPT  

Logvariable → SITUATION | SEVERITY | PRIORITY | MESSAGE  
| SOURCE_COMPONENT_ID  

Example_1: (SITUATION == parameter_not_accessible) OR (SEVERITY == 40)  
Example_2: (SEVERITY == 60) BETWEEN (SEVERITY == 70)  

The Example_1 and Example_2 illustrate rules created with log-variables situation, where “situation” specifies the type of the situation that caused the event to be reported and “severity” indicates the event status-severity-level with respect to the component that reports the event. Report situation is simulated while constructing rules with few operators like ‘==’, ‘!=’, ‘OR’, ‘AND’ and ‘BETWEEN” for the symptom catalog. However, the simulation can be extended for all possible situations listed in the next section 3.2.2, with additional set of operators.  

3. Action: This field shows the action to be performed for that rule. The options include EMAIL, SMS or a combination of both or NONE. This field indicates the default alerting action that is carried out when a match is found. This default alerting will let administrator know about the types of errors occurring in the application. The administrator is also provided with a facility using which administrator can reply to the alert using a command (basically an API) of the application. This command is then parsed and executed by the ASM to bring the system to a stable state.
4. **Symptom Editor**: The Symptom Editor is a tool which allows the system administrator to create and maintain rules for any application. The system administrator can create simple rule for monitoring a single event or can create complex rules containing sub-rules and operators between them to form expressions. These complex rules can be added, modified or deleted any time by using the symptom editor.

Figure 3.3 shows the Symptom Rule adder screen of the Symptom Editor. The **Field and Value** Combo-boxes allow the user to select events to monitor. The **Operator** field allows a user to select an operation between two such events. An event can be a simple event of the log-file or a pre-existing rule. In the above figure 3.3, each of the sub-panel allows user to create a rule (indicated by the Rule ID **R008, R009 and R010**). The figure 3.3 also shows the feature of the Symptom Catalog, wherein user can create simple rules (Rule **R008**) or can use simple rules to create a complex rule (Rule **R010**). The user-friendly way of storing symptoms bring in a great deal of flexibility in monitoring and maintaining the working of any application. These policies will allow the administrator to control the resources used and also optimize the throughput of the application.
3.2.2 Implementation details and working of the system

The proposed system is implemented on Windows XP platform and coding is done using Java Developers Kit (JDK) version used is v1.6. In order to demonstrate the working of ASM software the simulated log-file is generated continuously as long as the ASM software is running and monitoring function is being performed. Figure 3.4 shows the structure of simulated log file used demonstration.

![Figure 3.4 The structure of Simulated log file](image)

<table>
<thead>
<tr>
<th>Date_and_Time</th>
<th>Global_Instance_Id</th>
<th>Reporter_Component_Id</th>
<th>Source_Component_Id</th>
<th>Situation</th>
<th>Message</th>
<th>Severity</th>
<th>Priority</th>
</tr>
</thead>
</table>

The simulated log file is generated in a CBE format. As, the log-file is a simulated log-file, the contents of the logs would be set in such a way that only required log records look meaningful and the rest others as set of a particular character (‘x’) written in the format specified. This is done because the normal records written into a logs are not required for ASM software. It is only concerned with the erroneous set of records which it can identify. Also, this type of a format for the simulated log file simplifies the working of the ASM software.
The simulated log has eight fields in CBE format in accordance of their occurrence:

1) **Date_and_Time:** This element specifies the time at which this Common Base Event was created. It uses a type called dateTime as specified in XML schema. The lexical representation for this element is as follows:

   CCYY-MM-DDThh:mm:ss

   Where, CC represents the century, YY the year, MM the month, and DD the day. The letter T is the date and time separator and hh, mm, and ss represent hour, minute, and second, respectively.

2) **Global_Instance_Id:** The Global_Instance_Id element is a complex data type that represents the primary identifier for the event. This element uniquely identifies the event; therefore, it could be used as the primary key for the event.

3) **Reporter_Component_Id:** The Reporter_Component_Id element identifies the component that reported the event or situation on behalf of the affected component.

4) **Source_Component_Id:** The Source_Component_Id element identifies the component that was affected by the event or situation.

5) **Situation:** This element describes the category of event that was detected, providing autonomic managers with information to perform appropriate actions. Following are the possible situation types that can cause the event to be reported. Report situation is simulated while constructing rules with few operators like ‘==’, ‘!=’, ‘OR’, ‘AND’ and ‘BETWEEN’ for the symptom catalog. However the simulation can be extended for all possible situations with additional set of operators.

   - **StartSituation:** The StartSituation deals with the start-up process for a component. Messages that indicate that a component has begun the startup process, that it has finished the startup process, or that it has aborted the start-up process then all fall into this category. Existing messages include words like ‘starting’, ‘started’, ‘initializing’, and ‘initialized’.

   - **StopSituation:** The StopSituation deals with the shutdown process for a component. Messages that indicate that a component has begun to stop, that it has stopped, or that the stopping process has failed then all fall into this category. Existing messages include words like ‘stop’, ‘stopping’, ‘stopped’, ‘completed’, and ‘exiting’.

   - **ConnectSituation:** The ConnectSituation deals with the situations that identify aspects about a connection to another component. Messages that say a connection has failed, that a connection was created, or that a connection has ended then all fall into this
category. Existing messages include words like ‘connection reset’, ‘connection failed’, and ‘failed to get a connection’.

- **ConfigureSituation**: The ConfigureSituation deals with the components identifying their configuration. Any changes that a component makes to its configuration should be logged using this category. Additionally, messages that describe current-configuration-state fall into this category. Existing message includes words like ‘port number is’, ‘address is’, and ‘process id’.

- **RequestSituation**: The RequestSituation deals with the situations that a component uses to identify the completion-status of a request. Typically, these requests are complex management tasks or transactions that a component undertakes on behalf of a requestor and not the mainline simple requests or transactions. Existing messages include words like ‘configuration synchronization started’, and ‘backup procedure complete’.

- **FeatureSituation**: The FeatureSituation deals with the situations that announce that a feature of a component is now ready (or not ready) for service requests. Situations that indicate things like services being available and services or features being unavailable fall into this category. Existing situations include words like ‘now available’, ‘currently available’, and ‘transport is listening on port 123’.

- **DependencySituation**: The DependencySituation deals with the situations that components produce to say that they cannot find some component or feature that they need. This category includes messages about not finding the version of the component that was expected. Messages that say a resource was not found, or that an application or subsystem that was unavailable, also fall into this category. Existing messages include words like ‘could not find’, and ‘no such component’.

- **CreateSituation**: The CreateSituation deals with the situations documenting when a component creates an entity. Messages telling that a document was created, or a file was created, or an EJB was created all fall into this category. Existing message include words like was ‘created’, ‘about to create’, and ‘now exists’.

- **DestroySituation**: The DestroySituation deals with the situations documenting when an entity or component was removed or destroyed. Messages telling that a document was destroyed or a file was deleted all fall into this category. Existing message include words like ‘was created’, ‘about to create’, and ‘now exists’.

- **ReportSituation**: The ReportSituation deals with the situations reported from the component, such as heartbeat or performance information. Data such as current CPU
utilization, current memory heap size, and so on would fall into this category. Existing messages include words like ‘utilization value is’, ‘buffer size is’, ‘parameter not accessible’ and ‘memory threshold exceeded’.

- **AvailableSituation**: The AvailableSituation deals with the situations reported from the component, regarding its operational state and availability. This situation provides a context for operations that can be performed on the component by distinguishing if a product is installed, operational and ready to process functional requests, or operational and ready/not ready to process management requests. Existing message include words like ‘those that now ready to take requests’, ‘online’, and ‘offline’.

- **OtherSituation**: The OtherSituation category is to provide support for the situation that is product-specific requirement rather than the predefined categories.

**6) Message**: The message element is the text that accompanies the event.

**7) Severity**: The severity element is used to indicate the severity level of the event from the point of view of the component that reports the event. The predefined severity levels, in order of increasing severity, are as follows:

1. 0: Unknown
2. 10: Information MUST be used for cases when the event contains only general information and is not reporting an error.
3. 20: Harmless MUST be used for cases in which the error event has no effect on the normal operation of the resource.
4. 30: Warning MUST be used when it is appropriate to let the user decide if an action is needed in response to the event.
5. 40: Minor MUST be used to indicate that action is needed but the situation is not serious at this time.
6. 50: Critical MUST be used to indicate that an immediate action is needed and the scope is broad (perhaps an imminent outage to a critical resource will result).
7. 60: Fatal MUST be used to indicate that an error has occurred, but it is too late to take remedial action.

The associated values are 0 to 70. The reserved values start at 0 for Unknown and increase by increments of 10 to 60 for Fatal. Other severities MAY be added but MUST NOT exceed 70. This is an OPTIONAL property.
8) **Priority:** The priority element defines the importance of the event so that an event-consumer can establish a relative order in which the event records should be processed. The predefined priorities vary from 0 to 5 where 0 is low, 3 is medium and 5 is high.

The figure 3.5 shows the snapshot of contents of the simulated log-file with normal and error-record. As we can see, the normal records are written as records with character ‘x’ repeated as per format. Only error records are meaningful.

**Normal record**:

```
TXX:XX:XX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
```

**Error record**:

```
08-03-03T14:32:46 ASM-DL-EV-000004 ASM-DL-RS-000731 ASM-DL-RS-000452 Parameter_not_accessible Parameter_not_accessible_sample 50 02
```

Figure 3.5 snapshots of normal and error-records in simulated file

The ASM software implements four modules and the details are as follows

1. **Monitoring**

This module consists of code to retrieve contents of a log-file and monitor it. In this case, the log file is a simulated log-file being generated continuously in the CBE format which is shown in the figure 3.4 and 3.5. The simulated log is generated continuously through a java program that runs in the background and produces log-records continuously and these records are recorded into a file.

Another java program monitors this file, i.e. it reads the file and fetches predefined number of lines from the file. The fetching takes places continuously as long as the software is running. The pre condition required for this to happen is that the log file should be written continuously as long as software runs. This parallel working of the two programs is achieved by using the concept of serialization. After the log records are extracted, the records are parsed and fields are separated.

The figure 3.6 is a sequence diagram that illustrates the process of monitoring and creation of simulated log file.
1. Store normal record format in a String variable (refer figure 3.5).
2. Keep on writing normal records continuously after appending system time and date.
3. At a random time, pause the normal record writing and insert an error record (all possible error-records are stored in a file).
4. After this, restart the writing of normal records.
5. Keep on repeating the writing as long as the ASM software is running.

From the above steps it is observed that the normal records are written continuously and error records are inserted in between at random. This simulates a real application log generation.

2. Symptom Matching

This module consists of a code to extract the rules from the symptom-catalog and as well as the code to parse the log-fields-records and match them. The Figure 3.7 is a sequence diagram that demonstrates the symptom matching process.
1. Extract a set of records from the log-file being generated.
2. Extract one record out of the fetched records.
3. Separate and parse all the fields from a record.
4. Read the Symptom-Catalog and extract rules one by one.
5. Parse the Rule description of the extracted Symptom-Catalog record and extract the fields needed from source log –record for matching.
6. Analyze the source log fields and match against the symptom catalog’s rule description. Break down the complex rules if necessary.
7. Call the action module if a match is found.

From the above steps it is seen that the log-records are matched against symptom records and action module is called if match is found. This simulates monitoring of an application.

### 3. Action

This module consists of performing appropriate actions when a successful match occurs. The action is an alert to the administrator in terms of an EMAIL or an SMS. The administrator can also reply with a command back to the ASM. The ASM will then parse the command and perform the ordered execution if it is possible.
Figure 3.8 is a sequence diagram that demonstrates the symptom matching process.

1. Get the administrator’s details (like mobile number and email address) from the registration details.
2. For E-MAIL, Compute port address. Contact the mail server through that port.
3. Send a mail using the internet connection.
4. For SMS, connect to the external GSM modem.
5. Send a message using the GSM modem and its SIM card.
6. For receiving E-MAIL and SMS, use the built-in functions in a similar way. The ASM will receive replies both via E-MAIL and SMS.
7. After receiving the replies, parse it to extract the command.
8. Use the application APIs to execute the command.
9. If no replies are received for a period of time, execute the default actions stored.

From the above steps it is observed that an action is taken for every match found. The administrator is alerted, and the administrator can reply back with commands. These commands
are executed as a remedy for the error. If no reply is received, default action is performed. This simulates the self-healing of the autonomous system.

4. Symptom Editing:

This module consists of a code for creating and editing rules. All the rule-creation and maintenance is done through the interface as shown in Figure 3.9 and Figure 3.10.

![Symptom Catalog Creation](image)

Figure 3.9 Snapshot of Symptom Catalog creations

The symptom editing features include rule-addition, rule-deletion and rule-modification. The following are the details of the screen as highlighted in the above Figure 3.9.

1. Rule ID
   The label 1 indicates the id for the rule being created. If these rules are being used in creation of deletion of other rules, the rule IDs appear in the drop down lists.

2. Rule
   The label 2 indicates a symptom which might occur in the log files. The fields are in the standard CBE format. The operators used here are AND, OR, EXCEPT and BETWEEN. The operator “BETWEEN “applies to only Priority and Severity fields and when it is selected the logical NOT operator is disabled. The other operators apply to all the fields. The rules are nested correctly with brackets (“(“ and “)”).

3. Action
   The label 3 indicates the action to be taken. The following actions are available: SMS, EMAIL, and None.

4. ADD RULE
5. MODIFY RULE
6. DELETE RULE
7. OK
3. **Action**

The label 3 indicates the action to be performed for that rule. The options include EMAIL, SMS or a combination of both and NONE.

4. **Add Rule**

The label 4 indicates the Rule addition that calls a form to create a new rule. Figure 3.3 shows the symptom rule-adder form, which illustrates rule adding process in the symptom-catalog.

5. **MODIFY RULE**

The label 5 indicates a Rule modification that calls a form to modify the existing rule.

6. **DELETE RULE**

The label 6 indicates a Rule deletion that calls a form to delete a selected rule.

7. **OK**

The label 7 indication OK that confirms all the symptom rule creations

Figures 3.10 to Figure 3.12 show the Rule-addition, Rule-modification and Rule-deletion operations that can be performed using the Symptom Editor.

![Symptom Rule Adder Screen](image)

**Figure 3.10 Symptom Rule Adder Screen**

The following are the details of the screen as highlighted in the figure 3.10.
1. **Rule ID**

This label indicates the ID for the rule being created. If these rules are being used in creation of deletion of other rules, the rule IDs appear in the drop down lists.

2. **Not operator**

This label indicates the radio button (if pressed) to indicate negation of the selected field.

3. **Field and Value dropdown lists**

This label indicates field that allows user to select field and event-value for the rule being created. Various combinations are available for selection. The field marked ‘5’ also serves the same purpose.

4. **Operator field**

This field allows us to select an operator between the two event-fields. Various operators like ‘OR’, ‘AND’, ‘EXCEPT’ and ‘BETWEEN’ are available for a user to create a rule.

5. **Action field**

This field allows us to select the action we wish to set as a remedial action. The options include EMAIL, SMS or a combination of both.

6. **MORE**

This button allows us to enable one of the set of fields, i.e. it allows us to create one more rule. This is particularly useful when creating complex rules.

7. **RESET**

This button resets all the fields of the Rule-adder-screen.

8. **DONE**

This button writes all the rules created in the symptom-catalog file.

9. **CANCEL**

This button cancels the rule creation-task and takes the user back to symptom-catalog screen.

Figure 3.10 shows the Symptom-Rule-adder screen of the Symptom-Editor that is activated by enabling the button “ADD RULE”. The **Field and Value** Combo-boxes allow the user to
select events to monitor. The **Operator** field allows a user to select an operation between two such events. An event can be a simple event of the log-file or a pre-existing rule. In the above figure 3.3, each of the sub-panel allows the user to create a rule (indicated by the Rule ID **R008, R009 and R010**). The figure 3.3 also shows the feature of the Symptom Catalog, wherein the user can create simple rules (Rule **R008**) or can use simple rules to create a complex rule (Rule **R010**). The user-friendly way of storing symptoms bring in a great deal of flexibility in monitoring and maintaining the working of any application. These policies will allow the administrator to control the resources used and also optimize the throughput of the application.

Figure 3.11 shows the symptom-Rule-modifier-screen that can be brought up when user selects MODIFY RULE (after selecting a rule) from the symptom-catalog-screen. This screen will allow the user to modify an existing rule into a new rule as per the requirement.

The following are the details of the screen as highlighted in the snapshot above.

1. **Existing Rule:** This area shows the current rule that is going to be modified.
2. **Modified Rule:** This area shows the modified rule.

![Figure 3.11 Symptom Rule Modifier screen](image)

**Figure 3.11 Symptom Rule Modifier screen**

1. **RESET**
   This button resets the values selected in combo-drop-down lists of the modified rule.
2. **DONE**
   This button allows the user to verify whether the selected rule is valid or not.
3. **OK**
   This button writes the rule modified into the symptom catalog.
4. CANCEL

This button cancels the modification task and takes the user back to symptom catalog screen.

![Symptom Rule Deletion Screen](image)

Figure 3.12 Symptom Rule deletion Screen

Figure 3.12 shows the Symptom-rule-deletion screen that can be brought up when the user selects **DELETE-RULE** (after selecting a rule) from the symptom catalog screen. This screen will allow the user to delete an existing rule from the symptom catalog only if it is not used anywhere else.

The following are the details of the screen as highlighted in the snapshot above.

1. **RULE TO DELETE**
   
   This area shows the current rule that is going to be deleted.

2. **YES**
   
   This button when clicked will delete the rule selected from the symptom-catalog. Before deleting, the rule is first checked, i.e. whether the rule is being used by some other rules as a sub rule. If so, then the user is not allowed to delete the rule. Otherwise, the rule can be deleted.

3. **NO**
   
   This button allows the user to cancel rule-deletion task and go back to symptom-catalog screen.
1.2.3 Validation of the proposed work

The creation of ASM is inspired by the works of IBM on Autonomic Computing. IBM’s Autonomic Computing Toolkit also contains many features apart from its own Symptom Catalog. The traditional symptom catalog is complex because of its mesh like structure and to reference the symptoms and generate corrective actions many XML paths have to be accessed by writing XPATH expression [25] [43]. This results in increase time in initiating the action whenever the symptoms is matched The scope of this section is limited to comparison between the proposed ASM’s Symptom Catalog and the traditional ASM’s Symptom Catalog.

1. Proposed ASM’s Symptom Catalog

The Proposed ASM’s Symptom Catalog in the figure 3.2a and figure 3.2b has a simple yet effective structure for maintaining rules which are organized as a hash table in variable-record-length-field format. The structure of the symptom catalog is simplified by using the variable length record-field format, organized as hash table that has constant symptom access time. It is found that the symptom access time in terms of average path length would be $O(1) +|k|$, where $k$ is length of the chain.

For example:

Let us consider the ‘n’ symptom records represented in hash table

$L_{avg} = O(1)+|k| \text{ where } L_{avg} =$Average path length

$|k| = \text{length of the chain}$

if $k=1$ or $2$ then average path length is always approximated to $1.5$ then the Symptom access time in terms of average path length $L_{avg}$ is approximated to $2.5$.

2. IBM’s Autonomic Computing Toolkit (ACTK) Symptom Catalog

Figure 3.13 and 3.14 show the traditional IBM’s ACTK symptom-catalog that is in XML format along with symptom reference structure. Each symptom is composed of several structural elements and extension points.

These are:

• Symptom definition
• Symptom rule
• Symptom effect
• Symptom (instance)
• Symptom engine

In addition, there are auxiliary structural elements that are used in multiple symptom elements. These are: Identification, Versioning, Annotation, Location, Scope and Life cycle.
A symptom-catalog is designed to store and provide management to symptom definitions so that they can be consumed at run time by analysis and monitoring applications. The logical structure of a symptom reference is illustrated in Figure 3.15. This figure is used in the sections that follow to help locate, within the Symptom Reference format structure.

As compared to ASM’s Symptom Catalog, the Autonomic Computing Toolkit’s Symptom Catalog is quite complex because of its mesh like structure and to reference the symptoms and
to generate corrective actions many XML paths have to be accessed by writing XPATH expression. This results in increase time, in initiating the action for the symptom match.

1.2.4 Performance Analysis

The performance of proposed method is analyzed assuming realistic values of path lengths and their probability of references. The reference is confined to estimating only the average access time and not the insert/delete operations, as insert operation is one time activity. The estimated time in terms average path length would be approximated as follows:

\[ L_{avg} = \sum_{i=1}^{K} (pl_i * pt_i) \]

Where \( L_{avg} \) = average path length

\( K \) = number of possible paths

\( pl_i \) = Occurrence ith path

\( pt_i \) = probability of occurrence ith path

<table>
<thead>
<tr>
<th>Table 3.2 Approximated data values for traditional symptom reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Symptom paths</strong></td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

Table 3.2 shows the approximated data values for symptom reference which are estimated from the traditional symptom reference structure from the figure 3.14. It is observed that the symptom access time in terms of average path length is 3.3477. If it is compared with proposed ASM’s symptom catalog, there is a reduction of 25.30% theoretically and 53% practically in symptom access time was achieved. Figure 3.15 shows the comparison between Traditional and proposed hash based Symptom catalog with respect to symptom access time depending on the number of records. It is quite evident that the proposed hash based symptom catalog is effective.
with respect to searching of symptoms in the table. The details of the implementation and calculation are included in Appendix B. The insert and delete operations are one time activities hence, the effectiveness of the data structure with respect to these two operations has not been explored.

![Graph comparing access time for traditional and hash-based symptom catalogs]

Figure 3.15 Comparison of Traditional Symptom catalog with Hash based Symptom catalog.

The following Table 3.3 compares the features of the proposed Symptom catalog of ASM with IBM’s ACTK symptom catalog.

**Table 3.3 Comparison of proposed symptom catalog of ASM with symptom catalog of IBM’s ACTK.**

<table>
<thead>
<tr>
<th>FEATURE</th>
<th>ASM’s SC</th>
<th>ACTK’s SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symptom reference time</td>
<td>Improved</td>
<td>More</td>
</tr>
<tr>
<td>Action Execution</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Symptom Creating and Editing</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Rule Reusing</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Rule Expressions</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Administrator Alerts (by default)</td>
<td>Built in alerts</td>
<td>Not built in</td>
</tr>
<tr>
<td>Manual Action Specifying</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Following are the other features compared in the table:

1. **Action Execution**: Both ASM and ACTK allow actions to be taken when any breach of policy is noted.
2. **Symptom Creating and Editing:** Both ASM and ACTK allow the users to create and edit rules however, the structure is simple which is in variable-record length-field format as compare to complex XML structure of ACTK.

3. **Rule Reusing and Rule Expressions:** Both ASM and ACTK allow the users to create rule expressions by using simple events or by reusing existing rules. But creating specific expressions in ACT requires users to have knowledge about XML and XPATH, whereas ASM provides very user-friendly environment which even a novice administrator can make use of it with ease.

4. **Administrator Alerts:** Although ACTK can implement this feature by allowing the user to specify alerts as the action for rules by allowing the user to write alerting modules on his own, ASM provides this feature by default. A novice administrator may not be able to write his own module for alerts. This is where ASM proves to be more informative to the administrator in terms of alerting the administrators about the types of errors occurring in the applications.

5. **Manual action specifying:** ACTK by default does not let administrators to specify manual actions in real-time. ASM provides a facility for this feature by allowing administrators to reply back for the alerts sent with a command (basically an API of the application) which the ASM can parse and execute to bring the application to a stable state.

The comparison tells us that although most of the features are same in both the Symptom Catalogs, the distinguishing features of the ASM’s Symptom Catalog are its structure and provision for creation of rule expressions for policies in a simpler way. Also, the proposed ASM would allow more user-friendliness and manual specification of Actions. These features would help an administrator to monitor the application in a very flexible and simple way.

**Summary**

This work presents a simpler method of implementing IBM’s MAPE-K framework of Autonomic system that supports self-healing capability into a software product. The structure of the symptom catalog has been simplified that improves the symptom access time by 25.30% theoretically and 53% with practical implementation, as compared to traditional XML structure of symptom catalog. The software product would be monitored for healthy working by presenting novel approach for building a symptom catalog that contains possible symptoms and
associated actions. It also empowers the administrator with the ability to create and maintain primitive policies for any application in a convenient way. The symptom catalog presented in this work contains a variety of operators which will allow system administrator to have various combinations of events in creating rules and associated actions. Also, the manual action execution feature (for any error occurring) presents a method by which administrator can be ensured of full control of the working of any application.