Automatic Proactive Troubleshooting with IBM Rational Build Forge

Automatic repair and maintenance of Rational and third party toolsets with IBM Rational Build Forge

Spencer Murata
William Frontiero

November 8, 2010

Introduction

Refrigerator companies have often floated the idea of having intelligent refrigerators that would call in service requests for themselves when components were failing. The basis of this idea is that better diagnostics are driven by greater integration between computerized parts. This paper brings this idea to Rational products by using Build Forge to fix or "phone home" potential support issues proactively without user intervention. The premise of this is that the many support issues that Rational products encounter are essentially programmatic in nature. That is to say most Rational problems can be determined through step-by-step shell commands, monitoring the result and if require execute more commands. Moreover most of those problems can then be fixed through an additional set of shell commands. In most cases the process is already in place and IBM support uses those processes to diagnose and fix known issues. You could then use that process to determine the cause of the problem if they had the same knowledge of the product that the support technicians have. Currently we address that knowledge gap through content and training, but Build Forge can bring the internal support processes directly to the end users. Further through the automation of Build Forge, those processes can run autonomously and invisibly on the user system; fixing and/or reporting errors before the end user is aware of them. This paper will discuss the implementation Build Forge projects to proactively seek out those support issues.

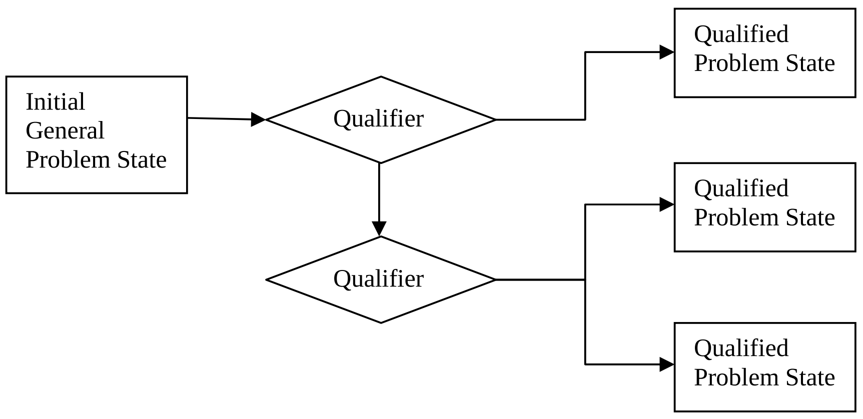
1 What is Build Forge?

Build Forge is an IBM Rational product that implements process automation. In Build Forge there are discrete units called Projects that contain collections of steps that contain atomic command line commands. The execution of the project will execute the steps one by one in sequence until the completion of the project. Build Forge is also particularly good at describing processes that require dynamic environmental input and generating new commands based on that information. So more than static processes that are immutable, Build Forge can describe dynamic processes that are non-deterministic.

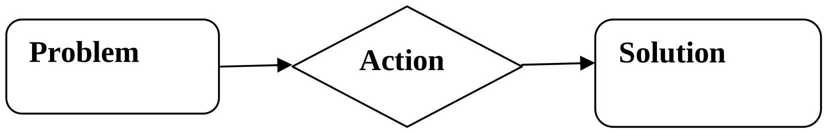
1.1 Non Deterministic Processes

The idea of the non-deterministic processes applies directly to the support process at a high level, as support diagnostics can be broken down into programmatic processes that are not necessarily static. IBM produces technotes, whitepapers, and other content to try to bring those diagnostic processes and knowledge out to the general user base. This content is generally presented as a process. The process usually starts with a simple diagnostic statement along the lines of "If you do x, then y occurs". The result of this first step determines two things: first, whether the problem described in the content will apply to the end user, or in other words: does the aberrant behavior that the content is describing exist on the end users system? Second, depending on the results, what parts of the process will apply or if none of it applies? At the beginning of the process though, the end state is not known making the application of the process dynamic and non-deterministic.

Diagnostic process:



From there the process will delve into enumerated steps that resolve the issue. Remember that different cases take different paths to the solution depending on what the exact nature of the problem is. That type of process plays directly to the strengths of Build Forge, which does not require a static process to run, but rather dynamically determines the process execution path depending on the information received during the execution of the process. Within IBM support much of the work is determining what diagnostics to perform and what information is required to repair an issue.



Often the issue could be resolved on the end user side, but is not due to insufficient knowledge on the end users side to bring the issue to resolution. Bringing the diagnostic and repair processes into Build Forge brings a unique opportunity to take the exact processes from IBM support and deliver them to customers. By creating a collection of Build Forge projects that encapsulate verified Rational support solutions we can eliminate end user confusion and improved overall environment health. The other benefit of translating support processes into Build Forge projects is that there is a much smaller cost to running those processes. So processes that may have been too complicated, time-consuming, or simply annoying to run may be run more often. Not only are we able to fix problems with the Build Forge projects, but we can carry out maintenance tasks that may have been ignored or put off.

2 Combining Build Forge with other toolsets

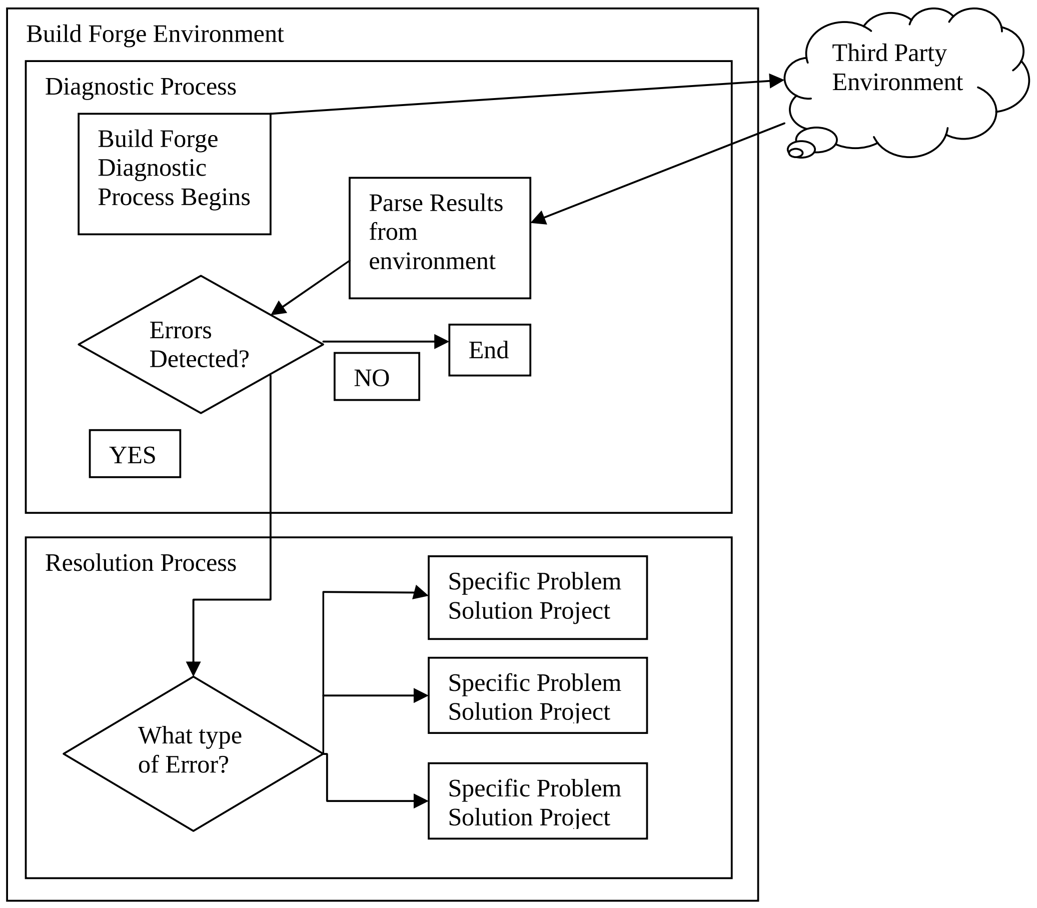
There are two main processes that construct the proactive repair maintenance solution: diagnostic and resolution.

2.1 Diagnostic Process

First Build Forge needs to mimic the end user detecting an abnormal condition. In other words, it has to find a condition where “If I do x, y happens, but I expected z” occurs. To do this Build Forge needs to continuously check for that error condition or set of conditions to occur. This continuous checking idea is known as continuous integration. The implementation of the Build Forge has a mechanism for dealing with this called an adaptor. Continuous integration is commonly associated with code changes, but the usage is broader within Build Forge. Specifically, the Build Forge adaptor mechanism is designed to monitor for a state change in the software environment and when that change occurs it fires another Build Forge process. This mechanism can be used to help monitor the software environment automatically. As mentioned before, the first part of the support process is to determine what the problem is based on what is observed in the environment. Build Forge here is a tireless watchdog that is looking for particular environments conditions to occur and trigger actions based on those conditions. So when known abnormal conditions occur, then Build Forge can detect them and diagnose them programmatically. Build Forge can then automatically use the wrong state data to fire the corresponding repair process for the specific problem that was detected.

2.2 Resolution Process

The second is the resolution process that is actually tailored to repair the specific problem. The resolution process is actually deterministic as when deciding to execute the resolution the problem is known and well defined. All the data that the resolution process requires should be collected in the diagnostic process so the specific resolution has everything that it needs. This process is comprised of only the repair steps needed to set the software environment back to a valid state.


Build Forge Environment
Diagnostic Process
Build Forge Diagnostic Process Begins
Third Party Environment
Parse Results from environment
Errors Detected?
End
NO
YES
Resolution Process
Specific Problem Solution Project
What type of Error?
Specific Problem Solution Project
Specific Problem Solution Project

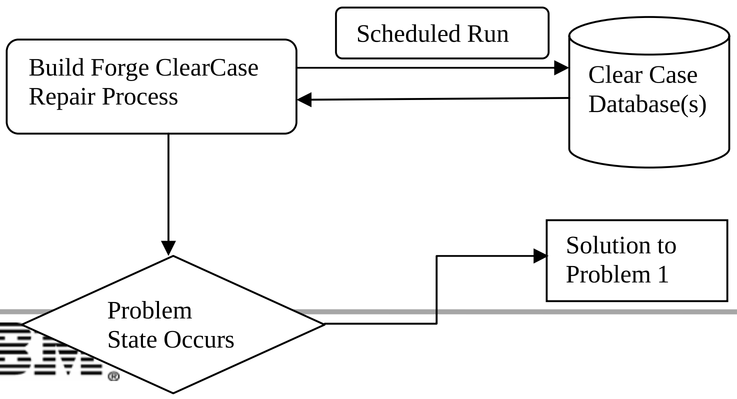
3 Practical Implementation

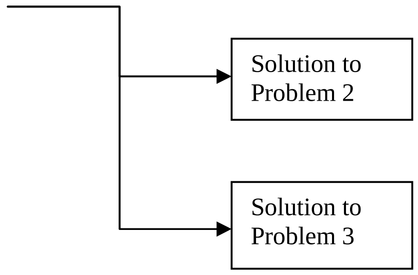
3.1 Diagnostic Implementation Details

The premise of this paper is very broad and far reaching, so a specific use case will be identified and implemented to demonstrate it. For this purpose a ClearCase broken hyperlink will be identified, removed, and IBM support will be notified of the issue. First we need to determine if a broken hyperlink exists in the ClearCase environment. A custom adaptor will look for the broken html link error. If one is detected it will find and store the necessary environment information to repair the problem. We will link the adaptor to the project that will handle the removal of the hyperlink and notification of support. This way, if and only if a broken hyperlink is found, the project will be triggered to run. The first adaptor part will call **`cleartool lsvob`** to get the list of ClearCase vobs in the ClearCase environment. After verifying that a Build Forge ClearCase view is present, the adaptor calls **`pushd M:\bfview && cleartool checkvob -ucm vob:$1.`** That command will change the directory into the view that was found and running the checkvob utility against each of the vobs that was found in the previous commands. The results of that checkvob are then run against a regular expression that is looking for the text **`Broken BaselineLbtype hyperlink in`** which would indicate that a broken hyperlink was found. The adaptor then will then call a script that will run the corresponding Build Forge project to the problem that was found. In this case it would be the broken hyperlink project.

3.2 Resolution Implementation Details

Once the broken hyperlink project has fired, it will then attempt to repair the error that was found. So in the case of the broken hyperlink, it will run **`cleartool dump -l`** with the component information from the error. This will get the details of the error. The project will then make a decision whether to fix the problem based on what the Build Forge user has set up for the environment. If the environment is set to fix the problem, the project will run a **`cleartool checkvob -hlinks -force`** command on the ClearCase object. That will clear out the broken hyperlinks automatically. However it should be noted that depending on the problem being diagnosed, the fix may not be that easy to apply. In those other situations, the fix maybe too complicated or potentially dangerous to apply automatically, but the problem does not have to be fixed entirely at this point. If the problem is not automatically repairable, the environment can be locked and the damage contained as much as possible. This way, problems that cannot be fixed on the spot can be prevented from spreading damage and small problems remain small problems rather than snowballing into big problems. Locking or repairing the environment will put everything into a stable situation for support to come in and properly diagnose, clean, and repair everything.





3.3 Notification Implementation Details

Support will be contacted automatically via Build Forge. Build Forge has the ability to email messages based on the results of the project builds. This way, when the project has reached a conclusion, we can take the action to notify the appropriate people and support about the issue. A customer could notify their internal support people about a particular type of problem that may be fixable internally. If the problem is severe enough, the email can be sent to IBM Support directly. Build Forge will contextually populate the notification with the exact information that support will require to fix the issue. The other possible notification implementation is to use a servlet to wrap around the IBM ESR tool. That way the data from Build Forge console could be sent directly to ESR and a PMR generated on the spot for the problem. With both IBM Support contact method and the customer internal support notification, the problem can be addressed before most end users are aware of it.

4 Conclusion

By using the concepts in the project outlined in this paper, Build Forge can be integrated with all Rational products and other third party toolsets to provide autonomic repair functionality. Build Forge can effectively heal other software environments without manual intervention. There will always be known problems that can be diagnosed and/or fixed in a programmatic process to provide autonomous repair support. Implementing those diagnostic and resolution processes will improve turnaround time on diagnosing issues as well as keeping software environments clean. By not requiring manual intervention to run and monitor diagnostic and repair processes, the processes can be run more frequently. Running those processes more frequently means issues that cannot be fixed immediately are caught early on and can be effectively contained. The small issues then will not grow to be big issues later on. Containment and control of issues means more stability and uptime for Rational or other software deployments, which means better productivity. Build Forge is a valuable tool for keeping a software environment, particularly a complex one, running smoothly. Build Forge can be the attentive administrator running diagnostics 24 hours a day maintaining the health and integrity of all the various subcomponents that may comprise the software ecosystem.

5 Master Check Adaptor and Perl script

This is the adaptor xml that implements the adaptor that will be used for the practical implementation of the diagnostic process for the ClearCase example. The clearcase_fix.pl is a perl services script that fires a given Build Forge project and environment with the given selector.

MasterCheck.xml

```
<?xml version="1.0"?>
<!-- (c) Copyright by IBM Corp. and other(s) [2003], 2007 All Rights
Reserved. -->
<!DOCTYPE PROJECT_INTERFACE SYSTEM "interface.dtd">
<PROJECT_INTERFACE IFTYPE="Source" INSTANCE="7.02">
<template>
      <env name="VOBSERVER" value="logical_server_name"/>
      <env name="SELECTOR" value="selector_name"/>
      <env name="view" value="view"/>
      <env name="vob_tag" value="tag"/>
      <env name="vob_storage" value="storage"/>
</template>
<interface>
      <run command="get_vobs" params="" server="$VOBSERVER" dir="/"
timeout="360"/>
      <ontempenv name="cc_errors" state="empty">
            <step result="FAIL"/>
      </ontempenv>
</interface>
<command name="get_vobs">
      <execute>
            cleartool lsvob
      </execute>
      <resultsblock>
            <match pattern="^\*?\s+(\/|\\.*?)\s+(.*?)\s+(.*?)$">
                  <setenv name="checkvoboutput" value=""
type="temp"/>
                  <run command="check_view" params=""
server="$VOBSERVER" dir="/" timeout="360"/>
                  <run command="checkvoboid" params="$1 $2"
server="$VOBSERVER" dir="/" timeout="360"/>
                  <run command="checkvob" params="$1 $2"
server="$VOBSERVER" dir="/" timeout="360"/>
            </match>
      </resultsblock>
</command>
<command name="check_view">
      <execute>
            cleartool startview bfview
      </execute>
      <resultsblock>
            <match pattern="View tag not found:">
                  <run command="create_view" params=""
server="$VOBSERVER" dir="/" timeout="360"/>
            </match>
      </resultsblock>
</command>
<command name="create_view">
      <execute>
            cleartool mkview -tag bfview -stgloc -auto
      </execute>
      <resultsblock>
      </resultsblock>
</command>
```

```
<command name="checkvoboid">
      <execute>
            pushd M:\bfview &amp;&amp; cleartool checkvob -ucm vob:$1
> ${BF_ROOT}ucmVobCheck.txt 2>&amp;1 ${BF_ROOT}ucmVobCheck.txt
            cat ${BF_ROOT}ucmVobCheck.txt
      </execute>
      <resultsblock>
            <match pattern="Scanning object: vob:(.*?) ...">
                  <setenv name="temp_vob_tag" value="$1"
type="temp"/>
                  <bom category="vobcheck" section="vob">
                        <field name="tag" text="$1"/>
                        <field name="server" text="$BF_SERVER"/>
                  </bom>
                  <bom category="vobcheck" section="error">
                        <field name="ucm_obj" text=""/>
                        <field name="state" text=""/>
                        <field name="error" text=""/>
                  </bom>
            </match>
            <match pattern="Bad modcnt VOB oid: (.*?) in (.*?)$">
                  <setenv name="bad_oid" value="$1" type="temp"/>
            </match>
      </resultsblock>
</command>
<command name="checkvob">
      <execute>
            cat ${BF_ROOT}ucmVobCheck.txt
      </execute>
      <resultsblock>
            <match pattern="Broken ComponentRootDir hyperlink in
(.*?)$">
                  <bom category="vobcheck" section="error">
                        <field name="ucm_obj" text="$1"/>
                        <field name="state" text="Broken"/>
                  </bom>
                  <setenv name="cc_errors" value="true" type="temp"/>
                  <run command="repairvob"
params="Fix_Broken_RootCompDir
vob_tag~${temp_vob_tag},VOBSERVER~${VOBSERVER},comp~$1"
server="$VOBSERVER" dir="/" timeout="360"/>
            </match>
            <match pattern="Broken BaselineLbtype hyperlink in
(.*?)$">
                  <bom category="vobcheck" section="error">
                        <field name="ucm_obj" text="BaselineLbtype"/>
                        <field name="state" text="Broken"/>
                  </bom>
                  <setenv name="cc_errors" value="true" type="temp"/>
                  <run command="repairvob"
params="Fix_Broken_BaselineLbtype
vob_tag~${temp_vob_tag},VOBSERVER~${VOBSERVER},bad_oid~${bad_oid},com
p~$1" server="$VOBSERVER" dir="/" timeout="360"/>
            </match>
            <match pattern="Bad modcnt VOB oid: (.*?) in (.*?)$">
                  <bom category="vobcheck" section="error">
                        <field name="ucm_obj" text="$1"/>
                        <field name="state" text="Bad OID"/>
                  </bom>
                  <setenv name="cc_errors" value="true" type="temp"/>
            </match>
            <match pattern="Has (.*?) broken (.*?) hyperlinks in
(.*?)$">
                  <bom category="vobcheck" section="error">
```

```
                        <field name="ucm_obj" text="$3"/>
                        <field name="state" text="Broken"/>
                  </bom>
                  <setenv name="cc_errors" value="true" type="temp"/>
            </match>
            <match pattern="cleartool: Warning:(.*)">
                  <bom category="vobcheck" section="error">
                        <field name="error" text="$1"/>
                  </bom>
            </match>
      </resultsblock>
</command>
<command name="repairvob">
      <integrate>
            perl clearcase_fix.pl $1 $SELECTOR $2
      </integrate>
      <resultsblock/>
</command>
<bomformat category="vobcheck" title="VOBs Checked">
      <section name="vob">
            <field order="1" name="tag" title="VOB Tag"/>
            <field order="2" name="server" title="VOB Server"/>
      </section>
      <section name="error" parent="vob" expandable="false">
            <field order="1" name="ucm_obj" title="UCM Object"/>
            <field order="2" name="state" title="State"/>
            <field order="3" name="error" title="Error Message"/>
      </section>
</bomformat>
</PROJECT_INTERFACE>
```

Clearcase_fix.pl

```
#$ARGV[0] = Project name
#$ARGV[1] = Selector name
#$ARGV[2] = Env List

use BuildForge::Services;

my $conn = new BuildForge::Services::Connection();
$conn->authUser("root", "password");

my $proj = BuildForge::Services::DBO::Project->findByName($conn, $ARGV[0]);
my $env = BuildForge::Services::DBO::Environment->findByName($conn,
"FixVob");
if (defined $env) {
      $env->delete();
}
$env = new BuildForge::Services::DBO::Environment($conn);
$env->setEnvironmentName("FixVob");
$env->setLevel(6);
$env->create();

#add env entries we need to the environment
#split up arg by commas <env arg>,<env arg>
@pass_entries = split(/,/,$ARGV[2]);
for ($i=0; $i<@pass_entries; $i++) {
      #split the token by ~, <var name>~<var value>
```

```
        @env_args = split(/~/, $pass_entries[$i]);
        $entry = new BuildForge::Services::DBO::EnvironmentEntry($conn);
        $entry->setEntryName($env_args[0]);
        $entry->setEntryValue($env_args[1]);
        $entry->update();
        $env->addEnvironmentEntry($entry);
}
$env->update();

$proj->setEnvironmentId($env->getEnvironmentId());
$proj->update();

BuildForge::Services::DBO::Build->fire($conn, $proj->getProjectId(),
$ARGV[1]);

#put everything back where it belongs

$proj->setEnvironmentId("");
$proj->update();
$env->delete();
```