Examensarbete

Automatic Software Integration

Axel Bengtsson och Ola Olsson

Institutionen för datavetenskap
Naturvetenskapliga fakulteten
Lunds universitet

September 2006

ISSN 1651-6389
LU-DV-EX: 2006-83

Automatic Software Integration

Axel Bengtsson
Ola Olsson

Examensarbete för 20 p, Institutionen för datavetenskap,
Naturvetenskapliga fakulteten, Lunds universitet
Thesis for a diploma in computer science, 20 credit points,
Department of Computer Science,
Faculty of Science, Lund University

Abstract

In large software projects with perhaps hundreds or even thousands of developers there is a need for good Configuration Management, often abbreviated as CM. Its main tasks of CM group or individual consist of organizing and controlling the development of the software.

One way to achieve this is to allow the CM team to have full control over the software source code. All source code that has not been accepted in the new software has to be delivered to the CM team, who update and integrate the software with the new code.

This implies that a developer has to deliver all software changes, to the CM team so it can be integrated into the system. One of the reasons for this is to ensure quality assurance, it is very important to always have fully functional and reliable code.

What if the developer would like some feedback on the new implementation and the CM team is not available at that particular time? Is it perhaps possible to classify deliveries, make simple builds, smoke tests and send the new build to system test automatic?

The purpose of this thesis project is to find a method that allows the CM group to use automatic integration. More builds on more cell phone variants as often as possible would save a lot of time as well as improving quality assurance in the various software projects. This would also facilitate faster feedback than before to the developers and tighten the loop from developers to the integration of the code.

We decided to write a program that fulfills these constraints and in order to make that possible, we had to gather knowledge about Sony Ericsson Mobile Communications (SEMC) systems, processes and tools and how they interact together. After some exposure to manual integration we wrote a program, AutoInt, which works in the same manner as manual integration.

One constraint on how AutoInt works, which was already known from the beginning, is the risk of the input/output from the other programs. If these protocols are changed, then our program will not work as intended.

A positive side effect that was not considered at the outset of the project is that all builds, even failed ones, decrease the build time for future builds by using the same files that are winked in from cache.

Some experiments were also set up to gather data on the effectiveness and efficiency of the tool.

Sammanfattning

I stora mjukvaruprojekt med flera hundra eller kanske till och med tusen utvecklare ställs det stora krav på konfigurationshantering, CM, som i stora drag handlar om att kontrollera utveckling av mjukvara.

Ett steg för att öka kontrollen av mjukvara är att låta ett CM-team ha full kontroll över integration och uppdatering och av den kod som kommer från utvecklare.

Detta betyder i sin tur att alla utvecklare måste skicka mjukvaruleveranser till CM-teamet för att få dem integrerade.
Är det i denna situation möjligt för en utvecklare att få uppgifter om integration även om CM-gruppen inte finns på plats när leveransen skickas?
Är det möjligt att automatiskt klassificera en leverans som enkel eller svår, göra enkla byggen, testa koden och skicka den nya mjukvaran till systemtest?

Vi bestämde oss för att skriva ett program, AutoInt, som uppfyller kraven. Detta krävde i sin tur att vi studerade Sony Ericsson Mobile Communications, SEMCs system, lärde oss deras program samt hur de fungerar tillsammans.

Vi har lyckats att klassificera leveranser och tyvärr hittade vi lite negativa saker med vårt program. Det är mycket svårt att flexibelt tolka in- och utdata från program och samtidigt vara beredd på uppdateringar från dessa program som används. Om protokollen till programmen ändras, är risken mycket stor att programmet inte fungerar.

En positiv bieffekt med programmet som inte var känd från början var att bygget cachar filer som sedan kan användas vid senare bygge. Detta sker även om bygget misslyckas vilket snabbar upp nästa bygge avsevärt.

Detta examensarbete presenterar en algoritm och en implementering av ett program som integrerar leveranser automatiskt.

Acknowledgement
We would like to thank Gillis Johnsson at SEMC for helping us through the project and answered tricky questions. The entire CM team at SEMC who were also a great help whenever we needed them. We would also like to thank Björn Berndin, Orvar, Andreas Göransson and David Loughnane for reading and commenting on our report.

Last but not least our families, and girlfriends for their support through all the years!

Prerequisites
This paper is mainly written for Sony Ericsson Mobile Communications Configuration Management team members, and for those who have taken a basic course in Configuration Management, CM.

Table of Contents

1 Introduction

As software development gets larger and more complex more changes will eventually be made, such changes can be in the form bug fixes, patches, new features and other deliveries. These changes imply more work for almost everybody in the company who is developing the software.

The software production cycle from developer to product is a complex task. A developer usually works in a group that takes care of certain functionality in the software. In large projects developers often work concurrently which means that when a developer wants to test newly written code, it has to be tested against an older stable version of the same software. It would be impossible for a developer to test the code if he/she had to tell all other developers to send over their latest version of the code. Imagine all developers working this way.

The older stable version of the code described above is called a baseline and this is a state where the software is stable enough to test new code against. To be able to use the baseline through a whole project, it is very important to update the baseline regularly.

When a developer has finished the newly written code, is satisfied with all module tests, then the integration step can be made. An integration consists of replacing the old code with the new, compiling the new software, and testing it. If the software passes the required tests the code can be checked in and a new baseline made.

The integration step is carried out by the Configuration Management team whose main task is to control and organize the evolution of the software. This includes version handling, reproducibility of a specific version in the system and releasing official software.

When the integration step is finished, system test takes place. System test consist of manual and automatic testing of the system.

We think that every process that is done repeatedly and in the same manner should be automated.

This statement is very important from two view points. Primarily, every repeated task will eventually be error prone and very repetitive. Secondly, a repeated task continually carried out in the same manner is possible to automate. If such task are automated, then a computer will almost certainly do it error free and quicker than a human being.

In the current integration process, a developer may send a delivery and if the CM team is unavailable, then he/she probably may have to wait until the next day to get a build result even if the delivery only contained an added comment in the source code.

However there are different solutions to this problem, though not all are viable solutions. One solution is to employ more resources and schedule them to work with nightly builds.

Another solution is to devise a time schedule when the developers are allowed to work but this would probably not gain anything. A third solution is to allow the developers to integrate the deliveries themselves. This would be an optimal solution to the problem but it only works in small projects. There are a number of reasons for this. The first is that quality assurance gets more and more complex as projects grow bigger. The second reason is that in small projects all people have some kind of overview within the project but as projects grow larger and larger, the need for specialists and CM knowledge grows. Often, large projects need both Configuration Managers and Quality Assurance Managers which mean that complex projects need dedicated people in charge of the integration.

We think that the loop from developers to the CM should be as tight as possibly.

This second statement is also very important for the entire software process. The developers receive faster feedback which reduces the frustration waiting for answer from the result of the integration. The baseline is updated faster and all test and new development can start earlier. The effect of this is that there is better synchronization with developers always having access to the latest baseline. The need for a complicated merge between files decreases.

A solution to this may be automatic software integration. When a developer makes a delivery, the delivery should automatically be identified, built, tested and integrated into the new software as if this was done manually.

SEMC have got their own delivery system, where developers send their deliveries through a web service. When delivering to this service a strict protocol must be followed. The delivery is placed in a database which the CM team has full control of. Unfortunately, they lack an automated integration process of deliveries.

The integration part is often a repeated task. The problem is that this new system has to be built and tested often, even on simple builds and different cell phone variants and this may take a very long time. A solution to the long build time is to always upgrade the build servers used, but the side effect is as usual, money loss.

Is it possible to make easy integrations automatically? Is it possible to parse the build logs for errors, report it, and automatically reject a delivery if it happens?

This thesis project describes the backgrounds of SEMC CM, existing software that solves an analogous problem and an implementation of a new program that indeed solves these problems. The intention is that this solves the problem with build servers operating at night when the CM team is unavailable. This would be an improvement because at this moment, SEMC have got many variants to build with each variant taking a lot of time to build. The build servers are not used 24 hours a day, such usage would have a number of benefits as we will show in this report.

1.1 *SEMC background*

This walkthrough of SEMC software process illustrate the source code life cycle, from developer to cell phone. After that, an example of a simple integration will be given and finally the programs used by the CM team at SEMC will be discussed.

This will serve to increase the reader's basic understanding of the company and their view of CM.

1.2 *Source code lifecycle*

The picture below describes a simplified picture of the source code lifecycle at SEMC.

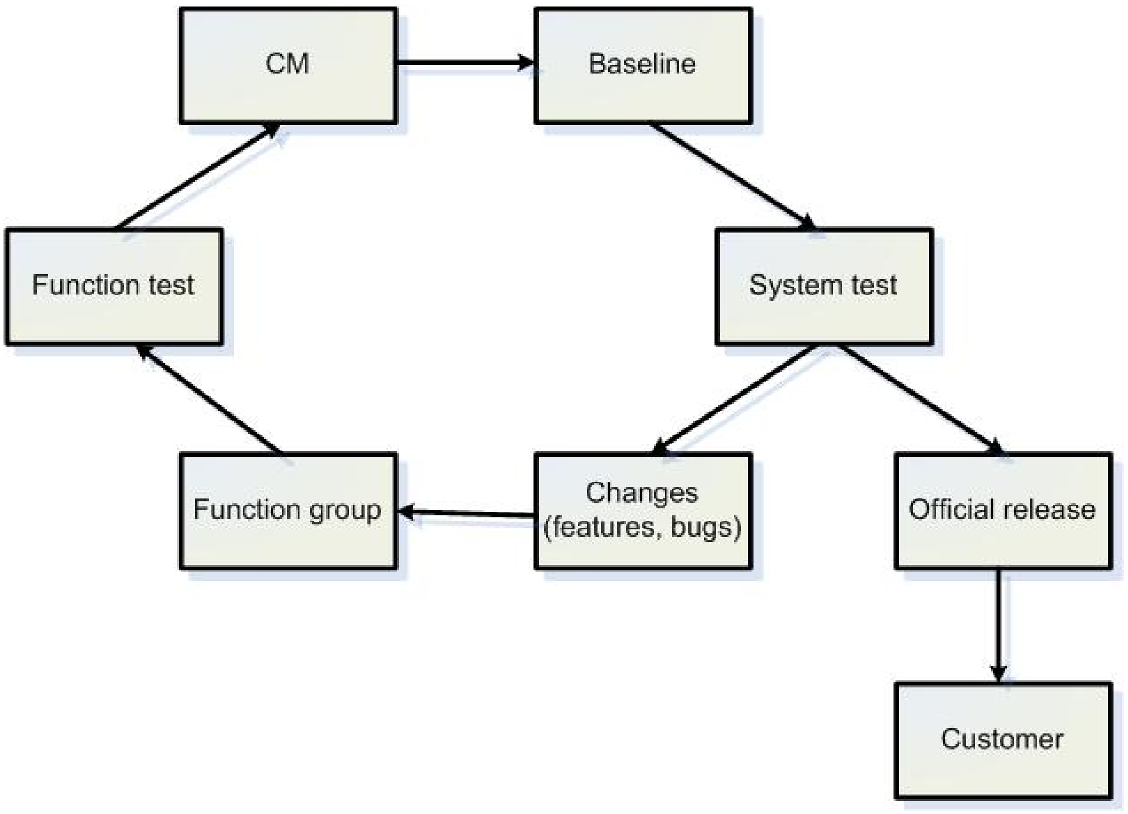


Figure 1.1, *The software developer life cycle*

In SEMC, a group of developers is called a function group and all function groups are responsible for their own piece of the code, i.e. software application area.

When developers have made a change they freeze the new code to a module freeze label and they send it to the function test group. The freeze label describes a particular code at the time when it was frozen. The function test group will then test certain functions made in this freeze label.

If all tests are successfully completed the Function group will send the same module freeze label to CM. CM will build the new software that includes this freeze label. If the software builds without error and the phone passes a basic sanity check the changes will be checked in.

System test is continuously carried out on the baseline. This test is very extensive and covers almost all functions in the phone. The system test is not only a bug finder, it is also very good way to establish quality assurance.

Official releases are continually made and tested throughout the project. Releases to the Customer are made at agreed time-lines.

1.3 *Integration*

This chapter describes the integration steps carried out by CM. A typical integration begins when a Function group makes a delivery and ends when the delivery is integrated into the baseline.

1.3.1 An Integration example

A new code delivery is delivered to CM via a web interface. The module freeze labels are stored in a database. A CM team is assigned to integrate each delivery.

A global configuration file containing all phone variants and versions of all code in the cell phones is updated with the information from the delivery. Before the CM team accepts the delivery, the entire project must be built and smoke tested.

If the build or smoke test fails the delivery gets rejected. This means that the function group must fix their code and then send a new delivery via the web interface.

As a final step, the CM has to check in the configuration file and set the delivery as integrated in the database.

Why is this complex process needed for an integration? A simple answer is quality assurance. The baseline is a state where the code has to be very stable, no serious bugs are allowed to exist here.

Figure 1.2 shows the integrations process in steps, the tools used will be described below.

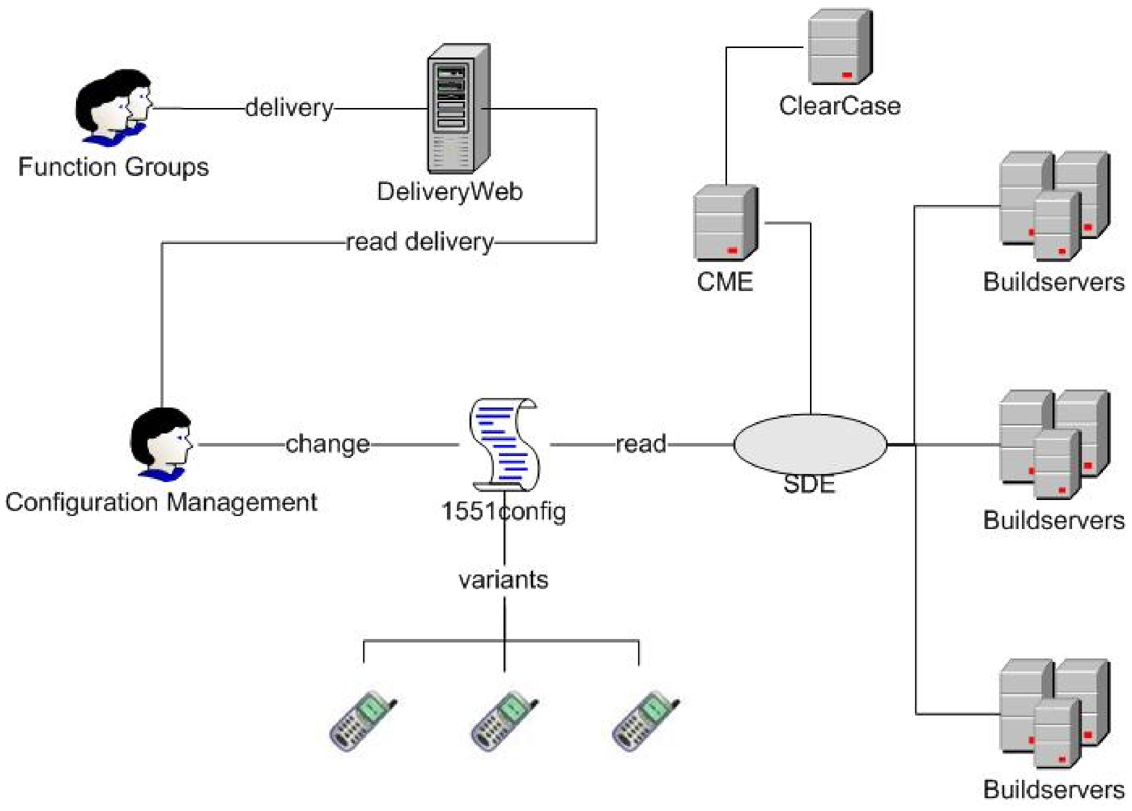


Figure 1.2, *Picture describing the programs used by the CM team.*

1.4 *Programs*

The following section describes the tools used by CM on a daily basis as well as their connection to each other.

1.4.1 Delivery Web

The Delivery Web is the gateway between the Function group and the CM team. The Function groups have to follow a strict protocol when they use Delivery Web to send their new applications or corrections to CM.

When a new delivery is made the CM team receives a notification via mail. The details of all deliveries can be seen on a web site that is connected to the database. CM can change the status of each delivery when the need arises. For example, the status may be changed from new to assigned, integrated or released.

1.4.2 Project configuration file

The project configuration file contains the module versions and the software variants. Instead of hard coding the different variants of cell phones into the code they are described as variants in this file.

In SEMC, the software is divided into modules. Each module has a specific task, which can be a phonebook, a protocol, a game etc. and each module in turn has its own project configuration file.

1.4.3 SDE

SDE is a SEMC in-house build tool. As input, it takes the project configuration file and then generates a make file depending on the variant that will be built.

The build servers used by SDE are chosen as in the Round Robin algorithm [1], which means that if all build servers are in use, the next delivery to be built will use the first build server in the list.

1.4.4 ClearCase

ClearCase [2] is a common version handling program. A version handling program stores the history of each file that is placed under version control. It is possible to compare different revisions of the same file to track changes that have been made in the code during the software developing cycle. This facilitates reproducibility and backup handling.

To alter a file in ClearCase, a checkout is made. When a file is checked out from ClearCase it becomes writable. When changes have been made, in accordance with a delivery, the file can be check in. The file becomes read-only and ClearCase makes a new revision of the file. It is possible to undo a checkout if for some reason the person working with the file does not want to check it in.

When working in ClearCase a view must be create and the correct settings applied. The settings needed are the CM module, rule on the checked out file, the category of the branch and the branch name.

1.4.5 CME

CME is an in-house developed software used as an interface to ClearCase. It is used by all developers and CM´s and allows for a clear and common interface to ClearCase.

1.4.6 BRAT

BRAT is an automated testing environment for testing mobile phones. The input to the program is the official build label made in SDE as well as a test description. The tests only check the basic functionality of the phones.

2 Analysis

In this chapter, all the problems and sub problems encountered are described. Other programs designed to solve similar problems are also discussed. It is intended to familiarise the reader with the problems that we encountered throughout the project.

2.1 *Problem description*

In all big software projects there is always a need for version controlled source code. If there is more than one developer there is a need to protect the code from all the other developers. The reason for the protection is described in Babich three classical problems; Double maintenance, shared data and simultaneous update [3].

To avoid these problems, most companies employ a CM team and purchase a version handling program that protects the baseline code from the developers to ensure that the baseline always is stable and that all code reaches a high quality.

Figure 2.1 shows that CM works as a layer between the baseline and the program. The programmer checks out from the baseline and works on module A. Before committing the changes, the developer updates the other modules in his/her private view and tests the new configuration. The developer is now able to commit the changes to the CM team. When the CM team is satisfied with the code, they commit the changes to the new baseline.

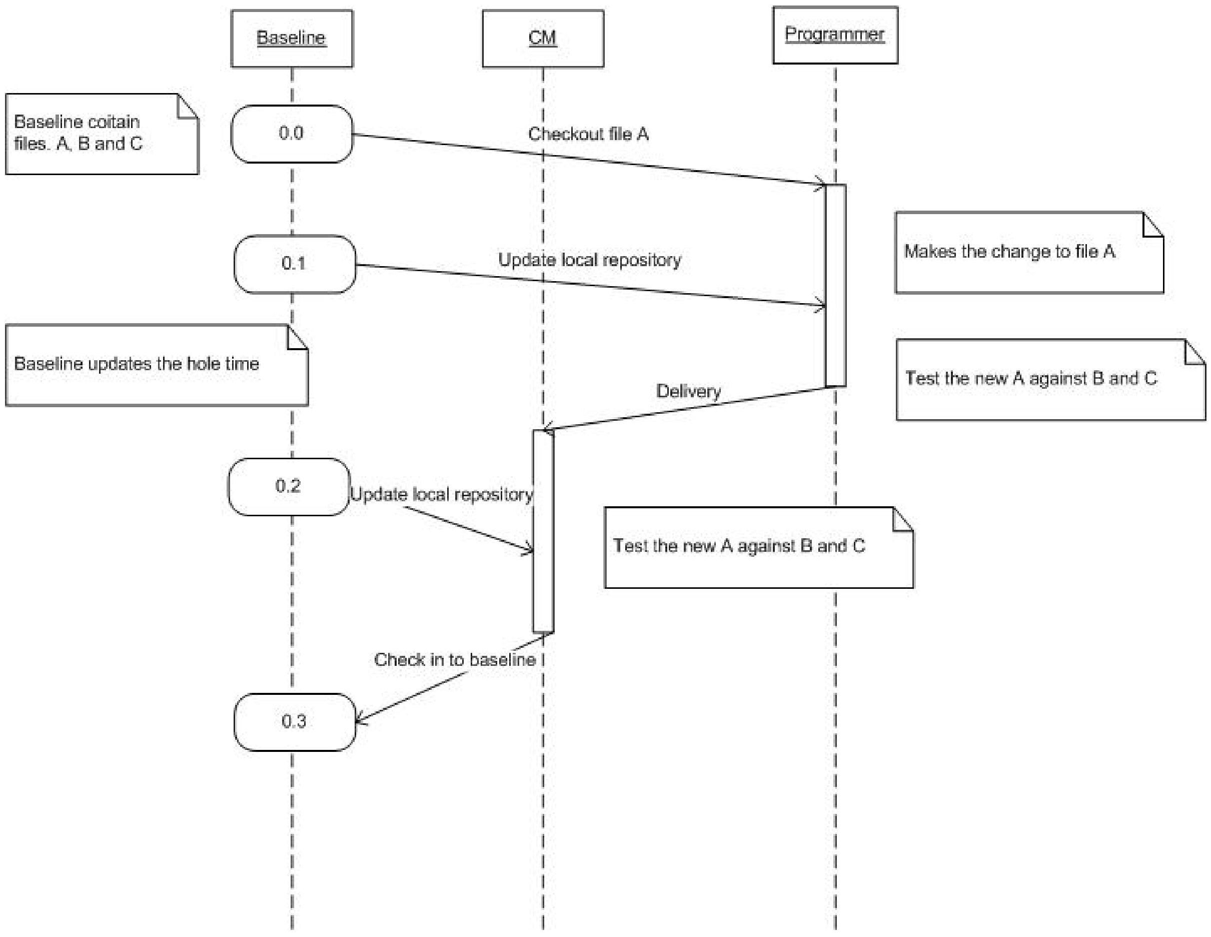

Baseline
CM
Programmer
Baseline coitain files. A, B and C
0.0
Checkout file A
Makes the change to file A
0.1
Update local repository
Baseline updates the hole time
Delivery
Test the new A against B and C
0.2
Update local repository
Test the new A against B and C
Check in to baseline
0.3

Figure 2.1, *Picture describing the way CM works*

2.1.1 Introduction

In a global company such SEMC there are developers working in different geographical locations throughout the world. This means that deliveries may be sent 24 hours per day.

When the CM team arrives at work, they often sort what can be termed ‘simple’ deliveries and often all these deliveries are built together. If, for some reason the build fails, then they have to find out which delivery caused the error.

The ‘complex’ deliveries, on the other hand, are often built one at a time.

A few problems arise here. How do we classify deliveries as ‘simple’ or ‘complex’ and how many deliveries can be built at one time to ensure that we have a successful build most of the times?

Many deliveries handled by the CM team are integrated in the same manner all the time. Is it possible to automatically integrate these deliveries once they are delivered, and if so, how many of the total deliveries can be integrated automatically?

Automation would result in many positive effects. One of them would be to fulfil a documented CM strategy, continuous integration [4]. Continuous integration encourages developers to check in code as often as possible. It would then be possible to build more releases, get more stable code and get faster feedback from testers and users. Automation would have the same effect as continuous integration even though it is not possible for the developers themselves to build and integrate. The source code lifecycle will be faster, the integration will be more secure and it would be more economical. This should enable the CM team to be more concentrated on the more difficult deliveries.

This thesis project is about developing a program to handle these tasks. The program should decide whether or not it is able to make an automatic integration, an automatic test and then create a report describing the integration. As outlined above SEMC would gain much from this.

2.1.2 Analysis on SEMC

Sometimes there are priority deliveries that must be included in an official release. If this delivery fails compile or link then CM group must wait for a new delivery from Function group who sent the failed delivery. The CM must then integrate new delivery. This may mean that the CM works late nights.

The difficulty when writing the script is that we have to control the surrounding programs. Many of the programs used are in-house developed programs. They are updated continuously, and therefore a change to an interface may occur. An interface change would probably mean that incorrect parsing of the output given to our program may occur. This may lead the program to make a decision that is incorrect.

Another potential problem area concern the delivery made by the developers through Delivery Web. The interface is a mix of checkboxes and textboxes. Before the developer sends the delivery, the program validates the delivery. However, it is possible to insert incorrect information in the textboxes which may lead to an error interpretation when parsing the text. At the time of writing this report there are some proposals at the CCB [5] that involve changes to the protocol.

The deliveries are saved in a key/value matrix in a database. To decide if it is possible to integrate a delivery, the combination of keys and values that exist in the delivery is carried out.

2.1.3 Summary of the problems

Is it possible to carry out an automatic software integration at SEMC? Is it possible to classify the types of deliveries made? What are the positive side effects of the implementation and what kind of demands do we have on the surrounding programs?

2.2 *Use cases*

Four different types of deliveries and the typical solution by the CM team to these will be described in this section.

2.2.1 Freeze label update

The easiest and most common integration is to update a freeze label in the project configuration file. This means that the developers have made a new update of their code and a new module freeze label to corresponding that change. A freeze is a specific code at a specific time. An simple example may look like this

Before	
A_feature.c	R1A001
After	
A_feature.c	R1A002

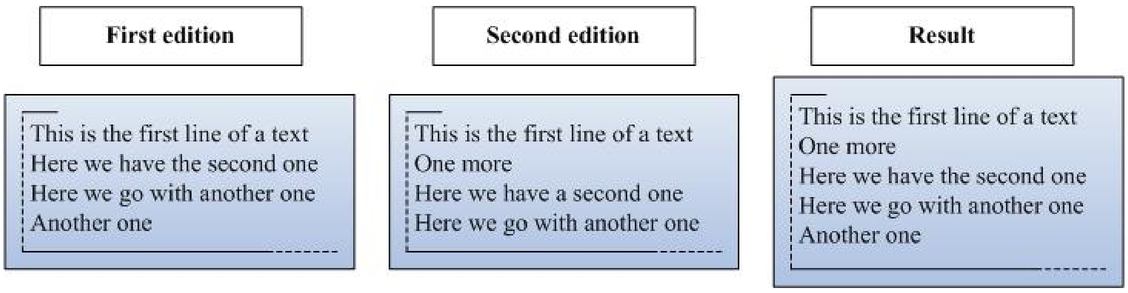
The file "A_feature.c" has been updated from revision R1A001 to R1A002 where the new code resides.

2.2.2 Dependency

A dependency describes a certain need, a delivery may need another delivery to be able to be integrated and work in the cell phone.
Suppose a module B is dependent on a module A, and therefore, the module A has to be integrated before B can be integrated. In this case B is dependent on A.
The CM team handles or waits until the module A has arrived. After that, A is integrated and after that B can be integrated as if the dependency did not exist.

2.2.3 Merge

A file merge is a try to unite two or more files into one. This can be explained easily by the following example.



First edition:

```
This is the first line of a text
Here we have the second one
Here we go with another one
Another one
```

Second edition:

```
This is the first line of a text
One more
Here we have a second one
Here we go with another one
```

Result:

```
This is the first line of a text
One more
Here we have the second one
Here we go with another one
Another one
```

Figure 2.2, *Easy merge*

Figure 2.2 describes a merge of two files which contain the same information except for an additional row in each. The result contains the same information as both of them, plus the two lines that differed.
The merge is not always expected to succeed, because conflicts may occur. A conflict may look like the example in figure 2.3.

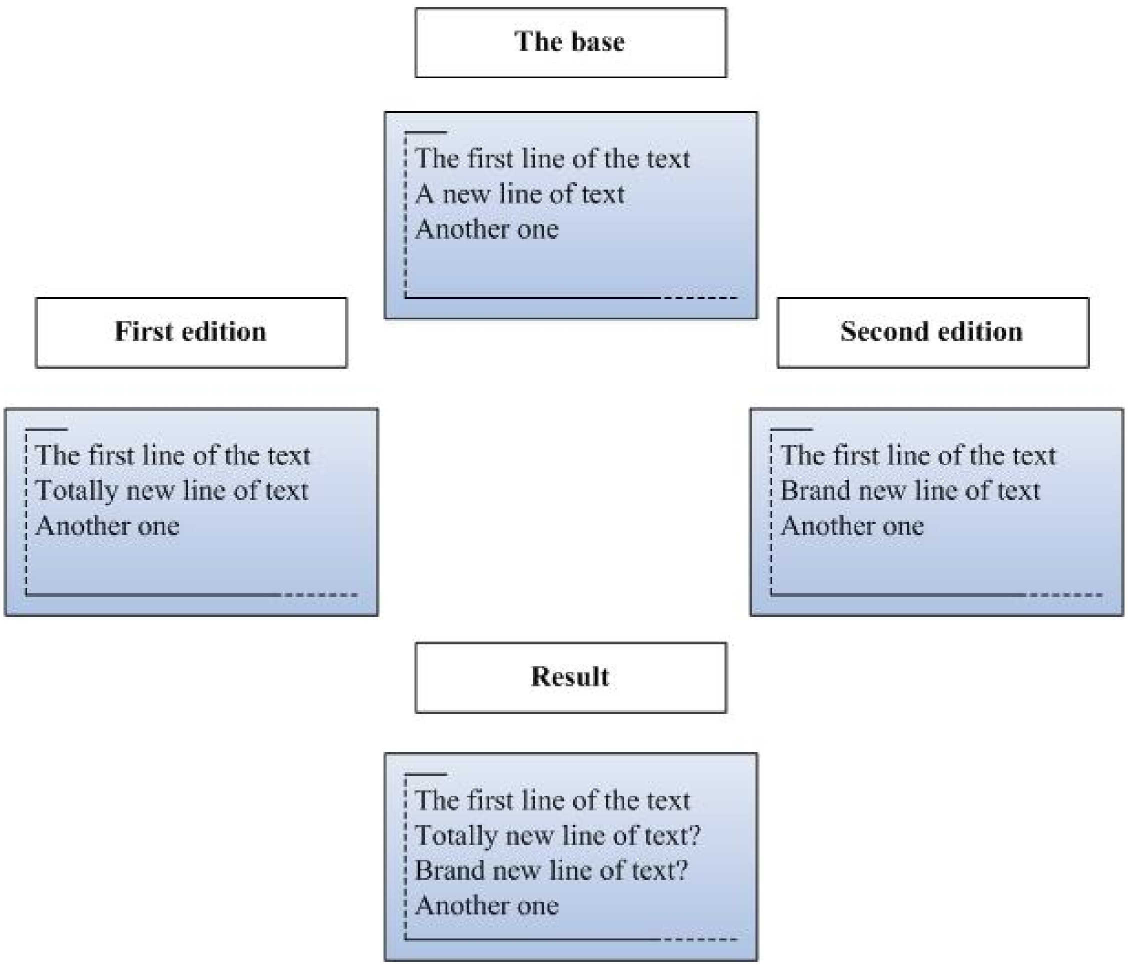


Figure 2.3, *Merge containing a conflict*

If two files originate from the same file and both of them change the same line in their own copy, the merging program may only guess which line is the right one. In this example, it is impossible to know whether the second line should be “Totally new line of text” or “Brand new line of text”.

When a delivery contains a merge, this is because a developer has changed something in the project configuration file. A Function group has branched out from the project configuration file and made some changes. In the mean time, the CM team has also updated the file, and when the function group wants to check in their version, a merge has to take place. The reason for making changes in the project configuration file and not to do this in the revisions (and make a freeze label update) is that files may have been added to the project and these files have to be in the project configuration file.

Figure 2.4 shows a branching of the project configuration file. After this, the developer makes some changes in his/her working copy and in the meantime, the main line also changes. When a merge has to take place, this could result in merging conflicts as described above.

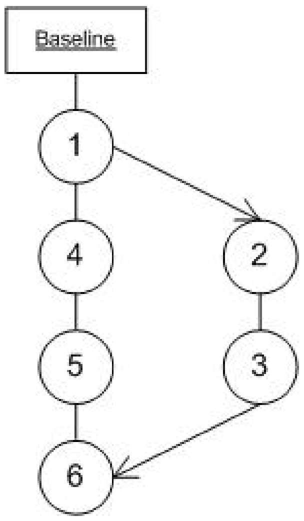


Figure 2.4, *A developer makes a branch (2) from the baseline (1) and makes some changes (3). In the mean time the CM team also makes their changes (4,5). The merge may be very hard and may result in a conflict (6).*

2.2.4 Hijacking

Whenever a source file in a project contains errors or the delivered file generates a problem for the CM team and they need a fast fix, they hijack the file which causes the problem. This means that the CM team makes their own changes in the particular file, just to get the software to compile and link.

However, it is not recommended to hijack a file, because it is very easy to introduce old bugs into new code and this reduces the code traceability.

2.3 *Related work*

All software projects have some problems related to the integration process and what solutions exist to take care of integration problems? Large open source projects such as Mozilla [6] and Apache [7] have the same type of problems as SEMC. The projects are also divided into modules, where each module has got a responsible person who has write access to the repository. Unfortunately automatic software integration does not exist in these projects.

A project called CruiseControl [8] solves the problem of building whenever the repository is updated. The program is possible to build after triggers and then it is possible to perform automated tests on the build result. This would be very interesting if we were not forced to write the integration part ourselves. However, the program inspired our build cycle and reporting system.

2.4 *Analysis of use cases*

This chapter analyzes the use cases described earlier.

2.4.1 Analysis of freeze label update

The work of the freeze label update is to find the particular module in the project configuration file and update the label.

2.4.2 Analysis of Merge

When the program detects a merge, it tries to merge the two project configuration files given. A merge can be very complex and sometimes very difficult to automate. Our program uses ClearCase to handle the merges which have no conflicts. If a conflict is detected, it stops automatically.

2.4.3 Analysis of Dependency

Practically, non cyclic dependencies are quite easy to handle. The program checks if the modules that the delivery is dependent on exist, and if they do, the delivery is classified as if the dependency does not exist. If the modules which the delivery is dependent on do not exist, the delivery is classified as "not possible".
Often, the dependencies are quite easy to handle but sometimes it is possible to find cyclic dependencies and the program is not capable of handling. There are also problems if we do not manage to integrate a dependency that awaits another label. The problem arises if new version of this label is not delivered. The new version gets integrated and we can never find that exact label we had a dependency to.

2.4.4 Analysis of Hijacking

It is not hard to solve the problem of removing a hijacked file from the project configuration file. A regular expression [9] is used to find the line where the module freeze label name resides, and then that line is removed.

2.5 *Summary requirements*

An automatic software integration program can be divided into three areas.

- The first is how to classify different integrations as simple or complex.

- The second problem is to develop a program to deal with simple integrations. Is it possible to control the surrounding programs as wanted? What demands do we have on them and what happens if the protocol for the program changes?

- The third question deals with the side effects of the program. How much time will a program eventually save for the CM team? Are there any negative side effects?

3 Design

This chapter will focus on the design for the program. Certain tradeoffs and implementation details will also be discussed.

3.1 *Background*

Automatic integration at SEMC is one of the ways to get closer to Continuous Integration, a documented CM strategy [3]. Continuous Integration is about regulary making small changes to a system, integrating them, building the new system and then releasing the new software. If developers are committing, integrating and testing their code every day, the deferred integration should be a historical process.
This implies that a relatively new software release always exists, and that the code has passed a basic sanity check. As a developer, it is always good to have the code fresh in the memory, as it will make it easier to find and correct bugs. It also means that the new features can be released faster and in turn this allowed for faster reports from customers about the new software.

There are however a couple of solutions to deal with the implementation of this problem. More staff can be hired to always have people present at the CM department. However, would probably result in too high costs.

Another solution is to get a time schedule when the developers are allowed to work but this would probably annoy developers. The third, and probably the least feasible solution would be to let the developers integrate their own deliveries. This would not achieve the desired quality in the project.

If it was possible to automate some builds and tests of the deliveries, SEMC could gain very much. Such benefit could be in terms of resources (staff + equipment) as well as less time in the development cycle.

The program developed in this thesis project will help to shorten the time taken in development cycle. It will also benefit the CM people, who can focus their attention on more meaningful integrations. The automatic integrations will also be less error prone, because redoing a simple task in the same manner will inevitably contain some human error.

3.1.1 Choice of programming language

The choice of programming language was not really a problem. Perl, Practical Extraction and Reporting Language [9] was chosen because of the nice text manipulation features.

Since we knew that we had to use a lot of regular expression, input/output, file access and http requests we both agreed on Perl at the project outset.

3.2 *Architecture*

The algorithm described is specific to SEMC but could easily be tailored into to a general algorithm.

When the program is started, it reads a special configuration file made for user specific information such as branch and view. Delivery Web is then used to assign all the deliveries found to the program, a ClearCase view is created and each delivery found is categorized. The project configuration file is checked out and altered with the information contained in the delivery. The new software configuration is built for different variants and if the builds are ok, the new product configuration file is checked in and a report is sent to the addresses outlined in the configuration file.

The algorithm is used to:

- Validate the config file
- Read the config file for dynamic variables
- Search through Delivery web for new deliveries and put them into a list
- Create a view
- Assign all deliveries in Delivery web
- For all deliveries:
    - Check out the product configuration file
    - Classify the delivery and decide what actions should be taken
    - Make all the necessary changes in the project configuration file
    - Build all selected variants.
    - If the builds are ok, then check in the product configuration file, if the build fails perform an undo checkout.
    - Make an appropriate label and set the status to New or Integrated in Delivery web.
- Create a HTML report.

3.3 *Classification of deliveries*

The deliveries seen on Delivery Web reside on a server. They are described with keys and values where each key correspond to a particular question in Delivery Web and all the values correspond to an answer to the questions.

To classify a certain delivery, the keys have to contain a certain value.
This means that we have to make constraints on these keys and values in order to classify them.

The program is able to handle four different kinds of situations, namely freeze label update, hijacking, dependency and a trivial merge.
The program will take different actions depending on which of the four situations arise in the delivery. However, it is possible that more than one action can be taken on a delivery. In such case these actions will be treated separately.

As already written, to classify the deliveries as to what action to take, the keys have to contain certain values and the classifications of the actions are written in XML.

The XML classification uses tags named as logical gates, “and”, “or” and “not” are allowed.
To handle a key, just add the logical gate, a value that the key has to have, and the desired action to make if the delivery satisfies the classification.

```
<DEMANDS>
        <AND>
                        KEY1 = VALUE1
                        KEY2 = VALUE2
                        FILE_ATTACHED=OFF
        </AND>
        <OR>
                        KEY3=VALUE3
        </OR>
</DEMANDS>
<ACTIONS>Freeze_update,Hijacking</ACTIONS>
```

The XML example above shows a simplified “classification file”. For every delivery using this classification XML file, the actions “Freeze_update” and “Hijacking” will be used if the keys of the delivery satisfy:
KEY1 has to have value VALUE1, KEY2 has to have value VALUE2 and the key FILE_ATTACHED has to have value OFF. These constraints have to be fulfilled by the delivery or if the key KEY3 has the value VALUE3, then the actions will take place.

The actions that are permitted in the XML-file are Freeze_update, hijacking, dependency and merge.

There is a very good reason why these classification schemes are not hard coded. If Delivery Web changes some of the key names, or adds some functionality, our program does not need to change. All that needs to be done is to make the proper changes in the classification schemes.

The reason for choosing XML, is to use a language that is relatively easy to both understand and parse. This could not be done in a usual text file because of the logical AND, OR and NOT statements.

3.3.1 Freeze label update

Practically, to solve the freeze label update, a regular expression [9] is made on the file name and it replaces the old freeze label with the new one. This may however be a little awkward, because it is very easy to 'trick' the program. This is discussed in more depth in section 5.1.1.

The freeze label update is the most common delivery and maybe the easiest to implement. We implement the update with a regular expression that parses the project configuration file seeking the module name.
If the module name is found, the freeze label is updated but if the name is not found the delivery is classified as "not possible".

3.3.2 Dependency checking

The dependency in a delivery is described as a list of module freeze labels. To integrate the deliveries, all of these module freeze labels have to exist in the project configuration file.
The project configuration file is parsed with a regular expression searching for the freeze labels. If all of them exist, the delivery gets integrated as if the dependency did not exist, otherwise the delivery will not be integrated.

To show our approach to the dependencies, three different cases are described in figure 3.1.

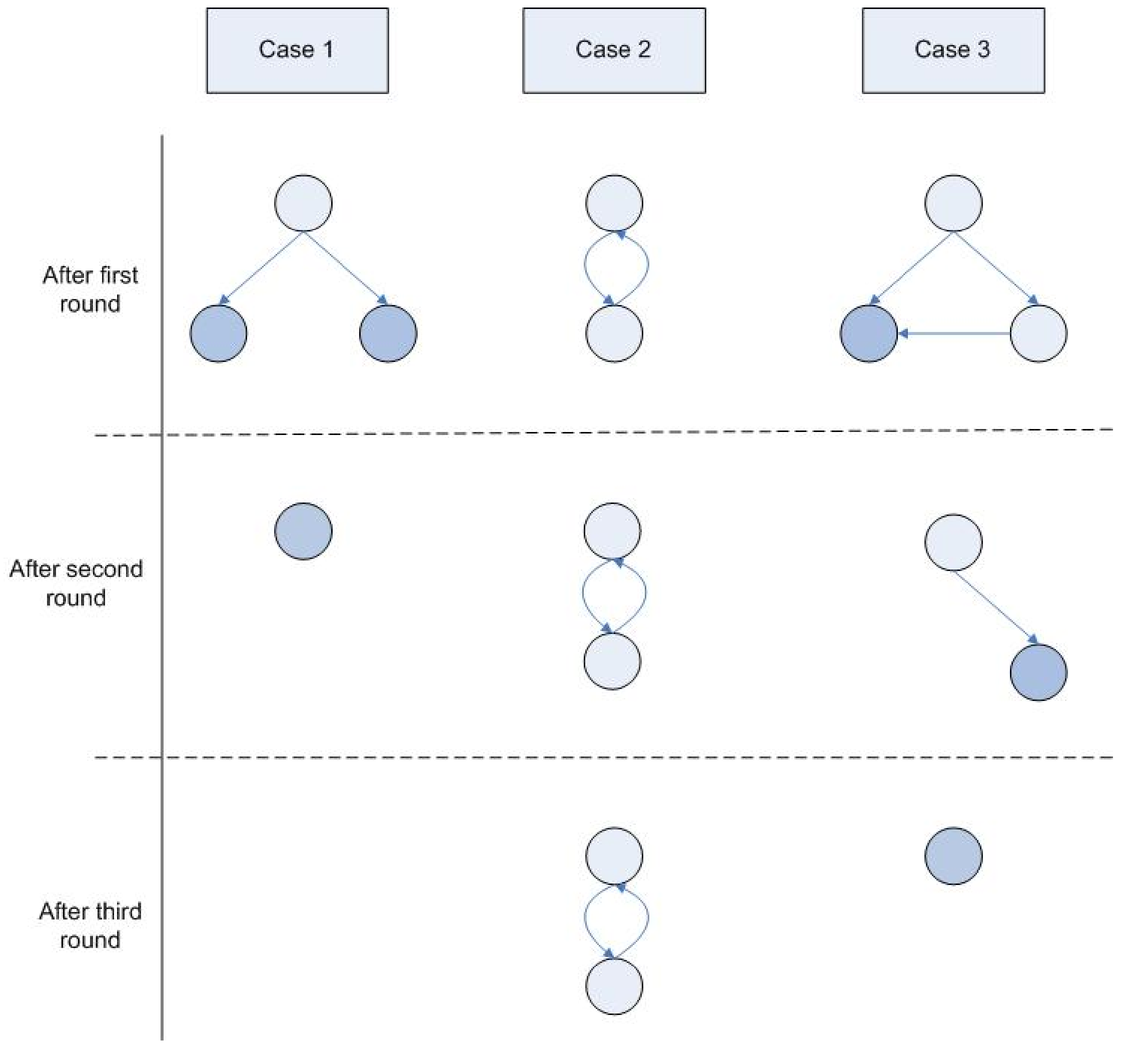


Figure 3.1, *The figure shows three cases of dependency after different runs of AutoInt.*

Case 1 describes a delivery that has dependencies to two other deliveries. The two deliveries will eventually be integrated by the program because they have no dependency to other deliveries. When they are integrated, the first one can also be integrated.

Case 2 describes a cyclic dependency between two deliveries and therefore, this kind of integration has to be manually integrated.

Case 3 illustrates an integration where two deliveries are dependent on one delivery. This is solved in the same manner as Case 1.
In the first round, the delivery which does not have any dependencies to the other modules will be integrated. In the second round, the next delivery is integrated and in the third round, only one simple delivery remains.

3.3.3 Merging

We implemented our merging through ClearCase. The merge program searches for files that are not identical and then it tries to merge them automatically. In case of a merge conflict, the merge stops immediately.

3.3.4 Hijacking

Hijacking is described in the project configuration file as an override of another file. The project configuration file is only concerned with the latest declaration of a file. Therefore, every hijacked file is described after the original file.
To take away a hijacked file, the latest version of the file is commented. This is done with a simple regular expression.

3.3.5 Other deliveries

The remaining deliveries are not possible to solve in the scope of this thesis. Some different examples of deliveries are:

- Deliveries that contain files to the phone file system.
- Platform deliveries.
- Deliveries that are described in natural language include non trivial parsing.

What all of these deliveries have in common is that most of them are special cases. In these special cases, changes must be made in many files and there is no static way to handle these deliveries. That is why they are difficult to script.

3.4 *Implementation*

When the implementation part began, a branch was created for two reasons. The first reason was to test the code without endangering code on the live integration branch. The other reason was to compare the result with the main branch.
The implementation was also tested in parallel with the live integrations.

3.4.1 Algorithm

The flow chart diagram in figure 3.2 shows the algorithm in AutoInt.

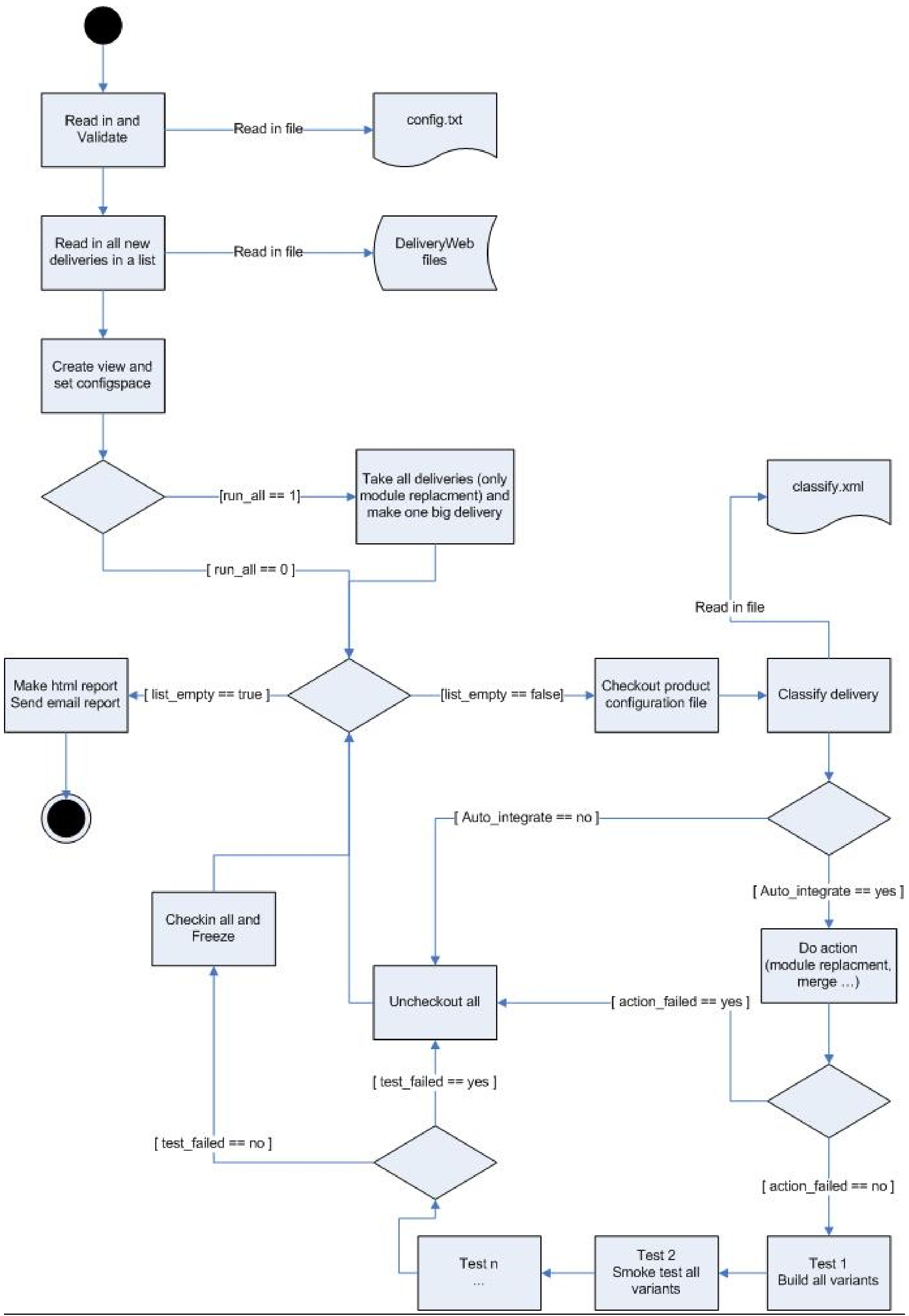


Figure 3.2, *Flow chart diagram of the algorithm.*

The program validates the config file and then checks DeliveryWeb for new deliveries.

If a special variable “run_all” is turned “on” in the configuration file, the program merges all deliveries found into one delivery. This is only done if all the deliveries contain the action “freeze module replacement”. The files are merged together according to the modules that are going to be replaced.
The reason to not use merge in “run_all” is when AutoInt gets two deliveries that should be merged. If the first merge succeeds and the second one fails, the version handling program has no idea which files are supposed to be checked in, and which files are supposed to be unchecked out.

The delivery is then classified by a parser which decides if AutoInt is able to integrate it or not.

If the classification succeeds, AutoInt tries to integrate the delivery. The integration may fail for a number of reasons, for example; a module label does not exist, the build does not succeed etc.

AutoInt is dependent on other programs and these programs input and output have to make reproducible results.
For a given input, the same output has to be given, even if the experiment is made hundreds or thousands of times. Otherwise it is impossible to interpret the results given.

The programs used by our program, must also be able to handle commands through the 32-bit windows shell cmd.exe, This is the easiest way to execute other programs from within Perl.

3.5 *Special modifications*

Under the implementation phase we discovered that there was a need for special modifications of the program.

3.5.1 Exclude list

An exclude list contains names of function groups that always deliver complex deliveries. The program ignores all deliveries made from this particular group and lets the CM group take care of these manually.

3.5.2 Second chance

The second chance option is good to have if the build servers fail for some reason. Such reasons may be failure in copying files, power shutdown or other causes, other than a problem with the new software.
A second chance variable states that if the total percent of failed build are less than a certain percent, the build that failed will be rebuilt.

3.5.3 Max deliveries

The max deliveries variable restricts the number of deliveries that the user wants to build. This can be useful when a person wants to integrate a group of deliveries together but wants to limit the number of deliveries in the group.

3.5.4 Time restrictions

A time restriction variable is good to set if the user wants to run the program within certain time period. This can be useful if the user needs to attend meeting or leave the job for the day.

3.5.5 Delivery name and status

There is a way to get the program to not only search for the new deliveries. Another way to use this tool is that a CM person sorts out the deliveries that should be automatically integrated. The CM person changes the status of the delivery to a special status such as “to_auto_int”. The program can then be prompted to search after “to_auto_int” status instead of “new”. This is a way to semi auto integrate. The CM classifies the deliveries and the program integrates and tests them.

3.5.6 Configuration of the program

The configuration of the program is made in a configuration file where each line consists of a key and a value separated by an equal sign, “=”. When the program starts, this file is put into a hash. The reason not to use XML or other solutions here is that the existing solution will achieve an simple way to put the configuration in a Perl hash.

The configuration file contains, keys described as:

- The name of the current view
- The name of the working branch
- Maximum number of deliveries to integrate
- If, for some reason you want to integrate all deliveries in one build
- A deadline time where the program has to be finished
- Which build servers to use
- Which Function group to exclude from automatic integration
- Which persons to send email to when the report is made

3.5.7 Configuration file validator

The configuration file validator is a syntax program made by us to correct spelling mistakes in AutoInt´s configuration file. Branches, build servers, variants and other information is checked against the version handling system, in order to correct any misspelled words.

3.5.8 Report

A report is sent out to the mail recipients whenever the program has finished integrating all the deliveries. The report contains all variables set up in the configuration file, the time each delivery was built, build logs and error reporting amongst others. The purpose is to distribute all the information about the integration.

4 Experimental evaluation

To get a good understanding of how AutoInt works, some statistics were taken. We made some experiments to get bug feedback and to collect data. After the experiments we realized that the program is very flexible and can be used to fetch many kinds of statistics that may be needed by the user.

4.1 *Proof of concept*

To interpret the reliability of the information from other experiments, AutoInt has to work as desired.

4.1.1 Premises

We created a branch from the main integration branch and let our program integrate deliveries that were integrated later on the main branch. After that, we compared our project configuration file against the main branches project configuration file. Various types of integration were carried out.

4.1.2 Result

All deliveries achieved the desired result, the project configuration files were identical after all builds. The CM group at SEMC used our branch to merge our project configuration file to the main branch.

4.1.3 Discussion

Our results show that automatic software integration is possible even in a complex environment with strict process such as SEMC. On the deliveries checked, AutoInt integrated them all correctly. It would be impossible to compare all deliveries with the manual integration.

4.1.4 Conclusion

AutoInt works as it was designed to. Users can trust the program.

4.2 *Which deliveries is the program able to build?*

What kind of deliveries is the program able to handle throughout a project? Is it possible to make statistics over a whole project to see in which phase of the project AutoInt is able to handle most deliveries? This experiment could be useful in order to decide if and when in a project the algorithm can be used to group deliveries.

4.2.1 Premises

A modified version of our program gathered statistics over an entire project in SEMC. The deliveries found were not integrated, only classified by our program i.e. no changes were made to the project configuration file and no build was made.

4.2.2 Result

The table below describes how many deliveries were sent to CM. Integrated and Not Integrated shows how many of them AutoInt could integrate. This filter was configured to only make Module replacement and Dependency as automatic integrated.

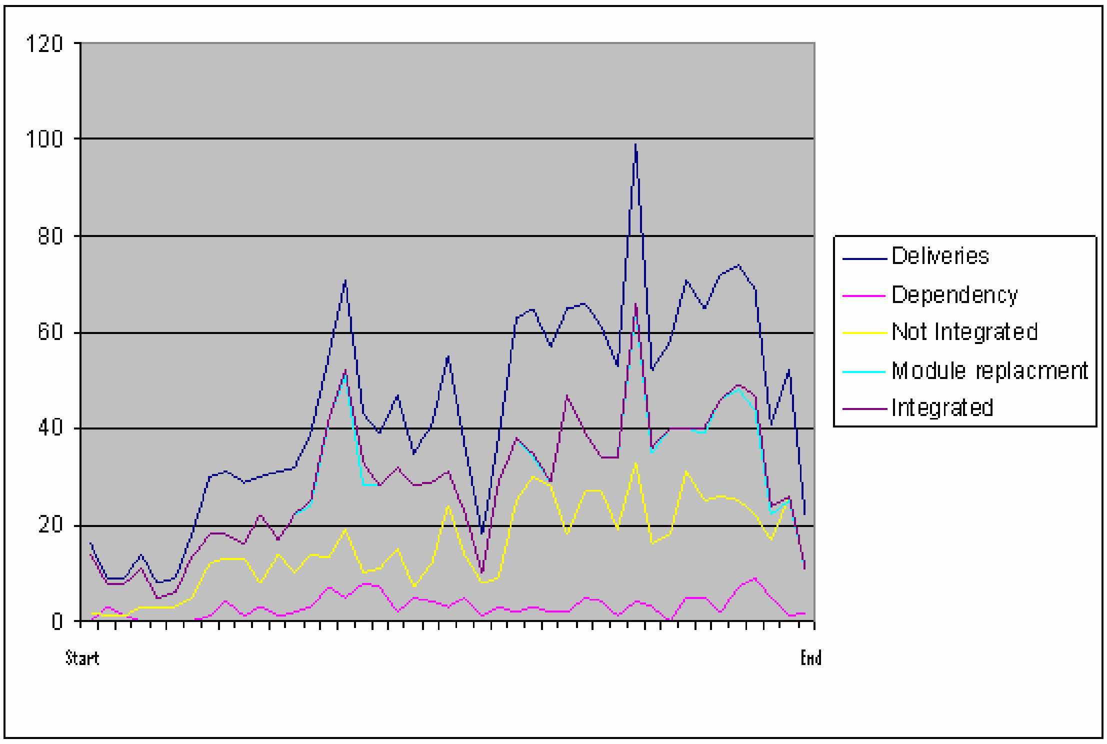


Figure 4.1, *The figure shows the various integrated deliveries made by AutoInt.*

The diagram below outlines the percentage how many percent of the deliveries that AutoInt is able to integrate.

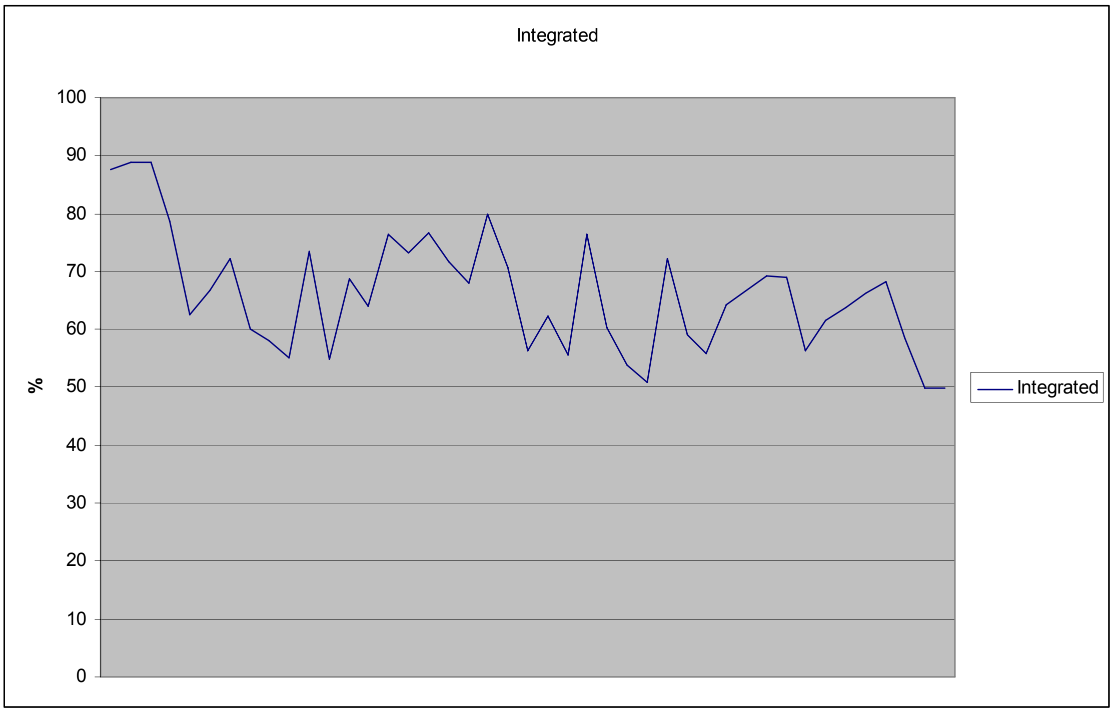


Figure 4.2, *The figure shows the percent of integrated deliveries on the y-axis. The x-axis represents time.*

4.2.3 Discussion

As the figure 4.2 shows, AutoInt can handle almost 90% of the deliveries at the start of the project but as the project precedes the deliveries that AutoInt can handle gets less. The reason for this could be statistical because there are not as many deliveries to integrate at both the start of and at the end of the project.

4.2.4 Conclusion

It would be a really good idea to use the program, since 2/3 of all deliveries at all times are integrations that AutoInt can handle. But that is not the whole truth. This experiment only decides if the program is able to integrate the delivery, the experiment did not try to integrate and build the deliveries. This would have given us more trustworthy results, but unfortunately, it would have taken too much time.

4.3 *Affected efficiency when building twice*

SEMC has got a wink-in function used to decrease build time. This experiment checks how much time would be saved by using this wink-in function. To make the experiment, we ran a delivery two times without using the wink-in function and then one more time using it.

4.3.1 Premises

We used build servers with equal speed and we used the same delivery both times.

4.3.2 Result

The first build was made five times and all builds took about the same time. When the wink-in function was used the time was only 41% of the original build time.

4.3.3 Discussion

We built five times to ensure that the build times were approximately the same on all occasions. The number five was a good compromise between the time it took to run the experiment and the result, which was almost the same on all five times.

4.3.4 Conclusion

Our advice to SEMC is to always use the wink-in function and to run nightly builds whenever the build servers are ready.

4.4 *Faster build with one delivery at a time or many?*

It is possible to build deliveries, either one at a time or to group the deliveries together and build them all at one time. It is good to know in which stage of the project it would be good to make one choice instead of the other. The purpose of this experiment is to find out the premises on which alternative to choose in various situations.

4.4.1 Premises

The experiment was carried out on the same build servers with the same deliveries. The same number of variants were built in both tests and then compared. We ran only one delivery and used the repository and five variants.

4.4.2 Result

The build where the deliveries were grouped together are indexed by the time 100. Here the repository was used both to put and get the files.

When a build was made after each delivery the result was index 50. After this build the other deliveries were built in index:

Build	**Time**
2	14
3	13
4	12
5	14

Summary: 103

4.4.3 Discussion

We can see that when we have five deliveries grouped together that a build is a little bit faster. If we would add more deliveries we most likely would see that we would have to add almost half an hour to every extra delivery if we were to build them one at a time.

The time we would have gained with the grouping algorithm would be insignificant if we added one more delivery because the additional files that are needed for the build are just winked in. The build itself is not made several times.

4.4.4 Conclusion

One risk with grouping together deliveries is that if the build fails, a new build has to be made. This build has to integrate every delivery one at a time to find which delivery caused the previous build to fail.
More time would certainly have been good to carryout more experiments in this area.

4.5 *Does the number of servers matter on the number of variants built*

The number of servers is often a constraint in the project. With a known number of servers, is it possible to choose a particular number of variants to maximize the use of servers without loosing time?

4.5.1 Premises

The build servers are of similar speed. The same delivery is built all the time but with different amount of deliveries. The wink-in function is used throughout the whole experiment.

4.5.2 Result

	Time (index)
1 variant built on 2 computers	100
2 variants built on 2 computers	104
3 variants built on 2 computers	201
4 variants built on 2 computers	171
5 variants built on 2 computers	291
6 variants built on 2 computers	264

4.5.3 Discussion

As the result show, it is recommended to always build the same amount of variants on all the build servers to utilise them as much as possible. This result was expected to a certain degree. However, it is noteworthy that the third run is faster than fourth and the sixth is faster than the fifth.

The statistics are best interpreted by grouping together (1,2), (3,4) and (5,6). A strange pattern which occurred in two of the couples shows that more variants can be built in less time than fewer variants.
This strange pattern has to depend on different workload or different times of the day when the build are run.

The formula derived from this experiment is

X = #build servers – (#variants modulus #build servers)

X = The number of variants that could be added to easily optimise the build

“#variants % #build” servers is the number of variants that will not use all the build servers. The number of build servers minus this number is the number of variants that can be added to optimise the number of variants.

The reason why we did not use more than two build servers was that SEMC used their build servers almost all the time, so we had to use the two computers dedicated to us.

4.5.4 Conclusion

Derived from the formula described in the discussion above, there are two noteworthy formulas.

To find the optimal number of servers, one has to allocate, either the same number of servers as variants, or the number of variants divided by two if this is not applicable. If this is impossible, divide this number by two again and so on.

5 Discussion

The effectiveness of AutoInt is a good place to start this discussion. After our experiment we know that approximately 2/3 of the integration can be taken care of. If we assume that we run AutoInt at night, the CM group can accept or reject all deliveries that AutoInt have made, when they arrive at work following morning. Hence, the developer sends a delivery before he/she goes home for the day the result can be ready first thing in the morning. This is practically instant feedback and is as close to our goal of Continuous Integration that is possible.

Even when the integration fails it saves time because then CM group know when something is wrong with these deliveries and can test these separately. This is equally important as a successful integration and saves almost the same time. If the integration fails or finishes successfully both results gives important information to CM.

Is it possible to let more deliveries be automatically integrated? We have chosen four that we thought were the most common ones and were easy to carry out automatically. There are of course other deliveries but we have found out that there exist no other common deliveries, most of these deliveries are "special cases". When we use the term "special case" we mean that it must be a manually integrated delivery.

One kind of integration test that we did not have time to implement in AutoInt was automatic smoke test. When this test is implemented in AutoInt, it can do the entire integration process independently.

AutoInt was tested in a live environment and the result of the test was satisfying. AutoInt integrated the deliveries over a weekend.

We don't know if SEMC will use AutoInt but we hope that they will release it.

5.1 *Program flexibility*

In this chapter we will discuss strengths and weaknesses of AutoInt. We will also discuss how easy or difficult it is to upgrade the program.

5.1.1 Easy way to trick the module freeze label update

The module freeze label update algorithm described in section 3.3.1 can easily be tricked by a developer that delivers a module label which is older than the one in the project configuration file. The existing label would not be checked against the new one in the delivery. There are two reasons for that:

1. The old freeze label is not included in the delivery.
2. It is hard to know in some situations which of the labels are the newest one.

It is quite easy if one of the labels are named R1A001 and the other one R1A002, but if the first one would be R1A015 and the other one was called R1B002, which one is AutoInt supposed to choose? One way to choose it would be to also give the old label as a parameter to delivery web.

5.1.2 Vulnerability of regular expressions

The regular expressions written in the programs are not as flexible as one may wish when implementing this kind of program. Because of this, the program is very dependent on the input and output as it is today in all programs, SDE, CME, ClearCase and Delivery Web. It would be quite easy to change the regular expression if the protocols for the programs changed but it would be annoying.

5.1.3 Build server update between builds

If the program finds many deliveries and one of them starts to build, one may realize that all the build servers in the configuration file will not be available all the time needed to build another delivery after the first one. Therefore it would be quite nice to reload the configuration file after each build. A problem with that is that it may cause file permission problems if updating it the same time as the program reads it.

6 Future work

Through the thesis work a few proposals to future work were found, some ideas that we didn't have time to create them ourselves, or other features that were not a part of the thesis project:

- If for some reason, SEMC decide not to use AutoInt as intended, they could certainly use it as collect statistics about all the projects. The statistics could be the same as the experiments made in this project, or other statistics that could be useful in different projects.
- If AutoInt was integrated in Delivery Web, all the automatic builds could be triggered as soon as a new delivery arrives.
- Whenever AutoInt succeeds, it would be good to let AutoInt know which bugs that were corrected in the delivery. This could be used to remove the bugs from the bug database, DMS.
- At the time this paper was written, the automatic testing environment BRAT was not fully implemented but it would be good to integrate it with AutoInt.
- An API with all system commands used in AutoInt and also how AutoInt parses the input from other programs.

7 Conclusions

The core of the project was to make a program to automatically integrate easy deliveries and an implementation in Perl was made.

Our program relies much on immutable static data from the programs we are using, but this static data will probably not last forever, as SEMC upgrade their programs sporadically. At the same time, it is almost impossible to guess or suspect how the other programs will act in the future, therefore we agreed on developing a very stable algorithm instead. This also implies that the program is almost impossible to use outside SEMC but the general algorithm can be altered to satisfy other purposes.

At this moment, our program is able to integrate more than 2/3 of the deliveries reaching the program, but this is not the whole truth. The reason is that the protocol where the developers send their deliveries is not as strict as we hoped it would be. It exists some text areas, where the developer is able to send information to the CM team, and this information is very hard to interpret without natural language processing, therefore, a proposal from us were made to change the protocol and the change is in the change control board right now.

The program had a very positive side effect which was not known when the thesis project was started. When a build is made, a central cache repository gets all files compiled by the build, and this makes other builds much faster. This repository is used even if the build does not succeed, therefore, even if our program did not succeed with all nightly build, the CM team had a great time loss when they built the morning after.

The build time is the most consuming part of an integration, and unfortunately, even simple module replacements have to be built and tested thoroughly. To tightening the loop from programmer to the integrations we think that in the future Auto Integration should be integrated to Delivery Web.

Glossary

Baseline – A stable version of the software code
Branch – A version handled copy of the code
CGI – Common Gateway Interface
Check in – To write a file to the version handling program
Check out – Make a file in the version handling program writable
CM – Configuration Management
Freeze label – A name of the code at a certain time
GUI – Graphical User Interface
Make – A build tool used by describing which components are built and how they are related to each other.
Rebase – Updating a working copy against the version handling program
Reproducibility – The ability to recreate software [11]
Revision – A new update of a file
SEMC – Sony Ericsson Mobile Communications, joint venture of Sony and Ericsson
Smoke test – A small test "to check if smoke comes out"
Uncheck out – Undo a check out
Variant – Different implementation of the same component [12]
View – Private workspace in a version handling program
Wink in – Using a compiled cached file.
XML – eXtended Markup Languge

8 References

1. Operating system concepts with java, sixth edition, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, ISBN 0-471-48905-0, 2004

2. The CM Challenge: Configuration Management that Works, David B. Leblang, ISBN:0-471-94245-6 Atria Software, Inc

3. Wayne A. Babish: Software Configuration Management – Coordination for team productivity, ISBN:0-201-10161-0

4. Martin Fowler, Last update 1st of May 2006. Last check 17th June 2006 http://www.martinfowler.com/articles/continuousIntegration.html#SignificantRevisions

5. M. A. Daniels: Principles of Configuration Management, Advanced Applications Consultants, Inc., 1985

6. http://www.mozilla.org/ Last visited 17th of June 2006

7. http://www.apache.org Last visited 17th of June 2006

8. http://cruisecontrol.sourceforge.net Last visited 17th of June 2006

9. Jeffrey E. F. Friedl, Mastering Regular Expressions, Second Edition, ISBN 0-596-00289-0, O´Reilly Media, 2002.

10. Learning Perl (Llama book) ISBN: 0596001320 O'Reilly Media 2001

11. Tom Milligan: Better Software Configuration Management Means Better Business: The Seven Keys to Improving Business Value, Technical Report, IBM (Rational), August 2003.

12. Chapter 3, Variants: Keeping Things together and telling them apart, ISBN:0-471-94245-6