Source Control…$100
Regression Script…$500
Good Automated Release Steps…$Priceless

Jeffrey Wren
Paradigm Works
300 Brickstone Square
Andover, Ma 01810
1-978-824-1400
jeff.wren@paradigm-works.com

ABSTRACT

Release management is critical to the success of every hardware development environment. However, it is typically the most overlooked and underestimated task in most development teams. In this ever increasing complex world of ASIC and FPGA designs, the ability to manage the changes made by both design and verification members in a sufficient way is needed where one can quickly determine faulty RTL, synthesis, schematic, and layout updates. This paper will address the drawbacks of a typical release flow, and will put forth a proven 5 step process a design team can implement which can be then be automated. . It also presents a case study, where a free open source software tool ReleaseWorks® [3] was successfully used to automate this 5 step process.

Categories and Subject Descriptors

B.6.3 **[Logic Design]:** Design Aids - *Verification*
D.2.7 **[Software Engineering]:** Distribution, Maintenance, and Enhancement – *Version Control*

General Terms

Reliability, **Standardization**, **Verification.**

Keywords

Release Management

1. INTRODUCTION

Release management is critical to the success of every hardware development environment. However, it is typically the most overlooked and underestimated task in most development teams. In this ever increasing complex world of ASIC and FPGA designs, the ability to manage the changes made by both design and verification members in a sufficient way is needed where one can quickly determine faulty RTL, synthesis, schematic, and layout updates. Often, the solution has been to write out a step by step process (parts of which are scripted) that one should follow before changes are committed to the main line of one's source tree. This formalizing of the release process is to be commended. However, the fact that it places the responsibility on the user to follow a manual procedure to verify changes is faulty. Frequently, the user may overlook a step and even if all the steps are followed may still make a mistake, e.g. forgets to check-in a file, or implements an environment variable which was set to get things to work and is only available in the user's local shell. The result is that the HEAD code of the development tree becomes broken. This can be relatively straightforward to debug in a small team environment. However, in large and possibly geographically dispersed teams where the database is changing rapidly, the ability to determine the broken area and to back out the changes can be time consuming.

Another issue is the release manager problem. This is where an engineer is tasked with making sure that all of the changes committed since a certain time are passing the verification testing. Again, this is faulty because the release manager has no easy way of isolating what is wrong and whose changes are the culprit for causing test failures. Also, this individual is most likely not well versed on the entire design and who to assign relevant issues to. The result is that many hours or up to several days can be spent cleaning up a snapshot release. Following which, the design has changed significantly and the whole onerous procedure has to begin again. This paper will address these drawbacks in release management, and will put forth a proven 5 step process a development team can implement which can be used to automate the whole release flow. The 5 steps to be outlined are as follows:

1. Create/Update Workspace
2. Modify Source Code
3. Submit
4. Integrate
5. Publish

A case study is provided detailing how utilizing a free open software tool, ReleaseWorks® [3], Company X went from a release process of up to 2 weeks between releases to just hours or at most a day.

2. Release Management Defined

Before continuing it is important that the concept of release management be defined. Ultimately, release management is the set of steps taken to guarantee that one's source code, schematic, layout, etc, is ready for distribution to the customer. The customer is defined as another team member, another division within the company, or a customer in the traditional sense.

3. Design and Verification Release Problems

There are three overall problems that a proper release methodology addresses, the user problem, the release manager problem, and the reproducibility problem.

3.1 The User Problem

Typically, users[1] want to be about the task of performing their job of designing the latest feature, writing tests to find bugs, etc. They don't want to be bogged down with the steps of performing release management of their code. However, most release methodologies place a great burden on the end user to perform a checklist of steps before they are allowed to commit their code changes to the repository. A common list of steps is as follows:

1. Creates workspace from Source Control.
2. Makes modifications
3. Runs local tests or regression list to verify changes
4. Update to latest changes on trunk
5. Re-run tests to make sure everything still works
6. Commits changes to source control.

The above steps are wrought with problems. For step 1, how does the user know that sandbox being generated even works, i.e. compiles and executes basic tests? Many project teams expect that if the above steps are followed, all one needs to do is create the sandbox to the HEAD revision. However, what if the user forgets to commit one of the files, or in order to get things to work locally, had an environment variable set? The result is that the HEAD is broken. Since the user cannot be guaranteed that the HEAD is clean, then he can spend time trying to debug an issue that is not even related to the changes that he made.

Step 4 is an attempt to solve the problem of two users submitting conflicting changes. However, in large teams, there is still a race condition that another user could commit changes during the time the first user is performing steps 4, 5, and 6.

The overall result is that the development branch becomes broken, and it takes time to debug and fix it. During this time, anyone who updates to the HEAD is also broken. The brokenness can be blatant such as the case of the testbench no longer compiles, or it can be subtle, where something functionally changed, and the behavior is modified where it could be hours before the problem is discerned and debugged.

3.2 The Release Manager Problem

Unlike the user whose focus is his own changes, the release manager's focus is on the entire project. Specifically, making sure that all changes from the team integrate together to make a deliverable to the customer. Just like the user, the release manager also has a checklist of tasks to perform. For example:

1. Determine latest good code.
2. Compile a workspace
3. Run regression suit of tests
4. Interpret results and resolve issues
5. Label the files
6. Notify project team
7. Repeat

In some teams the above list is performed by hand which can be prone to user mistakes. Often times the release manager will script several of the steps together to make their life easier. Regardless, the big issue is with step 4, interpreting the results. If everything passes then the job is done and the release manager can move on. However, what if it fails? What set of changes caused the problem? Who is responsible for the fix? Once the problem is found, should it be fixed and integrated in before moving forward, or should the change be backed out? What if the fix actually causes a different area of code to break? In the failing case of step 4, this can be a very time consuming process of determining the broken code and performing the fix. It can be iterative, and time consuming depending on how long the regression is. Also, what if the responsible user is out sick or on vacation? A more common issue today with distributed teams is the time zone problem. The release manager and the responsible code changer can be on opposite sides of the globe. Even for a simple fix it could be a minimum of 12 hours for the release manager to communicate the problem to the user and for that individual to submit a fix. All of this serves to keep the release from being performed.

While debugging any found issues, people may be committing changes. These changes could further hamper the release manager. For example, the release manager traces a problem to file foo.v. He contacts User A to fix the problem. However, during the time that User A is fixing the problem User B commits a change to foo.v. This change is then dependent on several other files. Now, the release manager has to update to another set of changes, which may cause other problems.

To address this problem, some teams have "code freezes". This is a time where no one is able to commit any changes to the repository until the release is performed. This actually can hinder development because it becomes difficult for team members to share changes and even though it doesn't stop development, it can slow it down significantly. This is because users will just do the minimum of what they can, and then they wait for the code freeze to end before they start being fully productive again.

3.3 The Reproducibility Problem

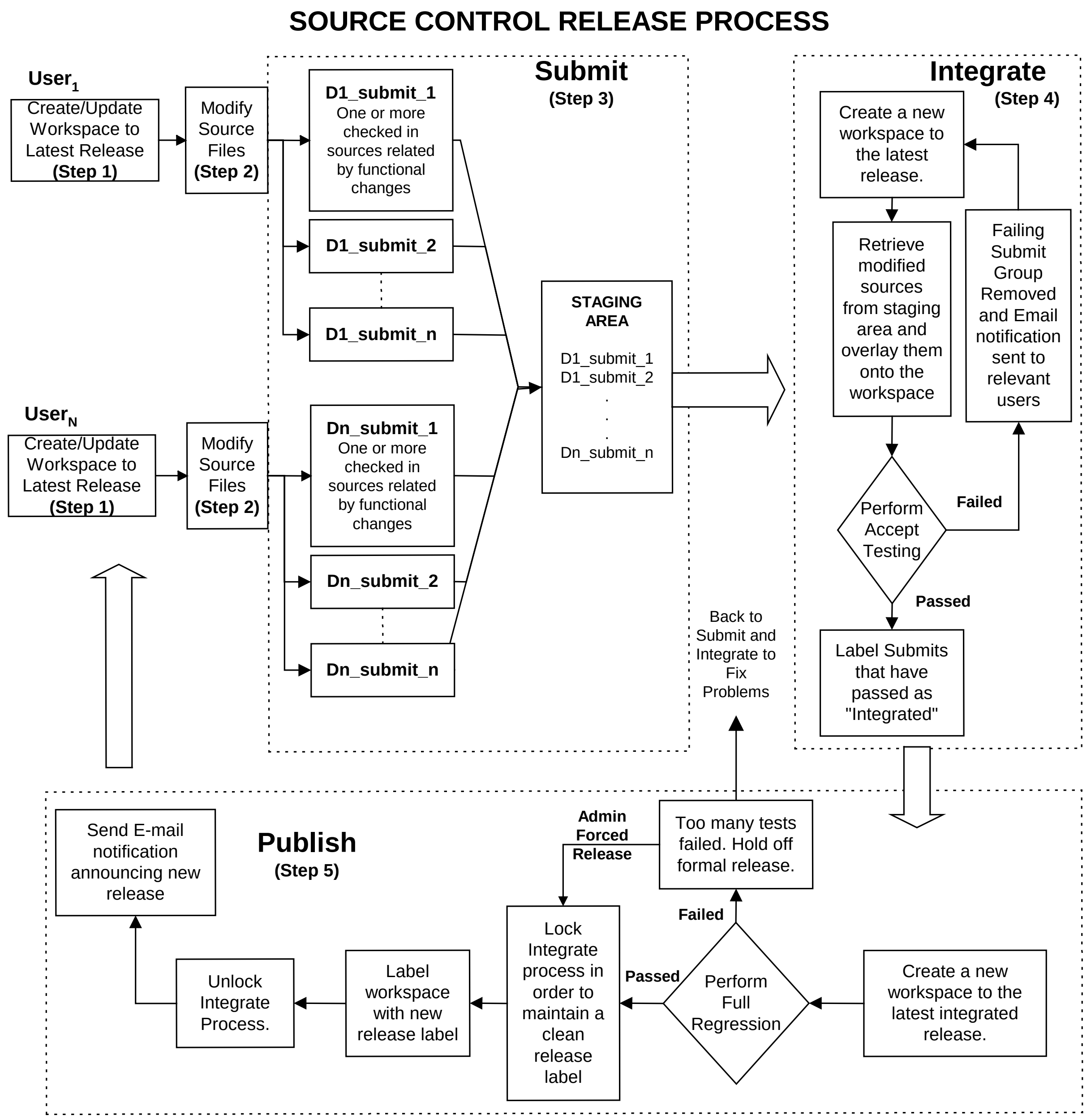
As verification methodologies have matured, the use of random test environments has increased. One of the downsides to this though is given the same seed value, a change to the source code can change the results of a test. If this happens, it can make it difficult for a team to reproduce a bug. One solution is to have the user who found the bug create a label that marks all of the relevant files. However, this can be time consuming, and also error prone, because for this to work everything has to be identical in the user's workspace as to what is trying to be regenerated. Basically, the user is performing many of the steps of a release and has several of the issues previously mentioned.

4 Five Steps for a Good Release Flow

A proper release flow will take the burden of acceptance testing out of the hands of the user, make the job for the release managers easier so that they can work on other project tasks rather than performing the release steps, and make the development area completely reproducible at any incremental stage of the design with minimal effort.

This section will outline 5 steps that can be used by any development team to construct a professional release flow. In actuality, this flow is utilized by many software teams. Since much of the hardware world is done in the software domain, this methodology works very well [1]. If done properly, the flow can be automated allowing for greater productivity.

[1] A user is defined as anyone who modifies the source files of the repository that make up the project. This can be a designer, a verification engineer, schematic entry tech, etc.



SOURCE CONTROL RELEASE PROCESS
User1
Create/Update Workspace to Latest Release (Step 1)
Modify Source Files (Step 2)
UserN
Create/Update Workspace to Latest Release (Step 1)
Modify Source Files (Step 2)
Submit (Step 3)
D1_submit_1 One or more checked in sources related by functional changes
D1_submit_2
D1_submit_n
Dn_submit_1 One or more checked in sources related by functional changes
Dn_submit_2
Dn_submit_n
STAGING AREA
D1_submit_1
D1_submit_2
Dn_submit_n
Integrate (Step 4)
Create a new workspace to the latest release.
Retrieve modified sources from staging area and overlay them onto the workspace
Failing Submit Group Removed and Email notification sent to relevant users
Perform Accept Testing
Failed
Passed
Label Submits that have passed as "Integrated"
Back to Submit and Integrate to Fix Problems
Publish (Step 5)
Send E-mail notification announcing new release
Admin Forced Release
Too many tests failed. Hold off formal release.
Failed
Passed
Perform Full Regression
Create a new workspace to the latest integrated release.
Lock Integrate process in order to maintain a clean release label
Label workspace with new release label
Unlock Integrate Process.

Figure 1: Release Process Steps

The following steps are the beginning of a specification that one could use to develop a release management tool. For example purposes a tool will be defined called release_tool that will be used to demonstrate how some steps could be implemented.

Figure 1 shows the five steps as a workflow.

4.1 Step 1: Create/Update to Known Working Label

4.1.1 Create Workspace

The first thing that needs to be done is to have a command that will create a user's sandbox/workspace to a known working label. What is meant by "working" is that at a minimum the workspace that is generated will allow one to compile and run any tests that are defined by the acceptance testing (Step 4). The default of the command should just create the sandbox to the latest released label that was created in Step 5. For example:

```
% release_tool create_sandbox sandbox_name
```

Where:

release_tool	The command line executable
create_sandbox	The subcommand call to the release_tool that performs the create
sandbox_name	A user argument that defines what the name of the sandbox should be

Having the tool automatically keep track of the release labels removes the burden of the user keeping track and trying to figure out what is the latest good code. For reproducibility purposes, the command should have an option that allows it to accept any previous release label.

```
% release_tool create_sandbox \
   –label PROJ-REL_1_2_3 sandbox_name
```

4.1.2 Update Workspace

For day to day use, in order that a user doesn't have to create a new sandbox every time a new label is generated, a command needs to be constructed that updates the workspace to the latest released code. For example, while in the existing sandbox, one could execute:

```
% release_tool update_sandbox
```

The default behavior should update the workspace to the latest release label from Step 5. However, a useful option would be one that allows one to update to the latest code that has passed the acceptance testing (Step 4), but has not yet passed the full regression testing (Step 5).

```
% release_tool update_sandbox –latest_accepted
```

4.2 Step 2: Modify Source Code

This is where the user performs the changes to the source code and then commits the changes to the source control tool. There are two methodologies that can be utilized. One, everyone works off the same branch. This is the mainline/trunk approach. Two, each user works off of their own user branch. The purpose of this paper is not to delve into this. Some pros and cons for each approach are listed below:

Mainline/Trunk
Pros:

- Files can be locked so that two users cannot modify the file at the same time. This can be specifically useful for binary files.
- Commits cannot be performed until changes made by another user are rectified. This can help to prevent merge conflicts.

Cons:

- Two users cannot work on the same file at the same time. More of an issue with distributed work teams.
- Commits are performed on same branch so that if two users need to touch the same file for their change group to pass the Integrate (Step 4) but user one's changes don't work, user two changes can be blocked.

User Branches
Pros:

- Two users can work on the same file at the same time.
- Commits to the user branch do not affect anyone else. Therefore, commits can be done frequently.

Cons:

- Files cannot be locked. This can be an issue for binary files.
- More merge conflicts when integration to the trunk is performed.

4.3 Step 3: Submit Changes

The Submit step is used to collect information on all the files that one has modified and group them together into a single submission to the release process. The requirement is that each of the files to be submitted must be checked into the repository. This is required because Step 4 uses the source control tool to generate the testing workspace. Depending on one's source control tool, the submit information can be a listing of each file and the relevant version, or it can be a label that contains the specific files that have changed. This information is then stored in a file in a staging area (A directory) until the release tool's acceptance testing (Step 4) is ready to process them. One should not have to wait for the acceptance testing process to start or end. One should be able to submit to the flow at any time.

One of the main purposes of a Submit is to collect all of the files that have changed and have dependencies on each other. For example, if three files have changed, and each file's changes are required in order to pass the accept list of tests, then one would submit all three files together. Do not submit each of the files independently. If there is a problem during the acceptance testing, then each of the individual submits would most likely fail, even if there was nothing wrong with the changes.

The final consideration before creating the Submit is to query the user for a release note. It is strongly encouraged that a detailed note describing the changes be entered. If the changes reflect a feature or bug fix in a change control system, then the change control reference number should also be included in the note. This information will travel with the submit group and will be included in the formal release notes.

4.4 Step 4: Integrate the Submit Groups

The Integrate step really is at the heart of the release flow. It is what keeps the code base in a known working state. It is used to verify changes that are submitted by the users. If the changes pass the acceptance list of tests the submitted group is promoted to the "Integrated" state. The acceptance testing should be made up of the minimum requirements that the submitted change groups should pass before being integrated into the release. This is usually made up of a compile of each of the relevant test benches and a simple test from each. Once the acceptance testing is complete, an email can be sent notifying the users that new integrated changes are available at which point they can perform an update workspace if they so desire (section 4.1.2). If the changes fail, then the release tool needs to determine which user's submitted group caused the failure. If there is only one Submit then this is easy. If there are multiple ones, then a process needs to be defined on how to determine the failing group. For this flow a straightforward binary search algorithm is used (section 4.4.1) until the failing submit group has been found. Once the failing submit group is found, it is labeled and an email detailing the failure and how it can be reproduced can be sent to the users involved.

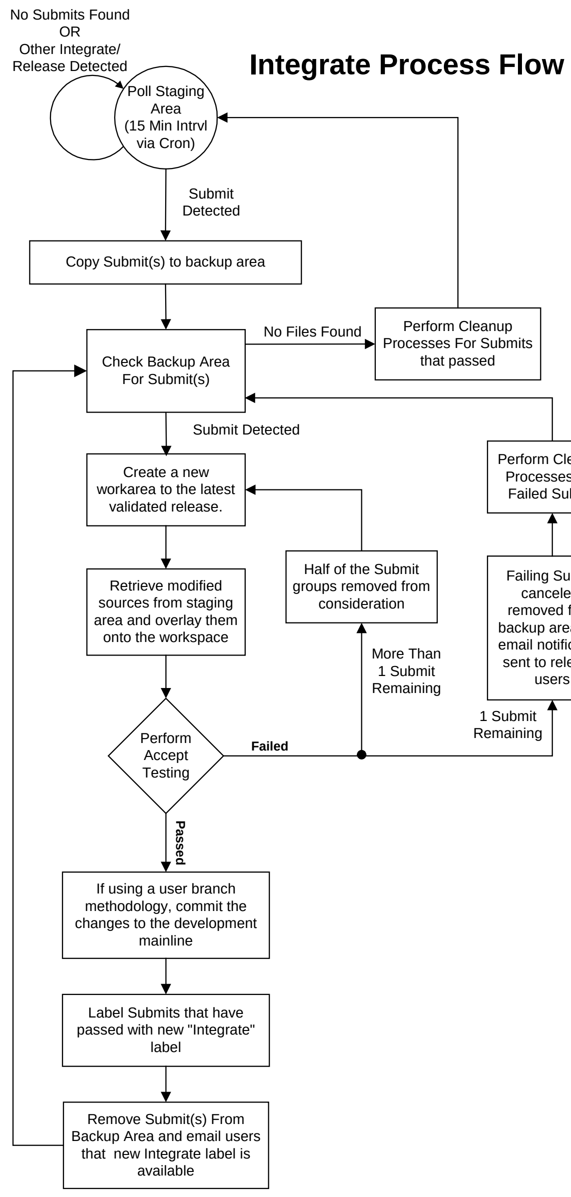
The reproducibility information is always available because the Integrate step knows what version of the workspace was in use at the time of the failure. In order to regenerate the problem, all the user needs to do is create a new workspace (Step 1) to the label at the time of failure. Then overlay the submitted change group.

4.4.1 Integrate – Step by Step

This section will go into more detail about each of the steps the Integrate process should perform in order that it can be automated. Figure 2 provides a flow diagram of each of the steps that are performed.

The first thing that the Integrate step does is to check the staging area for user Submit groups. If there are none, or another Integrate is running then the Integrate process will exit. This behavior allows one to place the Integrate step under cron control. That way the command can be kicked off at any interval that one defines (Typically 5 to 15 minutes) and if another process is running the exiting prevents process back-ups. If at least one change group exists in the staging area then the Integrate process will move the Submit group description files to a backup area. The Submit groups remain in the backup area until the Integrate step has successfully verified the changes and created a new source control label representing the changes. The Submit groups are not removed until everything is complete so that if there is an interruption in the process or a fatal error occurs, one can restart the flow.

Once the change group description files from the staging area are copied into the backup area, the Integrate enters into a loop process until all of the Submit groups have been processed. The first pass through this loop operates on all of the Submit groups that were initially found. A source control workspace is created, or updated if one already exists, to the latest Integrate label. Then all of the new change groups are overlaid on top. If one is using user branches, this is a merge operation. If one is using a mainline approach, the appropriate version of the file is retrieved from the repository and placed into the sandbox.



Integrate Process Flow
No Submits Found OR Other Integrate/ Release Detected
Poll Staging Area (15 Min Intrvl via Cron)
Submit Detected
Copy Submit(s) to backup area
Check Backup Area For Submit(s)
No Files Found
Perform Cleanup Processes For Submits that passed
Submit Detected
Create a new workarea to the latest validated release.
Perform Cleanup Processes For Failed Submit
Retrieve modified sources from staging area and overlay them onto the workspace
Half of the Submit groups removed from consideration
Failing Submit canceled, removed from backup area and email notification sent to relevant users
More Than 1 Submit Remaining
1 Submit Remaining
Perform Accept Testing
Failed
Passed
If using a user branch methodology, commit the changes to the development mainline
Label Submits that have passed with new "Integrate" label
Remove Submit(s) From Backup Area and email users that new Integrate label is available

Figure 2: Integrate Process Flow With Binary Search

Once the workspace has been brought to the correct state the acceptance testing is performed. If any process or test fails, then the Integrate is considered to have failed. If this happens, a check to see how many change groups were included in this Integrate pass should be done. If more than one change group is currently loaded then the process deletes half of the remaining Submit groups from memory.

The Integrate command then repeats the workspace creation and acceptance testing on the remaining Submit groups until the failing Submit group is found.—If at any time during this flow a Submit group or groups pass the acceptance testing, they are promoted to the "Integrated" state. All of the change groups that were integrated since the last formal release are labeled with a new source control label, email is distributed detailing that a new "Integrate" label is available, and the passing Submit group description files are removed from the backup area so that they are no longer operated on.—The following steps are performed on the failing gather group: An email is sent to the users who authored any of the changes to the files; Any appropriate cleanup processes are performed; and finally, it is deleted from the backup area so that no further processing is performed on it.

Finally, after all submit groups have been processed, if any submit groups passed then any appropriate clean-up processes are performed. The Integrate process then exits. The next Integrate is activated via the cron or by hand by the release admin.

Note: It is strongly recommended that the acceptance test suite be something that can run in an hour or less. This is especially true for large teams because large teams will have large numbers of Submits and it can take time to perform the binary search on such a large number. For example, consider the case of only 3 submit groups, one of which is failing. In a worst case scenario it can take up to four hours to find the failing submit group.

4.5 Step 5: Publish

The last step of the flow is the Publish. This is the step where a larger set of regression tests are performed. The Integrate phase is kicked off as frequently as needed to process user changes. However, the Publish step is performed less frequently because in the design world it can take many hours to execute (Nightly regression) or even days (Weekend Regression).

This step is more of blessing of the code that has already passed the integration step. If the Integrate label passes the larger set of testing then a new label is applied that represents this (See section 5, The Release Label). If it fails, then no new label is defined, and one is able to continue the Submit and Integrate steps until the next Publish is executed.

For larger development efforts it is recommended that the ability to perform a Publish concurrently with Integrates be established. This will allow for Integrates to be performed 24/7 which should keep the queue of Submits from getting backed up.

4.5.1 Publish – Step by Step

This section will detail the steps that should be performed during a Publish. Figure 3 provides a diagram of the Publish flow.

If in serial mode (Integrates and Publishes are performed serially), the first thing that Publish does is to make sure that no Integrate is currently running. If one is then the Publish will go to sleep for 5 minutes and will then check again. Once the Integrate has completed the Publish will lock out any other Integrates from occurring until the release process has completed. During the publication phase users are still able to commit changes to the repository and submit them to the staging area. If one configured the Publish to run concurrently with the Integrates then the Publish will just start regardless if an Integrate process is currently running or not, and Integrates will continue concurrently throughout the Publish.

Next the Publish will create a new workspace using the latest Integrate label (It is recommended that one always start with a clean workspace so that there is nothing left over from a previous Publish run). The Publish will then execute the full regression testing defined by the project. Once the testing is complete the test logs are parsed to determine the number of passing and failing tests. A watermark can be set that thresholds the number of allowable test failures (This can be convenient during development so that the release is not prevented even though there are some failures. As the delivery date gets closer the threshold can be reduced). If this threshold is reached then the Publish will perform any defined failure cleanup processes and will exit. The release manager will then need to determine if the publication should be forced, or if changes need to be made before continuing with the release.

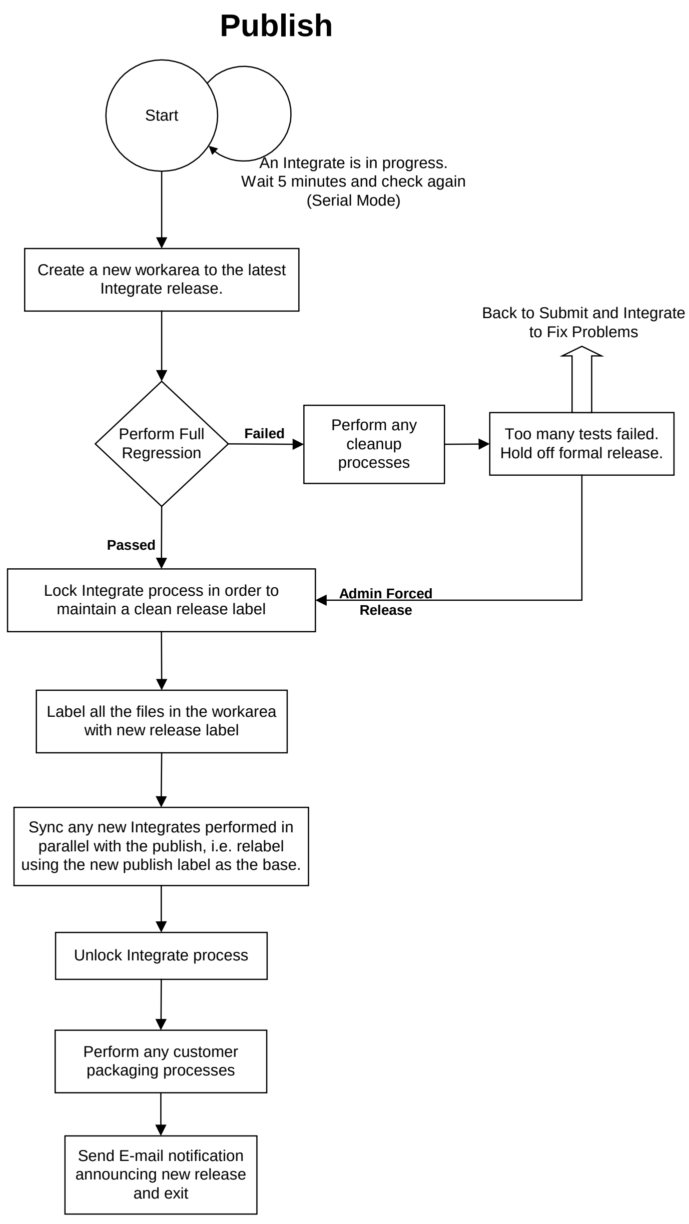

Publish
Start
An Integrate is in progress. Wait 5 minutes and check again (Serial Mode)
Create a new workarea to the latest Integrate release.
Back to Submit and Integrate to Fix Problems
Perform Full Regression
Failed
Perform any cleanup processes
Too many tests failed. Hold off formal release.
Passed
Lock Integrate process in order to maintain a clean release label
Admin Forced Release
Label all the files in the workarea with new release label
Sync any new Integrates performed in parallel with the publish, i.e. relabel using the new publish label as the base.
Unlock Integrate process
Perform any customer packaging processes
Send E-mail notification announcing new release and exit

Figure 3: Publish Flow Diagram

Upon successfully passing the test suite, the Publish will label the workspace with the new release label. Before performing the labeling, the Integrate process will be locked (Regardless if in parallel or serial mode) so that the labeling happens in a controlled manner. Once the labeling is complete the Publish will generate a release note containing all of the Submit groups in the release, a list of all the tests with their pass/fail results, and a dump of all the environment variables. This information is then emailed to those interested announcing the new release. Finally, the publish releases the lock file so that the Integrate process can continue.

5 The Release Label

In the release steps outlined, it can be seen that the methodology utilizes a labeling scheme to communicate to the user's the version of the repository that are stable. There is a plethora of labeling approaches that one can utilize. However, it is important to note, that a good labeling scheme can contain a great deal of important information of the state of the code, e.g. When was the label applied, what level of testing was performed (Integration Testing, Nightly Regression or Full Regression), what project does the label belong to, etc. A scheme that lends itself to hardware design is detailed here.

This labeling option provides for distinguishing between a nightly regression which contains a subset of tests, and a full regression that contains the full suite of testing. It consists of the following format:

<USER_PREFIX>-[DATE_STAMP]-REL_X_Y_Z

Where:

USER_PREFIX	Project/task unique string. It should be made up of alphanumeric characters separated by "_" characters. This prefix string should be locked so that only the release manager can apply it. That way, users cannot inadvertently change it or create a label that is not representative of the release. Example, MY_PROJ
DATE_STAMP	Contains the date that the last Publish label was generated (Step 5). It will be of the form MON_DD_YYYY, where:

• MON	Three character month abbreviation
• DD	The day of the month
• YYYY	The year

REL_X_Y_Z	Contains the release number info, where:

• REL	Short for Release
• X	The Major release number. This will be incremented by the Release stage (Step 5) when the release type option MAJOR is specified. Incrementing this value will zero out the Y and Z values. Typically used to designate that the full set of regression tests were run, such as a weekend regression.
• Y	The Minor release number. This will be incremented by the Release stage (Step 5). Incrementing this value will zero out the Z value. Typically used to represent that the code base has passed a subset of the full suite of tests, such as a nightly regression.

- Z The number of gather groups that have been successfully integrated since the last release. This number will be incremented by the Integrate stage (Step 4).

Example 1: Below is an example label followed by a list of information that we can discern from it.

```
MY_PROJ-JAN_01_2010-REL_15_3_8
```

- The MY_PROJ tells one that the label is the official label for the project, not a generic user one.
- The JAN_01_2010 string represents that the last Publish for this label occurred on JAN_01_2010
- The 15 represents that the code has passed the full suite of tests and is typically defined as the current "stable" release.
- The 3 represents that the code has passed 3 nightly regressions since the last full suite of tests were run.
- The 8 represents that eight Submit groups were successfully integrated since the last Release.
- Just by reading the above, we know that the last nightly regression label was MY_PROJ-JAN_01_2010-REL_15_3_0
- The date stamp would be different, but we know from the above that the last stable release label would have the string REL_15_0_0. The full label with date stamp would be easily determined by parsing the label history.

6 A Case Study

A simple case study on the impact of how automating the Integration and Publish steps of the release can have on a development team is provided. What is described is an actual case where Company X had developed a release management system (To keep the name of the tool confidential it will be called the release process tool, RPT), but it was incomplete. It solved the reproducibility problem, but it still had issues regarding the user and the release manager problem.

6.1 The User problem

The RPT was constructed as a wrapper around the source control tool ClearCase [2]. Each time the user wanted to commit their changes to the repository, they would have to execute the RPT tool with the appropriate files. At this point the tool would apply a label to the version of the files that were committed. This labeling approach kept others from seeing the changes until the files passed a regression performed by the release manager. There are two problems with this approach. One, the users were encouraged to run testing on the changes, but if the user actually performed the testing was not enforced. If the testing wasn't done, this could cause issues for the release manager when he went to perform the release. Two, even though this approach kept the new files from being automatically seen by other users it did not do a good job of managing changes as change groups. The user could commit one file or many, but there was no mechanism to tie them functionally together. This again could cause problems for the release manager when attempting to determine why a particular release was broken. This was due to the fact that release manager did not necessarily know what file dependencies were related.

6.2 The Release Manager Problem

The RPT tool had functionality to allow the release manager to control what user commits were to be tested. However, due to the complexity of the project, the release manager had to have detailed knowledge of how all of the design pieces fit together. The result was that the release manager was typically 1 person who was assigned to the task. If that person was unavailable then the release was not performed. Also, another issue was the unit level verses system level. For the unit level, the task of performing the release was faster because the time to execute the regressions was faster. However, at the system level, the size of the project made the time to execute a few tests several hours. This combined with having to integrate changes from several teams caused the time for implementing the release to increase.

For example, in the beginning of the paper it was stated that the time between releases could be up to 2 weeks. This was not caused by the fact that it actually took 2 weeks to perform, but due to the fact that the only a qualified individual could perform the release. In the case of the two weeks, it was a system level task and the release manager had been away for a week on vacation. Because it was a system level release, the regressions could take up to 24 hrs to complete. Also, if an issue was found that needed to be addressed by a unit level, this could take some time because that unit level team had to go through its release flow before handing off any changes. By the time the release manager had debugged the problems, had the appropriate people make the needed changes, another week had gone by. Granted this is a worst case scenario, but it should highlight the problem of being dependent on an individual performing the release by hand.

6.3The Reproducibility Problem

Due to the fact that labels were applied at the time of the commit, one could pass this label or combination of labels to others in order to aid in the reproducibility of bugs. This actually worked fairly well. It worked best, when the user committed all of the files together representing a single change group.

6.4 The Five Step Solution

After a couple of years of using RPT, it was decided that it was not an efficient enough solution. Also, more work was being done in a distributed manner using ClearCase multisite and the RPT would need to be revamped to handle this. A free open source software tool called ReleaseWorks® [3] was installed that utilizes the 5 steps outlined in this paper. Table 1 outlines the improvements.

Table 1: Company X's RPT tool vs. 5 Step Flow

	RPT	5 Step Flow
Acceptance Testing	Burden placed on user, but not enforced. Result was the release manager would need to perform debug if something broke	Burden taken away from the user. Changes only integrated into release if acceptance testing passed. The release manager no longer needed to debug inter-dependencies between multiple user changes
Feedback to user if changes successfully integrated into release	Dependent on release manager: Typical 1 day, worst case 1 week	Dependent on time to perform acceptance testing and binary search for failing Submit. For

		Company X this was a minimum of 2 hrs, worst case 1 day.
Release manager time to perform release	Typical: 0.5 day Worst Case: 3 days	Automated Release manager is free to perform other tasks
Nightly Regressions	Often these would crash because a user didn't properly submit the needed changes or due to incompatibility between user changes. A loss of feedback as to the state of the project of 1 day	Automated acceptance testing executed throughout the day guaranteed that the nightly regressions would execute cleanly. This gave the develop-ment team consistent feedback as to the state of the project
Time between Publishes	Dependent on release manager. Best case, 1 a day.	Automated: Consistent incremental Submit releases based on acceptance testing, with daily release based on nightly regression and weekly release based on full weekend regression.

7. Conclusion

There are several issues that can make a release flow inefficient, the user problem, the release manager problem, and the reproducibility problem. Each have been detailed and discussed. A robust 5 step methodology has been outlined that can be used by any production team to solve each of these problems. The case study provided shows how one company made significant performance improvements by moving to the automated 5 step flow.

A detailed cost analysis is not provided, but it should be readily seen from Table 1 that there are vast performance improvements that free up resources to perform other tasks rather than implementing the release, both for the user and the release manager. The larger the team, the greater the cost reward. These improvements to the release methodology could very well be priceless if they allow a company to reduce their time to market giving them an edge with competitors.

8. ACKNOWLEDGMENTS

Thank you to the DVCON committee for allowing this topic to be addressed.
Thanks to Jim Crocker, and Ambar Sarkar, for their valuable input while writing this paper.

9. REFERENCES

[1] Bays, M.E. *Software Release Methodology*. Prentice Hall PTR, Upper Saddle River, 1999.
[2] ClearCase, IBM Rational, http://www.rational.com
[3] ReleaseWorks®, ParadigmWorks^{TM}, http://releaseworks.sourceforge.net