SAND REPORT

SAND2010-6674
Unlimited Release
Printed September, 2010

Xyce™ Parallel Electronic Simulator Design

Developer Guide

Eric R. Keiter, Thomas V. Russo, Richard L. Schiek,
Heidi K. Thornquist, Eric L. Rankin, Ting Mei

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories

SAND2010-6674
Unlimited Release
Printed September, 2010

Xyce™ Parallel Electronic Simulator Design

Developer Guide

Eric R. Keiter, Thomas V. Russo, Richard L. Schiek,
Heidi K. Thornquist, Eric L. Rankin, Ting Mei

Sandia National Laboratories
Department 01437
Electrical and Microsystems Modeling

Abstract

This document is the **Xyce** Circuit Simulator developer guide. **Xyce** has been designed from the "ground up" to be a SPICE-compatible, distributed memory parallel circuit simulator. While it is in many respects a research code, **Xyce** is intended to be a production simulator. As such, having software quality engineering (SQE) procedures in place to insure a high level of code quality and robustness are essential. Version control, issue tracking customer support, C++ style guildlines and the **Xyce** release process are all described.

Acknowledgements

The authors would like to acknowledge the entire Sandia National Laboratories HPEMS (High Performance Electrical Modeling and Simulation) team, including Steve Wix, Carolyn Bogdan, Regina Schells, Ken Marx, Steve Brandon and Bill Ballard, for their support on this project.

Trademarks

Contacts

Bug Reports	`http://joseki.sandia.gov/bugzilla` `http://charleston.sandia.gov/bugzilla`
World Wide Web	`http://xyce.sandia.gov` `http://charleston.sandia.gov/xyce`
Email	`xyce-support@sandia.gov`

Contents

Figures

Tables

1. Introduction

Chapter Overview

This chapter contains an introduction to the **Xyce** developer guide.

1.1 Preface

The **Xyce** Parallel Electronic Simulator has been under development at Sandia since 1999. Historically, **Xyce** has mostly been funded by ASC, the original focus of **Xyce** development has primarily been related to circuits for nuclear weapons. However, this has not been the only focus and it is expected that the project will diversify. Like many ASC projects, **Xyce** is a group development effort, which involves a number of researchers, engineers, scientists, mathmaticians and computer scientists. In addition to diversity of background, it is to be expected on long term projects for there to be a certain amount of staff turnover, as people move on to different projects. As a result, it is very important that the project maintain high software quality standards. The point of this document is to formally document a number of the software quality practices followed by the **Xyce** team in one place. Also, it is hoped that this document will be a good source of information for new developers.

1.2 Xyce Project Statement

With the elimination of underground nuclear testing and declining defense budgets, science-based stockpile stewardship requires increased reliance on high performance modeling and simulation of weapon systems. Electrical systems and components are major elements in today's weapon systems. The goal of this project is to provide the tools that will allow the use of massively parallel modeling and simulation techniques on high performance computers in existing and future nuclear weapon electrical systems models.

1.3 The Xyce Project

The Xyce project is an ASC-funded project at Sandia designed to meet current and future electrical simulation needs of the weapons designers. It will provide a circuit modeling tool for Sandia designers capable of running efficiently on high-performance parallel computers using state-of-the-art algorithms. Some goals of this project are to

- Support Sandia-specific circuit and device models
- Support canonical industry-standard SPICE-based device models.
- Support commercial SPICE netlist interface, compatible with HSpice and PSpice when possible.
- Implement innovative, robust and scalable algorithms.
- Produce an efficient parallel implementation on a variety of architectures
- Couple to Charon device (PDE) simulator [1] and other simulation codes, to support multi-physics simulation.

The requirement that Xyce be massively parallel means that it needed to be designed, “from the ground up” in a fundamentally different way than SPICE. However, the requirement that Xyce be backward-compatible with commercial SPICE tools required that it be, in many respects, as similar to SPICE as we can be, without detracting from the other goals of the project. Finding and achieving this balance is one of the challenges of developing Xyce.

1.4 Document Organization

This chapter is the introduction. Chapter 2 contains basic information about the Xyce code repository, which (as of this writing) is based on CVS. Chapters 3, 4, and 5 all contain checklists for Xyce developers, pertaining to CVS checkin, closing bugs, and customer support, respectively. Chapter 6 contains a Xyce C++ source code style guide. Chapter 7 contains a description of matrix and vector access, which is necessary for device model development, and a potentially confusing issue for new developers. Chapter 8 contains a checklist for device model development. Chapter 9 contains frequently asked questions (FAQs) about Xyce development.

Chapter 10 and chapter 11 are about project management, with chapter 10 pertaining to the Xyce release process, and chapter 11 pertaining to the management of the third-party libraries used by Xyce.

If you are a new Xyce developer, you will probably need to read chapter 2 and chapter 6 first. The rest of the chapters are all important, but the release management chapter (chapter 10) and the third party software management chapter (chapter 11) can be read later. They will become more important once the new developer participates in a formal code release.

Document Change History

Version	Author	Modifications	Date
1.0	Scott Hutchinson and Rob Hoekstra	Initial Release	5/1/2001
1.1	Eric Keiter and Heidi Thornquist	Updates	3/7/2008
1.2	Eric Keiter and Tom Russo	Updates	3/31/2009

Table 1.1:

2. CVS Directions

Chapter Overview

This chapter contains descriptions and examples of CVS procedures which are commonly used by **Xyce** developers. For other details about CVS usage, see reference [2].

2.1 CVS Checkout

To check out a copy of of Xyce, you will first need to insure that you have accounts on the the CVS host machines. As of this writing, the **Xyce** source repository is split between two different machines, one on the Sandia Open Network (SON) and one on the Sandia Restricted Network (SRN). The SRN machine only contains parts of the code that are export controlled. It is possible to build and run **Xyce** without any of the SRN source code, but the opposite is not true. For the purposes of this document, given that computers get replaced from time to time, the machine on the open network will be referred to as son_machine, and the one on the restricted network will be referred to as srn_machine.

Once you have accounts, you need to do the following steps on your local machine. We use remote CVS, so you'll first need to set the environment variable RSH to ssh. The checkout command is:

```
cvs -d :ext:son_machine:/Net/Proj/Xyce/CVS checkout  Xyce
```

This gets you the non-export controlled parts. If you have not set up RSA keys in your accounts, you will be prompted for your password to son_machine.

If you want the SRN source code (you don't need it to compile **Xyce**), you will also have to do two other checkouts, into specific sub-directories below the top:

```
cd  Xyce/src/DeviceModelPKG
cvs -d  :ext:srn_machine:/Net/Proj/Xyce/CVS checkout SandiaModels
cd  ../../doc
cvs -d :ext:srn_machine:/Net/Proj/Xyce/CVS checkout  SandiaModelsDocs
```

Just like with the SON repository, if you have not set up RSA keys in your accounts, you will be prompted for your password, only in this case it will obviously be for the srn_machine.

To check out the regression test suite, the proceduce is very similar:

```
cd (somewhere other than where you did the last  command)
cvs -d :ext:son_machine:/Net/Proj/Xyce/CVS checkout  Xyce_Regression
cd Xyce_Regression
```

Similarly, if you want the SRN regression tests, you will have to check it out separately.

```
    cvs -d  :ext:srn_machine:/Net/Proj/Xyce/CVS checkout  Xyce_SandiaRegression
```

2.1.1 Important CVS options and the .cvsrc file

Certain options need to be specified so commonly with CVS commands that it is helpful to save them in a .cvsrc file so that they cannot be forgotten. Create a file in your home directory called `.cvsrc` and place the following lines in it:

```
    cvs -q
    update -d -P
    diff -u
```

The first line of these makes every cvs command less verbose, which helps eliminate some unimportant and distracting messages. The second line makes sure that every CVS update command brings in newly defined directories and purges directories that have been deleted.

2.1.2 Keeping up-to-date

As other people work on **Xyce**, you need to bring your copy up-to-date. This is done with the `cvs update` command while your current working directory is the top level of your Xyce checkout:

```
cvs update -dP
```

The “-dP” options make sure that files in newly created directories are updated, and that files in deleted directories are purged. If you have a `.cvsrc` file as described in the previous subsection, that “-dP” option may be left off as the .cvsrc file will insert it anyway.

2.2 Branching and Tagging

When doing significant development on Xyce, it is useful to first isolate those changes to a branch. Following guidlines for CVS management [2], use these commands to create and manage a development branch of code.

1. Tag the HEAD branch at the branch point:

   ```
   cvs -d :ext:son_machine:/Net/Proj/Xyce/CVS rtag TAG_CREATE_B1 Xyce
   cvs -d :ext:srn_machine:/Net/Proj/Xyce/CVS rtag TAG_CREATE_B1 SandiaModels

   cvs -d :ext:srn_machine:/Net/Proj/Xyce/CVS rtag TAG_CREATE_B1 SandiaModelsDocs
   ```

2. Create the new branch from the tag on the HEAD you just created:

   ```
   cvs -d :ext:son_machine:/Net/Proj/Xyce/CVS rtag -b -r TAG_CREATE_B1 BRANCH_B1
   Xyce
   cvs -d :ext:srn_machine:/Net/Proj/Xyce/CVS rtag -b -r TAG_CREATE_B1 BRANCH_B1
   SandiaModels
   cvs -d :ext:srn_machine:/Net/Proj/Xyce/CVS rtag -b -r TAG_CREATE_B1 BRANCH_B1
   SandiaModelsDocs
   ```

3. Create a new directory for your branch work, and check out your new branch:

   ```
   mkdir BranchWork
   cd BranchWork
   cvs -d :ext:son_machine:/Net/Proj/Xyce/CVS checkout -r BRANCH_B1 Xyce
   cd Xyce/src/DeviceModelPKG
   cvs -d :ext:srn_machine:/Net/Proj/Xyce/CVS checkout -r BRANCH_B1 SandiaModels

   cd ../../doc
   cvs -d :ext:srn_machine:/Net/Proj/Xyce/CVS checkout -r BRANCH_B1 SandiaModelsDocs
   ```

4. Merge changes from the HEAD the first time.

   After a few days, work on the HEAD will make your branch diverge. To bring your branch into sync with the HEAD while maintaining your new work, you must merge changes. The process you go through the first time you do this is different from the process on subsequent merges.

(a) First tag HEAD to establish a merge point, then do the merge.

In a directory where the HEAD is checked out:

```
cvs rtag TAG_HEAD_MERGE_TO_B1 Xyce
cd Xyce/src/DeviceModelPKG/SandiaModels
cvs rtag TAG_HEAD_MERGE_TO_B1 SandiaModels
cvs rtag TAG_HEAD_MERGE_TO_B1 SandiaModelsDocs
```

Since you're doing these commands in a directory where there is already a checkout, CVS knows what repository to use. You have to change to the SandiaModels directory to tag the SandiaModels component, because that is in a different repository than the rest of the code. You don't have to change to the SandiaModelsDocs directory to tag that component, because it is in the same repository as SandiaModels.

(b) Do the merge

In a directory where the BRANCH_B1 is checked out

```
cvs update -kk -dP -jHEAD > mergeout 2>&1
```

In this example we assume a BASH or SH shell, and redirect STDOUT and STDERR to a file so that the numerous messages that CVS emits will be preserved. You will look through this file to check for conflicts.

You may also have to merge SandiaModels and SandiaModelsDocs separately if your "mergeout" file created above doesn't clearly indicate that those directories were updated properly. If you need to do that, just "cd" into those directories and do the same update command:

```
cd src/DeviceModelsPKG/SandiaModels
cvs update -kk -dP -jHEAD > mergeout 2>&1
cd ../../../doc/SandiaModelsDocs
cvs update -kk -dP -jHEAD > mergeout 2>&1
```

(c) Correct conflicts

This is a critical step where you have to be very careful. When CVS detects that lines were modified on both branches, it can't figure out which version to use in the merge. It flags these "conflicts" by emiting a warning and by inserting *both* versions of the lines into the merged file with delimiters surrounding the versions from each branch. You will need to examine each file flagged as having conflicts for all such occurrences and choose the right version of the modified code to keep.

When you think you're done with all the conflict resolution, it's best to try a "cvs diff" to make sure you didn't miss any of the conflict delimiters. From your top level of the BRANCH_B1 checkout, do:

```
cvs diff -u > diffs
```

and look in the file "diffs" for occurrances of <<<<, ====, or >>>>, which are the things that CVS would insert when it finds conflicts. If you find any of these in the diffs, it means you missed some conflicts and need to go back and edit some more.

When you're done with the conflict resolution, you should *always* build the newly merged code and make sure it compiles. You should probably also run the test suite.

(d) Commit the merged code

After you've convinced yourself that the merged code builds and runs after your conflict resolution, you may then commit the merged code. From the top level of your checkout, simply issue the command:

```
cvs commit
```

Make sure you enter a meaningful and informative commit log about what you've done.

5. After the commit, tag the branch.

   In your Xyce BRANCH_B1 checkout directory, do these four commands:

```
cvs rtag -r BRANCH_B1 TAG_B1_MERGE_FROM_HEAD Xyce
cd src/DeviceModelPKG/SandiaModels
cvs rtag -r BRANCH_B1 TAG_B1_MERGE_FROM_HEAD SandiaModels
cvs rtag -r BRANCH_B1 TAG_B1_MERGE_FROM_HEAD SandiaModelsDocs
```

   As before, you can get away without specifying a repository ("-d :ext:...") because you're doing them from a checkout directory in which CVS has already saved files to tell it what repository to use. You need to be in the SandiaModels or SandiaModels-Docs directory for the SRN repository to be updated properly by the second and third "rtag" command..

6. Continue working on the branch until you want to merge in changes from the HEAD again

7. Merge changes in the HEAD that occurred between your last merge and now.

   (a) Merge from the old tag on the HEAD to the present.

This is mostly the same as the process you followed the very first time, except that this time you're making sure only to merge in changes that happened since the last time you did this. The tags you created (TAG_HEAD_MERGE_TO_B1 and TAG_B1_MERGE_FROM_HEAD) will mark the points from which we'll be updating.

In your top-level Xyce checkout of BRANCH_B1, do a CVS update from the head of all changes since the last merge:

```
cvs update -dP -kk -jTAG_HEAD_MERGE_TO_B1 -jHEAD > mergeout 2>&1
```

Note: If you leave out the `-kk` then keywords in the source file, like `Revision 1.xx` will cause conflicts. The `-kk` option causes cvs to ignore keywords. This is a sticky tag in that it will remain in effect for the brach files until a `cvs -A update -jBRANCH_B1` is done.

As before, we're capturing the standard output and standard error streams to a file so we can review them for conflict messages later.

This command merges in all the changes from the last merge time to the HEAD of the main trunk. Once again, you should review the "mergeout" file carefully and make sure that the entire tree was updated properly, including the SandiaModels and SandiaModelsDocs tree. You should also make sure that all directories that were created on the HEAD since the last merge were created and populated properly.

(b) Move the merge tag on the main trunk

AFTER you are convinced that the update did everything you expected it to do, move the tag:

```
cvs rtag -F TAG_HEAD_MERGE_TO_B1 Xyce
```

cd src/DeviceModelPKG/SandiaModels

```
cvs rtag -F TAG_HEAD_MERGE_TO_B1 SandiaModels
cvs rtag -F TAG_HEAD_MERGE_TO_B1 SandiaModelsDocs
```

Note: If you do this routinely immediately after the update and then find a mistake in the merge (such as neglecting to have "update -d -P" in your .cvsrc file or forgetting to use "-dP" in the update command) you will be unable to re-do the update operation to correct the mistake because the tag will no longer be in the right place. It is therefore essential that you only move the tag when you are convinced that the merge was done correctly and you no longer need the previous merge point to be tagged. This has happened to team members on occasion and it is tedious to correct it, so you should double and triple check that your merge is complete before moving the tag on the HEAD branch.

(c) Fix conflicts.

Again, this is the step that requires careful work, but the steps are exactly the same as the first time you did it on the branch. See above for details.

(d) Commit the merged code The commands to do this are exactly the same as they were the first time you merged. See above for the exact commands.

(e) Update the tag on the branch

After you commit the merged code to the branch, move the tags on the branch that mark the post-merge point:

```
cvs rtag -F -r BRANCH_B1 TAG_B1_MERGE_FROM_HEAD Xyce
cd src/DeviceModelPKG/SandiaModels
cvs rtag -F -r BRANCH_B1 TAG_B1_MERGE_FROM_HEAD SandiaModels
cvs rtag -F -r BRANCH_B1 TAG_B1_MERGE_FROM_HEAD SandiaModelsDocs
```

(f) Now update the branch to remove the sticky tags

```
cvs -A update -dP -jBRANCH_B1
```

8. If more work is to be done on the branch then go to step 4, otherwise go to the next step.

9. When the work on the branch is complete and it is time to migrate the work back to the main line, merge changes from the branch to the HEAD.

   In a directory with the HEAD version of Xyce checked out:

   ```
   cvs update -dP -jBRANCH_B1
   ```

10. Fix the conflicts and commit

11. Tag the branch as dead ended. This is not essential, but sometimes comes in handy. Tag the branch code with the special tag: TAG_BRANCH_B1_END_OF_LINE:

    In the directory with the BRANCH_B1 checkout:

    ```
    cvs rtag -r BRANCH_B1 TAG_BRANCH_B1_END_OF_LINE Xyce
    cd src/DeviceModelPKG/SandiaModels
    cvs rtag -r BRANCH_B1 TAG_BRANCH_B1_END_OF_LINE SandiaModels
    cvs rtag -r BRANCH_B1 TAG_BRANCH_B1_END_OF_LINE SandiaModelsDocs
    ```

    Doing this makes sure that anyone viewing the graphical representation of the revision tree (as, for example, through bonsai) will see the end of line marker on the branch and know that the branch is no longer active.

3. CVS Checkin Procedure

Chapter Overview

This chapter contains a pre-checkin checklist for **Xyce**.

3.1 Guidelines

As a means of ensuring the integrity of the **Xyce** code repository, the **Xyce** developers are encouraged to do everything reasonably possible to make sure that any changes they've made to the code do not adversely affect the quality of the code and do not impede the other developers. In general always remember:

- This is a group project, so it is critical that you don't interfere with the work of others.
- Anytime a code checkin has an adverse effect, it will present an opportunity cost to the group. Once the code in the repository is broken, it causes confusion among all the developers. If developer 'A' breaks the code, developer 'B' may spend a inordinate amount of time, incorrectly assuming the code failure is their fault. This can happen even when the original bug is minor.

Thus, developers should follow the checklist listed below when checking in code.

The suggested good-citizen rules may be broken into two categories: 1) pre-check-in and 2) post-check-in. The "pre-check-in" tasks are performed in the developer's "sandbox" where the code is edited, etc. The "after check-in" tasks are performed in a completely separate mirror of what's in the repository. Here's the suggested process:

1. Develop a test for feature and/or bugfix. Do this before developing code. This is part of the philosophy of test-driven development. The passage of this test will be the criteria for declaring a code feature completed. For more on bug and issue resolution, see chapter 4.
2. In the developer's sandbox (i.e., development directory), edit the code, manually test it, etc.
3. In a build directory, build and test the code using the source from step 2. Please note that if any development has been done which the developer believes may require a special build/platform combination, those should be tested. For example, if development has been done which may impact the MPI-enabled portion of the code, an MPI-enabled build and successful regression test with a parallel run should be performed. Another example, if the developer has modified any code within an `#ifdef` block, the code should be compiled and tested with this `#ifdef` enabled and also with this block disabled.

4. Iterate through items 2 and 3 as needed until the test(s) created for this feature to pass.
5. In a high-level directory (top of the Xyce source, usually), issue a “cvs commit” with no other arguments. This assures that all the files you have changed get committed, regardless of their package subdirectory.
6. After performing the commit, go to a “pristine” build-and-test directory (i.e. one where you have a separate CVS checkout in which you have not been editing code) and perform a “cvs update” to pull in the newly committed code.
7. In a build directory, do a completely default build using the code that got updated in 6.
8. Run the regression test suite on this executable. This can be done with the run_xyce_regression script in Xyce_Regression/TestScripts
9. Any new failures might require debugging back in the development directory (i.e. “goto 2”).

The purpose of checking out in a pristine directory when you’re done is to make sure you’ve actually committed all the necessary changes. This is especially important if you’ve added or removed files; if you forgot to issue a `cvs add` command then your code will build properly in your original development directory, but not in your pristine directory. Taking this step assures that you will catch and correct the error before other developers start getting build failures.

Here’s an example shell session for this process:

```
> #The "~/Xyce_Development" directory is our development sandbox
> cd ~/Xyce_Development/Xyce/src/SomePackage/src
> #the point below is that you’re editing some source code
> emacs/vi/gvim/’cat > ’/ed/whatever SomeSource.C
> cd ~/Xyce_Development/build_dirs/some_obscure_build
> ../../Xyce/configure --some-obscure_option --enable-complete_hosing
> make
> run_xyce_regression (see Xyce_Regression/TestScripts)
> cd ~/Xyce_Development/Xyce
> cvs commit
> # now test it in a pristine checkout
> cd ~/Xyce_Build_and_Test/Xyce
> cvs update
> cd ~/Xyce_Build_and_Test/build_dirs/default_build
> ../../Xyce/configure
> run_xyce_regression
```

Again, this process should help ensure the integrity of the repository, provide for fewer nightly test failures and allow the team to continue to perform at a high-efficiency level.

4. Bug Resolution Checklist

Chapter Overview

This chapter contains a checklist for resolving **Xyce** bugs and issues.

4.1 Preface

This document contains a checklist for processing bugs and/or issues for the **Xyce** Parallel Circuit simulator. **Xyce** development issues are tracked using bugzilla. Currently, there are two different bugzilla servers, one on the Sandia Restricted Network (SRN) and the other on the Sandia Open Network (SON):

```
http://charleston.sandia.gov/bugzilla/    SRN
http://joseki.sandia.gov/bugzilla/        SON
```

In general, any bug that contains export-controlled information will be on the SRN version of bugzilla. Most other bugs should go into the SON version.

Bug/issue resolution is a very important part of **Xyce** code development and it is crucial that resolved bugs be well documented and that the resolution of any bug follow a standard process. Doing so helps insure code quality, and it insures that code development is well documented for outside auditors.

4.2 Checklist

This checklist provides guidance on documenting and closing bugs assigned to a developer in Bugzilla. A related flow chart is shown in figure 4.1. The process for fixing and documenting bugs can be thought of as being part of the release process, which is described in chapter 10.7.1.

1. When a bug is entered into bugzilla it will either start out in the `UNCONFIRMED` state, or the `NEW` state. Bugs submitted by customers are usually entered in the `UNCONFIRMED` state, and so the first step is to confirm the bug to change its state to `NEW`. This is usually done by the QA lead. Confirming a bug means that the person doing so actually verifies that the bug exists and is reproducible.
2. If the bug is in the `NEW` state, the QA lead may assign the bug to another developer. Once the bug is assigned, the bug assignee needs to formally “accept” the bug AS SOON AS POSSIBLE. Doing so lets the bug reporter know that the bug has been recognized and acknowledged. Accepting a bug does not necessarily mean that it will be fixed immediately. Instead, it means that the bug has been acknowledged and

is in the system. It is really important that bugs be accepted in a timely manner, as doing so requires very little work.

3. At this stage, many bugs may, for one reason or another, be closable immediately. For example, it may be immediately apparent that the submitted bug is due to a netlisting error rather than a source code error. There are three primary resolutions for bugs of this sort:

   (a) `RESOLVED WONTFIX`: If the issue is a feature request that we have no intention of implementing, or if the reporter has reported a feature as if it were a bug, this is the resolution to use. Generally, setting a bug to this resolution is saying "Yes, we know that, and it's going to stay that way."

   (b) `RESOLVED INVALID`: This resolution is for issues where the user has a netlisting error or other issue that is not actually a bug in Xyce.

   (c) `RESOLVED WORKSFORME`: This signifies that the bug assignee (or QA contact) is unable to reproduce the error reported by the bug reporter.

   If the bug is worthy of one of these summary resolutions, the bug should be marked as such as early as possible. If a bug is marked `WONTFIX`, `INVALID`, or `WORKSFORME`, there should be clear documentation entered into bugzilla as to why it is being marked this way, and proceed to step 8.

4. At this stage (after accepting the bug, and after determining that it is NOT a `RESOLVED WONTFIX`) the bug assignee needs to do the work to fix the issue, as it fits the project schedule and priorities. If possible, the bug assignee should attempt to follow a "test-driven development" model:

   (a) Create a simple test case that reproduces the error reported in the bugzilla item. You will use this test case to verify that your bug is fixed, and will also use it as a regression test to assure that this bug does not get reintroduced by later development. This test case will fail when run with unmodified code.

       If the bug was input by a customer, there may be a customer-submitted circuit that demonstrated the original bug, and sometimes such customer-submitted circuits can be used as certification tests. However, it is usually better practice to simplify customer-submitted test netlists, as they can be unneccessarily complex. Ideally, tests which certify a bug should be the absolute minimum complexity required to demonstrate the bug. Doing this simplification substantially reduces the testing load. Additionally, by simplifying the test circuit, it is much easier to understand the true nature of the bug. Developing a simple, focused test case that targets the bug under consideration saves time and improves code quality in the long run.

(b) Fix the code so that your new test case runs properly.

(c) Verify that the fixed code solves the customer's original problem, too. If you properly diagnosed the problem and developed your test case correctly, this should involve little more than running the fixed code on the customer's problem. If it doesn't solve the customer's problem completely, then you have to diagnose the additional issues, going back to step a.

(d) Add your new test case (or test cases) to the regression suite

In some rare cases, it is not possible to develop a test for a bug. If a test cannot be developed for the bug, then this should be documented in the bugzilla database and commented on in:

```
Xyce_Regression/Xyce_SandiaRegression/Netlists/Certification_Tests/
BUGS_NOT_NEEDING_TEST_CASES
```

then proceeded to step 8.

Note that "it is difficult to produce a test case for this bug" is not a good reason to enter it into BUGS_NOT_NEEDING_TEST_CASES. There are only a few good reasons for a bug to be entered in this file. An example of a valid reason for being there would be when an existing test case begins failing on some platform; the test case for the fix is that the existing test case stops failing, so a new one is not needed. Other bugs that have legitimate reasons to be in the BUGS_NOT_NEEDING_TEST_CASES file are those that changed only some internal handling, and which have no externally verifiable change on Xyce output.

5. If the bug's test case does not involve any export-controlled material, then commit the bug to the Xyce_Regression repository, which resides on the SON. If the test case involves any export controlled material, then it should be placed in the Xyce_SandiaRegression repository instead, which resides on the SRN.

6. Most tests should be included as part of automated valgrind testing. In addition to passing the test as designed, such tests need to also pass valgrind in order to be marked "fixed". To add a test to automated valgrind testing requires that the tag `valgrind` be added to that tests `tags` file. Do not add long, slow tests to valgrind testing.

7. Verify that the new test is run in nightly or weekly testing as appropriate, and ensure that any needed tags are set properly in the test's "tags" file and "options" file. If the "tags" and "options" files are not set up, then a test will be run in all environments. Generally, for a test to be run nightly, it needs to be an inexpensive test that runs in seconds to minutes (preferably seconds). Long tests, or very large tests requiring a lot of memory should be consigned to weekly testing.

8. Comment on this bug in the bugzilla database including reference to any tests that were added to the regression suite. Then close the bug as appropriate, if the test reports (as well as manual testing) indicate that the tests associated with this feature are consistently passing. When you have committed your fixed code to the repository, mark the bug `RESOLVED FIXED`.

9. Run the test suite and make sure that your new test case runs and passes. Wait for an overnight nightly test run and make sure that your test passes on all platforms. At this point you may mark your bug `RESOLVED VERIFIED`.

10. You or the QA lead must now Assign another developer to verify that the test does indeed test the feature in the bug report in an appropriate manner, and does pass on all platforms. This developer should be entered into the bug as the “QA Contact.” When this person performs the verification, he or she should mark the bug `CLOSED`.

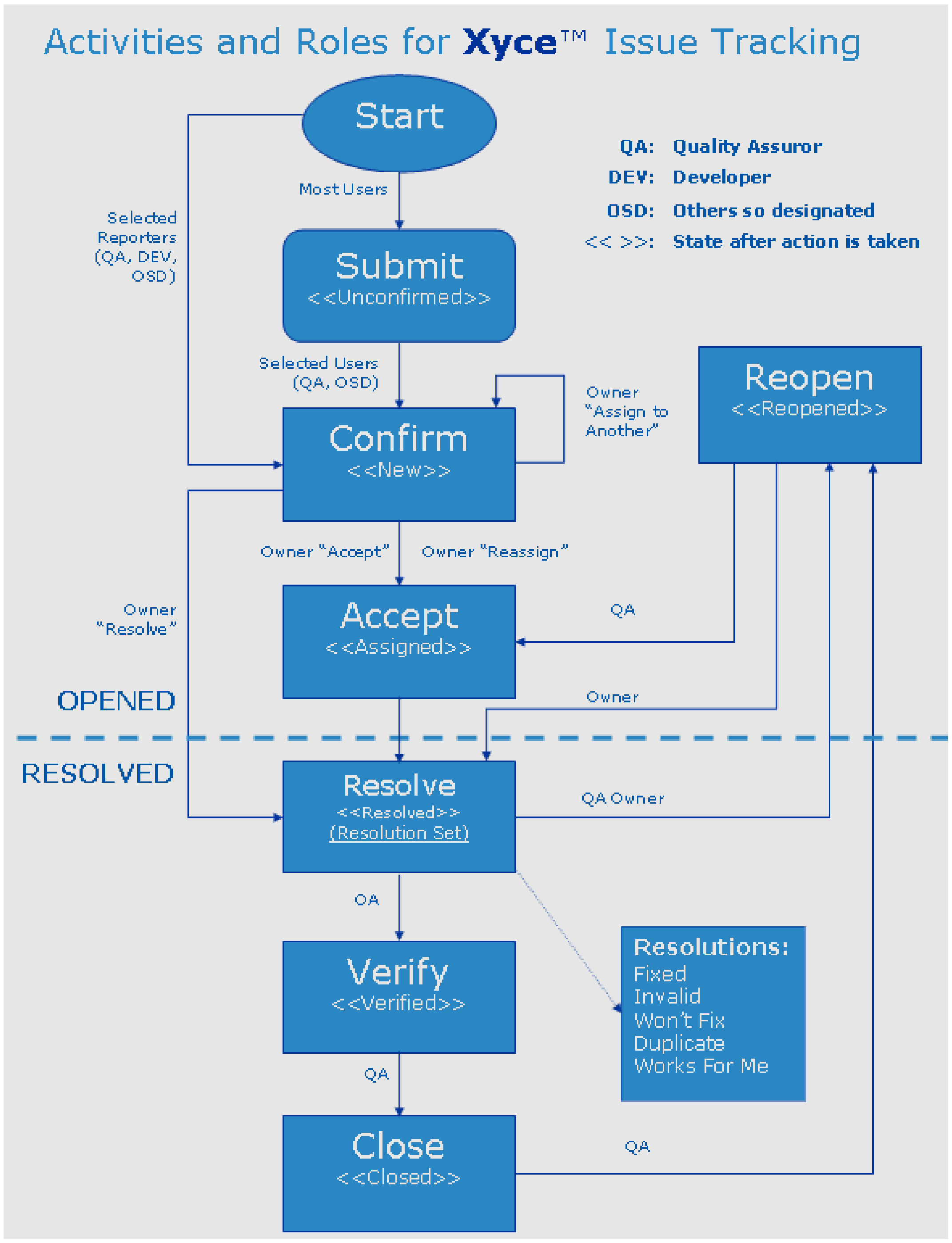

Figure 4.1. Issue Tracking Flowchart.

5. Customer Support

Chapter Overview

This chapter describes activities related to customer support. It includes a checklist that should be followed by **Xyce** developers.

5.1 Guidelines

This checklist provides guidance on collecting customer data (i.e. feedback and future requirements) and using that data to guide **Xyce** development.

1. Collect the customer's request through direct consultation with the customer. Detail is important at this stage. If the customer is reporting a bug, then try to gather circuits that demonstrate the bug. If the customer is requesting a new feature, device model, analysis type, then try to understand how or the customer plans to use this feature.

2. Document the customer's request. Since **Xyce** is a multi-developer project, the best avenue to documenting a bug or new development request is to enter the request into Bugzilla (charleston.sandia.gov/bugzilla). Despite the name, Bugzilla is used to track the development of new features and reported bugs. In the new Bugzilla entry, describe what the customer's requirements are and add in any deadlines or time constraints if they are known. You may also include the customer in the `cc` list so that they are updated on the status of this issue as it progresses. Also, any circuit files, meeting notes, diagrams etc. can be attached to the Bugzilla entry to further document the request.

3. If Bugzilla is not an appropriate means to document the request – for example, the development goals are more research focused and may change, then the customer's input should still be recorded. In such cases, a Xyce developer may attend project meetings or have one to one meetings with the customer to better define Xyce's role in a project. Here, meetings notes maintained in a notebook should document the customer's needs. If or when customer requirements solidify then they can be entered into Bugzilla.

4. Once a change or new feature is implemented for a customer, it is important for the Xyce developer to ensure that the new feature works as the customer desired. If the feature has been recorded in bugzilla (and most should be), the "bug checklist" given in chapter 4 should be followed, which should include certification tests which prove that the feature is implemented and working correctly.

5. Continued interaction and feedback with the customer should continue after feature requests have been implemented, to insure that the customer is satisfied with the result. If not, then the Xyce developer should repeat the above process.

6. Xyce C++ Style Guidelines

Chapter Overview

This chapter contains C++ coding guidelines to be used by **Xyce** developers. These guidelines should be followed in the creation of **Xyce** source code.

6.1 Preface

This chapter is intended to document the C++ coding style used in source code level development of the **Xyce** parallel circuit simulator. While there are many references, this chapter has primarily been inspired by reference [3].

Unfortunately coding standards and guidelines can often start ”religious wars” among code developers. However, setting consistent standards is critical in production codes, which can often have many different developers. As of this writing, **Xyce** has had over 20 unique developers, and will likely have more in the future.

Most of the code guidelines in this chapter were developed at the beginning of the **Xyce** project in 1999, and were based on commonly used style guidelines of the time. Unfortunately, the original **Xyce** style guide no longer exists, and we have had to reverse engineer this one, so it may not map perfectly to the original.

Long-term **Xyce** developers have usually followed these style guidelines closely. In general, if you follow the rule that you code to the pre-existing style of the file you are working in, you will wind up following much of what is documented here. As **Xyce** is a mature project, most developers will tend to work in files that are old, and already have a pre-existing style to them.

Note that when a code team adopts a coding style, the important thing is not that it perfectly please all the developers (as that would be impossible), but rather that it be something everyone is willing to live with and actually practice. Ultimately, the goal is for any **Xyce** developer to be able to open any **Xyce** source file, and have it look familiar.

In general keep please the following in mind:

- This is a group project. As such, at some point, someone other than yourself will need to understand, and possible debug or rewrite, code that you have written. This is true of everyone who writes source code in **Xyce**.
- Code refactors, new solvers, etc., are a fact of life on a large project. So is staff turnover.
- It is your responsibility to produce source code that is clear and comprehensible to other developers.

6.2 References

Good books to use for C++ style (and other technical guidance) include the following Scott Meyers books: “Effective C++” [4], “More Effective C++” [5] and “Effective STL” [6]. Additionally for more style suggestions, see [7], [8]. [9], and [10],

6.3 Naming convention

6.3.1 Variable names must be in mixed case starting with lower case.

For example:

line, savingsAccount

This is common practice in the C++ development community, and makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration Line line;

6.3.2 Named constants (including enumeration values) must be all uppercase using underscore to separate words.

For example:

MAX_ITERATIONS, COLOR_RED, PI

This is common practice in the C++ development community.

6.3.3 Avoid Using Preprocessor Constants. Use Enumerated types instead

In general, the use of preprocessor constants should be minimized, as they are invisible in a debugger. When possible, enumerated types instead.

6.3.4 Names representing methods or functions must be verbs and written in mixed case starting with lower case.

For example:

getName(), computeTotalWidth()

This is common practice in the C++ development community. This is identical to variable names, but functions in C++ are already distinguishable from variables by their specific form.

6.3.5 Private class variables and functions should have underscore suffix.

For example:

```
class N_DEV_Resistor
{
  public:
    N_DEV_Resistor (); // constructor
    virtual ~N_DEV_Resistor (); // destructor

  private:
    void calculateResistance_ ();

  public:
    string name;

  private:
    double resistance_;
}
```

Apart from its name and its type, the scope of a variable is its most important feature. Indicating class scope by using underscore makes it easy to distinguish class variables from local scratch variables. This is important because class variables are considered to have higher significance than method variables, and should be treated with special care by the programmer. A side effect of the underscore naming convention is that it nicely resolves the problem of finding reasonable variable names for setter methods and constructors:

```
void setDepth (int depth)
{
  depth_ = depth;
}
```

It should be noted that scope identification (via underscores) in variables has been a controversial issue for quite some time. It seems, though, that this practice now is gaining acceptance and that it is becoming more and more common as a convention in the professional development community.

6.3.6 All names should be written in English.

For example:

```
fileName; // NOT: filNavn
```

English is the preferred language for international development.

6.3.7 Variables with a large scope should have long names, variables with a small scope can have short names.

Never, ever use single characters for variable names, unless they are very local indexing integers isolated to a few lines of code. This is extremely important, as single character names are almost impossible to search for in a large file.

In general, always ask the question, “what would happen if I grep’d for this variable?”. If grepping for a variable yields most of the file, then you need a more unique name.

6.3.8 The name of the object is implicit, and should be avoided in a method name.

For example:

```
line.getLength(); // NOT: line.getLineLength();
```

The latter seems natural in the class declaration, but proves superfluous in use, as shown in the example.

6.3.9 The terms get/set must be used where an attribute is accessed directly.

For example:

```
employee.getName();              // NOT: employee.pleaseGiveMeTheName();
employee.setName(name);          // NOT: employee.changeName(name);
matrix.getElement(2, 4);         // NOT: matrix.retrieveElement(2,4);
matrix.setElement(2, 4, value); // NOT: matrix.saveElement(2,4);
```

This is common practice in the C++ development community. In Java this convention has become more or less standard.

6.3.10 The term compute can be used in methods where something is computed

For example:

```
valueSet->computeAverage();
matrix->computeInverse()
```

Give the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.

6.3.11 The term find can be used in methods where something is looked up

For example:

```
vertex.findNearestVertex();
matrix.findMinElement();
```

Give the reader the immediate clue that this is a simple look up method with a minimum of computations involved. Consistent use of the term enhances readability.

6.3.12 The term initialize can be used where an object or a concept is established.

For example:

```
printer.initializeFontSet();
```

The American initialize should be preferred over the English initialise. Abbreviation init should be avoided. For a related issue, see section 6.7.

6.3.13 Plural form should be used on names representing a collection of objects.

For example:

```
vector <Point> points;
```

Enhances readability since the name gives the user an immediate clue of the type of the variable and the operations that can be performed on its elements. Additionally, the plural form should be avoided when the name does not represent a collection of objects.

6.3.14 The prefix “is” should be used for boolean variables and methods.

For example:

```
isSet, isVisible, isFinished, isFound, isOpen
```

This is common practice in the C++ development community and partially enforced in Java. Using the is prefix solves a common problem of choosing bad boolean names like status or flag. isStatus or isFlag simply doesn't fit, and the programmer is forced to choose more meaningful names. There are a few alternatives to the is prefix that fits better in some situations. These are the has, can and should prefixes:

```
    bool hasLicense();
    bool canEvaluate();
    bool shouldSort();
```

6.3.15 Complement names must be used for complement operations

For example:

```
get/set
add/remove
create/destroy
start/stop
insert/delete
increment/decrement
old/new
begin/end
first/last
up/down
min/max
next/previous,
old/new
open/close
show/hide
suspend/resume
etc.
```

Reduce complexity by symmetry.

6.3.16 Abbreviations in names should be avoided, except when the word is primarily known that way

For example:

```
computeAverage(); // NOT: compAvg();
```

There are two types of words to consider. First are the common words listed in a language dictionary. For example, always write:

command	instead of	**cmd**
copy	instead of	**cp**
point	instead of	**pt**
compute	instead of	**comp**
initialize	instead of	**init**

etc.

Then there are domain specific phrases that are more naturally known through their abbreviations/acronym. These phrases should be kept abbreviated. For example, always write:

html	instead of	**HypertextMarkupLanguage**
cpu	instead of	**CentralProcessingUnit**

etc.

6.3.17 Negated boolean variable names must be avoided

For example:

```
bool isError;   // NOT: isNoError
bool isFound;   // NOT: isNotFound

if (!isError)   // easy to understand
if (!isNotError) // less easy to understand
```

The problem arises when such a name is used in conjunction with the logical negation operator as this results in a double negative. It is not immediately apparent what !isNotFound means.

6.3.18 Functions (methods returning something) should be named after what they return and procedures (void methods) after what they do

This increases readability, and makes it clear what the unit should do and especially all the things it is not supposed to do. This again makes it easier to keep the code clean of side effects.

6.4 File, Function and Class Headers

Xyce uses standard headers for files, functions and classes. These must be used consistently, and the various metadata fields (creator, date, etc.) need to be correctly filled out. These can be found in the source code repository, in Xyce/src/headers. Examples of each are in the following subsections.

6.4.1 C++ File Header

The file header is particularly important, as it contains fields that are modified by CVS. These fields are “Revision Number”, “Revision Date”, and “Current Owner”. It is not necessary to change these fields manually, as CVS will do this automatically.

```
//-----------------------------------------------------------------------------
// Copyright Notice
//
// Copyright (c) 2000, Sandia Corporation, Albuquerque, NM.
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Filename       : $RCSfile$
//
// Purpose        : Describe the purpose of the contents of the file.
//                  If the contains the header file of a class,
//                  provide a clear description of the nature of
//                  the class.
//
// Special Notes  : Specify any "hidden" or subtle details of the
//                  class here. Portability details, error handling
//                  information, etc.
//
// Creator        : {Name of File Creator}, {Affiliation}
//
// Creation Date  : {mm/dd/yy}
//
// Revision Information:
// ---------------------
//
// Revision Number: $Revision$
//
// Revision Date  : $Date$
//
// Current Owner  : $Author$
//-----------------------------------------------------------------------------
```

6.4.2 Function Header

```
//------------------------------------------------------------------------
// Function      : N_DEV_CCCS::factory
// Purpose       :
// Special Notes :
// Scope         : public
// Creator       : {Name of Function Creator}, {Affiliation}
// Creation Date : {mm/dd/yy}
//------------------------------------------------------------------------
```

6.4.3 Class Header

```
//------------------------------------------------------------------------
// Class         : N_DEV_AsrcInstance
// Purpose       :
// Special Notes :
// Creator       : {Name of Class Creator}, {Affiliation}
// Creation Date : {mm/dd/yy}
//------------------------------------------------------------------------
```

6.5 Forward Declarations

Use forward declarations as much as possible. This will reduce compile times and dependencies between files. For a full explanation of this, see Scott Meyers' books [4], [5]. In general, if a header file only contains pointers or references to a class, rather than an explicit declaration, then the header only needs minimal information and can be forward declared. For example, if you have a pointer to the class "myClass" in a header file then use:

```
class myClass;
```

instead of

```
#include<myClass.h>
```

in the header file. The include statement will still be needed in whichever source file the pointer gets allocated.

6.5.1 iostream Forward Declarations

There is a special case of forward declaration, for the C++ standard library IO stream classes. To forward declare, use the alternate include statement:

```
#include <iosfwd>
```

instead of:

```
#include <iostream>
```

6.6 Pointers vs. References

Marshall Cline: “Use references when you can, and pointers when you have to.” Frequently Asked Question 8.6 in [11].

In general, if it is possible to use a reference instead of a pointer, then it is always the better choice, and will result in code that is more memory-safe. (references cannot be deallocated or reassigned) For a complete discussion of this, see Scott Meyers’ books [4], [5], as well as Marshall Cline’s C++ FAQs [12].

6.7 Late class/package initialization is very important!

In a complex code it is best to avoid initializations in class constructors. This is because many classes are interdependent and it is easy to have circular dependencies. Also, classes are not always guaranteed to be allocated in a particular order. So, it is best to delay initialization of member objects until later, when the entire code is ready to do so. In **Xyce**, you can see this in the functions N_CIR_Xyce::doAllocations and N_CIR_Xyce::doInitializations.

6.8 Avoid C-style raw arrays. Use the Standard Library instead

The C++ standard library (formerly called the standard template library, or STL) supports many types of container objects such as stl::vector, stl::list, stl::map, etc. Each of these is much safer and more convenient to use than raw arrays. So, for example use:

```
vector<double> stlExampleVector;
stlExampleVector.resize(100,0.0);
```

instead of:

```
double rawArrayExample[100];
```

In general raw arrays are much less safe and will lead to code that is more fragile than if you use STL equivalents.

6.9 Avoid Raw Pointers

Raw pointers can be dangerous. Avoid these as much as possible, by either using STL, reference-counted pointers (RCPs), or both in combination.

Occasionally, when using legacy APIs, it may seem that raw pointers are necessary to support the API function calls. However, there is nearly always a workaround. Some example workarounds are given in the following subsections.

Note, for efficiency purposes, the Xyce device package (as of 2009) now uses a lot of raw pointer-based loads. It is easier to optimize cache memory usage this way. However, all the raw pointers are extracted from STL and Epetra objects to maintain memory safety. This is one exception where raw pointers can be OK.

6.9.1 Interfacing with legacy code that requires raw pointers

It is common for a legacy library (particularly one written in C) to expect function arguments that are raw pointers to doubles or chars. Using STL libraries, you can always extract the raw pointer from the STL object.

String Example

```
string name("eric");

// in this case the legacy library wants something like:
// doSomethingLegacyLibrary(char *, int size);

doSomethingLegacyLibrary(name.c_str(), name.size());
```

Double-precision array example

```
vector<double> exampleVector(100,1.0);

// in this case the legacy API wants this:
//doSomethingLegacyLibrary(const double*, int size);

doSomethingLegacyLibrary(  &(exampleVector[0]), exampleVector.size()  );
```

This is described in some detail in Meyers [4].

6.9.2 RCPs

Reference counted pointers (also known as smart pointers) should be used when two classes have a persisting relationship with each other with respect to the data being passed as a pointer. If two classes do not have a persisting relationship, it is much better to use a reference in the argument list. **Xyce** uses the Trilinos Project's Teuchos::RCP class for reference counted pointers. The main advantage to using reference counted pointers is that each pointer keeps track of how many references there are to it. When that reference count goes to zero, it deletes itself, thereby preserving memory and removing deletes from the code. It is a form of automatic memory management and it avoids many of the common memory-leak mistakes in writing C++ code. Reference counted pointers are also very useful when creating new objects since you don't have to figure out when to delete them, they take care of themselves. The following examples demontrate its use.

RCP Example

```
#include <Teuchos_RCPDecl.hpp>
using Teuchos::RCP;
using Teuchos::rcp;

// Declare the type for arrayPtr as a Reference Counted Pointer.
RCP< std::vector<double> > arrayRCPtr;

// Wrap a "new" call with rcp to safely handle memory.
arrayRCPtr = rcp( new std::vector<double> );

// Access the underlying object as normal for a pointer.
int size = arrayRCPtr->size();

// Pass them to functions by copying, just like pointers
int result = someFunction(arrayRCPtr);

// Get the underlying pointer out:
std::vector<double>* arrayPtr = arrayRCPtr.get();

// Check if a RCP is null:
if ( arrayRCPtr == Teuchos::null) {
  std::cout << "Error, arrayRCPtr is null!" << std::endl;;
}

// Dereference a refcounted pointer just like normal:
int result = someOtherFunction(*arrayRCPtr);

// Create a reference to the underlying object just like normal:
std::vector<double>& array = *arrayRCPtr;

// If you have a raw pointer and you need to pass it to a function
// that takes a RCP, but you don't want the object to get
// deleted, then use the extra "false" argument to rcp which tells
// it to leave the pointer alone when the reference count goes to
// zero:
N_LAS_Vector* tempVectorPtr;
tempVectorPtr = rcp( tempVectorPtr, false);
int result = someNewFunction(tempVectorPtr);

// If you need to pass a raw pointer to a function, the syntax gets a
// little different.  The "*" de-references the pointer and the "&"
// gets its address.
int result = someOldFunction(&*arrayRCPtr);

// One of the main advantages of reference counted pointers is in the
// use of factories.  These are classes or functions that create new
// objects for you based on parameter lists or other input
// specifications.  They are particularly useful for complex objects.
// In this case, you have allocated new memory for the new object,
// but you don't have to delete it because the reference counted
// pointer will take care of that for you.
RCP< ComplexObject > complexObjectPtr =
  createNewComplexObject(paramList);
```

6.10 Files

6.10.1 C++ header files should have the extension .h. Source files should have the extension .C

```
MyClass.C, MyClass.h
```

6.10.2 A class should be declared in a header file and defined in a source file where the name of the files match the name of the class

```
MyClass.h, MyClass.C
```

This makes it easy to find the associated files of a given class. An obvious exception is template classes that must be both declared and defined inside a .h file.

6.10.3 All definitions should reside in source files

```
class MyClass

  public:
    int getValue () return value_; // NO! ...

  private:
    int value_;
```

The header files should declare an interface, the source file should implement it. When looking for an implementation, the programmer should always know that it is found in the source file. The obvious exception to this rule is of course inline functions that must be defined in the header file.

6.10.4 File content must be kept within 80 columns

Source file text must be within a width of 80 characters, with only very rare exceptions. Most terminal windows default to this width, and 80 columns is a common dimension for editors, terminal emulators, printers and debuggers, and files that are shared between several people should keep within these constraints. It improves readability when unintentional line breaks are avoided when passing a file between programmers.

6.10.5 Never Use Tabs

There should be no tabs used, because many editors assume different indent levels for tabs. All spaces should be true spaces. Editors (like Vim and emacs) can be configured to automatically do this. Vim and emacs configurations are described in the next two sections.

Vim editor configuration

The "vim" editor is an update to the standard Unix "vi" editor, and stands for "vi improved". It is backward compatible with "vi" and has a lot of useful extensions. Most Unix and Linux machines these days will have "vim" instead of "vim", and the /usr/bin/vi command is usually softlinked to an installation of "vim". It is rare to find the original "vi" anymore.

If you use the "vim" editor, you can enforce tabbing by doing the following [13], [14]. Add the following commands to your .vimrc file:

```
set expandtab
set tabstop=2
set shiftwidth=2
```

The first command (expandtab) will automatically force any uses of the "tab" key to insert spaces instead. The second command (tabstop) will set the number of spaces used for each tab. The last command (shiftwidth) will force the editor to automatically indent to two characters.

emacs editor configuration

Emacs has "C" and "C++" modes that automatically enable certain behaviors when emacs detects you're editing files of that type. Unfortunately, by default it uses GNU coding standards that are not entirely compatible with the Xyce style. The most obvious difference between Xyce and GNU coding standards is that GNU standards indent the opening curly braces of a block, and further indent the contents of the block. This behavior is easily set to the Xyce style by addition of a few lines to the file ".emacs" in your home directory.

```
(require 'cc-mode)
(c-set-offset 'substatement-open 0)
(c-set-offset 'arglist-intro 3)
(setq-default indent-tabs-mode nil)
```

The first guarantees that the "C" mode is loaded upon launching emacs (instead of on demand), and the subsequent lines change internal parameters of the mode. The "substatement-open" line is the one that disables indentation of the curly braces. The "arglist-intro" line sets the indentation of the first line of argument lists of function calls, and the last disables use of tabs for indentation. See the documentation for "CC mode" in the Emacs Info tool (Control-H i to enter Info, then `m cc<tab><enter>` to locate and select the "CC Mode" info).

6.10.6 The incompleteness of split lines must be made obvious

```
totalSum = a + b + c +
           d + e;

function (param1, param2,
          param3);

setText ("Long line split"
         "into two parts.");

for (int tableNo = 0; tableNo < nTables;
     tableNo += tableStep)
{
  ...
}
```

Split lines occurs when a statement exceed the 80 column limit given above. It is difficult to give rigid rules for how lines should be split, but the examples above should give a general hint. In general:

- Break after a comma.
- Break after an operator.
- Align the new line with the beginning of the expression on the previous line.

6.11 Include Files and Include Statements

6.11.1 Header files must contain an include guard

```
#ifndef COM_COMPANY_MODULE_CLASSNAME_H
#define COM_COMPANY_MODULE_CLASSNAME_H
...
#endif
```

Here is a specific example:

```
#ifndef Xyce_N_DEV_Resistor_h
#define Xyce_N_DEV_Resistor_h

....

#endif
```

The construction is to avoid compilation errors. The name convention resembles the location of the file inside the source tree and prevents naming conflicts.

6.11.2 Include statements should be sorted and grouped

Sorted by their hierarchical position in the system with low level files (standard includes) included first. Leave an empty line between groups of include statements. If this is a header file, then have the foward declarations last. Each block should have comments to label them (Standard Includes, Xyce Includes, Forward Declarations).

```
// ---------- Standard Includes ----------
#include <vector>
#include <string>
#include <map>
#include <set>
```

```
// ----------   Xyce Includes   ----------
#include <N_UTL_Misc.h>

#include <N_UTL_OptionBlock.h>

#include <N_IO_PkgOptionsMgr.h>

#include <N_DEV_Device.h>
#include <N_DEV_SolverState.h>
#include <N_DEV_ExternData.h>
#include <N_DEV_DeviceOptions.h>

...

// ---------- Forward Declarations ----------
class N_DEV_DeviceBuilder;
class N_DEV_DeviceInstance;
class N_DEV_ModelBlock;
class N_DEV_InstanceBlock;

class N_UTL_OptionBlock;
...
```

In addition to showing the reader the individual include files, this also gives an immediate clue about the modules that are involved. Include file paths must never be absolute. Compiler directives should instead be used to indicate root directories for includes.

6.11.3 Include statements must be located at the top of a file only

This is common practice in C and C++ codes. Avoid unwanted compilation side effects by ”hidden” include statements deep into a source file. Also, include statements should only be for true header files. (ie, NEVER include a *.C file).

6.12 Types

6.12.1 Types that are local to one file only can be declared inside that file

This enforces information hiding.

6.12.2 The parts of a class must be sorted public, protected and private

All sections must be identified explicitly. Not applicable sections should be left out.

The ordering is ”most public first” so people who only wish to use the class can stop reading when they reach the protected/private sections.

6.12.3 Type conversions must always be done explicitly: Use C++-style Casts

C++ has many options for casting, more so than C. dynamic_cast, static_cast, etc. These can be necessary to convert (for example) between pointers to base class objects to derived class objects, or from one derived object to another. However, casting is also used for much less exotic purposes as well, such as converting a double to an int, or an int to a char.

When converting (for example) from a double to an int, use this format:

```
double doubleExample = 10.0;
int intExample = static_cast<int> (doubleExample);
```

Do NOT do a C-style cast:

```
double doubleExample = 10.0;
int intExample = (int) (doubleExample);
```

Also do NOT do this (a fully implicit cast):

```
double doubleExample = 10.0;
int intExample = doubleExample;
```

The main reason for avoiding C-style casts is that they are hard to search for and/or grep. Also, using C++ style casting helps reinforce the other types of casting available in C++, which are very powerful. Also, implicit casting makes it very easy to not notice that you are even doing casting at all, which can lead to errors. By using C++ casting, the programmer indicates that he is aware of the different types involved and that the mix is intentional. This is covered in a lot of detail in Meyers [4].

It is possible to using the gcc compiler warnings to prevent the use of C-style casts. Simply add -Wold-style-cast to CXXFLAGS, and the compiler will catch all instances of old style casts.

6.13 Variables

6.13.1 Variables should be initialized where they are declared

This ensures that variables are valid at any time. Sometimes it is impossible to initialize a variable to a valid value where it is declared:

```
int x, y, z;
getCenter(&x, &y, &z);
```

In these cases it should be left uninitialized rather than initialized to some phony value.

6.13.2 Variables must never have dual meaning

Enhance readability by ensuring all concepts are represented uniquely. Reduce chance of error by side effects.

6.13.3 Use of global variables should be minimized

In C++ there is no reason global variables need to be used at all. The same is true for global functions or file scope (static) variables.

6.13.4 Class variables should never be declared public

The concept of C++ information hiding and encapsulation is violated by public variables. Use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C struct). In this case it is appropriate to make the class' instance variables public.

Note that structs are kept in C++ for compatibility with C only, and avoiding them increases the readability of the code by reducing the number of constructs used. Use a class instead.

6.13.5 Implicit test for 0 should not be used other than for boolean variables and pointers

```
if (nLines != 0) // NOT: if (nLines)
if (value != 0.0) // NOT: if (value)
```

It is not necessarily defined by the C++ standard that ints and floats 0 are implemented as binary 0. Also, by using explicit test the statement give immediate clue of the type being tested.

It is common also to suggest that pointers shouldn't test implicit for 0 either, i.e. if (line == 0) instead of if (line), but the latter is regarded so common in C/C++ that it can be used.

6.13.6 Variables should be declared in the smallest scope possible

Keeping the operations on a variable within a small scope, it is easier to control the effects and side effects of the variable.

6.14 Loops

6.14.1 Only loop control statements must be included in the for() construction

This is correct:

```
sum = 0;
for (i = 0; i < 100; i++)
{
  sum += value[i];
}
```

Initializing the "sum" variable in the for() construct, as below, is incorrect:

```
for (i = 0, sum = 0; i < 100; i++)
{
  sum += value[i];
}
```

Increase maintainability and readability. Make a clear distinction of what controls and what is contained in the loop.

6.14.2 Loop variables should be initialized immediately before the loop

Readability is enhanced when the initialization of loop variables occurs immediately before the loop rather than at the top of a function.

This is desirable:

```
isDone = false;
while (!isDone)
{
  [...]
}
```

and this is less readable:

```
[... block of declarations ...]
bool isDone = false;
[... more declarations ...]
[... many lines of code ...]
while (!isDone)
{
  [...]
}
```

6.14.3 do-while loops can be avoided

Do-while loops are less readable than ordinary while loops and for loops since the conditional is at the bottom of the loop. The reader must scan the entire loop in order to understand the scope of the loop. In addition, do-while loops are not needed. Any do-while loop can easily be rewritten into a while loop or a for loop. Reducing the number of constructs used enhance readability.

6.15 Conditionals

6.15.1 Complex conditional expressions must be avoided

Introduce temporary boolean variables instead.

```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry)
{
  [...]
}
```

instead of

```
if ((elementNo < 0) || (elementNo > maxElement)|| elementNo == lastElement)
{
  [...]
}
```

By assigning boolean variables to expressions, the program gets automatic documentation. The construction will be easier to read, debug and maintain.

6.15.2 The nominal case should be put in the if-part and the exception in the else-part of an if statement

```
bool isOk = readFile (fileName);
if (isOk)
{
  [...]
}
else
{
  [...]
}
```

Makes sure that the exceptions don't obscure the normal path of execution. This is important for both the readability and performance.

6.15.3 The conditional should be put on a separate line

Place the "if" part of a conditional on a separate line from the statements to be executed when the condition is true, and always use braces to mark the block of executable code even if it is a single line:

```
if (isDone)
{
  doCleanup();
}
```

Do not compress the conditional into a single line, as below:

```
if (isDone) doCleanup();
```

Doing so decreases clarity. When writing on a single line, it is not apparent whether the test is really true or not.

6.15.4 Executable statements in conditionals must be avoided

Executable statements should not be placed in the conditional of an "if" statement. A correct usage would be:

```
File* fileHandle = open(fileName, "w");
if (!fileHandle)
{
  [...]
}
```

while the following incorrectly puts the "open" function call into the conditional:

```
if (!(fileHandle = open(fileName, "w")))
{
  [...]
}
```

Conditionals with executable statements are very difficult to read. This is especially true for programmers new to C/C++. Additionally, this makes stepping through code in debuggers such as gdb more difficult.

6.16 Miscellaneous

6.16.1 The use of magic numbers in the code should be avoided

Numbers other than 0 and 1 should be considered declared as named constants instead. If the number does not have an obvious meaning by itself, the readability is enhanced by introducing a named constant instead. A different approach is to introduce a method from which the constant can be accessed.

6.16.2 Floating point constants should always be written with decimal point and at least one decimal

```
double total = 0.0;   // NOT: double total = 0;
double speed = 3.0e8; // NOT: double speed = 3e8;
double sum; : sum = (a + b) * 10.0;
```

This emphasizes the different nature of integer and floating point numbers. Mathematically the two model completely different and non-compatible concepts. Also, as in the last example above, it emphasizes the type of the assigned variable (sum) at a point in the code where this might not be evident.

6.16.3 Floating point constants should always be written with a digit before the decimal point

Floating point constants should always have a digit before the decimal point:

```
double total = 0.5; // NOT: double total = .5;
```

The number and expression system in C++ is borrowed from mathematics and one should adhere to mathematical conventions for syntax wherever possible. Also, 0.5 is a lot more readable than .5; There is no way it can be mistaken for the integer 5.

6.16.4 Functions must always have the return type explicitly listed.

Always declare the return type of a function explicitly:

```
int getValue()
{
  [... function definition ...]
  return someValue;
}
```

NOT

```
getValue()
{
  [...]
  return someValue;
}
```

If not explicitly listed, C++ implies int return type for functions. A programmer must never rely on this feature, since this might be confusing for programmers not aware of this artifact.

6.16.5 goto should not be used

Goto statements violates the idea of structured code. Only in some very rare cases (for instance breaking out of deeply nested structures) should goto be considered, and then only if the alternative structured counterpart is proven to be less readable.

6.16.6 “0” should be used instead of “NULL”

NULL is part of the standard C library, but is made obsolete in C++.

6.17 Layout

6.17.1 Basic Indentation Length should be Two Spaces

```
for (i = 0; i < nElements; i++)
{
  a[i] = 0;
}
```

Indentation of 1 is too small to emphasize the logical layout of the code. Indentation larger than 4 makes deeply nested code difficult to read and increase the chance that the lines must be split. Choosing between indentation of 2, 3 and 4, 2 and 4 are the more common, and 2 has been chosen to reduce the chance of splitting code lines.

6.17.2 Block Layout: Placement of braces (curly brackets)

In **Xyce** the preferred braces placement is the “ANSI C++ style”:

```
if (a $>$ 5)
{
  // This is ANSI C++ style, which Xyce uses.
}

if (a $>$ 5) {
   // This is K&R style  (not used by Xyce)
}

if (a $>$ 5)
  {
    // This is GNU style  (not used by Xyce)
  }
```

Most of the **Xyce** source complies with the ANSI C++ style.

6.17.3 The class declarations should have the following form

In particular, the public, protected and private data and functions should be in separate blocks, with the most public members first.

```
class SomeClass : public BaseClass
{
  public:
    ...
  protected:
    ...
  private:
    ...
}
```

This follows partly from the general block rule above, as well as the sorting rule for public, protected, and private data 6.12.2.

6.17.4 Method definitions should have the following form

```
void someMethod()
{
   ...
}
```

This follows from the general block rule above.

6.17.5 The if-else class of statements should have the following form

```
if (condition)
{
  statements;
}
```

```
if (condition)
{
  statements;
}
else
{
  statements;
}

if (condition)
{
  statements;
}
else if (condition)
{
  statements;
}
else
{
  statements;
}
```

This follows partly from the general block rule above. The else statement is put on a separate line from the left curly bracket, This should make it easier to manipulate the statement, for instance when moving else clauses around.

6.17.6 A for statement should have the following form

```
for (initialization; condition; update)
{
  statements;
}
```

This follows from the general block rule above.

6.17.7 A while statement should have the following form

```
while (condition)
```

```
{
  statements;
}
```

This follows from the general block rule above.

6.17.8 A do-while statement should have the following form

Note that do-while statements are discouraged in general.

```
do
{
  statements;
} while (condition);
```

This follows from the general block rule above.

6.17.9 A switch statement should have the following form

```
switch (condition)
{
  case ABC :
    statements; // Fallthrough
  case DEF :
    statements;
    break;
  case XYZ :
    statements;
    break;
  default :
    statements;
    break;
}
```

Note that each case keyword is indented relative to the switch statement as a whole. This makes the entire switch statement stand out. Note also the extra space before the : character. The explicit Fallthrough comment should be included whenever there is a case statement without a break statement. Leaving the break out is a common error, and it must be made clear that it is intentional when it is not there.

6.18 White Space

6.18.1 Miscellaneous

- Conventional operators should be surrounded by a space character.
- C++ reserved words should be followed by a white space.
- Commas should be followed by a white space.
- Colons should be surrounded by white space.
- Semicolons in for statements should be followed by a space character.

```
a = (b + c) * d; // NOT: a=(b+c)*d

while (true) // NOT: while(true)
{
  ...
}

doSomething(a, b, c, d); // NOT: doSomething(a,b,c,d);

case 100 : // NOT: case 100:

for (i = 0; i < 10; i++) // NOT: for(i=0;i<10;i++)
```

This makes the individual components of the statements stand out, and enhances readability. It is difficult to give a complete list of the suggested use of whitespace in C++ code. However, the examples above should give a general idea.

6.18.2 Method names can be followed by a white space when it is followed by another name

```
doSomething (currentFile);
```

This makes the individual names stand out and enhances readability. When no name follows, the space can be omitted (doSomething()) since there is no doubt about the name in this case. An alternative to this approach is to require a space after the opening parenthesis. Those that adhere to this standard usually also leave a space before the closing parentheses: doSomething(currentFile);. This do make the individual names stand out as is the intention, but the space before the closing parenthesis is rather artificial, and without this space the statement looks rather asymmetrical (doSomething(currentFile);).

6.18.3 Logical units within a block should be separated by one blank line

```
Matrix4x4 matrix = new Matrix4x4();

double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

matrix.setElement(1, 1, cosAngle);
matrix.setElement(1, 2, sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2, cosAngle);

multiply(matrix);
```

Enhance readability by introducing white space between logical units of a block.

6.18.4 Methods should be separated by three blank lines

By making the space larger than space within a method, the methods will stand out within the class.

6.18.5 Variables in declarations can be left aligned

```
AsciiFile* file;
int        nPoints;
float      x, y;
```

This enhances readability. The variables are easier to spot from the types by alignment.

6.18.6 Use alignment wherever it enhances readability

```
value = (potential * oilDensity) / constant1 +
        (depth * waterDensity) / constant2 +
        (zCoordinateValue * gasDensity) / constant3;

minPosition     = computeDistance(min, x, y, z);
averagePosition = computeDistance(average, x, y, z);
```

There are a number of places in the code where white space can be included to enhance readability even if this violates common guidelines. Many of these cases have to do with code alignment. General guidelines on code alignment are difficult to give, but the examples above should give a general clue.

6.19 Comments

6.19.1 Tricky code should not be commented but rewritten!

In general, the use of comments should be minimized by making the code self- documenting by appropriate name choices and an explicit logical structure.

6.19.2 All comments should be written in English

In an international environment English is the preferred language.

6.19.3 Use // for all comments, including multi-line comments

C-style comments (/* */) should not be used.

```
// Comment spanning
// more than one line.
```

There should be a space between the ”//” and the actual comment, and comments should always start with an upper case letter and end with a period.

7. Matrix and Vector Access

Chapter Overview

This chapter is intended to provide a description of the vector and matrix indexing that is used by **Xyce**. This is required information for developers responsible for implementing new devices in **Xyce**. The primary responsibility of a device (such as a resistor or diode) in **Xyce** is to sum numerical entries into the Jacobian matrix, residual vector, and related linear algebra entities. To perform this summation, each device must access the Jacobian matrix, and other entities, through an abstract interface. This abstract interface will be described in this chapter. Additionally, to set context, some explanation of the general design of **Xyce** will be included.

7.1 History and Motivation

Xyce is a circuit simulator, similar to other circuit simulator codes like Spice3f5 [15]. **Xyce**, however, is unique in that it is designed from the ground up to be distributed memory parallel. The message passing implementation imposes extra overhead and infastracture on **Xyce** which would not be required in a serial code like Spice3f5. Figure 7.1 illustrates how a circuit problem is distributed across multiple processors. Only certain circuit nodes and devices (voltage nodes and device nodes) are owned by the local processor, while others are owned by other processors. This results in "cuts" in the graph, and the need for some information to be passed back and forth between the various processors.

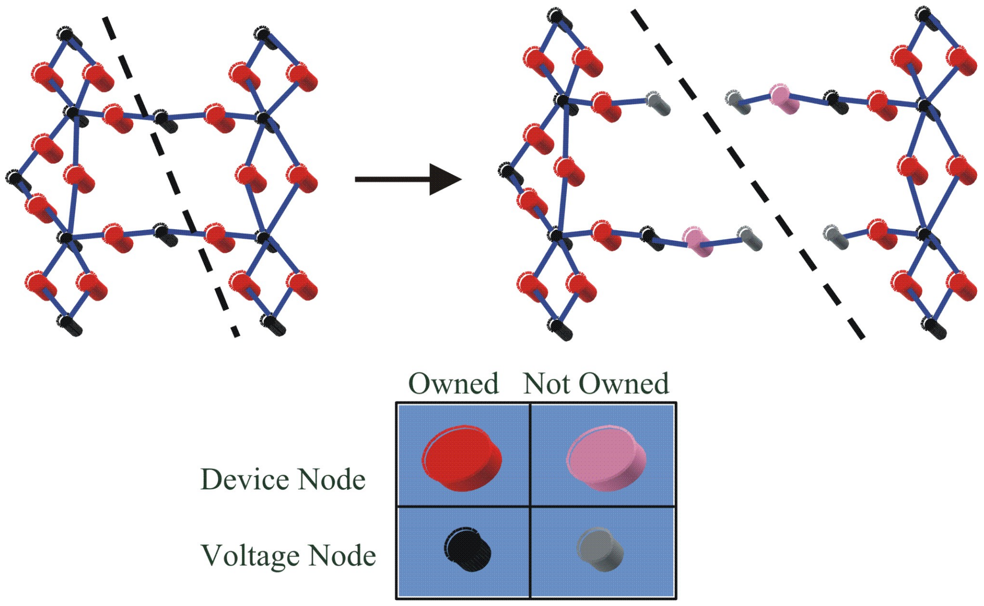

Figure 7.1. Nodes shared by two processors. The dashed line indicates the boundary between two processors.

Fortunately, many of the details related to the parallel implementation are hidden behind abstract interfaces in the **Xyce**. This allows many (but not all) developers to do their work without having to spend time worrying about parallel issues.

Global and Local Indexing

There are two indexing systems used in **Xyce**. One is referred to as Global ID (GID) and the other is referred to as Local ID (LID). For the remainder of this chapter, they will be referred to by their respective acronyms. Previously, each device in **Xyce** has load functions implemented that can use either indexing system. However, a few years ago most of the GID-based loads were removed, as they have been somewhat deprecated for a couple of years, and were redundant with the much faster LID-based loads. In a few cases, GID loads are still used, as they are much easier to set up.

GID was the first indexing system implemented in **Xyce**. It is easy to understand, but is relatively slow. It requires the linear algebra package to perform copies during each matrix load, and it also relies on relatively slow accessor functions. Most egregiously, the GID-based linear algebra accessors don't perform any index checking, so every GID load call in the device package is bracked by if-statements. An example of a GID-based matrix load is illustrated by the code fragment in Figure 7.2.

The if-statements required by GID are to insure that off-processor and ground-node loads are skipped. (if they are not skipped, the code will core dump) The if-statements bloat the device code, and (as is to be expected of branching statements) prohibitively impede the load procedure.

Spice3f5 uses very fast pointer-based accessor to the Jacobian and residual. An example of a Spice3f5 matrix load is illustrated by the code fragment in Figure 7.3. Note how much smaller this fragment is than the one in Figure 7.2.

Compared to Spice3f5, the GID-based **Xyce** loads did not perform well. Using a pointer-based accessor in **Xyce** was not practical, given the requirements of distributed memory, but the poor GID load performance required that a new, faster, indexing system (and interface) be developed for **Xyce**.

LID was developed to address this performance issue, and is much faster. Most importantly, the LID approach handles off-processor and ground elements automatically, behind the interface, so LID-based loads do not require ”owned” if-statements. LID relies on inlined, overloaded bracket operators, so usage is very compact. LID indexing is uses indices which are local to the processor, so the linear algebra package does minimal translation of indices when running in parallel. Also, for the matrix load, LID obviates the need to use intermediate data copies in the linear algebra package, as the LID approach directly utilizes the compressed row format (CRF). An example of a LID-based matrix load is illustrated by the code fragment in Figure 7.4.

```
// row associated with the drain node
count = 0;
if (drainConductance != 0.0)
{
  irow = ADrainEquDrainNode_I;
  for (i=0;i<imax;++i) {cols[i]=-1;vals[i]=0.0;}
  if (ADrainEquDrainNode_I != -1 &&
      ADrainEquDrainNode_J != -1)
  {
    cols[count] = ADrainEquDrainNode_J;
    vals[count] = drainConductance; ++count;
  }
  if (ADrainEquDrainPrimeNode_I != -1 &&
      ADrainEquDrainPrimeNode_J != -1)
  {
    cols[count] = ADrainEquDrainPrimeNode_J;
    vals[count] = -drainConductance; ++count;
  }
}
if (count != 0)
{
  bsuccess = bsuccess && JMatPtr->sumIntoRow
    (irow, count, &vals[0], &cols[0]);
}
```

Figure 7.2. Example of GID load code fragment. Note the relatively large number of lines of code, compared with the much smaller fragments in Figure 7.3 and 7.4. Several if-statements test if indices are equal to -1, which by convention indicates if (1) the node is ground or (2) off-processor.

This fragment was taken from the BSIM3 MOSFET Jacobian load function. The sumIntoRow function call is the accessor function into the Jacobian matrix. The two arrays, vals and cols are temporary storage arrays, which store the compressed row structure. cols contains the global column indices. irow is the global index for the matrix row. Note that all the GID-related load code is no longer present in the **Xyce** BSIM3 source, but this example is still useful.

```
/* Row corresponding to the drain node */
*(here->BSIM3DdPtr)  += BSIM3drainConductance;
*(here->BSIM3DdpPtr) -= BSIM3drainConductance;
```

Figure 7.3. Example of Spice3f5 load code fragment. Note the small number of lines, and the lack of if-statements. This fragment was taken (slightly modified for clarity) from the BSIM3 load function. Also, note that the load places the conductance values directly into the pre-determined memory locations.

```
// Row corresponding to the drain node:
(*JMatPtr)[li_Drain][ADrainEquDrainNodeOffset]
   += drainConductance;
(*JMatPtr)[li_Drain][ADrainEquDrainPrimeNodeOffset]
   -= drainConductance;
```

Figure 7.4. Example of LID load code fragment. This fragment was taken from the BSIM3 load function. Similar to the Spice3f5 fragment, there are no if-statements, and the specification is relatively compact. Unlike the Spice3f5 load, pointers are not used, and the overloaded bracket operator are used to hide details related to parallelism.

The usage of LID has two separate categories, Direct Vector Access (DVA) and Direct Matrix Access (DMA). DVA was much easier to implement and understand, so it was implemented and made the default first. This chapter will describe both, but as DMA is more complex, it will be the focus.

Note that conceptually, it is convenient to think of GID as the main indexing system for the topology package, and LID to be the main indexing system for the device package.

7.2 Global to Local Index Relationship

The relationship between global and local indices into the residual (or solution) vector is shown in figure 7.5.

Global indices into the residual and solution vector go from 0 to $N-1$, where N is the number of unknowns. In a distributed memory environment, each processor will own, and have direct access to, a fraction of the entire solution vector (and Jacobian matrix, residual vector, etc.). On any given processor, the global indices in use will probably not start at 0, and may not be continuous.

Local indices on each processor always start at 0, and run continuously up to $M-1$, where M is the number of unknowns owned by the processor. The local vector, however, is larger than M, and has extra storage locations starting at $i = M$. These extra locations in the local vector are for values that need to be discarded, or possibly communicated off-processor. These extra storage locations preclude the use of GID-style if-statements. For the ground node, it is faster to simply load numerical values into a dummy location than to use an if-statement to skip the load.

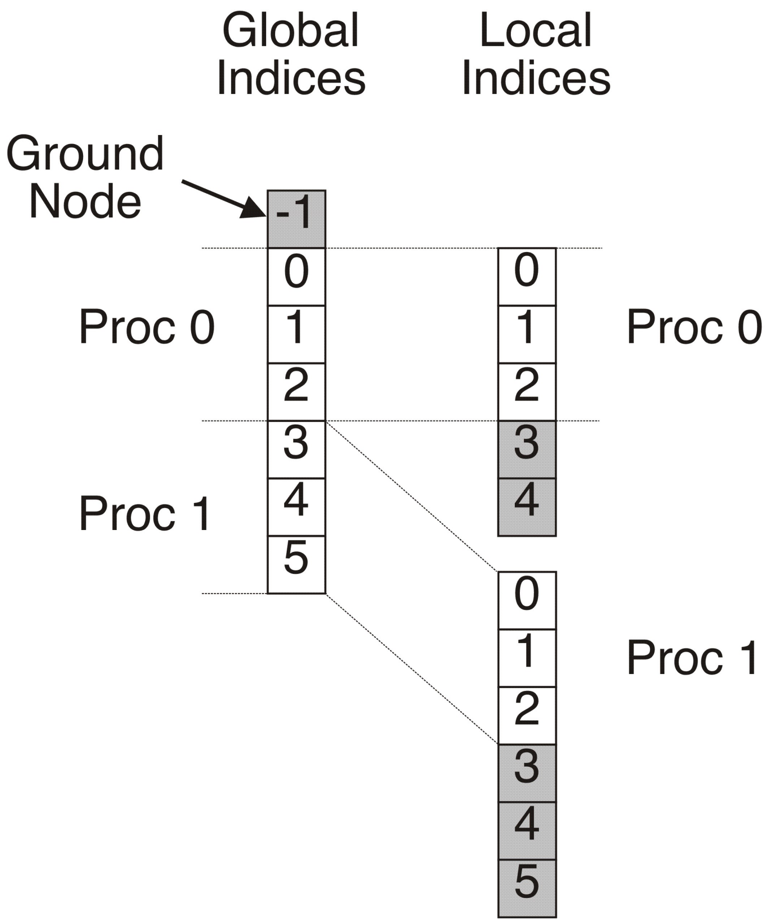

Figure 7.5. Example of a global to local mapping for a vector. The ground node is given a global index equal to -1, but there is no location for it in the global vector. For this reason, access into global vectors has to test for -1 first.

Each local vector has a few extra elements at the end, to hold values for either ground, or off-processor variables. These auxilliary entries are handled behind the linear algebra interface, and are not directly included in the linear system.

7.3 External Variables vs. Internal Variables

A circuit network is comprised of circuit nodes and devices. This is ilustrated in Figure 7.6. Associated with each circuit node is a voltage variable, which is usually shared by two or more devices. These voltage variables are the most common example of external variables in **Xyce**.

Internal variables are variables which are not shared between devices, but instead are completely under the jurisdiction of one device. Many simpler devices (such as the resistor) do not have any internal variables, but more complex devices (such as transistor devices) often do.

In addition to internal variables not being shared between devices, they are also different from external variables in that they are always completely local. The parallelism of **Xyce** cuts the circuit graph in between nodes (voltage and device). As of this writing, cuts the circuit graph are never through a device. Any device node is primarily owned by one processor, and communication between processors only concerns external variables.

External variables are easy for the topology package to handle, as they can be determined directly from the circuit connectivity. Internal variables are a not known by the topology package a priori, so topology relies on the device package for this information. The interaction between the device and topology packages is somewhat complex, and is the subject of the next section.

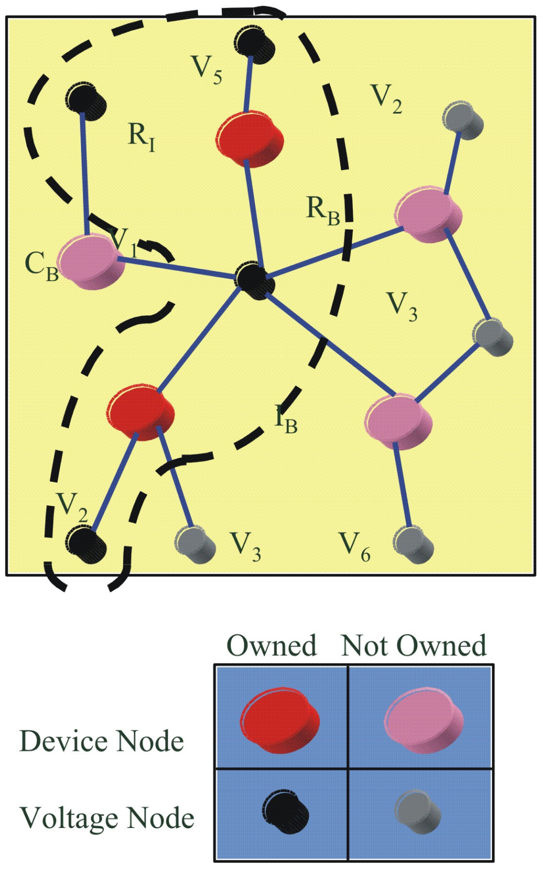

Figure 7.6. Voltage Nodes and Devices Nodes. Voltage nodes all (except ground) have associated external solution variables. Some device nodes have associated internal solution variables. Device nodes are on the edges of the graph, and voltage nodes are on the vertices. The dashed line indicates the boundary of the local processor.

7.4 Topology and Device Package Interaction

The **Xyce** topology package is responsible for translating the circuit topology into a linear system topology (among other things). In order to do this, it is neccessary for it to keep track of all of the internal and external variables of a simulation, which devices those variables are associated with, setting up all their local and global indices, and providing all of this information to the linear algebra, parallel services, and device packages.

7.4.1 jacStamp Example

For this example, assume that a device is being implemented that has the following Jacobian stamp structure:

$$\begin{bmatrix} A & B & 0 & 0 \\ B & 0 & B & 0 \\ 0 & B & C & 0 \\ B & 0 & C & C \end{bmatrix} \tag{7.1}$$

The full Jacobian matrix, of course, will be much larger than this stamp, but the stamp represents the part of the matrix that the hypothetical device cares about. This contribution will be summed into the full matrix at each Newton step.

During the setup phase, all the device (and the device developer) really know about is the abstract sparsity pattern of the Jacobian stamp. In **Xyce**, the sparsity pattern is represented by the jacStamp data structure. For this example, the jacStamp structure will be as follows:

Row 0, with nonzeros in column 0 and column 1:

$$\text{jacStamp}[0][0] = 0 \tag{7.2}$$
$$\text{jacStamp}[0][1] = 1 \tag{7.3}$$

Row 1, with nonzeros in column 0 and column 2:

$$\text{jacStamp}[1][0] = 0 \tag{7.4}$$
$$\text{jacStamp}[1][1] = 2 \tag{7.5}$$

Row 2, with nonzeros in column 1 and column 2:

$$\text{jacStamp}[2][0] = 1 \tag{7.6}$$
$$\text{jacStamp}[2][1] = 2 \tag{7.7}$$

Row 3, with nonzeros in column 0, 2 and 3:

$$\text{jacStamp}[3][0] = 0 \tag{7.8}$$
$$\text{jacStamp}[3][1] = 2 \tag{7.9}$$
$$\text{jacStamp}[3][2] = 3 \tag{7.10}$$

As one can see from the jacStamp, the jacStamp structure specifies the sparsity pattern for the device's Jacobian Stamp in a compressed row format. Note that for every device, the jacStamp will start at index zero, as it is just for specifying the local sparsity pattern.

Topology will obtain this jacStamp structure during the setup phase. Once topology has enough information (the number of internal variables for each device, and the jacStamp sparsity pattern for each device), it will integrate them into the global linear system. For the hypothetical device in this example, the matrix entries of interest could theoretically be anywhere in the global system:

$$\left[\begin{array}{ccccccccc}
 & & c1 & & c2 & & c3 & & c4 & \\
 & & \vdots & & \vdots & & \vdots & & \vdots & \\
r1 & \ldots & A & \ldots & B & \ldots & \ldots & \ldots & \ldots & \ldots \\
r2 & \ldots & B & \ldots & \ldots & \ldots & B & \ldots & \ldots & \ldots \\
r3 & \ldots & \ldots & \ldots & B & \ldots & C & \ldots & \ldots & \ldots \\
r4 & \ldots & B & \ldots & \ldots & \ldots & C & \ldots & C & \ldots \\
 & & \vdots & & & & \vdots & & \vdots &
\end{array}\right] \tag{7.11}$$

The global system, being stored in a compressed row format, will have a small (compared to the size of the system, N) array of values for each row. Each array will contain (only) the nonzero elements of the row.

After the topology package creates the matrix structures, it calls the function, "registerJacLIDs" for each device. The purpose of this function is to pass in the LID-based compressed row column offsets for each row. The form of registerJacLIDs is similar to that of the jacStamp - the difference is that the integer values owned by the data structure correspond to the column offsets relative to the entire global matrix, rather than the column offsets for the sparsity pattern of a single device.

7.4.2 Function registerJacLIDs Example

Figure 7.7 shows a very simple example of the registerJacLIDs function, taken from the resistor device. This function is called by the topology package, but only after all the final matrix details have been determined. So, it is called long after jacobianStamp is called. jacLIDVec, the two-dimensional array which is passed in as an argument, follows the same structure as the jacStamp, except that the contents of it are offsets into the global matrix rows, rather than row indices into the local stamp.

This is an important point. The matrix storage is based on a compressed row format, meaning that only nonzero entries are stored. Row offsets are not global (or local) row indices. They are just offsets into a specific row-vector. This means that the offset corresponding to row J in one row may not be the same as the offset corresponding to row J in another row. In other words,

$\texttt{APosEquPosNodeOffset} \neq \texttt{ANegEquPosNodeOffset}$

So, it is very important that different offset variables are for every single row, even if they refer to the same columns.

```
//-----------------------------------------------------------
// Function      : N_DEV_ResistorInstance::registerJacLIDs
// Purpose       :
// Special Notes :
// Scope         : public
// Creator       : Robert Hoekstra
// Creation Date : 08/27/01
//-----------------------------------------------------------

void N_DEV_ResistorInstance::registerJacLIDs
   ( const vector< vector<int> > & jacLIDVec )
{
  N_DEV_DeviceInstance::registerJacLIDs( jacLIDVec );

  APosEquPosNodeOffset = jacLIDVec[0][0];
  APosEquNegNodeOffset = jacLIDVec[0][1];
  ANegEquPosNodeOffset = jacLIDVec[1][0];
  ANegEquNegNodeOffset = jacLIDVec[1][1];
}
```

Figure 7.7. Example of function registerJacLIDs.

8. Device Development Checklist

Chapter Overview

This chapter gives guidelines for **Xyce** device model development. It attempts to address development of both industry-standard legacy devices and also of Sandia-developed devices. The goal of this chapter is to provide standard checklists that should be followed for **Xyce** devices, and to help new developers understand the criteria for declaring a device to be completed.

8.1 Introduction

Xyce is a SPICE-compatible simulator, and as such many of the device models are standard in the electronics industry. On the other hand, as Sandia has a number of unique modeling needs, there are a number of original models in **Xyce** as well. As such, there are 2 categories of devices that go into **Xyce**.

1. Original device models developed from scratch specifically for Sandia's unique needs. Example: photocurrent models, biological models, etc.
2. Legacy devices from SPICE (or spice-like codes), where the source is publically available. Examples: level-1 MOSFET, level-1 JFET, level-1 BJT, level-1 Diode, BSIM3, BSIM4, BSIMSOI, etc. Spice3f5 is in the public domain, has been a defacto industry-standard for years, and many commercial simulators are derrived from it. As such, the models in Spice3f5 are standard to nearly any circuit simulator. Similarly, many newer models, developed under the auspices of the Compact Model Council [16] can also be considered industry standards.

This checklist is divided as follows. Section 8.2 contains a checklist that applies to any device, whether it is a legacy device or not. Section 8.3 lists additional tests specific to transistor devices. These also apply to any device, whether it is an original device or a legacy device. Section 8.4 contains a checklist of specific tests for legacy devices.

8.2 Checklist for Device Certification

The following set of tests must pass for any device developed in **Xyce**, whether it represents original work or not.

8.2.1 Any device development must have a corresponding issue in bugzilla

This is really important. If no bugzilla issue exists, you (or someone else on the team) should create one before doing any CVS checkins. Ideally, the issue should be recorded in bugzilla before any code development even starts. From the point of view of our development process, if work is not recorded in bugzilla, it doesn't exist.

8.2.2 For any device to be considered complete, tests must exist to certify it

This requirement goes hand-in-hand with item 8.2.1. For any bugzilla issued to be declared "fixed", and later "verified", certification tests, which will be part of nightly and weekly regression testing, must be created to prove that the issue is actually resolved.

Guidance for appropriate tests can be gleaned from the other items in this document. A short, minimal, necessary-but-not-sufficient test list is given below, with links to sections that describe these tests in more detail. Note, these are tests that a bare minimum for automated regression testing. A lot of other testing (such as numerical Jacobian testing 8.2.3) may be more appropriate for manual hand-testing during development.

1. All states of the device must be tested. (saturation, inversion, forward active, reverse bias, etc.)
2. All model options must be exercised (optional nodes, optional equations, etc)
3. All parameter defaults must be tested.
4. Non-default values of parameters must be exercised as well.
5. Tests against analytic solutions (when possible) should be created.
6. All relevant analysis types must be exercised (DCOP, TRAN, etc).
7. Any derivative device must pass all the antecedant tests.

8.2.3 Numerical Jacobian tests must pass

The numerical Jacobian test must pass (with some exceptions) for a variety of operating conditions for the device i.e. all equations modeling the device should be exercised in the numerical Jacobian test through tests involving more than one circuit.

There are cases in which incorrect Jacobian terms can be justified. An example of a good justification is if there are clear round-off errors in the numerical Jacobian terms. Another example would be if it can be demonstrated that doing the more correct and complete derivative actually causes the solvers to be less robust. This can sometimes happen if the derivative term in question causes the matrix to be more poorly conditioned.

In the old days (1960's), codes were sometimes designed with inaccurate derivatives (or even constant Jacobians) because computing the derivatives required too much memory and/or floating point operations. This was an issue when computers were much slower and smaller than they are today, and should not be used as a justification for imprecise Jacobians in Xyce.

Because there are (rare) examples in which imprecise Jacobian terms might be desirable, this test is not appropriate as a catch-all regression test, and should be done manually as part of device development. Currently, it is up to the individual developers to insure that this happens.

8.2.4 For a variety of circuits, the new device must pass valgrind tests

This is to check for memory leaks, etc., uninitialized variables, etc.

8.2.5 Both time integrators need to be supported by any new device

As of **Xyce** release 4.0, the new-DAE time integrator will be the default time integrator. The old integrator will be maintained for one or two more releases, and then probably discarded. Until then, however, both must be supported. That means that any and all of the tests in this chapter must pass for both the netlist option:

```
    .options timeint newdae=1
```

and also:

```
    .options timeint newdae=0
```

If any test fails for either time integrator, the device is not a finished device. In practice, this means that the old-DAE load functions: loadAnalyticJacobianBlock and loadRHSBlock must be completed, and the new-DAE load functions: loadDAEFVectorBlock, loadDAEQVectorBlock, loadDAEdFdxMatrixBlock, loadDAEdQdxMatrixBlock need to be completed. This is discussed, somewhat in section 9.5.8 of the FAQs.

8.2.6 Derivative devices must pass antecedant tests

It is common to develop new devices by adding physics to an existing device. Hence, we have devices such as the level-4 BJT, which consists of the level-1 BJT plus photocurrent effects. It is important that the behavior of the original device be maintained, so in this example, it is necessary for the level-4 BJT to pass all the level-1 BJT tests to be considered a completed device. Of course, this can only be done for derivative devices that have been modified only from adding effects, rather than replacing or modifying existing effects.

8.2.7 Compile with the gcc option -Wshadow

This will warn about shadowed variables, which is an easy mistake to make (especially when importing a SPICE device into **Xyce**). For the device to be considered finished, there should be zero shadowed variables.

8.2.8 Device must follow **Xyce**'s style guide

The style guide is described in chapter 6.

8.2.9 All artifacts of reference devices must be completely removed from source

It is common to create a new device by initially copying an existing device, and then doing global searches and replaces on the class names, and then removing most of the internal details and replacing them. A common mistake has been to leave behind artifacts of the original device, and the result of this mistake is for other developers to find code and or comment fragments in one device, that obviously originated with another one. This can be very, very confusing to any developer who later tries to understand the code and should be avoided at all costs.

For example, a developer might be assigned the BSIM4 device. A common starting point would be to copy the BSIM3, and rename it BSIM4. Then, after that, to carefully remove BSIM3 code and comments (as needed) and replace with BSIM4 code and comments. As the two devices are very similar, this can jump-start development, but it is really easy to inadvertently mix code from the two models. If this development path is taken, it is best to remove as much of the source code from the original device as possible, before starting code development on the new device.

Any device that still has old code artifacts left around cannot be considered finished. Remember, this is a team project, and it is important to have your code be as comprehensible as possible.

8.2.10 Source code must reside in the appropriate directory

Source code for devices in **Xyce** are stored in one of several sub-directories in the `DevicePKG` directory. `DevicePGK/src` and `DevicePKG/include` should be used for non-export controlled models as this source code is stored on the SON. `DevicePKG/SandiaModels` should be used for any export controlled models as this data is held on a server on the SRN. If your new device uses automatic differentiation then it should reside in `DevicePGK/AD` or `DevicePKG/SandiaModels/AD` as appropriate.

8.3 Additional requirements for transistor devices

8.3.1 Both N-type and P-type must be tested individually.

Test circuits that run the device as an N-type and as a P-type must run on their own.

8.3.2 Circuits that contain both N-type and P-type must run robustly.

More specifically, this means N-type and P-type for BJTs, and NMOS and PMOS for MOSFETS. Circuits of this type are often fundamental digital building blocks such as inverters, nand gates, ring oscillators, etc. In general, circuits which contain both device types, and which run through a variety of states, are much more challenging numerically than circuits that contain only one type of device. Single device circucits are usually just set up to look at I-V curves, which are necessary but not important. However, I-V curve circuits are usually so simple as to not be much of a numerical challenge. There are many examples of devices, that ran find in an I-V circuit, but had mistakes that rendered an inverter non-robust.

Many examples of such circuits exist in the Xyce test suite, which can be checked out as `Xyce_Regression`. Often, good robustness tests can be created by simply copying an appropriate circuit for a similar device, changing the level numbers, and running with default parameters.

8.3.3 Automated voltlim tests must also pass

Voltage-limited solves will have an extra limiter term in the residual. That extra term has to be stored (in the Jdxp vector for old-DAE, and the dFdxdVp and dQdxdVp vectors for new-DAE integrator). The automatic test compares the contents of these vectors against matrix-vector multiplies. As with the numerical jacobian test, it is important to test voltlim with a variety of test circuits. This should be with a single device as well as more complex circuits including both N- and P-type devices.

8.3.4 For both N-type and P-type, the numerical jacobian, valgrind and voltlim tests must pass for all the different model options supported by that device.

For example, the BSIM4 has 4 different gate models, 2 different rds models, etc. Some of these options significantly change the equations solved by the model, so it is important to test all of them. Exceptions can be made if it is known that a certain model option is not needed by any current users. For example, the reference implementations of BSIM3 and BSIM4 have options that enable a non-quasi-static (NQS) model; this NQS model is not required by anyone at Sandia, so we have not implemented that part of the model and we don't have any tests for it.

8.4 Additional tests for Legacy Devices

8.4.1 **Xyce** Legacy devices Must Match Legacy Simulator

In most cases, the legacy simulator is SPICE. This of course requires that the developer have a copy of the original source that they can run. Spice3 or chilespice. If the original legacy simulator is SPICE, then a highly modified version of Spice3f5 source can be found in our repository to use for detailed code comparisons.

Here is a list of detailed comparison tests (between Xyce and SPICE) that will need to pass for a legacy device implementation in Xyce to be considered complete.

- All the default parameters must match between the Xyce implementation and the legacy code implementation. This sounds obvious but is can sometimes be a source of error, particularly if the starting point for a new Xyce device is a copy of an old device (see section 8.2.9).
- During the DCOP. SPICE and Xyce (for a variety of circuits) must match exactly over many Newton iterations. This means that every Jacobian matrix will match to machine precision, and every intermediate solution vector will match to machine precision as well, and every corrected residual vector will match to machine precision.

  The residual vector can be tricky, as SPICE and Xyce handle voltlim differently. (for good reasons).
- Same, as previous item, but for transient. For the first two time steps out of the DCOP, all the Jacobians and vectors should match, if the time integration is forced to match. To do this, force both Xyce and SPICE to use backward euler integration, and force them to use constant step sizes, with the same step size. This should work for both new- and old-DAE.

9. Frequently Asked Questions

Chapter Overview

This chapter contains a list of frequently asked questions (FAQ) from **Xyce** developers. Most of it centers around device model implementation, but there are also many questions pertaining to other **Xyce** development subjects. In addition to questions about other modules of the source, there are also questions pertaining to **Xyce**'s purpose, mandate, and general philosophy. The original FAQ was written in 2004, and has been updated several times for this chapter.

9.1 Where to find other information

9.1.1 References

For more information, consult:

- **Xyce** Math Guide. [17].
- **Xyce** Users Guide [18].
- **Xyce** Reference Guide [19].
- **Xyce** Reference Guide [20].

9.1.2 I can't find the answer to my question in this FAQ, and I can't find it in the other references. What do I do?

Ask someone. The authors of this document are a good place to start.

Xyce has been in development since 1999. At this point, there around 50 device models in **Xyce**. If you are confused about an issue, it is unlikely that you are the first person to encounter it. You probably aren't even the second person to encounter it. Someone on the project has probably already figured it out, possibly years ago, and that person may consider it to be an officially "solved problem".

Xyce isn't perfect, but most of the code that is there was put there for a deliberate reason. Most of the code in **Xyce** was written by people who are still here.

If you get stuck on an issue, don't sit on it for long. There's no point in re-inventing the wheel. It is better for everyone involved if you seek out other developers and ask questions. Even if the issue is truly a new one (unlikely, but possible), the more people on the team who know about it, the better. Your development process will go a lot faster (and result in much better code) if you proactively seek advice.

9.2 General Philosophy and Miscellaneous

9.2.1 What is the goal of this FAQ?

The purpose of this chapter (and the other chapters) is to provide a guide to **Xyce** software developers. In particular the hope is that this FAQ contains information not covered by other chapters of this document, or by other **Xyce** developer documents. This will prevent new developers from “reinventing the wheel”, as many issues encountered by a new developer have probably already been addressed by others. This FAQ was originally written with device model implementation in mind, so much of the focus is on that subject, but other topics are addressed as well. Issues addressed in the FAQ include (in no particular order):

- Historical **Xyce** design decisions.
- Context; the relationship of **Xyce** to other circuit simulators, both commercial and public domain.
- Random “gotcha” issues that have confused everyone.
- Code quality.
- Code consistency. To the extent possible, devices should follow a standard blueprint.
- Verification. Or: making sure that for a given set of equations, you are actually solving them correctly and reliably.
- Numerical stability. This is really important! **Xyce** runs huge circuits. The larger the circuit, the more crucial this is.

In general, this chapter does not address:

- Detailed mathematical description of **Xyce**. For a mathematical description, see the **Xyce** Math Guide [17].
- How to write “best practice” C++ code. (for this see chapter 6, which contains the **Xyce** C++ style guidelines).
- Model validation issues.

- Original Model creation. In other words, if you are trying to determine or derive a new set of equations for your new organic thin-film transistor (TFT) model (for example), this chapter will not tell you how to do that. However, if you have a set of equations in mind, it will probably help you set them up for **Xyce** to solve them.

9.2.2 What are the requirements of **Xyce**?

This question is addressed in chapter 1. **Xyce** has mostly been funded by ASC, and is intended to support the Nuclear Weapons community at Sandia. Requirements include:

- To be SPICE-compatible, which includes supporting SPICE legacy devices, and also using a netlist-based input file.
- To be a high-performance computing application (massively parallel, where "massively" means up to 1000's of processors).
- To include specialized device models and algorithms, unique the the weapons community at Sandia.

Much of **Xyce**'s design is based on these requirements. The second requirement (high-performance computing) necessitates that **Xyce** be fundamentally different than SPICE, in many respects. The first requirement, that we be compatible with SPICE, requires that we be the same (or at least similar) to SPICE in other respects.

To illustrate this issue consider:

- Common **Xyce** user question: "Why doesn't my PSpice (or HSpice, AimSpice, or SmartSpice) circuit work in **Xyce**?"
- Common question from ASC (and other) review panels: "How is **Xyce** different from PSpice (or any other commercial simulator)? In other words, why are we funding it? Can't we just buy one for less money?"

For the **Xyce** project to be successful, we have to be able to give good answers to both of these questions.

9.2.3 Who are the customers for **Xyce**?

At Sandia National Laboratories, the most common type of engineer is electrical, and there are literally hundreds (possibly thousands) of engineers who use some form of circuit simulator. Most of these users are actually not **Xyce** customers, but the internal **Xyce** download list has about a hundred unique users, and can be expected to grow. Some of these users are relatively low-maintenance, and others require a lot of attention, because they are applying **Xyce** to problems which push the edge of **Xyce**'s capabilities.

Xyce is a relatively large software project, but we don't have the resources to support (for example) 500 customers, so we have always presented **Xyce** as a niche tool, to be used as appropriate. To date, **Xyce** has been designed for very large circuit problems (beyond the capability of serial SPICE simulators), and which include hostile radiation effects. Nearly all of our users also use commercial tools, such as HSpice, PSpice, SmartSpice, etc. From our point of view, that is fine. We aren't trying to compete with commercial tools for every user. We're primarily trying to provide capability that is unavailable elsewhere.

9.2.4 Why do devices in **Xyce** need to precisely match SPICE devices? Don't people always complain about SPICE devices? Surely we could come up with better ones.

Yes, theoretically, we could come up with better ones, and in many cases we have. However, most Sandia devices are intended to address physics not addressed in the larger circuit community. There aren't very many companies in the world who care about neutron effects, for example, so the only option is for us to develop such models. For most non-specialized physics, it is more practical to use the same canonical set of industry models that everyone else uses.

As noted in the question 9.2.3, most circuit designers jump between circuit simulators, depending on their needs. For example, a designer might do some early work in SPICE, to take advantage of the parts library, but then switch to **Xyce**, once they need to add their small PSPICE-developed circuit, to a much larger **Xyce** circuit.

What this means is that a designer needs to be confident that they can use the same model, and model parameters in code A that they used in code B, and still get the same answer. Switching codes should never result in significant changes in the answer. Minor differences due to platform differences, error tolerances, etc., can be acceptable, but ultimately, a trusted result from code A should match a result from code B. This means that we need to make the models precisely the same, to the extent that this is possible.

Also, while legacy SPICE models are often “not very good”, their source code has been debugged to death. After 30 years, and thousands of users, you can bet that just about any numerical bug has been found already. Most of the time, the fastest way to get a canonical SPICE device correctly implemented in **Xyce** is to refer to the SPICE device source extensively.

9.2.5 You said in the last question that SPICE devices have been “debugged to death”. What is a bug?

This may seem like a silly question, but it isn’t. It is important to define terms.

When I say “bug”, I mean something like a sign error. Or, an incorrect derivative in the Jacobian. Or, a divide-by-zero error that causes the code to crash. Or, an inconsistently implemented gmin resistor.

A bad model derivation, however, is not a bug. For example, suppose you had a SPICE device where an exponential term should clearly (based on you understanding of the physics of that device) be an error function. That is not a bug. That is an invalid model.

9.2.6 When you say “precisely match SPICE”, what does that mean?

It means that if we force SPICE and **Xyce** to run with equivalent solver options, they should give exactly the same answer, out to (something like) 8 digits. This should be true, even many time steps into the run. It should also be true at intermediate points in the solve. For example, the 10th Newton step of the 10th time step should yield identical Jacobian matrices in SPICE and **Xyce**.

To get SPICE and **Xyce** to run equivalently for the DCOP phase is trivial. If **Xyce** has `.options nonlin searchmethod=0`, `.options linsol type=ksparse`, and `.options device voltlim=1`, then it should run the same as SPICE. (Note these options are no longer the defaults in **Xyce**, since we use KLU as the default solver now.)

To get SPICE and **Xyce** to run identically in transient is harder, as they can use different time stepping algorithms. The only way to force them to do the same thing it to force both codes to run with constant time stepping enabled, and to force both codes to use backward Euler integration. In **Xyce** this is set with `.options timeint conststep=1 maxord=1`.

For any device, **Xyce** and SPICE should match in this manner for a variety of circuits and device parameters. This variety of circuits should include different levels of complexity, nonlinearity and size. It isn't sufficient to show that **Xyce** and SPICE "precisely match" for just one circuit.

Note that this does NOT mean that running in SPICE mode is generally a good idea. It is only something you should do if you are debugging a legacy device.

9.2.7 What is a "legacy device"? What are some examples?

Sometimes I also use the term "canonical device", or "canonical SPICE device", and I mean the same thing by these terms as "legacy device".

Legacy devices are devices for which the following is true:

- developed outside of Sandia (probably).
- are considered to be industry standards.
- can be found in most commercial and free simulators.
- are widely used by circuit designers.

Some examples of legacy devices include:

- any "level=1" model - the level-1 BJT, the level-1 diode, etc.
- any model from the Berkeley BSIM group. (BSIM3, BSIM4, etc.)
- any model developed under the auspices of the compact model council. (this includes the BSIM group, and also models like the Mextram BJT, developed at Phillips)
- any model that can be found in free versions of Spice3.

Some examples of devices which are not legacy devices are:

- Any proprietary, or export-controlled models developed at Sandia.
- Any models developed under contract with RPI, or other university or organization.
- Most rad-aware models.

By necessity, a large fraction of **Xyce**'s device library are legacy models. Models in this category are easier (in some respects) to work with, as most of the difficult model development work has been done by others, and they've already been used extensively by (literally) thousands of users. As such, putting them in **Xyce** is more a code development exercise than a model development exercise. After all, we didn't invent the BSIM3, a group at UC-Berkeley did.

Despite their extensive vetting by the industry, legacy models aren't perfect by any means. However, based on our experience, they are relatively solid numerically and in terms of software quality. They aren't likely to have silly bugs, like sign errors, or units mistakes, or Jacobian mistakes, etc., although we have found mistakes like this occasionally. What deficiencies they do have are more likely to be in the conception of the model, not the implementation. You can certainly argue if any of them include the correct physics, but that is a different discussion.

What this means, from a **Xyce** development standpoint, is that a large part of the device code development will involve very detailed comparisons to SPICE. The job of implementing a legacy model in **Xyce** is mostly a code development job, not a model development or model validation activity.

For models that are not legacy models, then the development process needs to be very different. A developer should make every effort to think critically about the model, and all its implementation issues. For models developed here, it is more important that an attempt is made at V&V.

9.2.8 Are all legacy models from SPICE?

No. At this point, many are not, and this trend will continue, as the industry has moved towards specifying models in Verilog-A format. The advantage of this is that it separates (mostly) the fundamentals of the model from the details of the simulator. At this point, most compact model groups (if they put their models in the public domain) do so in Verilog-A form, rather than SPICE. The disadvantage (to us) of the trend towards Verilog-A is that we have historically implemented models from SPICE source, and have less experience with Verilog-A (see then next question, below).

SPICE has been the de-factor industry standard for a long time, and SPICE itself is in the public domain, so most of the models that have gone into **Xyce** so far have been SPICE-based models, either taken from the original Spice3f5 code, or from separate Spice3 implementations, such as the BSIM3.

At this point, we've nearly run out of public domain models that come in SPICE form. We've implemented most of the models from Spice3, and have implemented most of the models from the BSIM group at Berkeley. The one remaining BSIM model (as of this writing), that we have not implemented is the BSIM5, and while many papers have been published about this model, the BSIM5 source code isn't in the public domain yet.

9.2.9 What do we do about Verilog-A specified models?

As of this writing we're in the process of setting up the ADMS model compiler [21] to work with **Xyce**. This requires doing an extensive customization of the back end of ADMS. The function of a model compiler is to take a model specified in Verilog-A form, and the convert it into the detailed source code of a specific simulator. Once the back end to ADMS is set up, we should be able to plug most Verilog-A models directly into **Xyce**. Potentially, this will mean that a large number of models will come into the code in short order.

9.2.10 What about V&V?

V&V = Verification and Validation. Loosely speaking, validation means answering the question "am I solving the right equations?" and verification means answering the question "given a set of equations, am I solving them correctly?"

Following good software quality engineering (SQE) practices is part of verification. On the **Xyce** project, we have historically done a pretty good job with SQE. The real gaping hole has been with validation. Prior to FY05, the **Xyce** project essentially did no formal validation, primarily due to lack of funding. This situation has turned around, but validation methods are still under development. Also, it has been necessary to prioritize the work.

In terms of priorities, validation work needs to address models that are high risk. Generally, this has meant focusing on compact models developed at Sandia, which primarily focus on abnormal or hostile effects, including radiation environments (gamma, neutron, x-ray), high temperatures, etc. Legacy models have received less validation attention, mostly due to resource constraints, and also because there is some implicit trust in models that have already been vetted by the world at large.

To do good validation, verification has to be done first. Doing a lot of validation work on broken (un-verified) code can be waste of time. So, when implementing a device, worry about getting it set up correctly and consistently first. Validation (if it happens) comes later.

9.2.11 Shouldn't users always use the best models they can?

Yes. But remember, **Xyce** (and other circuit simulators) are primarily engineering design tools. They are not "basic science" or "research" tools. A compact model is generally not going to be high fidelity enough to learn much new about the physics of a single transistor. Instead, a compact model is going to contain a discrete representation of the physical behavior of that transistor. The compact model is in some ways the result of physics research, not the driver of it.

Ironically sometimes a circuit designer or analyst will use a bad model, knowing full well that it is a bad model. Why? Because bad models are often really simple and easy to understand. The level-1 MOSFET, for example, is a very simple transistor model, and a typical designer can plausibly estimate model parameters for it, without much difficulty.

The BSIM3 MOSFET, on the other hand, is a much "better" model, in terms of how much physics it includes, and how numerically stable it is. The drawback to the BSIM3, however, is that it has over 300 model parameters, and many of them are not particularly intuitive. If a designer is in a hurry, and just wants a qualitative answer, it may be easier to just use the level-1 MOSFET.

Sometimes a designer is using a model, not to model a specific discrete part, but to model the behavior of a larger, more complex component. For a case like this, all a designer may want is something that "qualitatively behaves like a MOSFET". For a case like this, a simple model is often preferable.

As long as a user has a good understanding of a model's deficiencies, and that awareness is incorporated into their work, then it can be OK to use one of the older, simpler device models.

9.2.12 I'm implementing a legacy device, and I've noticed that books and papers that describe this device don't fully describe what is in SPICE. What's the deal?

You will see this a lot. Generally, textbooks are written for people who want to use SPICE, rather than people who want to write SPICE (or **Xyce**). As such, textbooks will tend to focus on the "high points" of the model. Also, textbooks usually aren't going to focus much (at all) on code implementation details - you aren't their target audience.

Like books, papers are often a very good reference, if you are interested in learning how a device model was originally conceived. However, often times a model has been iterated upon for years after the original publication of the paper. For example, the Gummel-Poon BJT model was originally described in a paper dated 1970. Most SPICE development happened after that, and as SPICE evolved the model was enhanced. It is still referred to as the "Gummel-Poon BJT Model", but the original paper does not fully describe it.

Code implementation details are almost never in books and papers. You will not see a description of voltage limiting in the definition of any transistor model, for example.

In general, if you want to understand the assumptions used in developing a model, and to understand the physics of a device, books and papers are very useful. If you want to understand, in detail, how a given model is set up in SPICE, the best place to look is in the SPICE code. If the SPICE code and a textbook appear to disagree, you should trust the SPICE code first.

Why? Consider the consequences for mistakes. If a textbook has a mistake in it, the result is that some students might (or might not) get confused. If SPICE has a mistake, the result is that the code won't run reliably, or possibly at all. A sign error in a textbook won't significantly change how a human being understands the Gummel-Poon BJT, but it will be catastrophic in SPICE.

9.2.13 How we do we know that the BSIM3 (or any model) in a commercial code is the same as the model we've put into **Xyce**?

Technically, we don't know for sure, but if the vendor's documentation claims that they are using (for example) the "industry standard BSIM SOI model, version 3.2", we've generally believed them. Usually, vendors document what differences there are between models, if any.

Unfortunately, we don't have access to Silvaco's (or Agilent's, or Cadence's) source code, so we don't have much choice but to trust the documentation. Our experience using a variety of circuit codes (both commercial and free) is important here as well. In practice, **Xyce** has consistently gotten similar enough results to other simulators, that it seems unlikely that there's much difference in the device models.

This issue is one of the motivations for Sandia to develop its own code - with **Xyce** a Sandia user can always find out what is in the source.

In general, the EDA (electrical design automation) industry by and large depends on models being consistent across simulators. Many device models are industry standards, which have been developed jointly by a consortium of companies. The industry has attempted to develop and impose standards (starting at least 10 years ago), and this was necessary for the industry to mature. In practice, electronics designers frequently jump from one tool to another, using similar data sets and/or model parameters. If there were significant differences between commercial codes, designers wouldn't trust them, and this is bad for business (the EDA vendor's business, that is).

Most commercial simulators are derived from Berkeley SPICE. Some, like HSpice, were derived from Spice2, and others, like PSpice, were derived from Spice3. Even if the commercial vendors made model changes, they had the same starting point that we did.

Note that there has been one such case (that we're aware of) in the history of **Xyce**, where a commercial simulator used a slightly different model than Spice3. This one case concerned how breakdown voltage is handled in the PSpice level 1 diode, compared with the Spice3 and **Xyce** level 1 diode. The vendor's documentation didn't spell out "here's how our model is different from Spice3"; the documentation instead said, "here are the parameters and equations for our model". This particular difference didn't get noticed until users reported "the **Xyce** diode doesn't recognize the parameter IBV". In this case, however, as long as users didn't try to use this parameter, PSpice appeared to be using the same model as Spice3 and **Xyce**. We concluded that this particular parameter represented an enhancement to the Spice3 diode, not a major departure.

One final point: From a commercial vendor's point of view, it is a selling point for them to support an industry-standard model. That's a necessary "hook" to lure customers away from rival codes. The EDA industry is very competitive, and it wouldn't make sense for a vendor to secretly make significant changes to their implementation of the BSIM3, or any other industry-standard device. If they wanted to make changes, they'd probably just create a new model, and advertise it as such - "the new Silvaco high-frequency BJT" or something like that.

9.2.14 So, if the models in **Xyce** are the same as commercial simulators, how is **Xyce** different?

The answer to the question 9.2.13 doesn't mean that **Xyce** isn't different from commercial codes, just that the differences are generally not in the legacy device models. Commercial simulators are mostly different with regard to:

- solver technology.
- the GUI.
- output file format.
- code structure.
- analysis capabilities (most support `.AC`. We currently do not, for example.)
- platform support (a lot of commercial simulators are mainly for Windows).
- large model parts libraries (parameter set databases).
- a lack of hostile environment models (radiation, etc.)

9.2.15 I am really convinced that the legacy SPICE code is calculating this small term incorrectly. Can I delete it?

Probably not, for several reasons.

First, you're probably wrong. Really. It is extraordinarily rare for the legacy code to be so wrong that a particular effect should be removed. It is most likely that you have coded it incorrectly, and you should assume that this is the case until you can absolutely prove otherwise. In almost every case that we've had where a developer suspected the legacy code was bad, it has turned out to be something else. It is very easy to become convinced that certain terms are wrong, especially after an extended period of pouring over debugging output in the small hours of the morning.

But it is not unheard of that legacy devices have coding errors. If you can prove that there is in fact an issue in the legacy code that is causing a problem, it is usually the case that the real issue is that Xyce solves things differently than SPICE does, and certain types of errors that kill Xyce simply don't matter to SPICE. An error of this sort was once found in the MOSFET level 3, where a pair of charge-fitting expressions used in different ranges of voltage were supposed to be continuous at the transition — and were not because of a mistake in the constant term. In this case, since only the derivative really mattered (i.e. the current, which matched on either side of the transition), this error somehow had no effect on SPICE. When Xyce attempted to do its numerical derivative of charge to get the current, the discontinuity caused it to see a massive impulse current as soon as the voltage crossed the transition point — at which time it would repeatedly scale back the timestep, see an even bigger current, and spiral down the timestep until it crashed.

When finding such an error — which takes some intensive detective work and detailed debugging using both SPICE and Xyce, and not just hand-waving — it is not correct to delete the offending expressions, but to correct them and equally important document the correction. At a minimum there should be detailed explanations of your changes in the commit log, and comments in the code. It is also usually wise to leave the incorrect code enclosed in an "ifdef" so that the traces of your change are obvious.

But really, you're probably mistaken, and the error is not in the legacy device. "Fixing" a port of a SPICE model to Xyce almost always means undoing the mistakes you made in porting it.

For a related question, see question 9.8.10.

9.2.16 Do we support bypass?

No, we don't. We've never really supported bypass, although it was included in the initial **Xyce** implementation of the BSIM3 a long time ago (circa 2001). At the time, we experimented with it and concluded it was a bad idea. The bypass code was then removed from the BSIM3 and bypass has not been implemented in any other **Xyce** device.

The idea behind bypass is to check the devices to see if the inputs to a device have changed since the last load. If they haven't changed (to within a tolerance), then the device doesn't recalculate its currents or conductances, it just re-loads them. This requires storing the previous load values and keeping them around until the end of the run.

In SPICE, this works a little better than in **Xyce**, as the bypass stuff is intimately tied to its convergence tests. By definition, in SPICE, if a device us "bypassing" that means that it is converged. In **Xyce**, this isn't necessarily true, which can cause problems. The tolerances used to invoke bypass in **Xyce** have nothing to do with the tolerances used by the solvers.

In practice, this tended to lead to the **Xyce** nonlinear algorithm not making any more progress towards convergence, once bypass was turned on for a device. This was a undesirable effect, so we've never supported bypass.

Also, note that modern circuit codes apparently don't use bypass either. Kundert [22] refers to bypass as a "dubious" optimization, which was "not implemented in Spectre". According to Kundert, "bypass has been found to improve performance by 15-30%. In the worst case it degrades performance, injects noise into the circuit ..., causes convergence problems and wastes a considerable amount of memory."

So, anyway, don't try to copy SPICE's bypass stuff into a **Xyce** device - ignore that stuff.

9.2.17 What should we do about SPICE convergence checking code?

In most SPICE devices you'll often see a block of code that checks how much the device inputs have changed with respect to a tolerance (voltTol, currentTol, etc.) In **Xyce** all of our convergence checking is done in the solvers, not the devices. In keeping with our code design, solver issues should not be part of the device package in **Xyce**.

The tolerance stuff is also used by SPICE as part of bypass (see question 9.2.16). If **Xyce** supported bypass, it might make sense to include it in a **Xyce** model implementation. However, we do not support bypass, so any code performing tolerance checks in SPICE should not be imported into **Xyce**.

9.2.18 Can we create a module that will allow us to plug SPICE devices directly into **Xyce**, without modification?

This has been suggested before, but it isn't likely. The fact that **Xyce** uses a different linear system (see section 9.4) makes doing this difficult, although not impossible.

However, as the code has evolved, we've been gradually moving further away from SPICE, rather than closer. The new-DAE formulation (see question 9.5.8) is a good example of this. It is hard to imagine how one would "plug in" a SPICE device to that formulation. In that formulation, the charge (and flux) terms are stored separately in the Q-vector, which is organized around circuit nodes (like the F-vector) rather than around capacitor branches, like the SPICE state-vector. In theory, it might be possible to make an automatic mapping between the SPICE state vector and the **Xyce** Q-vector, but it would be difficult.

In general, as we'll be moving further from SPICE in the future, plugging SPICE devices directly into **Xyce** will become even more difficult. Given that we're nearly out of SPICE devices anyway (and are re-directing our focus to using Verilog-A model compilers like ADMS [21]), we aren't likely to ever plug SPICE devices directly into **Xyce**.

9.2.19 Why are we moving farther away from SPICE? Wouldn't it be easier to follow SPICE more closely?

Yes, it would be easier, but then part of the rationale for this project would go away.

Part of the point of **Xyce** is to develop a code that pushes the "bleeding edge" of circuit solver technology. SPICE has (by modern standards) a cryptic, restrictive, 20-year old, design. If we followed this design too closely, it would stifle numerical innovation. As it is, we're still pretty restricted. SPICE is a long-term industry standard, so users expect **Xyce** to be bug-compatible with SPICE.

Rewriting device models from SPICE to **Xyce** is difficult, but it isn't impossible. That's why we hire PhDs.

Finally, don't compare this task to adding SPICE devices to a SPICE-based code (like ChileSPICE, or your locally hacked version of Spice3f5). If the model is SPICE-based, and the target application is SPICE-based, then the job of adding the model is trivial. All the hard work was done by the SPICE model developer. All the code developer really has to do, in that case, is make minor modifications to a few source files and the Makefile.

9.2.20 Why is the BSIM3 level=9 rather than level=8?

There is no gold standard for level numbers. Or at least, many model level numbers are not very standardized.

9.3 Equation set

9.3.1 Do SPICE and **Xyce** solve the same equations?

The short answer is yes. Both codes rely on the "modified KCL" equation set. This is also referred to as "modified nodal analysis". See the **Xyce** Math document for a detailed explanation.

The fact that both codes use modified nodal analysis means:

- Most equations are Kirchhoff current law equations.
- Most solution variables (that are part of the linear system, not a post-process) are voltage node values.
- The only exceptions to the first 2 items in this list are devices which have non-Ohmic currents (currents which are a function of voltage). The most common example is the independent voltage source.

Note that **Xyce** may someday use a different type of analysis, such as tableau analysis, but this has not happened yet.

9.4 Linear System

9.4.1 How is the **Xyce** linear system different from SPICE?

Both SPICE and **Xyce** use Newton's method to solve the nonlinear system. Both **Xyce** and SPICE solve a linear system at each iteration of Newton's method. The linear systems used by the 2 codes are related, but not the same. Most traditional nonlinear solvers (including **Xyce**) will solve this nonlinear system at each Newton iteration:

$$\mathbf{J}\Delta\mathbf{x}^{k+1} = -\mathbf{f} \tag{9.1}$$

$$\mathbf{x}^{k+1} = \mathbf{x}^{k} + \Delta\mathbf{x}^{k+1} \tag{9.2}$$

The index, k, is the Newton iteration step number. $\mathbf{J}$ is the Jacobian matrix, and $\mathbf{f}$ is the residual vector. $\mathbf{J} = \delta\mathbf{f}/\delta\mathbf{x}$.

In contrast, the SPICE nonlinear iteration is accomplished by solving this equivalent linear system:

$$\mathbf{J}\mathbf{x}^{k+1} = -\mathbf{f} + \mathbf{J}\mathbf{x}^{k} \tag{9.3}$$

The SPICE equation, given by equation 9.3 is algebraically equivalent to the traditional system, given by equations 9.1 and 9.2. Note that the term $\mathbf{J}\mathbf{x}^{k}$ is a vector term. Using equation 9.3 has one less step involved, so it probably requires slightly fewer floating point operations to execute. This slight optimization <u>might</u> by why SPICE uses it. (or it might not)

Equation 9.1 and/or equation 9.3 are solved by invoking a linear solver such as KLU, or AztecOO. Equation 9.2 is solved simply by the vector addition of $\mathbf{x}^{k}$ to $\Delta\mathbf{x}^{k+1}$.

For either approach, the code is trying to obtain a correct solution vector, $\mathbf{x}$, and declares success when (by some measure) the norm of the $\mathbf{f}$ vector is small. In circuit codes, the matrix $\mathbf{J}$ and vector $\mathbf{f}$ are provided by the device models.

As you can see, a SPICE device model has to provide $\mathbf{J}$ and $\mathbf{f}$, and the same is true of a **Xyce** model. However, the right hand side of equations 9.1 and 9.3 is different, and as such a different system is handed off to the linear solver.

SPICE devices load $-\mathbf{f} + \mathbf{J}\mathbf{x}^k$ directly into a vector structure - they don't set up a separate $\mathbf{f}$ vector and $\mathbf{J}\mathbf{x}^k$ vector. Unfortunately, SPICE doesn't even go out of its way to spell out which device contributions are part of $\mathbf{f}$, and which ones are part of $\mathbf{J}\mathbf{x}^k$. This issue is the main thing that makes converting a SPICE device to a **Xyce** device difficult.

9.4.2 Are there any other right-hand-side issues to be wary of? Or, how do we handle linear resistors?

YES! Linear resistors are a potential "gotcha".

Because SPICE loads $-\mathbf{f}+\mathbf{J}\mathbf{x}^k$ for the right hand side, it means that any terms in $\mathbf{f}$ that are purely linear are cancelled when $\mathbf{J}\mathbf{x}^k$ is subtracted off. This has been a significant source of coding errors in converting SPICE models to Xyce.

As an obvious example, consider the simple resistor, whose $\mathbf{f}$ vector is simply $\frac{(v_{pos}-v_{neg})}{R}$, or $(v_{pos} - v_{neg}) * G$. This is precisely the same as the Jacobian multiplied by the current solution vector, so the SPICE load function would add zero in to the right-hand side. Rather than wasting code to add in and subtract the same thing, there is simply no code in SPICE related to loading the right-hand side for the resistor. In Xyce, however, it is necessary to calculate and load this term.

This issue comes up immediately for any semiconductor device model that contains parasitic resistance models, such as the BJT or any of the MOSFET models. In SPICE, the terms involving the parasitic resistors are missing from the right-hand side load, but are present in the Jacobian. Another common place for this issue is when GMIN is added to Jacobian elements. These result in additional linear right-hand side terms that are usually not included in the SPICE implementation. It is essential that the Xyce developer be aware of these missing linear terms and add them in appropriately.

9.4.3 What is GMIN, anyway?

GMIN is a SPICE-ism, that we've decided to adopt. It is apparently intended to improve solver stability, but we've never been able to prove (to ourselves) that it actually helps much. However, for the sake of SPICE compatibility, we've made it part of **Xyce**.

GMIN is a global device package parameter. It is a small conductance, which has a default value of 1.0e-12. In SPICE, you can set GMIN from the netlist with:

```
.options gmin=1.0e-7
```

In **Xyce**, it is set like this:

```
.options device gmin=1.0e-7
```

Many SPICE transistor devices include "gmin" resistors. These resistors are not connected to anything physical - they are just there to allow a small amount of current. This probably adds some degrees of freedom to the circuit solve, and it could be argued that might make things easier for the nonlinear solver. Also, having them there might improve the conditioning of the Jacobian, thus helping the linear solver. However, in practice, we haven't seen **Xyce** behave much differently with them there vs. not there.

In SPICE, it is impossible to set gmin to zero from the netlist, but you can always hack SPICE code to force gmin=0.0 if you you need to. If you don't hack SPICE, then SPICE will always force it to some nonzero number (I think the minimum is 1.0e-14). In **Xyce**, however, it is possible to make it truly zero. **Xyce** doesn't impose a limit, nor do we plan to implement a limit.

Often, the last remaining issue in a device, the one final thing that prevents a **Xyce** device from completely matching SPICE, are the gmin resistors. As they are very small, it is often difficult to notice that they are missing. If you are seeing subtle differences between your **Xyce** device and the equivalent SPICE device, check to see if the GMIN resistors are correctly set up.

Note, as they are linear resistors, you need to understand the answer to question 9.4.2.

9.4.4 GMIN is such a small term. Why does it matter so much?

If you implement gmin correctly, then it doesn't change the answer very much, for most circuits.

The real problem comes if you implement gmin in a mathematically inconsistent way. Generally, if you have gmin in your device, you must set it up correctly, or your device will be numerically unstable. The device, if it converges, may give an answer that looks reasonable, so it is easy to not notice this type of mistake at first. However, a gmin mistake always means that the device will numerically diverge a lot more often than it should.

That may sound hard to believe, given that gmin is (typically) 1.0e-12, but it has been observed for many devices, during the many years that **Xyce** has been under development.

Linear resistors are handled much differently in **Xyce**, compared to SPICE. (see question 9.4.2). Because of this, a very common error in **Xyce** devices happens when a **Xyce** developer naively copies the SPICE Jacobian (which includes gmin), but doesn't set up the corresponding right hand side terms. In general, it is better to leave gmin out of the device altogether, than it is to set it up with this mistake.

9.4.5 What is GMIN stepping?

Gmin stepping is a nonlinear solver algorithm used by SPICE. It is a continuation algorithm, similar to some of the ones we use in **Xyce** with LOCA. The name, however is deceptive. Continuation algorithms typically work by sweeping a parameter over a range of values, and having the nonlinear solver do a Newton solve at each value. At the end of the parameter sweep, the solved solution is a solution to the desired problem that was originally specified by the user.

The reason the term "gmin stepping" is deceptive is that it implies that the various GMIN resistors, present in many SPICE devices, are set to some artificial value and progressively modified over the course of the sweep.

What actually happens is that a variable, called diagGmin, is summed on to the diagonal of the Jacobian matrix in SPICE. Initially, this diagGmin variable is set to a scalar of gmin. By default, it is 10 orders of magnitude larger, so by default diagGmin is initially 1.0e-2. The continuation sweeps over diagGmin, going one order of magnitude at a time. On the final step, diagGmin is set to zero, so for the final solve, the original circuit problem is being solved.

In effect, what this does is put a large resistor in between every single voltage node and ground. On the first step, given that diagGmin is 1.0e-2 by default, this resistor is 100 ohms by default. Over the course of the continuation, the resistor gets larger and larger and larger, until it is essentially infinite (or the current between this node and ground is zero).

Given that diagGmin is put onto every diagonal in the matrix, that means that diagGmin is occasionally added to voltage drop equations, rather than KCL equations. Recall that in SPICE and **Xyce**, most equations are Kirchhoff current law (KCL) equations, and that most variables are voltage node variables. Occasionally, a device (like an independent voltage source) exists that doesn't have an Ohm's law I-V relationship. For devices like this, it is necessary to add a voltage drop equation, and a corresponding current variable. Anyway, the "gmin stepping" algorithm puts diagGmin on every single diagonal of the matrix, so it is added to a handful of equations that are associated with a current variable, rather than a voltage variable.

For such equations, adding diagGmin is not adding a resistor to ground - it is doing something else. It is essentially adding a (const*I) term, instead of a G*V term.

GMIN stepping has been implemented in **Xyce**, but it is not automatically invoked like it is in SPICE. In SPICE, when the traditional DCOP fails, the code automatically attempts GMIN stepping and if that fails it automatically attempts to use source stepping to solve the DCOP. In **Xyce** these options have to be turned on manually.

9.4.6 Why didn't **Xyce** just use the same linear system as SPICE?

Because we think SPICE's linear system is weird. Hardly any other implicit numerical codes use it. This is in part because many, many nonlinear solver algorithms depend upon being able to manipulate $\Delta \mathbf{x}$. In SPICE's approach, there is no $\Delta \mathbf{x}$ vector, so you can't do anything to it directly. Also, a lot of numerical algorithms depend on testing the $\mathbf{f}$ vector, and in SPICE's linear system, there is no true $\mathbf{f}$ vector.

From a pragmatic point of view, **Xyce** depends a lot on the ASC Algorithm libraries, which includes the NOX nonlinear solver library, and LOCA (library of continuation algorithms). (actually, LOCA is part of NOX, but whatever) Both of those libraries, as they are designed with a variety of applications in mind, not just circuits, expect to perform a traditional nonlinear solve, not a SPICE-style nonlinear solve. For us to use these libraries (and we need to use them), **Xyce** needs to set up a traditional nonlinear solve.

9.4.7 Is there an easy way to reverse engineer the SPICE residual?

Yes. Or, at least there is a rule of thumb that will work most of the time.

If you look at the load functions of a SPICE device, you'll probably notice that there are a lot of currents being calculated, and a lot of conductances being calculated. Conductances are generally derivatives of current with respect to voltage ($\delta I/\delta V$), so in general, they belong in the Jacobian, $\mathbf{J}$. However, you'll notice that they are used in the right-hand-side vector a great deal, via $G \cdot \Delta V$ terms. For example, you will frequently see expressions like this:

```
here->BSIM3csub = Isub - (Gbb * Vbseff + Gbd * Vds + Gbg * Vgs);
```

(this example is take from the SPICE BSIM3 device). Here, `Isub` is a current variable. `Vbseff`, `Vds`, and `Vgs` are all junction voltages. (i.e. the difference between two nodal voltages). `Gbb`, `Gbd` and `Gbg` are all conductance variables. So, variables starting with the letter `I` are usually currents, variables starting with the letter `V` are usually voltages, and variables starting with the letter `G` are usually conductances. (note: there are, of course, exceptions to this!)

`csub` is the variable that will actually be put into the right-hand-side vector from the SPICE BSIM3. Near the bottom of the BSIM3 load function (the SPICE version), you'll find this:

```
ceqbs = -here->BSIM3csub;
```

And later on, you'll see this:

```
(*(ckt->CKTrhs + here->BSIM3bNode) -=(ceqbs + ceqbd + ceqqb));
```

(Note: these terms have been simplified, somewhat.) What th is all means is that `csub` corresponds to $-\mathbf{f} + \mathbf{J}\mathbf{x}^k$. Therefore, `Isub` corresponds to $-\mathbf{f}$, and all the `G*V` terms correspond to $+\mathbf{J}\mathbf{x}^k$.

`csub`, starting with the letter `c`, is kind of a current, but only kind of. It has the units of current (amps). However, given that most conductances in the BSIM3 are nonlinear, terms like `Gbb * Vbseff` are not intended to provide currents that correspond to real currents in the device. They are there to make the linear system consistent.

So, here is the rule of thumb: Assume that current variables (starting with `I`) are part of $\mathbf{f}$, and that conductance-voltage terms (`G*V`) are part of $+\mathbf{J}\mathbf{x}^k$.

9.4.8 In **Xyce**, what do we do with $\mathbf{J}\mathbf{x}^k$?

In reading the answer to the previous question, it might seem that the $\mathbf{J}\mathbf{x}^k$ term isn't needed in **Xyce**. After all, it doesn't appear in equation 9.1. However, we don't throw it away. If we want to use voltage limiting (a nonlinear solver enhancement that SPICE uses) then we need to keep it, in a modified form. When voltage limiting is enabled, the linear system solved at each Newton step is given by:

$$\mathbf{J}\Delta\mathbf{x}^{k+1} = -\mathbf{f} + \mathbf{J}\Delta\mathbf{x}_{voltlim} \tag{9.4}$$

The effect of $\mathbf{J}\Delta\mathbf{x}_{voltlim}$ is to force the nonlinear solver to include the changes due to voltage limiters in the nonlinear step.

If **Xyce** is run with this netlist option:

```
.options device voltlim=0
```

then **Xyce** solves equation 9.1, and it doesn't need anything like the $\mathbf{J}\mathbf{x}^k$ term. However, if **Xyce** is run with this netlist option:

```
.options device voltlim=1
```

then voltage limiting is enabled, and **Xyce** solves equation 9.4 instead of equation 9.1. (Note that this is the default in **Xyce**.) In this case, a term like $\mathbf{J}\mathbf{x}^k$ is needed, but the equivalent term in **Xyce** is given by $\mathbf{J}\Delta\mathbf{x}_{voltlim}$. This translates, in the **Xyce** source code to something that looks like this:

```
csub = Isub;
if (devOptions.voltageLimterFlag) // set from the netlist
{
  csub_Jdxp = - (   Gbb * (Vbseff-Vbseff_orig)
  + Gbd * (Vds - Vds_orig)
  + Gbg * (Vgs - Vgs_orig));

  csub += csub_Jdxp;
}
else
{
  csub_Jdxp = 0.0;
}
```

In the above code fragment, the term `csub_Jdxp` is the $\mathbf{J}\Delta\mathbf{x}_{voltlim}$ term, while `Isub` is the $\mathbf{f}$ term. Compare this code fragment with the fragment in question 9.4.7. If you can see the pattern match between these two code blocks, you'll be able to convert most devices from SPICE to **Xyce** quickly.

9.4.9 What is voltage limiting?

Voltage limiting is a unique circuit nonlinear solver enhancement. For a mathematical description, see the **Xyce** Math document. The premise for it is to prevent junction voltages from changing too much from Newton step to Newton step. For some devices, this is crucial. In particular, devices with an exponential current-voltage relationship are very sensitive to minor changes in voltage, and will often "blow up" numerically.

Voltage limiting is only set up in nonlinear devices, such as diodes, BJTs and MOSFETs. There are 3 functions that are commonly used: `fetlim`, `pnjlim`, and `limvds`. These 3 functions are available in **Xyce**, and while some devices (like the BSIM SOI) use other functions, these are by far the most common.

Voltage limiting can be thought of as the code "going into denial". During a device load, a device will first obtain nodal voltages from the solution vector of the current nonlinear iteration. What follows is a simplified example, for a diode. For any device, the first step is to obtain relevant nodal voltages from the solution vector. This is done near the top of the `updateIntermediateVarsBlock` function:

```
    Vpp = (*solVectorPtr)[li_Pri];
    Vn  = (*solVectorPtr)[li_Neg];
```

Voltage is a relative quantity, so currents in the diode are calculated based on voltage drops. These are calculated next:

```
Vd = Vpp - Vn;
```

For safekeeping, this junction voltage is saved:

```
Vd_orig = Vd;
```

At this point, the device "limits" the voltage:

```
Vd = devSupport.pnjlim(Vd, Vd_old, Vte, tVcrit, &ichk);
```

The input value, `Vd`, was just calculated. `Vd_old` is from the previous Newton iteration, if it exists. `Vte` and `tVcrit` are inputs that determine the extent to which `Vd` should be limited. (they help determine the shape of the I-V curve, which determines how much `Vd` should be allowed to change) This function call returns a new value for `Vd`. Sometimes, the same value of `Vd` is returned that was passed in. If `Vd` is changed by this function call, then `Vd-Vd_orig` is nonzero.

At this point, now that the limiter functions have been called, it is safe to reset `Vd_old`:

```
Vd_old = Vd;
```

From this point onward, the device uses the new value for `Vd` in all of its current and voltage calculations. The problem with this is that the current and voltage calculations are not consistent with the contents in the solution vector. **Xyce** is not aware of this change to `Vd`, except inside this device.

To correct for this, this change to `Vd` is incorporated into the right-hand-side vector load of this device. The correction term (referred to previously in this chapter as $\mathbf{J}\Delta\mathbf{x}_{voltlim}$) forces the nonlinear solver to include this change in the $\Delta\mathbf{x}$ vector, when it is calculated for the current nonlinear solver iteration.

So, when the final load is performed for the diode, in the function `loadRHSBlock`, the load has this form:

```
double Gd_Jdxp = 0.0;
double Vd_diff = Vd - Vd_orig;

if( devOptions.voltageLimiterFlag && !origFlag)
{
  Gd_Jdxp = -( Gd + Gcd ) * Vd_diff;
}

coef = Id + Icd + Gd_Jdxp;
(*extData.RHSVectorPtr)[li_Neg] += coef;
```

Note that this example has been simplified.

9.4.10 Why do we sometimes turn voltage limiting off? SPICE doesn't

Because when voltage limiting is on, the right hand side vector (`RHSVector`) is not $\mathbf{f}$ (the residual). Some nonlinear solver algorithms depend on this vector structure being $\mathbf{f}$, and nothing but $\mathbf{f}$. Also, some nonlinear solver algorithms need to be able to load and re-load $\mathbf{f}$, multiple times at each Newton step, and reliably get the same $\mathbf{f}$ for the same $\mathbf{x}$. Voltage limiting is not set up in a sophisticated enough way to do this without introducing hysteresis into $\mathbf{f}$.

Xyce actually does create another vector (`fvector`) which contains nothing but $\mathbf{f}$. However, it is difficult to make NOX (or any solver) understand that it needs to use `fvector` for some uses and `RHSVector` for others. Generally, these libraries are set up to use one vector for everything.

The main solver algorithm that is inconsistent with voltage limiting is line search, and all of its variants. Currently, there is no easy way to use any type of line search and voltage limiting at the same time. In other words, you should never see this in the input file:

```
.options device voltlim=1
.options nonlin searchmethod=2
```

If `voltlim=1`, then `searchmethod` should never be anything other than `0`.

Voltage limiting could be implemented with better bookkeeping, to get line search to work with it. However, given that line search and voltage limiting are essentially trying to accomplish the same thing (reduce the nonlinear step size), the benefits of combining the two methods are not clear.

SPICE doesn't give you the option to turn off voltage limiting. This simplifies the SPICE source code, and for most circuits voltage limiting is an effective algorithm. SPICE doesn't have nearly as many solver options as **Xyce**, so it doesn't have to accommodate them.

9.4.11 What other things are enabled by voltage limiting?

The netlist specification, `.options device voltlim=1` turns on several things that use the voltage limiter machinery in **Xyce**.

Anything in a device, that modifies junction voltages to values other than what would have been obtained directly from the solution vector has to be treated as a voltage limiter. This includes:

- Initial junction voltages, that are set on the very first Newton step of the first DCOP solve.
- The SPICE way of applying `IC=`.
- bypass, if we used it.

Each of these things will change junction voltages to non-solution-vector values. As such, these changes need to be propagated back to the nonlinear solver, in the same manner as voltage limiter changes.

When viewed in this way, it is more accurate to think of `.options device voltlim=0` really meaning, “remove all hacks that artificially change device junction voltages”. It could also be thought to mean “Force the RHSVector to contain $\mathbf{f}$, and only $\mathbf{f}$.

9.4.12 Does it matter if I use the same voltage limiter functions in my **Xyce** device as are used in the equivalent SPICE devices?

In general, yes, it does matter.

It is true that the exact limiter function used doesn’t really change the model. Voltage limiting is really a solver enhancement, so one could make the argument that using a different limiter function (from the original SPICE device) is OK. So, for example, in your implementation of the diode, you could use a new function ”newlim” instead of ”pnjlim”, and your diode would still be a diode.

However, the problem with doing this is that it makes comparing to the original SPICE device much more difficult. In practice, the fastest way to debug a **Xyce** model is to make it identical to the SPICE equivalent. If you use a different limiter function, you’ve rendered that debugging approach nearly impossible. The benefits of doing a lot of detailed, direct comparisons to SPICE far outweigh any benefit that you could possibly get from using your own limiter. If you must come up with a new limiter, do it much later. Do your initial development and debugging using the same limiter as SPICE. Once you are 100% certain that the model is correct, then you can consider using a different function, but only then.

Another issue is that one common bug report is, ”this circuit worked fine in PSpice (or Spice3, or ChileSPICE, or whatever), but fails in **Xyce**.” If you make certain to use the same limiter functions, and apply them in an equivalent manner, this type of bug report is a lot less likely. Generally, the SPICE limiters have been set up that way for a reason - they’ve been well-tailored to that device, and the particular I-V relationships of that device.

To put it bluntly, if you try to come up with your own limiter, you are probably making a lot of pointless work for yourself. Instead of just copying over a few lines of SPICE (which should be trivial), you’ll have to invent something yourself. Chances are, whatever you invent won’t work as well, unless you spend a lot of time thinking about it.

You should only try to come up with a different limiter if you have a compelling reason to think the original SPICE limiter is inadequate. One example of a valid, compelling reason would be if a user has a very important circuit that won't converge with the original SPICE limiter. If this is your justification, make 100% sure that it is the SPICE limiter's fault, and not some other bug in the device. One good test is to see if the users' circuit runs in Spice3, or ChileSPICE, or some other SPICE variant. If it runs in SPICE, then the issue is not the limiter - you probably have a bug in the implementation of the device.

9.4.13 Are the SPICE and **Xyce** Jacobians the same?

The short answer is yes, they should be identical at every Newton step. This comes with a few caveats:

- The variable ordering will be different - for example the (row=5, col=5) element of a **Xyce** matrix may actually correspond to the (row=3, col=3) element of a SPICE matrix.
- Jacobians are 100% the same for the DCOP case. For transient, they will only be identical if the time step size is identical. For testing purposes, it is easy to force this to happen, by using the constant step size option in both codes.
- **Xyce** has a lot of solver options that SPICE lacks. Only compare Jacobians if you are running with equivalent options.

By default, **Xyce** runs with options that are roughly equivalent to SPICE.

9.4.14 Are the SPICE and **Xyce** right-hand-side (RHS) vectors the same?

The short answer is no. The one exception is the very first residual (RHS vector) of the very first Newton solve. At this stage of the solution, the initial guess to the solution vector is all zero's. Thus, by coincidence, the $\mathbf{x}_{orig}$ vector can always be considered to be zero, and the **Xyce** and SPICE linear systems become identical. (for this case $\mathbf{x}^k = \Delta\mathbf{x}_{voltlim}$) This will not be true for any other nonlinear iteration.

9.4.15 Are the SPICE and **Xyce** solution vectors the same?

The short answer is yes. The same rules hold true as for the Jacobian. Variable orders will be different, and in transient time step sizes may be different. Also, make sure the same solver algorithms are used. Other than that, the solution vectors should match at every Newton step.

9.5 Time Integration

9.5.1 How are time derivatives calculated in **Xyce**?

The answer to this question pertains to the old time integrator. There is a new time integrator (see the new-DAE formulation, question 9.5.8), for which this question does not apply. Also, note that the old time integrator was removed from **Xyce** in 2008. This question is left in place merely for historical purposes.

Time derivatives are calculated (for now, with the old time integrator) via the state vector(s). Any quantity that needs to be differentiated with respect to time should be set up as a state variable, and put into the state vector. (`extData.nextStateVector`). This placement into the state vector should happen in the updatePrimaryStateBlock function.

After the updatePrimaryStateBlock function has exited, the time integrator will calculate time derivatives of the entire vector, and put the result into the `extData.nextStateDerivVector`. Using the same state vector index, you can then obtain the time derivative that you need.

For a concise example of all of this, see the capacitor device (N_DEV_Capacitor). Most time derivatives are of capacitor charges, q.

So, the procedure should be, to calculate dq/dt, do the following.

- In updateIntermediateVarsBlock, calculate the most up-to-date C and q that you can. In the capacitor device, C is a constant provided by the user, and q=C*V. In other devices such as the BSIM3, C and q are calculated independently from a bunch of other quantities, and $C = \frac{dq}{dV}$. In still other devices (notably the level 1 and level 3 MOSFET devices), q is calculated from the capacitance and voltage by a simple approximate integration (this is sometimes referred to in comments by the imprecise term “Meyer Back-Averaging”. See question 9.5.6 for more details.).
- in updatePrimaryStateBlock, place q in the state vector. It will look something like:

```
 staVectorPtr = *(extData.nextStaVectorPtrPtr);
(*staVectorPtr)[li_QState] = q0;
```

- in the updateSecondaryStateBlock function, obtain dq/dt. It will look something like this:

```
        i0 = (*(*extData.nextStaDerivVectorPtrPtr))[li_QState];
```

Once you have `i0` (or whatever you call it) you can use it to set up the residual vector.

For a related question, see question 9.6.4.

9.5.2 What should go into the state vector?

Anything you want to differentiate with respect to time, and not much else. Occasionally, you may need to access the previous history of a device, and you can use the state vector to do this. This second justification (previous history) is rare.

This issue is sometimes confusing for people who have looked at SPICE code, as SPICE uses the state vector for all sorts of things. We don't. Most times that a SPICE device puts stuff in the state vector, it is not necessary for us to do that.

Remember, as **Xyce** is a C++ code, most data an individual device needs can easily be stored in said device's instance or model class. It isn't necessary to bother with the state vector. The entire state vector gets differentiated, many times, over the course of a run, so anytime you put stuff in it that doesn't strictly need to be there, you'll just be adding extra work to the time integrator.

For example, it is almost never necessary to put DC (or transient, for that matter) conductances in a state vector. It is almost never necessary to store junction voltages either, although this can be handy for enforcing voltage limiting from one time step to another.

9.5.3 SPICE puts variable x into the state vector. Should I also put variable x into the state vector?

Probably not, unless you need to obtain dx/dt. See question 9.5.2.

Occasionally, there will be other state vector uses, but they are rare. For example, the transmission line device needs to refer to previous points in the time history, and the amount of time it needs to "look back" is completely arbitrary and specified by the user. So, the only easy way to support this is via the state vector. For this type of exception, it is OK to use the state vector.

It is not appropriate to put things into the state vector, just for safekeeping. You should be able to store most of your information in the local instance class.

9.5.4 What is `solState.pdt`?

This term is used for capacitor current Jacobian terms in **Xyce**. "pdt" is an acronym for "partial time derivative". Capacitor currents are in **Xyce** given by:

$$I_{cap} = \frac{dq}{dt} = \frac{\alpha}{\Delta t}(q_{i+1}) + \text{other terms} \tag{9.5}$$

The index $i+1$ is intended to denote the most recent time point. It is still unknown, and subject to the (in progress) nonlinear solve. The "other terms" contain information about previous time points (i, $i-1$, etc., depending on the integration method). These previous time points have already been obtained, and are no longer unknowns from the perspective of the nonlinear solver. Thus, from the nonlinear (and linear) solver's point of view, the only term in equation 9.5 that will give a nonzero derivative with respect to the solution vector is the leading term, $\alpha q_{i+1}/\Delta t$.

The backward differentiation formulas (BDFs) used by the old **Xyce** time integrator all have the form given by equation 9.5. `solState.pdt` is given by:

$$\texttt{solState.pdt} = \frac{\alpha}{\Delta t} \tag{9.6}$$

`solState.pdt` is used for capacitor Jacobian contributions (capacitor conductances). The conductance for any capacitor in **Xyce** is given by $C\alpha/\Delta t$, or (in other words), `C * solState.pdt`.

This is an approximation for capacitor conductance that isn't quite correct for nonlinear capacitors. In **Xyce** (and SPICE) the derivative of equation 9.5 with respect to V is approximated by:

$$G_{cap} = \frac{dI_{cap}}{dV_{i+1}} = \frac{dq_{i+1}/dt}{dV_{i+1}} = \frac{CdV_{i+1}/dt}{dV_{i+1}} = \frac{C\alpha V_{i+1}/\Delta t + \text{other terms}}{dV_{i+1}} = C\frac{\alpha}{\Delta t} \tag{9.7}$$

Note that for Backward-Euler, $\alpha = 1.0$, and the "other terms" are just $q_i/\Delta t$. For other BDFs, α can be other numbers. Equation 9.7 is an approximation that depends, in part, on $dq_{i+1}/dt = CdV_{i+1}/dt$. This assumption is only true for linear capacitors, but equation 9.7 is used in **Xyce** (and SPICE) for nonlinear capacitors anyway. In practice, it has been good enough.

To set up the capacitor conductance for a **Xyce** device, a pattern like this should be used (taken from the BJT device):

```
if (!solState.dcopFlag)
{
  gCapBEdiff = capBEdiff * solState.pdt;
}
else
{
  gCapBEdiff = 0.0;
}
```

Never use the actual time step size directly to set this up. In other words, don't set `gCapBEdiff = capBEdiff/solState.currTimeStep`. This will only be correct if α coincidentally happens to be 1.0, and there is no guarantee that this will be true. The **Xyce** device package doesn't have a way to directly obtain α.

NOTE: In the BSIM devices, the variable `ag0` is the equivalent of `solState.pdt`. When implementing a BSIM device, make sure the **Xyce** version sets ag0 to solState.pdt. So, if you see a line like this:

```
gcgmdb = -cgdo_local * ag0;
```

The variable `gcgmdb` is a capacitor "conductance", and the variable `cgdo_local` is a capacitance, and `ag0` is $\alpha/\Delta t$.

9.5.5 Why can't I put a differentiation formula directly in a device?

Some backward-differentiation formulas (BDFs) are easy to set up (in particular, Backward Euler). SPICE obtains time derivatives by calling `NIintegrate` whenever it needs one. When implementing a SPICE device in **Xyce**, it can be tempting to just hardwire a BDF in the corresponding locations in the **Xyce** source. Sometimes, using the "state" functions can be a hassle, as they apply some constraints on when you can obtain derivatives.

However, you should never hardwire BDF formulas in devices, unless you have a really, really good reason to do so. All the time stepping and error control is handled by the time integration package, and it can't do it correctly if you aren't using its time derivatives. Also, **Xyce** allows users to set the integration method from the input file, as well as time integrator tolerances. If you hardwire time derivatives into the device package, you won't use any of those options.

Finally, philosophically, calculating time derivatives really isn't the job of the device package. The device package is a physics package, not a solver package. Most issues pertinent to a solution method should be done in the various solvers packages.

There are a few exceptions to this in **Xyce**, in which time discretizations are hardwired directly inside of device models. The intention is to eventually get rid of them. These exceptions include the handling of Meyer capacitors in the level 1 and level 3 MOSFETs (see question 9.5.6), and the handling of excess phase in the BJT (question 9.5.7).

9.5.6 What is “Meyer Back-Averaging?”

The simplest MOSFET models include computation of “Meyer capacitances” and the charges on these. Since the Meyer capacitances depend on voltage, it is not correct to calculate the charges stored in these parasitic capacitors as $q = C * V$. The correct way to do it is via integration, $q = \int_{v0}^{v1} C(V)dV$. In the level 1 and level 3 SPICE MOSFET models, this is done by using a simple trapezoid rule approximation to the integral:

$$q(t1) = q(t0) + \frac{1}{2}(C(V(t1)) + C(V(t0))) * (V(t1) - V(t0))$$

It is essential to perform the charge computation in this way to guarantee charge conservation.

Unfortunately, the way the original SPICE code was written this approximation looked very much like it was averaging the old and new capacitance. This lead to a number of comments in the Xyce code referring to this as “Meyer Back-averaging,” but it is in fact nothing more than an obfuscated coding of a simple approximate integration.

At some point Xyce may support performing integrations over such quantities through the time integrator in a manner that is consistent with the order of the time integration being performed on the DAE. At that time (if it happens) it might be appropriate to update these simple MOSFET devices to use that facility. On the other hand, for strict SPICE compatibility it might be desirable to leave it alone.

9.5.7 What is the BJT excess phase term?

The excess phase term of the level-1 BJT is one **Xyce** example of a hardwired time derivative function. Explaining excess phase in detail is beyond the scope of this document. From a physical point of view, excess phase is due to distributed phenomena in the base region of the BJT, which causes an extra phase shift than would be predicted by the Gummel-Poon model.

From a mathematical point of view, the excess phase term is represented by a second-order differential equation:

$$\frac{d^2 I_{FX}}{dt^2} + 3\omega_0 \frac{dI_{FX}}{dt} + 3\omega_0^2 I_{FX} = 3\omega_0^2 \frac{I_{CC}}{q_B} \tag{9.8}$$

Being a second-order equation, it doesn't fit naturally into the capabilities of the old time integrator, and for that reason this expression is hardwired (using a Backward-Euler BDF) into the **Xyce**. However, for the new-DAE version of the BJT (see question 9.5.8, the equation has been recast as a pair of first-order equations, and for that implementation, the differentiation is handled by the time integration package.

Note that the new-DAE version of excess-phase is not used by default, even when the new time integrator is being used. The reason for this is efficiency. Using the new form of excess phase requires an extra two solution variables per device, and for circuits with a lot of BJT's in them, this noticably slows down the linear solve time. So, by default, the old, state-vector method is used. If MPDE or harmonic balance are used, then the new form must be used, and that is the default for those types of analysis.

9.5.8 What is the deal with the new-DAE formulation?

This is a new time integrator in **Xyce**. The old time integrator was, essentially, an ODE (ordinary differential equation) time integrator. Note that as of 2008, the old time integrator has been removed from **Xyce**.

The new integrator is more appropriate for DAE (differential algebraic equation) problems, which is what circuits actually are, as it is possible for some (or many) of the equations to lack time derivative terms, thus making them algebraic constraints rather than differential equations. In terms of stability and accuracy, it is better to use algorithms that were designed for DAEs instead of ODEs.

Another advantage of the new time integrator (theoretically, anyway) is speed. The new integrator is variable timestep, variable order, while the old one had a fixed order The new integrator has shown dramatic improvements over the old, in terms of simulation time, and accuracy for some circuits.

Also, the new time integrator is required for the Multi-time PDE (MPDE) algorithm and also the harmonic balance (HB) algorithm, which we are supporting in **Xyce**.

As of this writing the old integrator has been removed. Prior to its removal, it was necessary to support two versions of every device, one for each integrator. After removing the old integrator, the "old" implementations were mostly removed, but there may still be a few devices that have not been purged yet.

The new-DAE formulation casts the problem like this:

$$\mathbf{f} = \frac{d\mathbf{Q}}{dt} + \mathbf{F}(\mathbf{x}, t) \tag{9.9}$$

In this formulation, two different vectors ($\mathbf{Q}$ and $\mathbf{F}$) are set up by the device package. The time integrator is then responsible for summing them together in an appropriate manner to create $\mathbf{f}$

In terms of device implementation, the new-DAE algorithm has a few key differences. For both integrators, the updatePrimaryStateBlock function (and hence, the updateIntermediateVarsBlock function) will always get called.

However, for the new-DAE integrator, updateSecondaryState, and loadRHSBlock and loadAnalyticalJacobianBlock are ignored, i.e. not called. Instead, the new-DAE time integrator will call the vector loaders (the B-vector is combined with the F-vector so it doesn't get its own load function):

- loadDAEQVectorBlock
- loadDAEFVectorBlock

It will also call the matrix loaders:

- loadDAEdQdxBlock
- loadDAEdFdxBlock

The residual (or RHS) vector will be assembled as a linear combination of the FVector and the time derivative of the QVector. (see equation 9.9). The Jacobian matrix will be assembled as a linear combination of the dFdx matrix, and the time-derivative of the dQdx matrix. These linear combinations are handled outside of the device package, and are not the device package's responsibility.

One nice thing about this is that the function call sequence for a device load will be less confusing. Another nice thing is that the separation of the physics (the device package) from the numerical solver techniques (in the time integration package) will be more complete.

Explaining the F and Q vectors in detail is beyond the scope of this document. They all follow the form of KCL equations, like the residual vector. Concisely, one can think of them as:

- The Q-vector is for any KCL contribution that needs to be differentiated with respect to time. It is similar to our old state vector, but has the ordering of a solution vector. Q is dependent on the solution (x) and time (t).
- The F-vector is dependent upon x, and if the device contains sources, it may also be dependent on time. This vector contains KCL contributions to a DCOP calculation.

For simple examples of how this is set up, see the resistor, capacitor, and voltage source. For a complex example, see the BSIM3 (N_DEV_MOSFET_B3) device.

9.6 Code structure

9.6.1 Why do file names and class names start with N_?

This is historical. Our naming convention was set up to be hierarchical. The first letter indicates the "componenent" and the next 3-letter string indicates the "package". N_ indicates the "numerical" component. Originally, on the project we intended to have a graphical user interface (GUI), which would have been another component, possibly labeled as G_. However as the project evolved we never had the resources (or priority) to create a GUI.

Examples of packages include the "device model package", labeled as DEV. So a file or class in the device package would have the prefix N_DEV_ in the name.

9.6.2 What is the loader package for?

The idea of the loader package is to provide a layer of insulation between the device (or physics) package, and the solver packages.

However, the encapsulation mostly goes one way. When solvers need to call the device package, they call the loader, which subsequently calls the device package. However, when the device package calls a solver, it just calls it directly. (this will someday change) Almost all of the calls from the device package to the solvers come from the device manager class.

9.6.3 What does N_DEV_DeviceMgr::setupSolverInfo_ do?

This function calls the solvers (nonlinear and time integrator, mainly) to get things like the current time step size, the current time, the Newton step number etc. It is supposed to be one of the only places in the device package that calls the solvers. Anyway, all of the random "bookkeeping" information is obtained here. This function is called at the beginning of every loadRHS and loadJacobian function call.

9.6.4 What happens when N_DEV_DeviceMgr::loadRHSVector is called?

Note that this answer pertains to the old time integrator, not the new-DAE time integrator. (see question 9.5.8). For the old time integrator, a bunch of functions in the device package are called:

- The RHSVector is set to zero.
- N_DEV_DeviceMgr::setupSolverInfo_ is called.
- The device manager loops over all the device instances, and the updatePrimaryStateBlock function is called for each one. Each updatePrimaryStateBlock function calls the updateIntermediateVarsBlock function.
- After the updatePrimaryState functions, the device manager then tells the time integrator to calculate time derivatives, by calling:
  ```
  tiaMgrPtr_->updateDivDiffs();
  tiaMgrPtr_->updateDerivs();
  ```
- updateSecondaryState is called for all the device instances, so each device can obtain the time derivatives it needs.
- loadRHSBlock is called for every device instance, to complete the load. loadRHSBlock is the function in which each device instance actually sums quantities into the RHSVector.

The most important thing to understand here is the calls to the time integrator that happen in between the updatePrimaryState and updateSecondaryState calls. Once the time integrator functions have completed, all the time derivatives needed to complete the RHS load are available.

9.6.5 That sounds like a convoluted set of function calls. Why?

The code was written in a hurry.

Also, it happened because the time integrator was designed to do everything (calculating time derivatives, etc.) as all-at-once vector operations. This required that the loadRHS process include a "pause", during which the time integrator is called, after which the code returns to the device package, to complete the RHS load.

A lot of this confusing structure can (and will) go away once we complete the switch to the new-DAE formulation. (see question 9.5.8). The biggest thing that will change for new-DAE, is that once the device package exits, and control goes back to the time integrator, it will stay there. The code will not go back to the device package for the final RHS load. The final assembly of the RHS vector will happen in the time integrator, which is actually a more logical place for it to happen.

9.6.6 Why do a lot of function names end with "Block"?

A lot of device instance functions once had counterparts in the device class. (Each device has a device class, a device model class, and a device instance class) It used to be that when a load was called (say loadRHS) for a device, the function that was called was the device class function. Inside that function, there would be a nested loop structure that looped over device models and instances, to perform the loadRHS for each instance.

There wasn't much reason to do it that way, other than to mimic SPICE, so I eventually stripped out the device class functions and set up equivalents down in the device instance classes. To distinguish between the old device class "loadRHS", and the new device instance version, I added the word "Block" to the end of the function. I'm not sure why I bothered to add the word "Block". The class name by itself makes it clear that this is a different scope of function.

Anyway, for example, `N_DEV_Resistor::loadRHS()` was replaced by
`N_DEV_ResistorInstance::loadRHSBlock ()`.

Note - this may seem like a minor change, but when it was propagated to every device, for every load-related function, I was able to reduce the total number of lines in the device package by about 15-20,000 lines.

9.7 Device Parameters

9.7.1 Where are default parameters set?

They are set (currently) in two places:

1. In the initializations of model and instance constructors. (these are just to make sure the data is not uninitialized).
2. After the parTable, in the constructors of models and instances, when the overloaded device entity function “addPar” is called. These “addPar” functions are where the defaults are set for real.

There are a few notes of interest here:

- Double-precision parameters are set up in the static parTable object, which is a local data structure to any device constructor. This table is looped over and addPars are called for each row. As this is a static structure, it only needs to be processed for one instance or model of the device type.
- Non-double-precision parameters are not in the parTable, and are handled via individual addPar calls.
- The addPar function is an overloaded function of N_DEV_DeviceEntity, and there is a version available for every type of parameter data (double, int, string, etc).
- Some devices have documentation for each parameter embedded in their parTables and some devices do not. addPar is sufficiently overloaded to handle either kind.
- There is also a “processParams” function in every device instance and model class. This function is nearly deprecated at the moment. In earlier versions of **Xyce**, all the hard work in parameter processing happened here. This was changed when the parTable approach was first implemented. However, as the parTable approach breaks the C++ guidelines of having initializations delayed(see 6.7), and not tied to the constructor, this will probably get refactored in the future, to make processParams a meaningful function again.
- processParams still gets called from the constructor, even though in many cases it is a no-op.

- The instance version of processParams includes calls to updateTemperature for a lot of devices, so in that case it is not a no-op.
- The parTable approach actually violates a number of C++ style guidelines, and is somewhat unsafe in terms of memory usage. Its one advantage is that it ties parameter information directly to memory offset locations in the class. Most parameters have a corresponding class variable, which gets set to their value. By using the table, the parameter has a direct mapping into the memory location, and this reduces the number of times it has to be stored. This is one of the rare exceptions when having static data is OK.

At some point, the parameter handling in the device package will need to go through a major refactor, in order for the code to follow best-practice C++, and for the code to be easier to maintain and extend. As the device package is large, this will require a substantial effort.

9.7.2 Is it important to have the same parameter defaults as SPICE?

Yes!

Make sure your defaults match SPICE's defaults, assuming you are working on a SPICE-based model. The most commonly used value for any model parameter is its default value. Defaults don't necessarily correspond to a specific "real" device, but any device model put into **Xyce** should run to convergence with all the default values, and produce the same answer as SPICE.

Even when users specify a lot of model parameters, they usually don't set every single parameter, just a subset. For us to match SPICE results, default parameters must be the same. If the defaults are not the same, then effectively, it is not the same model.

9.7.3 Some devices have a lot of parameters and/or variables - do I really have to type them all into the various model and instance constructors?

Yes.

As noted in question 9.7.1, it is necessary to initialize all variables, or else some C++ compilers will barf.

It is recommended to use scripts, global replaces, and other editor tricks as much as possible. You probably won't be able to do 100% of the constructor using scripts, but you can take care of the worst of it. If you are clever about using scripts, then setting these constructors up shouldn't be that hard.

For variables coming from SPICE, copy over the relevant header file(s) from SPICE, and edit it down to a list of variable names. Do some global replaces to get rid of the extra SPICE-centric stuff. For example, a variable like "JFETbeta" will probably need to be changed to just "beta". (the name "JFET" will be implicit from the name of its owner class). Eventually, you'll be left of a file that has nothing but a list of (for example) instance variables used by the SPICE model, one variable per line. Once you have this, you can use this file a lot to set up the **Xyce** header file, and the constructors.

9.8 Debugging

9.8.1 What's with the Xyce_DEBUG_DEVICE macro?

Most debug output in **Xyce** is inside of ifdefs. If you want to use it, you have to configure and compile **Xyce** with it enabled. To use code inside the Xyce_DEBUG_DEVICE ifdefs, you need to configure and compile with (at least):

```
configure --enable-debug_device
```

9.8.2 I've been debugging a device. I've mostly been looking at the *.prn file. Is that enough?

No. For debugging the *.prn file should be considered "necessary but not sufficient".

The *.prn file contains output specified by the `.PRINT` line in the netlist. While this output needs to be correct, it really isn't "debug" level output - it's user output.

Data in the *.prn file is supposed to represent a converged answer. If the problem is non-linear, that means that even for a single converged data point, **Xyce** has gone through numerous nonlinear iterations. If (for example) it takes 5 Newton steps to obtain convergence, then the code has (at a minimum) performed 5 linear solves, 5 matrix loads, and 6 (not 5) residual loads.

Most errors in device models will be invoked at load time. For a 5-iteration solve, an error in the residual load will be exercised at least 6 times. Possibly many more than that, depending on the algorithm you choose. By the time you obtain a converged answer, the error will have been invoked enough times that it will be nearly impossible to diagnose it, if all you look at is *.prn file data.

Note - of course, ultimately, the *prn file data needs to be correct, so certainly look at it - but realize that any errors you have will be very hard to diagnose from the data in this file.

9.8.3 If the *prn file is the wrong place to look for data to debug, where should I look?

Look at matrices and vectors for individual Newton steps, and do it mainly for the first 2 Newton steps of the initial DCOP.

If you have an error in a device model, then there is a good chance that error will be in place on the first Newton step of the first Newton solve. In other words, look at the very first Jacobian load and the very first residual load, and nothing else.

Once you have satisfied yourself that the first set of loads are 100% correct, then, AND ONLY THEN, move on to examining the second Newton step in detail. The first step is, in some sense "special", in that there are things that happen on the first step that happen on no other step. The second step, however, is not special, in terms of the code it exercises, so if it is correct, most of the other steps should be correct.

In general, if the first step is 100% correct, and the second Newton step is 100% correct, then probably the entire DCOP calculation will be correct. Once you are confident that the DCOP calculation is right, then start debugging the transient. As with the DCOP, debug the first Newton step of the first time step, then the second Newton step of the first time step.

In summary:

- Debug the first Newton step of the DCOP.
- Debug the second Newton step of the DCOP.
- Check that the full DCOP is correct.
- Debug the first Newton step of the first time step.
- Debug the second Newton step of the first time step.

These are the "critical points" of the calculation. If you get these right, you'll get most of the simulation right.

9.8.4 How do I look at matrices and vectors in detail?

The easiest way is to use the old nonlinear solver, which is specified in the netlist by setting `.options nonlin nox=0`. (NOX is the newer solver). To dump matrices and vectors out to files, you need to be using a version of **Xyce** that has been configured and compiled with (at least):

```
configure --enable-debug_nonlinear --enable-verbose_nonlinear
```

By default, if you have compiled with these enabled, and you use the old nonlinear solver, you will get a lot of matrix and vector files dumped out every time you run **Xyce**. The amount of output is dependent upon the `debuglevel` parameter. For example:

```
.options nonlin nox=0 debuglevel=-1
```

will result in no matrix or vector output files. Setting:

```
.options nonlin nox=0 debuglevel=1
```

or

```
.options nonlin nox=0
```

will result in files at every Newton step, with unique names for each Newton step. Setting:

`.options nonlin nox=0 debuglevel=2` (or higher)

will result in files at every Newton step, with unique names set by both the Newton step number and the time step number. This is by far the most verbose option.

The files of interest are matrix.*.txt, rhs.*.txt and solution.*.txt. In the default mode, the "*" part will be a 3-digit number, set to the Newton step number. For most debugging purposes, this is enough - I'm usually only interested in debugging one particular nonlinear solve, not several.

9.8.5 What is the namesMap.txt file?

This file will tell you the mapping between solution variable index and solution variable name. Most solution variable names are determined by voltage node names, set in the netlist. A typical namesMap.txt file looks like this:

```
0                4
1      vmon_branch
2       vdd_branch
3           source
4            drain
5                5
6             gate
```

The first column is the solution vector index, and the second column is the variable name. You can get a nice, useful output if you do a Unix “paste” of the namesMap.txt file to one of the solution sized vector files. For example:

```
 paste namesMap.txt rhs.002.txt <return>
```

will give you (for example):

```
0                4      8.673617379884035472e-19
1      vmon_branch      0.000000000000000000e+00
2       vdd_branch      0.000000000000000000e+00
3           source      4.541535243477754641e-03
4            drain     -4.656753028599916293e-03
5                5      8.267041565201971309e-19
6             gate      1.152177851223489614e-04
```

In this example, you have the index in the first column, the variable name in the second column, and the residual vector value (from the second Newton iteration) in the last column.

Xyce will not output a namesMap.txt file, unless it has been configured to do so. To enable this capability, **Xyce** must be configured with:

```
configure --enable-test_soln_var_map
```

When **Xyce** has been configured to do this, it will always produce this file. This capability will probably not work correctly in parallel.

9.8.6 How do I use the numerical Jacobian?

You can use the numerical Jacobian by adding this to a netlist:

```
.options device voltlim=0 numjac=1
```

Voltage limiting must be turned off (by `voltlim=0`), for this to work. In general, voltage limiting can be thought of as an option that forces the right-hand-side (RHS) vector to contain the residual (f-vector) and only the f-vector.

As the Jacobian, $\mathbf{J} = d\mathbf{f}/d\mathbf{x}$, the numerical Jacobian is calculated by perturbing individual values of the $\mathbf{x}$ vector, recalculating $\mathbf{f}$, and doing a simple finite difference for each entry. This is, of course, much, much slower than using hand calculated analytical expressions for these derivatives in the code, which is what **Xyce** normally does. However, calculating Jacobians in this manner can help a great deal in tracking down Jacobian mistakes.

Do NOT assume that your Jacobian is perfect, just because you can get a circuit to converge in the nonlinear solve. Convergence should be considered necessary, but not sufficient evidence for any device you claim is completed in **Xyce**.

Always do at least a few numerical Jacobian tests on any device you add to **Xyce**. It is easy to do this. Simply run a relevant circuit with these options:

```
 .options device voltlim=0 numjac=1
.options nonlin nox=0 searchmethod=0
```

Then run it again with these options:

```
 .options device voltlim=0 numjac=0
.options nonlin nox=0 searchmethod=0
```

Note that with `voltlim=0`, a lot of circuits won't converge. That's OK. Whether it converges or not, **Xyce** should behave almost exactly the same with these 2 options. For circuits that fail to converge, they should still fail in (almost) exactly the same way. If you are running a circuit that doesn't converge, it is often best to set the maximum number of nonlinear solver steps to a low number, just to save time. (if the circuit isn't going to converge anyway, don't bother calculating a lot of steps, just look at the first few). You can set the number of nonlinear solver steps to (for example) 3 steps like this:

```
 .options device voltlim=0 numjac=0
.options nonlin nox=0 maxstep=3 searchmethod=0
```

Note the addition of `maxstep=3` to the nonlinear solver options line.

In general, the first few numerical Jacobian matrices should be nearly identical, to their corresponding analytical Jacobian matrices. (i.e. the first numerical Jacobian should match the first analytical Jacobian, the second numerical Jacobian should match the second analytical Jacobian, etc.). As the solve progresses, minor differences will start to emerge, due to roundoff differences, so the match will be less perfect later in the solve.

In general, for circuits that do converge, the code should take approximately the same number of nonlinear iterations. If the final number of iterations is off by more than one, you probably have a mistake. Also, (as with failed circuits) if you see discrepancies in Jacobian matrices, you probably also have a mistake. Occasionally, there can be differences due to inaccuracies in the numerical Jacobian calculation, but these inaccuracies will not cause dramatic differences in the solve.

In general, if your device does not pass the above test, then either the device has a bug (likely) or the numerical Jacobian calculation has a bug (not very likely).

9.8.7 What should I do to compare **Xyce** to SPICE?

For any legacy device, this will be the focus of model development. There's a lot to be said here. In short:

- This is more important (for legacy devices) than the numerical Jacobian test. (see question 9.8.6) The numerical Jacobian tests don't voltage limiting, which needs to be tested.
- Compare Jacobians, solution vectors, and residual vectors, especially on the first two or three Newton steps of the initial DCOP.
- You will find over half of your mistakes in the first few Newton iterations, and it is easier to see them if you examine the artifacts of those iterations in detail. This means....
- Comparing the final output, in a *prn files (or their equivalents), is not adequate. You will not find any subtle errors this way.
- When comparing transient, make sure to force a constant stepsize in both codes.

In general, when comparing **Xyce** and SPICE, if you are comparing a small DCOP calculation, the two codes should get nearly identical answers, for every variable in the solution vector. Consider:

- SPICE always uses a direct solver. For small circuits, **Xyce** also uses a direct solver. Direct solvers give exact answers, with the exception of roundoff error. For a small problem, roundoff error should be negligible.

- The linear systems solved by **Xyce** and SPICE are algebraically equivalent, even though they look different.

- For the DCOP phase, there are no time integration differences to consider.

Solver tolerances are slightly different between **Xyce** and SPICE, so **Xyce** may take a slightly different number of nonlinear iterations than **Xyce**. If this happens, the final "converged" answer will be different, and actually should be different. Make sure to only do detailed comparisons for the same nonlinear iteration. For example, if **Xyce** takes 12 iterations, and SPICE takes 10 iterations, compare the 10th iteration of **Xyce** to the 10th iteration of SPICE.

Unfortunately, **Xyce** and SPICE will order their solution variables differently, so comparing matrices, solution vectors, etc., of the two codes can be tricky. Fortunately, there is an easy way to do this, which is described in the answer to the next question.

9.8.8 Is there an easy way to compare vectors and matrices?

Yes. Use the programs matcmp, veccmp, and mapMerge, which can be found in the Xyce/utils directory. They can be compiled by running the script "Build" , from the directory. There is a README in that directory, which explains how to use them.

Note, these programs can be used to compare a **Xyce** matrix to a SPICE matrix, and also a **Xyce** vector to a SPICE vector. However, you'll need a hacked up version of SPICE (or ChileSPICE) to do it. This capability is really, really useful, as these programs can automatically track what the mapping is between a SPICE variable ordering and **Xyce**'s variable ordering, for a given problem. This can save a lot of time, especially for larger problems.

The ordering issues are resolved with the **Xyce** namesMap.txt file (see question 9.8.5) , and an equivalent ”names" file from SPICE. The mapMerge utility takes both of them as input, and comes up with an index map to translate between the two codes.

9.8.9 I'm trying to compare Xyce with analytic Jacobian to Xyce with numerical Jacobian, not Xyce and Spice. How do I use matcmp and veccmp?

You need to edit basecmp.h and un-ifdef the line "#define BOTH_FILES_ARE_XYCE 1", then recompile matcmp and veccmp. Once done, you no longer need the "mergedMap.txt" file — the utilities will use the one namesMap.txt file and work on both matrix and vector files as if they came from Xyce.

There is no automatic detection by matcmp and veccmp that both files are from the same simulator — you need to recompile them.

9.8.10 What is the most common source of errors in **Xyce** devices?

Errors in translation.

Most **Xyce** devices are based on legacy SPICE devices. Most SPICE devices have been around a really, really long time, and have been debugged by (literally) thousands of users and developers. By the time we look at them, they are usually rock solid.

In general, when we implement SPICE models in **Xyce**, one way to think about it is that we are "translating" them from one code to another. We aren't changing the functionality of the model - we're changing how it looks. If mistakes exist in a **Xyce** translation, it is most commonly an error in the translation, not in the original SPICE model.

Note, when I say the old SPICE models are well-debugged, I'm not talking about how valid or accurate the models are. In that respect, a lot of legacy devices are terrible, and everyone knows it. The types of errors I'm referring to involve:

- sign errors.
- errors relating to "type" (N-type vs. P-type) - these are often sign errors.
- errors relating to the mode of the device.
- matrix and vector indices.
- mistakes in voltage limiting terms.

- mistakes in bypass terms (which we don't support! see question 9.2.16).
- incorrectly evaluating time derivatives.
- incorrectly handling gmin.
- incorrectly calculating derivatives for the Jacobian.

In general, old SPICE models wouldn't work correctly, and be so widely used, if they had continued to have errors like this over the years. It is really, really rare to find them in SPICE models. As **Xyce** is a much newer, much less mature code, these types of errors usually originate with **Xyce** developers, not SPICE.

So, from a development point of view, when implementing a **Xyce** model, it behooves you to translate the SPICE code, while changing it as minimally as you can.

For a related question, see question 9.2.15.

9.9 Test circuits

9.9.1 Is there a good program for comparing waveforms?

Try using xyce_verify.pl. It can be found in `Xyce_Regression/TestScripts` . It is good for comparing both transient and DCOP data. Its usage can be determined by running it with no arguments:

```
> xyce_verify.pl
xyce_verify.pl
Usage: ./xyce_verify.pl [options] netlist goodfile testfile [plotfile]
```

This script is the main comparison script used by our nightly and weekly regression testing. It works pretty well. In addition to comparing data, it also checks to insure that all the requested variables have been output. Also, for DC sweeps it will check that the correct number of distinct sweep steps have been output.

9.9.2 My new device appears to get the correct answer in a small test circuit. Am I done?

Probably not. If you've only been working with one test circuit, and are only looking at the final answer (the contents of a *prn file, for example), that is in the category of "necessary but not sufficient".

There have been many observed cases on this project, in which a device appeared to give the right answer for a small test problem, but in fact was very, very broken. You may be wondering, why? Here's why:

- Full coverage of the device. A lot of small tests don't fully exercise a device. For example, I've seen a number of simple test circuits apply zero-value junction voltages to a subset of the input terminals. Any current that depends on one of these junction voltages isn't getting tested - it will just be zero, whether you set that current term correctly or not. Also, some devices have optional internal variables, that are turned on or off by certain parameters. For devices like this, it isn't possible to fully test the device with one test circuit. Every mode, and combination of modes, available to the device needs to be tested.

- DCOP vs. Transient. If your test circuit is a DCOP or DC sweep circuit, then you haven't tested everything. Transient has all the same terms as DC, with capacitors added. You need to test both analysis modes before you can declare a device finished.
- Numerical stability. If you have the correct residual vector, but an inconsistent Jacobian, then your device is "correct", in the sense that if it converges it will give the right answer, but it is incorrect in that it won't converge very often. Small, simple test circuits are usually very easy to solve, from a numerical point of view. Bugs due to mathematically inconsistencies will not be caught by small tests.
- Numerical stability, part 2. If your test consists of your new device, and a minimal driver circuit, then (probably) all the nonlinearity in the test circuit is in your new device. In the real world, your device will be used inside of a meaningful circuit, which will have lots of nonlinearities. If your device has any mistakes in it (which negatively affect the nonlinear solve), those mistakes will cause a lot more problems in a complex, nonlinear circuit than they will in a tiny, almost linear circuit.
- N-type vs. P-type. If your device is a transistor, and your test just has only one device in it, then you are either only testing a P-type device, or an N-type device. You need to test both.
- N-type vs. P-type, part 2. If your new device is a transistor, you need to have some test cases that contain both an N-type and a P-type device, in the same circuit. Almost any digital component will consist of an equal number of multiple instances, of both types.

Unfortunately, mistakes are sometimes discovered, not during development, but after code has been distributed to users. A user attempts to use **Xyce**, and reports back, "A circuit that ran just fine in PSpice totally bombs in **Xyce**". When this happens it is embarrassing for the project, and degrades trust among the **Xyce** user community.

Xyce is a relatively mature code at this point. Most of the issues pertaining to making **Xyce** match SPICE are well-known issues. If we declare that a device is available and supported, and said device is an old, legacy device, we should never get the "it ran in PSpice but not **Xyce**" bug report

Before we declare that a device is available for users, it needs to be bulletproof. The penalty for releasing bad code is much worse than the penalty for delaying a release.

Note: of course, we will always get bug-reports from users, but we should strive to only get bug reports having to do with "bleeding edge" capabilities, or capabilities that we don't yet support.

9.9.3 Where can I get better test circuits?

A really good place to get ideas is `Xyce_Regression`. Most transistor devices can be mixed and matched (somewhat), so you can often take a test circuit that was originally designed for another, similar device, and modify it slightly to use your new device.

For example, if you are implementing a new MOSFET device, there are many MOSFET tests in the `Xyce_Regression` sub-directories. As a first step, grab one of them, change the level number, and run with default parameters. Even if it doesn't run right out of the box, it can still be a valuable test. Run it in SPICE and compare. If it fails in **Xyce** it should also fail in SPICE, in nearly the same way. If it doesn't you still have work to do.

Make sure that you do some tests involving multiple instances of your new device. In particular, MOSFET devices need to be tested in circuits that contain both NMOS and PMOS device types, as CMOS technology is based on "complimentary logic". For CMOS technologies, there will be no meaningful circuits that don't have both types.

We've used inverter chains a lot in our testing of **Xyce** devices, because it is an easy problem to scale to arbitrary size. There are perl scripts that can be used to create arbitrary-length chains. Some perl examples can be found in
`Xyce_Regression/Xyce_SandiaRegression/Netlists/stress_tests/BSIM3/INVERTERS`.

If your device is a transistor device, then any circuit of traditional digital building blocks can work pretty well as a test circuit. This includes inverters, NAND gates, ring oscillators, etc.

9.9.4 I don't have any valid model parameters. What do I do?

If you are implementing a industry-standard SPICE device, you don't need "valid" parameters, at least not at first. You mainly need to do a lot of comparisons to SPICE. Remember, if you have been tasked with implementing a legacy device model, your primary job is code development, not model validation.

Model validation is, of course, very important, and it may be necessary later. **However, you can't do good model validation until you've properly implemented the model! Any attempts to "validate" a buggy model will be a waste of time!**

For any new device, start out comparing **Xyce** with SPICE, in detail, for the device's default parameters. This is a good first step. Do this for a variety of circuits.

As you work more with the model, you'll probably develop an understanding of what some (or most) of the parameters are supposed to do. As you develop this understanding, start experimenting with modifying your test circuits away from the defaults. **Xyce** and SPICE should still match.

Also, remember that "valid" parameters just means that someone has gone to the trouble of fitting the device model to a particular "real-life" electronic part. For debugging (verification) purposes, a hypothetical device is just as good.

Finally, if you have been asked to implement a legacy model, that probably means that a user has requested it, and has a particular circuit problem in mind, that they want to solve. Once you are ready (i.e. once the device is mostly implemented), ask that user (or users) for the model parameters they plan to use, and try them out. Again, run them in **Xyce** and SPICE, and compare.

9.9.5 **Xyce** and SPICE don't match for my test legacy device circuit, unless I tweak model parameters. Is this OK?

By "legacy device" I mean, "device developed outside Sandia, that can be found in most circuit simulators". See question 9.2.7.

If this is a DCOP simulation, then no, this is not OK under any circumstances. You have a mistake that you need to fix.

If this is a transient simulation, then it might be a solver tolerance issue. Try making the two codes match by setting stricter solver tolerances in both codes. Also, try running both codes in constant time step mode, with the same stepsize. If neither of these works, then you have a mistake in your **Xyce** device that you need to fix.

9.9.6 In my **Xyce** model, I left out a small term that was in the legacy SPICE model. It doesn't seem to change the answer for my test circuit. Is that OK?

Absolutely not!

If it is the SPICE model, it needs to be in the **Xyce** model. In general:

- Just because a term is small in your test circuit, that doesn't mean it will be small in other circuits.
- Even if the term is always small, that doesn't mean that it isn't important. For example, the excess phase term in the level=1 BJT is usually small, but a lot of circuits won't run without it. (see question 9.5.7.
- Terms were put into a SPICE model for a reason. If you don't have a clear, well-argued reason to exclude it, then you haven't justified removing it. The vague reason, "It seems like a small effect." is not an adequate reason.
- Claiming, "this term is a really bad approximation for this effect" is also not a valid reason. If it is such a bad approximation, the correct way to handle it is to come up with a better one, and make the new approximation a optional (non-default) choice. Do not delete the old approximation - a lot of users have extracted parameter sets that were fitted to that approximation, so it needs to be supported.

Valid reasons for excluding a term include:

- The term is zero by default, and is only nonzero if a certain device option is invoked, and you know for a fact that **Xyce** users will not be invoking that option. An example of this is the NQS term in the BSIM3. It has never been supported in **Xyce**, but it is an option that none of users has ever used. If a user were to need it, we would need to add it.
- If the term is wrong to a point of breaking the device and rendering it unusable. This has happened a couple of times on the **Xyce** project, but it is incredibly rare. (See question 9.2.15.) It will almost never happen, if you are developing a legacy device. Usually, if your device is broken, it is your fault, not the model's.

In general, if your model is going to deviate from SPICE, you need to have a really strong justification for it. If your best justification is, essentially, “it seemed to work”, you’ve gone to the dark side.

This question mainly refers to legacy SPICE models. For new models that are under development at Sandia, it is a different can of worms. For new devices under development, you aren’t trying to match an industry standard.

9.10 Testing

9.10.1 How do I run the regression test suite?

To run the test suite, you'll need to run a script called `run_xyce_regression`. It can be found in `Xyce_Regression/TestScripts`. It can be run directly from that directory. If you do not specify a path to a **Xyce** executable, it will use the first one it find in your path. You can double-check which **Xyce** is in your path by typing `which Xyce`.

9.10.2 Can you give some explanation of what is going on in the test suite?

The top test suite directory is `Xyce_Regression`. The main subdirectories directly under it are: `Netlists`, `OutputData`, and `TestScripts`. Once you start running it, you'll start to see other directories and files, related to test results.

The test scripts have evolved considerably since the beginning of the **Xyce** project. In the old days, we had two separate repositories, one for regression, and one for certification tests.

9.10.3 What sorts of tests should go into the nightly regression test suite, and when should I add them?

The point of the nightly regression tests is to make sure that the code doesn't get accidentally broken. It is not intended to be complete proof that **Xyce** gets the right answer. Of course, it is best if developers make an effort to insure that the "gold standard" files contain, to the best of our knowledge, "correct" answers, but this isn't the primary goal.

Regression tests should usually be tests that can run very quickly, in a few seconds. Generally, the smallest test you can come up with, that exercises the capability, is what you should shoot for. Of course, we need to have some tests that are large, in part to test the parallel capability of **Xyce**. Tests that are large, and take minutes or more to run, should be given the "weekly" tag.

9.10.4 What sorts of tests should go into the `stress_tests` directory, and when should I add them?

Generally, the original idea behind `stress_tests` was to have a place to store test circuits that were not ready for regular regression testing. Often they have been circuits set up as part of a test-driven development process, meaning that initially **Xyce** is incapable of running them. At this point, however, the test framework is sophisticated enough to place any test into any `Netlists` subdirectory, without necessarily adding it to regression testing. This can be accomplished, either by using an "exclude" file in the local netlist directory, or by including the "exclude" tag in the "tags" file.

At this point the `stress_tests` directory is deprecated and is only kept around for historical reasons. Nowadays tests which are still under development, and not ready for nightly testing, should be developed in `Xyce_Regression/Netlists/` with appropriate exclude tags. Once they are ready, then the developer can simply change the tags for that test.

10. Xyce Release Process

Chapter Overview

This chapter presents a high-level description of the **Xyce** Parallel Electronic Simulator Release and Distribution Management Process. The purpose of this process is to standardize the manner in which all **Xyce** software products progress toward release and how releases are made available to customers. Rigorous Release Management will assure that **Xyce** releases are created in such a way that the elements comprising the release are traceable and the release itself is reproducible. Distribution Management describes what is to be done with a **Xyce** release that is eligible for distribution.

10.1 Preface

This chapter presents a high-level description of the Xyce Parallel Electronic Simulator Release and Distribution Management Process. The purpose of this process is to standardize the manner in which all Xyce software products progress toward release and how releases are made available to customers. Rigorous Release Management will assure that Xyce releases are created in such a way that the elements comprising the release are traceable and the release itself is reproducible. Distribution Management describes what is to be done with a Xyce release that is eligible for distribution.

10.2 Introduction

10.2.1 Document Purpose

The purpose of this chapter is to provide a high-level description of the Xyce Parallel Electornic Simulator Release and Distribution Management (RDM) process. This chapter describes the process elements of RDM, but it does not describe their specific implementation. No specific tool is recommended or discussed in detail. There are several implementations that would successfully carry out the process described in this chapter. Therefore, even if the implementation or the toolset changes in the future, this process remains stable.

10.2.2 Scope

The Xyce Release and Distribution Management (RDM) process described in this chapter is required for each Xyce product. The main sections of this chapter focus on the tasks and activities that are required for RDM. Templates and project specific extensions are included in the sections and the end of this chapter. This RDM process applies to all of the work products of application engineering, software development, and verification. A subset of these work products and artifacts are included in the product release that is distributed to customers. The specific subset of work products and artifacts included in the product release is dependent upon the product being released, the intended use, and the customer. These dependencies should be captured in the Requirements Management Process. RDM consists of two processes: the Release Management Process (Section 10.4) and the Distribution Management Process (Section 10.2.2). The Release Management Process identifies the activities that must be addressed for a product release to become eligible for distribution. The Distribution Process describes the activities that must be addressed when distributing a product release to customers.

Release Management Process

The Release Management Process identifies the objectives, goals and activities that need to be addressed for a product release to be eligible for distribution. The output of the Release Management Process is a product release that is eligible for distribution. The Release Management Process relates to activities that occur internally to the Xyce project. It includes activities that must begin when a new software release is initially foreseen, as well as activities up to and including the point in time when the new product release is eligible for distribution to customers.

The outputs of the Release Process include:

- Xyce product release(s), comprising features to meet customer requirements, enhancement requests and potentially fixes for specific defects.
- Installation notes, describing installation activities for the release.
- Release notes, describing new features, changes to existing features, defect fixes, deficiencies, known defects and limitations.

The Release Management Process is described in Section 10.4. The complete set of artifacts that comprise a product release are not defined in this process. Refer to Section 10.7 for a list of the interfaces between this document and other Xyce processes.

Distribution Management Process

The Distribution Management Process describes the objectives, goals and activities that need to be addressed once a product release is eligible for distribution to customers. The output of the Release Management Process, a product release, is an input to the Distribution Management Process. The outputs of the Distribution Management Process are data items that are used to track and manage product distributions (i.e. track and manage which customers have which release artifacts).

The Distribution Management Process describes the needs for a product becomes eligible for distribution, when the product is distributed, and when a product is withdrawn. The Distribution Process is covered in Section 10.2.2.

Roles

Roles are functional divisions of the process responsibilities and they are defined based on the activities that comprise RDM. Roles are covered in Section 10.6.

10.2.3 Goals

Since 2001, Sandia National Laboratories (Sandia) has been developing a DOE-ASCI funded parallel electric circuit simulation code named Xyce. The Xyce Parallel Electronic Simulator has been written to support, in a rigorous manner, the simulation needs of the Sandia National Laboratories electrical designers. The code has been developed using an object-oriented design and modern coding-practices that ensure that the Xyce will be maintainable and extensible far into the future.

Developing and modifying the Xyce products in time frames responsive to the customer can be a significant effort, so the procedures to manage the items that make up the Xyce products need to be as clear and non-intrusive as possible so that efficient response to the customer can be maintained.

The primary goal is to document the Release and Distribution Process for the managers and developers of Xyce.

Thus, the main goals of the RDM process are to:

- Ensure that all Xyce releases contain the proper artifacts;
- Ensure that all Xyce releases are reproducible;
- Ensure that all phases of the Xyce release life cycle are adequately tracked and documented;
- Define a process that is adequate for current Xyce needs and which can be enhanced with additional procedures at a later date if desired;
- Ensure that distributions of all Xyce product releases can be tracked; and
- Allow prior releases of Xyce products to be withdrawn if needed.

10.3 Abbreviations and Definitions

artifact A deliverable or work product that is the output of some phase of the software development life cycle. A configuration-controlled artifact is an artifact that is stored in a corporate repository (library), and changes to it are controlled.

authorized maintiner The person responsible for installing and maintaining the Xyce application on the platform or specific environment. This person is also a customer.

baseline (verb) To capture a snapshot of a controlled item (or group of controlled items) at a reference point within the item's development life cycle. (noun) A reference point in the development of a controlled item or the snapshot of the controlled item captured at a reference point within the item's development life cycle.

change request Documentation (formal or informal) of enhancements, modifications, or bug fixes being succested for the system. The change request must be approved prior to the work specified in the change request beginning. Examples include formal change control documents, issues (bugs), and enhancement requests.

check in (commit) To put the initial or a new revision of an element into a version control system

check out To extract a revision of an element from a version control system.

components Tightly coupled/interdependent sets of modules that provide functionality. Capabilities trace to components.

customer Any person who wants to use the software application, report a problem, request an enhancement, or request a specific version of the application.

distribute To make a product release available to a customer. Depending on the customer's needs, the customer may pull the product distribution or Xyce may push the distribution to the customer.

distribution management Establishing, maintaining, and tracking procedures, roles, and responsibilities for distribution of products to customers.

element Lowest level (atomic) item that is subject to version control.

element version The revision of an element. Used when referring to a specific revision of an element; such as, "What version of the file is in the release?"

emergency patch A patch that has to be implemented and released as quickly as possible because critical operations, decisions, or results are impacted. Contrast to immediate patch.

full release A release that is self contained (i.e. the release may be installed and used without access to any previous release).

freeze To prevent additional changes to a specific version of a version controlled item.

immediate patch A patch that needs to be implemented before the next scheduled primary release, but the updates are not critical. Contrast to emergency patch.

label An identifying marker that can be associated with specific versions of version controlled elements. Multiple elements can be assigned the same label as a means of grouping the elements.

locked An indicator that a specific reversion of a version controlled element may not be modified.

major release A scheduled or planned release of a product that comes out when there are extensive new product features, when there is a significant redesign of the product or when customers of the product are required to make significant changes in how they use the product or in support elements of the product such as the version of a supporting commercial application or third party software. See definitions for minor release and patch release, Figure 10.1.

minor release A scheduled or planned release of a product that comes out when a product feature has been added or significantly modified from its original documented behavior. A minor release usually does not imply significant redesign of the product although there may be redesign of some of its components. A minor release also should not require the customer to make significant changes in how they use or support the product. See major release and patch release. See Figure 10.1.

module Smallest coherent unit of a product. Requirements trace to modules. Modules are composed of one to many version controlled elements.

partial release A release that contains some subset of a products elements and/or components. Partial releases must be installed over, or in conjunction with, a previous full release. See full release.

patch An as-needed update to one or more of the elements that comprise a product for the express purpose of fixing critical or function-impacting defects. See Figure 10.1.

patch release A product release that is generated due to a patch. See release and primary release.

primary release A scheduled and planned release of the product which is either a major release or a minor release. A release is categorized as either a primary release or a patch release. See release and patch release.

product Items offered for use by licensed customers. Xyce products will be the Xyce Framework, ASCI Applications that utilize the Xyce Framework, third-party libraries, and Xyce Tools.

product release (formal) A captured occurrence of a product. Release is used in referring to the identification of a product; such as, "What is the release of the product?"

promotion Moving from one release development life cycle activity to a higher release development life cycle activity.

resource distribution A distribution of the application provided as a resource on specific simulation platforms, for specific environments, for use by authorized persons with accounts on the platform. These distribtutions are maintained by an authorized maintainer.

release An integrated set of one to many products that will, when ready, be made available for distribution to customers. See full release, partial release, primary release, and patch release.

release management Establishing, maintaining, and tracking procedures, release development lifecycle activity, roles, and responsibilities for releasing product to customers.

release number The alpha-numeric identifier given to a specific product release.

role A functional division of responsibilities.

subsystem An architecturally motivated organization of components.

bf tasks The major activities that are performed in the release process.

version One of a sequence of copies of an element, each incorporating modifications. See product release and element version.

version control Identifying, maintaining, and tracking versions of the components of a product and versions of the product itself.

version-controlled file A file that has been placed under the version control system.

version number A numeric identifier assigned to a specific occurrence of an element or product. See product release and element version.

withdraw To make a specific product release ineligible for distribution. A product release that has been withdrawn is no longer supported.

10.4 Release Process

10.4.1 Objective

The objective of the Release Process is to define the activities in the Release Process with adequate detail to facilitate the process implementation.

10.4.2 Goals

The goals of the Release Process are to:

- Define Release Process activities;
- Identify roles that are responsible for the activities.

10.4.3 Process

A simple definition of a release is a version of a product that will, when ready, be made eligible for distribution.

This definition of a release is very general and has broad application. For example, the release might be a single element, a group of elements, an application, or a suite of applications.

Xyce products that are subject to this process include the Xyce Parallel Electronic Simulator as well as any associated third-party software (TPS). Xyce release customers are internal Sandia analysts and designers as well as non-Sandians associated with the ASCI program. This may be expanded in the future to include external customers.

Prior to defining the activities in the Release Process, it is necessary to explain release types, release numbering and baselining.

Release Types

Software products are frequently refined, enhanced, and fixed. Consequently, new versions of products are apt to become available for release. Releases may be preplanned, where the features are outlined in an overall strategy for the product, or the releases may be extemporaneous to fix issues in a current release. Full releases will contain all product components, and partial releases will contain some subset of product elements and components.

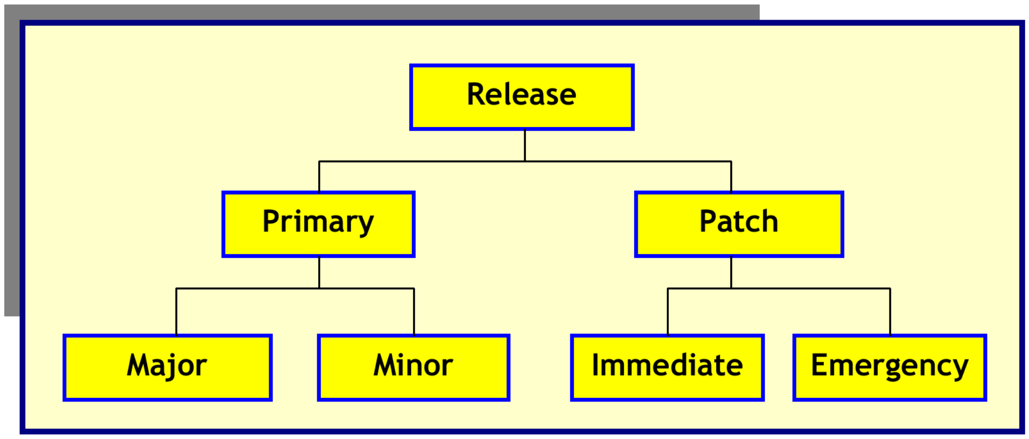

Figure 10.1. Release Hierarchy.

Primary Release

A release that is pre-planned is called a primary release. A primary release can be either a major release where the product has had significant changes made to it or a minor release where the changes are more incremental in nature.

All primary releases need to be made available as full releases (they contain all components required to install and use the product). A primary release must include the features and fixes of all previous primary and patch releases, unless the feature is being discontinued.

Xyce primary releases will occur periodically. The specific release schedule for primary releases is outside the scope of this document.

Patch Release

An extemporaneous release that is implemented to fix issues in an existing release is known as a patch release. The two types of patch releases are immediate patch and emergency patch. Patch releases may be full or partial releases (depending on project need) but will generally be partial releases.

An immediate patch is indicated when the patch release must be eligible for distribution prior to the forecast distribution date of the next planned release, and the need for the patch release is not critical enough to justify an emergency patch. All normal Release Process steps are followed for an immediate patch.

An emergency patch is indicated when the patch release must be eligible for distribution as quickly as possible. When generating an emergency patch, some of the activities normally required for a product release may be deferred. Deferral of process activities involves risk management decisions and must be preceded by appropriate approvals. All deferred activities must be completed retroactively after the release. Emergency patches should be rare as they are only generated in critical situations.

Release Numbering

Release numbering is the way a release is named and labeled so that it can be uniquely identified and referenced. The release number format for Xyce is the following:

(baseline) MajorRelease.MinorRelease.Patch.ModificationType

Table 10.1 describes the terms in the release number format. Refer to Table 10.5 for information on the release development life cycle activities.

Term	Description
Baseline	Indicates the intermediate release development life cycle activity (S for STABLE, Q for QA). This term is not used for PROD releases.
Major Release	A numeric term that is required for all releases.
Minor Release	A numeric term that is required for all releases.
Patch Release	A numeric term that is optional if the patch number is 0.
Modification	A numeric term that is used for baseline identification and is always suppressed in PROD release labels.
Type	Alpha-numeric term that is used to identify sub-varieties of the release. This term is intentionally flexible as release types may surface at any point during the release life cycle and packaging.

Table 10.1: Release Numbering Terms

Table 10.2 provides a few examples of baseline and release numbers for a fictitious Xyce product.

Description	Sample Release Number
Stable release of Xyce 1.0	(S) Xyce 1.0
QA release of Xyce 1.0	(Q) Xyce 1.0
Production release of version 1.0	Xyce 1.0
First production patch release	Xyce 1.0.1
First production minor release	Xyce 1.1
Version 1.1 packaged on CD for run-time-only licensed customers	Xyce 1.1 RTO-CD

Table 10.2: Release Numbering Examples

Baselining

Baselining is defined as capturing a snapshot of a controlled element (or controlled elements) at some reference point. Baselining provides a mechanism for logical groupings of elements, such as the elements that compose a product release. Baseline creation can be triggered by events in the release development life cycle as well as date or milestone events.

Baselines are comprised of a single revision of at least one version controlled element. A single revision of an element may be a member of multiple baselines. However, only one revision of each element may be in any specific baseline. In other words, if a revision of an element is denoted as being in a specific baseline, no other revision of that element can be in the same baseline.

What to Baseline

aAll elements that comprise Xyce product releases will be baselined during the appropriate release activity. There is a hierarchy of relationships that goes from the element level to the product:

- Element - the lowest level (atomic level) object that is subject to version control. Examples of elements include: source code files, build dependency files, build scripts and important outputs of release activities (artifacts).

- Module - composed of one to many elements. Modules are the smallest coherent unit of a product.
- Component - Tightly coupled/interdependent sets of one to many modules that provide functionality. Capabilities trace to components.
- Subsystem - An architecturally motivated organization of one to many components.
- Product - Items offered for use by licensed customers. Xyce products will be the Xyce Parallel Electronic Simulator and associated third-party libraries. A product is comprised of from one to many subsystems.

Release Baselines

A release baseline is created when a Xyce product, or a component of a Xyce product, completes the STABLE, QA, or PROD release development life cycle activities.

Reference Baseline

A reference baseline is created any time a traceable and reproducible reference point is desired. Possible examples include representing the product state as of a specific date or milestone.

Modifying a Baseline

With only one exception, baselines are never modified. The single exception relates to release baselines. Release baselines may be modified in that they may have elements that represent artifacts of the current activity added to the baseline (e.g. test results, installation instructions). Once a product release has been moved to the next release development life cycle activity, all previous release development life cycle activity release baselines are permanently frozen.

Baseline Abandonment

There are times when events require that a release baseline be abandoned. Abandoning a baseline implies that the product release is being returned to the previous release life cycle activity. Reference baselines are never abandoned.

For example, if a release baseline fails to pass all acceptance criteria, then that release baseline is not accepted. Notification that the release baseline has failed is passed back to the previous development life cycle activity along with results for the criteria that were failed. The specific release baseline is said to be abandoned. The release baseline is retained for historical record.

When a release baseline fails acceptance criteria and is abandoned, element fixes occur in the previous release development lifecycle activity. A new release baseline is created that includes the fixes. The baseline modification term is incremented so that every baseline is uniquely identified.

Baseline Identification

The standard that will be used for identifying baselines should be flexible enough to uniquely identify all varieties of product and reference baselines. The Xyce baseline identification format is:

ID Type release-number

Term	Description
ID	An alpha-numeric string that identifies the product for a release baseline, and is a freeform identifier for reference baselines. Legitimate values for release baselines will be Xyce and ¡TPS NAME¿.
Type	A literal string that is used to indicate the type of baseline. - Rel for release baselines - Ref for reference baselines
Release-number or text	The number described in Table 1. This term is used for release baselines only (i.e. it is not used for reference baselines).

Table 10.3: Baseline Identification Terms

Since the baseline identification must be flexible enough to accommodate both product and reference baseline labeling, a few examples may be useful. Table 10.4 contains baseline identification examples

Description	Baseline Identifier
Second DEVEL iteration for release baseline of Xyce version 1.0.	(D) Xyce Rel 1.0.0.1
First Stable release baseline of Xyce version 1.0.	(S) Xyce Rel 1.0.0.0
Fourth QA release baseline of Xyce version 1.0.	(Q) Xyce Rel 1.0.0.4

Description	Baseline Identifier
Production release baseline of version 1.0.	Xyce Rel 1.0.0
Date specified reference baseline (DOE literal is intended to represent some request from DOE to checkpoint progress at a point in time).	DOE Ref Jan 4, 2001
Reference baseline for Xyce product that is not based on the release life cycle. The text “for media test” is the term “release-number or text.”	Xyce Ref for media test

Table 10.4: Baseline Identification Examples

Release Activities

The Release Process for Xyce consists of activities and the roles that are primarily responsible for those activities. Each activity has entry and exit criteria. Activities are performed in a specific order, with a limited number of paths to completion of the release request.

Products can be in multiple release cycles at the same time. For example, if a product has a release that is in the Release Development Lifecycle PROD activity and new development has been started since the release, the product will also be in the Release Development Lifecycle DEVEL, STABLE, or QA activities. If the product is also being patched then there will be another patch release in one of the same activities. This represents three different releases (current release, next primary release, patch release) for the product.

Reference [23] provides a good visual overview of the Release Process flow, but it is not detailed enough to guide implementation. Table 10.5 gives more detail for the activities in the Release Process and identifies the role that is responsible for each activity. Additional information is provided for the four execute activities (DEVEL, STABLE, QA, PROD) that follow. Roles are defined in Section 10.6

Activity	Description	Role Responsible
Submit	Submission of the Product Release Request begins the Release Process. The release request must contain specific enough information to allow the PTL to review and scope the request.	PTL or CCB

Activity	Description	Role Responsible
Review	Review includes scoping and requires careful study of the request, requirements, and other release requests. The request may be: ■ Returned to the originator for more information; ■ Combined with other requests; or ■ Forwarded for approval.	PTL
Approve	Approval is required before the request moves into the activities that consume more resources. The request may be: ■ Approved; ■ Returned for more information; ■ Deferred; or ■ Rejected.	CCB
Plan	The planning activity includes: ■ Resource planning and allocation; ■ Generation of schedules; and ■ Milestone identification.	PTL

Activity	Description	Role Responsible
Execute	The execute activity encompasses all activities of the Release Development Lifecycle. The activities in the Release Development Lifecycle are: ■ DEVEL; ■ STABLE; ■ QA; and ■ PROD	DEV & QA
Certify	Certification indicates that this is an appropriate time for the release to be made eligible for distribution.	RL
Notify	Interested parties are notified that the release is "eligible for distribution". One key interested party that is always notified is the Distribution Lead (DL)	RL
Close Request	Process improvement tasks are performed. The release request is administratively closed.	RL

Table 10.5: Release Process Activities

Execute: DEVEL

The first activity executed in the Release Development Lifecycle is the DEVEL activity. During this activity releases are in said to be in development. Releases in DEVEL are not subject to mandatory version control (although version control is recommended), or mandatory baselines. Work is done to add, enhance, fix, or remove functionality.

Releases may be returned to this activity from the Release Development Lifecycle STABLE activity.

All work done during this activity must be directly linked to some form of a change requests (new reqreuiement, bug reports). Any work that exceeds the scope of existing change requests mandates that new or updated change request(s) be generated.

Execute: STABLE

The second activity in the Release Development Lifecycle is the STABLE activity. Releases that have reached this activity are ready for element and component level testing. This is the first Release Development Cycle activity with a mandatory baseline, and from this point on all Release Development Lifecycle activities require a baseline.

Releases that fail to meet acceptance criteria are returned to the DEVEL activity with error descriptions and/or change requests. Similarly, releases may be returned to the

STABLE activity from the Release Development Lifecycle QA activity. All work done during this activity must be directly linked to change requests. Any work that exceeds the scope of existing change requests mandates that new or updated change request(s) be generated.

Execute: QA

The third activity in the Release Development Lifecycle is the QA activity. During this activity releases are built in an environment as close to the target platform as possible. All acceptance tests are executed.

If the release fails to pass any test criteria it is returned to the Release Development Lifecycle STABLE activity along with error descriptions and change requests.

Execute: PROD

The final activity in the Release Development Lifecycle is the PROD activity. The primary reason for this activity is to indicate that the release is ready to be certified. Various tasks may be completed during this activity including:

- Release notes;
- Installation documentation; and
- Planning for release packaging.

Product Release Progression

Products go through a progression of releases as time goes on. Major releases are succeeded by minor releases. Major and Minor releases are “patched”. Figure 10.3 shows a typical product release progression.

10.5 Distribution Management Process

10.5.1 Objective

The objective of the Distribution Process is to define the activities in the distribution management process with adequate detail to facilitate the process implementation.

10.5.2 Goals

The goals of the Distribution Process are to:

- Outline activities, documentation and roles associated with providing released versions of the product(s) to the customers in a documented and controlled manner.
- Define high-level Distribution Process activities and the roles that are responsible for those activities;
- Establish documentation requirements for distribution management

10.5.3 Process

Scope

This distribution management process applies to all Xyce-related product releases.

General

Once a release has successfully completed the Release Management certification activity it becomes eligible for distribution. Releases that are eligible for distribution may be moved to an electronic distribution system, packaged for distribution, shipped to authorized customers, or installed as a resource on specific simulation platforms by authorized maintainers (to be accessed by customers).

Distribution management deals with the packaging, shipping, tracking and notifications associated with releases. Distribution management does not allow for any changes to be made to the underlying release as part of the Distribution Process.

For the period of time that a release is to be available to customers the release is said to be in distribution. If, for any reason, it is determined that a specific release should no longer be available for customers that specific release is said to be withdrawn. Withdrawn releases must not be distributed to any customer.

Requirements

The requirements for distribution management are as follows:

- All Xyce-related product distributions shall be controlled under this Xyce Distribution Process. As part of the release notes (and preferably as part of a product license) release customers should be advised that they are not allowed to redistribute any part of any Xyce-related product.
- Only those products that have successfully completed the certification activity of the Release Process are eligible for distribution.
- All requests for distributions and all actual distributions will be tracked. The minimum information required for distribution tracking includes customer name, customer location, Xyce product(s), release number, distribution date, media type, and target platform.
- Any release may be withdrawn from distribution. For example, an old release might be retired when a new release becomes eligible for distribution.
- Distribution must take place in such a way that only authorized customers receive the release.

10.5.4 Distribution Packaging

Prior to being distributed releases are packaged. Distribution packaging consists of:

- Identification of the correct product elements;
- Selection of media type based on customer requirements;
- Selection of a package format; and
- Packaging.

The key requirements for packaging are:

- All release packages will be properly labeled (physically and/or electronically) with the proper release identifier;
- The contents of all release packages will be properly documented; and
- Other than documentation activities (e.g. Release Notes), releases are never created or modified during the Distribution Process.

Media Type

The media type identifies the physical storage mechanism used by the release. Examples of media type include: CD, tape, and distribution server.

Package Type

Package type identifies a storage format. The three package types are multiple files, archive files, and mixed files.

A multiple file package contains all of the elements required to install the release with each file independently stored on the media.

An archive file is a single file that contains multiple files. Examples of archive files are tar files, ZIP files, and self contained installation files.

Mixed file packages will contain some combination of multiple files and archive files.

10.5.5 Distributing Releases

Once a release is eligible for distribution, customer requests for the release may be considered. Figure 10.4 illustrates the Distribution Process and the role(s) responsible for the activity. Roles are defined in Section 10.6.

Figure 10.4 shows the process flow and the role(s) responsible for each activity. Additional details of each activity, the role with primary responsibility for the activity, and the transitions between activities are covered in Table 10.6.

Activity	Description	Role Responsible
Submit	The request must include the customer name, product release number, media type, and target system. The request may be automatically generated from customer license lists (standing orders), generated by an electronic customer interface, or generated as a result of verbal or textual communications with the customer.	Customer or DL
Check	All distribution requests are checked for completeness. The request may be returned to the submitter or transitioned to the next activity.	DL
Evaluate	Distributions may only be made to customers who are licensed and authorized to receive the specific release requested. In addition, only requests for releases that are "eligible for distribution" may be considered. Requests may be rejected or transitioned to the next activity.	DL
Distribute	There are numerous ways that the actual distribution may occur. While the method of distribution and the specific packaging may vary, in no case shall the underlying release be modified. See Section 4.6.1 and Section 4.6.2 for additional details.	DL
Track	All distribution requests are tracked. The minimum data that is retained is customer name, request date and request status.	DL
Close	Perform process improvement and administratively close the distribution request.	DL

Table 10.6: Distribution Process Detail

Distribution Mechanisms

Examples of distribution mechanisms include;

- Releases may be automatically packaged, copied to selected media, and shipped to specific customers;
- Releases may be packaged and electronically distributed (pushed) to specific cus-

tomers; and

- Releases may be placed on a distribution server and made available for customers to “pull” at their convenience. In all cases, the distribution server software must authenticate customers prior to distribution.

10.5.6 Distribution Logging

All distributions will be logged. The minimum information that will be logged is:

- Customer name;
- Authentication source;
- Specific release (product and version);
- Distribution media;
- Distribution packaging;
- Shipment mechanism; and
- Distribution date.

10.5.7 Withdrawing Releases From Distribution

At times there will be need to withdraw Xyce releases from distribution. There are two general reasons that a release might be withdrawn.

First, old releases may become out of date as new releases become eligible for distribution.

Second, a release that is determined to contain significant issues may influence the CCB to decide to withdraw that specific release from distribution. In this case the release may be withdrawn immediately or it may be withdrawn at some time prior to the end the normal distribution cycle (i.e. after a patch release is available). In all cases release tracking information is referenced so that all customers of the release can be notified.

Table 10.7 provides more detail on the withdraw activities, descriptions, and the role primarily responsible for the activity.

Activity	Description	Role Responsible
Submit	A formal request is made to withdraw a specific release from distribution The request must at a minimum specify the product, the release(s) impacted, and the reason for the request.	CCB or PTL
Check	The request is checked for completeness. The request may be returned to the originator or transitioned to the next activity.	DL
Evaluate	The evaluator(s) must determine if it is appropriate to withdraw the release. Requests may be refused, deferred, or approved.	CCB
Withdraw	The release is withdrawn from distribution.	DL
Notify	All interest parties are notified that the release has been withdrawn. Interested parties include: customers, PTL, and CCB. Customers must be advised that withdrawn releases are not supported.	DL
Close	Perform process improvement and administratively close the distribution request.	DL

Table 10.7: Withdrawing a Release

10.6 Roles

10.6.1 General

Roles represent a functional division of process responsibilities. Roles are defined based on the current activity, and the tasks that need to be accomplished.

Individuals may function in more than one role; however, an individual needs to know in what role he/she is functioning at any single point of time so they can verify that the proper set of tasks is being executed.

10.6.2 Role Definitions

Table 10.8 identifies the roles that are required for the Release Process and the Distribution Processes. Additional detail for each role is provided in the sections following Table 10.8.

Role	Description
Change Control Board (CCB)	A group of individuals who are responsible for making high-level decisions regarding releases.
Product Team Lead (PTL)	An individual who is responsible for the product architecture, product development, and managing resources assigned to product related activities.
Developer (DEV)	Responsible for designing, writing, and maintaining the source and related documentation files that comprise some part of a product.
Quality Assurance (QA)	Individuals who are responsible for assuring that application software conforms to project standards, and for the testing and evaluating releases. This includes the Quality Assurance Lead (QAL), the Quality Assurance Builders (QAB), and the Quality Assurance Testers (QAT).
Release Lead (RL)	Responsible for overseeing and documenting all releases.
Distribution Lead (DL)	Responsibly for the majority of the tasks in the Distribution Process.
Customer	Recipient and/or User of the product. Customers primary reponsibilities include requesting releases.

Role	Description

Table 10.8: Roles

10.6.3 Role Details

Table 8 defined the roles required for RDM. The following sections expand, clarify, and illustrate those roles.

CCB

The CCB has defacto management authority of overall project direction and development resources. The CCB is comprised (at a minimum) of a representative from management, a representative from product management, and the product team lead (PTL).

Examples of the tasks that fall under the authority of the CCB include:

- Approving and certifying release requests;
- Evaluating withdraw requests;
- Resolving project level technical issues;
- Managing project resource availability; and
- Approving release timing.

PTL

The product team lead is responsible for day-to-day oversight of product technical issues and resources assigned to the product.

Examples of the tasks that fall under the authority of the PTL include:

- Review and scope release requests;
- Plan releases; and
- Direct release development.

DEV

The developer is responsible for designing, writing, and documenting code.

Examples of the tasks that fall under the authority of the developer include:

- Receiving assignments from the PTL to create, enhance, or debug product functionality;
- DEVEL activity of the Release Development Lifecycle including identification of the elements that go into the STABLE baseline; and
- Notifying QA when the release is ready to be transitioned to the STABLE activity.

QA

Quality Assurance has several sub-roles. The QA Lead (QAL), QA Builder (QAB, and the QA Tester (QAT).

QAL

The QA Lead (QAL) is responsible for:

- Managing evaluation of the Release Development Lifecycle STABLE and QA activities; and
- Verifying that all work done to support a release relates to specific change requests.

QAB

The QA Builder (QAB) is responsible for creating QA product releases at the direction of the QAL. Activities include:

- Documenting which element versions are included in the STABLE and QA
- Release Development Lifecycle activities;
- Creating the STABLE and QA release baselines; and
- Performing and evaluating the STABLE and QA builds for targeted release item environments.

QAT

The QA Tester (QAT) is responsible for preforming tasks to evaluate releases at the direction of the QAL. Activities include:

- Manual testing of releases; and
- Automated release testing.

RL

The release lead is responsible for overseeing the Release Process and for documenting all releases. Activities include:

- Ensuring inclusion of release notes and installation notes;
- Creating the PROD release baseline. Verifying contents of the PROD baseline;
- Documenting the date that the release becomes eligible for distribution; and
- Closing release requests.

DL

The distribution lead is responsible for the majority of the tasks during the Distribution Process. Activities include:

- Notifying of customers when a release becomes eligible for distribution;
- Notifying customers when a release is withdrawn from distribution; and
- Documenting and tracking distributions.

Customer

The customer has few (if any) official responsibilities related to RDM. Examples of the activities in which the customer may engage include:

- Obtaining appropriate licenses;

- Receiving and installing releases;
- Submitting product trouble reports;
- Submitting product enhancement requests; and
- Calling the help desk with issues.

10.7 Interfaces to Other Processes

Release and Distribution Management (RDM) has interfaces to other Xyce processes.

10.7.1 Issue Tracking

Release requests may be initiated as a result of defect reports or when the PTL requests a defect patch. Both require that an issue be reported to issue tracking. The QA Lead (QAL) may allow a release to proceed (with a defect) while reporting an issue (that will eventually generate another release request). The DL uses issue tracking to report media and packaging issues

The PTL (product team lead) is called the Team Lead (TL) in the Issue Tracking process (see [24]).

The work required for individual issue tracking is descibed in chapter 4. A flowchart (from that chapter) documenting issue tracking, and its relationship to the release process phases is shown in figure 4.1.

10.7.2 Requirements Management

Release requests may be initiated by a variety of sources. Release requests may include references to both new and existing requirements. Any new requirements that arise from release requests must be handled through the Requirements Management Process [25].

All release request links to requirements need to be documented sufficiently to allow requirements base testing of release functionality.

The PTL (product team lead) is called the TL in the Requirements Management document [25].

10.7.3 Third-Party Software (TPS)

The Xyce Parallel Electronic Simulator has dependencies on third-party software (TPS). The TPS required to build Xyce is typically utilized in the form of statically linked libraries that can be categorized in the following manner:

- Unmodified third party software (UTPS)
- Modified third party software (MTPS)
- Xyce specific external software (XSES)

Depending on which category of which a given library is a member, the maintenance and distribution of this software is handled slightly differently as is described in the Third-Party Software Configuration Management Plan, given in chapter 11. Here we simply specify that TPS artifacts are baselined and tracked in coordination with the Release Process.

10.7.4 Configuration Management

Product release and product distribution are components of Configuration Management (CM). Proper implementation of RDM requires that another component of CM, version control, is properly defined and implemented. Since version control procedures are not defined in this document, the assumption is made that proper version control will be performed for all items that are needed for Xyce releases. That is, all instances of all controlled Xyce elements (e.g. source files, build files, test files, data files, thirdparty libraries, documentation, release notes,...) will be placed under version control.

The Release Management Process assumes that a Configuration Management Process is properly implemented. Version control of elements that compose releases is of particular importance to the Release Development Lifecycle DEVEL, STABLE, QA, and PROD activities.

10.8 Guidelines for Release Notes

Suggested Outline for Release Notes

The following is a generic outline for release notes for a software product. The philosophy implied by this generic outline is that the release notes will provide an overall description of the product and then provide information just about the current release of the product.

10.8.1 Scope/Product Definition

This section would have an overall description of the product. This description should be general enough that it applies to all releases of the product. In other words, this section is release-independent.

10.8.2 Hardware/Software Information

Supported Platforms

This section gives the currently supported hardware/operating-system combinations.

Hardware Requirements

Minimum hardware requirements required for running the code. These may be platform dependent.

Software Requirements

Minimum software requirements for running the code, including any standard or third party libraries required by the code.

10.8.3 Release Documentation

10.8.4 New Features and Enhancements

Highlights

This section describes major features of the product that should be highlighted. In essence this is the "sales pitch" section. Like the Product Description section, the highlights described in this section should be general enough to apply to all releases of the product. Features specific to a release that should be highlighted will be included in the next section, New Features.

Package-Specific Features and Enhancements

This section describes the new features and enhancements at the UML package level.

10.8.5 Defects Fixed in this Release

This section details the defects that were fixed specifically in the noted release. Since a sub release should include all the fixes in the patches that preceded the subareas, descriptions of the defects fixed by those patches do not have to be included.

10.8.6 Known Defects and Workarounds

This section is optional. It highlights significant defects that were not fixed in the release. This section should be included if there are some major defects that are still outstanding or if before the completion of the release, a release was advertised to fix certain defects, but the fixes were not included. It should also include workarounds, if available.

10.9 Release Certification Tests

10.9.1 Checklist for Release Certification

The following checklist outlines the test necessary for certification of a Xyce release. As the Xyce code matures, this list will expand and evolve to match its capabilities. These tests encompass the standard Xyce test suite and add in several other tests which exercise other aspects of Xyce not covered by the current regression tests.

10.9.2 General Directions:

The release certification date should be filled in immediately to initiate this process. The initiation date must be the date of printing. The indicated steps may be completed in any reasonable order. The process shall be considered complete when all steps have been completed.

10.9.3 Specific Directions:

No specific form of review is required (i.e. Fagan Inspection). The project leader shall determine the scope of the planning and the method of conducting the review. The project leader shall notify planning participants and provide them with the applicable material.

This checklist is complete when the project leader determines its disposition. The project leader may consult with the participants to determine the final disposition.

10.9.4 QA

Process Ownership:

This document and associated procedure are owned by the Project Leader who must approve any changes. This document is version 1.0 and supercedes all previous versions. This document is under version control. All previous versions will print with a banner “OBSOLETE”.

Authority to perform process:

Any team member may initiate a meeting using this procedure.

If this process or associated document is superceded after an activity has been initiated, the process shall be completed under this version unless specifically notified otherwise by the team leader.

Stage Entrance and Stage Exit:

The stage entrance is the process initiation by any authorized person. All necessary process inputs are built into the checklist. The stage exit is the completion of this form (all steps completed and metrics entered.) Note (see Specific Directions above) that the project leader can terminate the process once the disposition can be determined to avoid wasted effort.

10.10 Team Checklists

10.10.1 Checklists for Release Process Activities

The following are a set of checklist tables for use with Xyce releases to help ensure completion and documentation of necessary activities. This section is focused on checklists for team-level activities. Following each table is a set of directions as well as QA information for the list. Note that these checklist are available as separate documents to be used and archived for each release cycle.

10.10.2 Release Planning Checklist

Release Number:____________

Release Planning Date:____________

Release Planning Participants:____

____

Activity	Description	Owner	Completion Date
Plan Target Capabilities	Generate list of target capabilities for current release. If possible, these should be traced to specific requirements and/or change requests.		
Develop Acceptance Criteria	Specify acceptance criteria for each target capability and/or functionality.		
Plan Target Functionality	Generate list of target functionality for current release. If possible, these should be traced to specific requirements and/or change requests.		
Activity Dates	Establish dates for each activity: ■ Code repository branch ■ Production build ■ Release notes completion ■ Release certification ■ Release notification Note that these activities may be specified in a relative manner (e.g., relative to the process initiation or an activity completion).		

Activity	Description	Owner	Completion Date

Table 10.9:

Plan Completion: Release plan complete

Completion Date:____________

General Directions:

The release planning date and participants should be filled in immediately to initiate this process. The initiation date must be the date of printing. The indicated steps may be completed in any reasonable order. The process shall be considered complete when all steps have been completed.

Specific Directions:

No specific form of review is required (i.e. Fagan Inspection). The project leader shall determine the scope of the planning and the method of conducting the review. The project leader shall notify planning participants and provide them with the applicable material.

This checklist is complete when the project leader determines its disposition. The project leader may consult with the participants to determine the final disposition.

QA

Process Ownership:

This document and associated procedure are owned by the Project Leader who must approve any changes. This document is version 1.0 and supercedes all previous versions. This document is under version control. All previous versions will print with a banner "OBSOLETE".

Authority to perform process:

Any team member may initiate a meeting using this procedure. If this process or associated document is superceded after an activity has been initiated, the process shall be completed under this version unless specifically notified otherwise by the team leader.

Stage Entrance and Stage Exit:

The stage entrance is the process initiation by any authorized person. All necessary process inputs are built into the checklist. The stage exit is the completion of this form (all steps completed and metrics entered.) Note (see Specific Directions above) that the project leader can terminate the process once the disposition can be determined to avoid wasted effort.

10.10.3 Release Configuration Management (RCM) Checklist

Release Number:______________

RCM Initiation Date:______________

RCM Initiation Owner:____

Activity	**Description**	**Owner**	**Completion Date**
Baseline CM Code Repository	The code CM repository should be baselined (tagged) according to the labeling convention established in the Release and Distribution Management.		
Create Release Branch	The code CM repository shall be branched in support of the Release Development Lifecycle.		
Track Release Development Lifecycle	The RDL shall be tracked in a manner consistent with the Release and Distribution Management document. This shall be accomplished by baselining each promotion through the RDL.		
Baseline Final Release Configuration	At the completion of the RDL (PROD stage), the repository shall be tagged according to the labeling convention established in the Release and Distribution Management guide.		

Table 10.10:

RCM Completion: RCM complete

Completion Date:______________

General Directions:

As release activities proceed through the RDL, this checklist must be updated/completed. The initiation date and owner shall be filled in immediately to initiate this process. The initiation date must be the date of printing. The indicated steps may be completed in any reasonable order. The process shall be considered complete when all steps have been completed.

QA

Process Ownership:

This document and associated procedure are owned by the Project Leader who must approve any changes. This document is version 1.0 and supercedes all previous versions. This document is under version control. All previous versions will print with a banner "OBSOLETE".

Authority to perform process:

Any team member may initiate a meeting using this procedure. If this process or associated document is superceded after an activity has been initiated, the process shall be completed under this version unless specifically notified otherwise by the team leader.

Stage Entrance and Stage Exit:

The stage entrance is the process initiation by any authorized person. All necessary process inputs are built into the checklist. The stage exit is the completion of this form (all steps completed and metrics entered.) Note (see Specific Directions above) that the project leader can terminate the process once the disposition can be determined to avoid wasted effort.

10.10.4 Release Development Lifestyle (RDL) Checklist

Release Number:_______________

RDL Initiation Date:_______________

RDL Initiation Owner:____

Activity	Description	Owner	Completion Date
DEVEL	During DEVEL for a given RDL, work is performed to add, enhance, fix or remove functionality. Completion of this stage is determined by the assigned developers and results in promotion to the STABLE stage.		
STABLE	Releases that have reached this activity are ready for element and component level testing. This is performed according to associated test plans. Failure in this stage requires returning to the DEVEL stage with associated change requests will be entered.		
QA	During this phase, acceptance tests are performed on all target platforms. In addition, testing and related activities may be performed by other QA individuals. Failure here will result in returning to the STABLE stage and possibly the DEVEL stage with associated change requests will be entered.		
PROD	At this final stage, the product is ready for certification. Here, release notes and other documentation may be initiated.		

Table 10.11:

RDL Completion: RDL complete

Completion Date:_______________

General Directions:

For each of possibly multiple passes (minimum one) through the RDL, this checklist must be completed. The initiation date and owner shall be filled in immediately to initiate this process. The initiation date must be the date of printing. The indicated steps may be completed in any reasonable order. The process shall be considered complete when all steps have been completed.

QA

Process Ownership:

This document and associated procedure are owned by the Project Leader who must approve any changes. This document is version 1.0 and supercedes all previous versions. This document is under version control. All previous versions will print with a banner "OBSOLETE".

Authority to perform process:

Any team member may initiate a meeting using this procedure. If this process or associated document is superceded after an activity has been initiated, the process shall be completed under this version unless specifically notified otherwise by the team leader.

Stage Entrance and Stage Exit:

The stage entrance is the process initiation by any authorized person. All necessary process inputs are built into the checklist. The stage exit is the completion of this form (all steps completed and metrics entered.) Note (see Specific Directions above) that the project leader can terminate the process once the disposition can be determined to avoid wasted effort.

10.10.5 Release Certification Checklist

Release Number:____________

Certification Initiation Date:____________

Certification Participants:____

Activity	Description	Owner	Completion Date
Perform Acceptance Tests	Acceptance test, as specified in the Release Plan, are performed and documented.		
Complete and Review Documentation	Documentation (Release Notes, updated Guides, etc.) is generated and reviewed by assigned participants.		
Certify Release	Release is certified assigned reviewers and management. The review form is archived.		

Table 10.12:

Certification Completion: Release Certification complete

Completion Date:____________

Certification Signatures:

QAL: ________________________

PTL: ________________________

PI: ________________________

Mgt. Rep.: ________________________

General Directions:

The certification date and participants should be filled in immediately to initiate this process. The initiation date must be the date of printing. The indicated steps may be completed in any reasonable order. The process shall be considered complete when all steps have been completed.

Specific Directions:

No specific form of review is required (i.e. Fagan Inspection). The project leader shall determine the scope of the planning and the method of conducting the review. The project leader shall notify planning participants and provide them with the applicable material.

This checklist is complete when the project leader determines its disposition. The project leader may consult with the participants to determine the final disposition.

QA

Process Ownership:

This document and associated procedure are owned by the Project Leader who must approve any changes. This document is version 1.0 and supercedes all previous versions. This document is under version control. All previous versions will print with a banner “OBSOLETE”.

Authority to perform process:

Any team member may initiate a meeting using this procedure. If this process or associated document is superceded after an activity has been initiated, the process shall be completed under this version unless specifically notified otherwise by the team leader.

Stage Entrance and Stage Exit:

The stage entrance is the process initiation by any authorized person. All necessary process inputs are built into the checklist. The stage exit is the completion of this form (all steps completed and metrics entered.) Note (see Specific Directions above) that the project leader can terminate the process once the disposition can be determined to avoid wasted effort.

10.10.6 Distribution Management Checklist

Release Number:______________

Distribution Initiation Date:______________

Distribution Owner:____

Activity	Description	Owner	Completion Date
Build Platform Executables	For each supported platform, build executables from the baselined release.		
Distribute Release	Distribute supported executables and associated artifacts according to distribution plan.		
Notify Customers	Notify customers of Release and provide release artifacts (either actual artifacts or instructions on access).		

Table 10.13:

Distribution Completion: Distribution complete

Completion Date:______________

General Directions:

Complete this checklist as part of the overall distribution process. The initiation date and owner shall be filled in immediately to initiate this process. The initiation date must be the date of printing. The indicated steps may be completed in any reasonable order. The process shall be considered complete when all steps have been completed.

QA

Process Ownership:

This document and associated procedure are owned by the Project Leader who must approve any changes. This document is version 1.0 and supercedes all previous versions. This document is under version control. All previous versions will print with a banner “OBSOLETE”.

Authority to perform process:

Any team member may initiate a meeting using this procedure. If this process or associated document is superceded after an activity has been initiated, the process shall be completed under this version unless specifically notified otherwise by the team leader.

Stage Entrance and Stage Exit:

The stage entrance is the process initiation by any authorized person. All necessary process inputs are built into the checklist. The stage exit is the completion of this form (all steps completed and metrics entered.) Note (see Specific Directions above) that the project leader can terminate the process once the disposition can be determined to avoid wasted effort.

10.11 Individual Roles Checklists

10.11.1 QA Roles

The Quality Assurance Lead, Builder, and Tester are responsible for ensuring that each product release conforms to project standards by compiling, testing, packaging, and evaluating release components in a rigorous and repeatable manner. The following checklists may be used as a guide for QA activities.

Release Number:____________

Quality Assurance Initiation Date:____________

Quality Assurance Lead:____

Quality Assurance Lead (QAL)

Activity	Description	Owner	Completion Date
Test Harness	Prepare the tools and environments necessary for testing prior to beginning QA activities.		
Test Coverage	Confirm that at least one test exists for every RESOLVED VERIFIED bug/feature.		
Initiate Testing	Provide lists of the build configurations, run-time configurations, and required tests to the QAB and QAT.		
Manage Testing	Check build and test reports for completeness. Initiate successive QA rounds as needed. Coordinate QAB and QAT activities.		
Verification	Verify that each build configuration passes its required set of tests.		

Activity	Description	Owner	Completion Date
Sign-off	Sign official release management documentation to certify that the QA process is complete.		

Table 10.14: QAL Roles

Quality Assurance Builder (QAB)

Activity	Description	Owner	Completion Date
Packaging	Compile and bundle release packages requested by the QAL.		
Verification	Verify that each release package is correctly formatted, containing proper documentation, static/shared executables, bundled libraries, support software, etc.		
Documentation	Document the configuration used to create release artifacts. This includes all data necessary to duplicate each QA round.		

Table 10.15: QAB Roles

Quality Assurance Tester (QAT)

Role	Description	Owner	Completion Date
Test	Use a combination of the automated test framework and manual testing to produce results for all run-time configurations and tests requested by the QAL.		

Role	Description	Owner	Completion Date
Verification	Verify that each release package passes all required tests.		
Archive	Archive configuration and test results in persistent storage. This includes all data necessary to reproduce each QA round.		

Table 10.16: QAT Roles

QA Completion: QA complete

Completion Date:________________

10.12 Release Activity Timeline

The preceding sections have given an overview of the **Xyce** release process from a high level. However, such an overview can be confusing to the typical code developer. What follows in this section is (in plain English) the steps typically taken to produce a formal a **Xyce** code release.

10.12.1 Release Version Numbering

Historically, the project has nominally aimed for two major releases a year, and they alternate between the *.0 and *.1 version number suffixes. So, for example, one could have major releases **Xyce** version 5.0 and **Xyce** version 5.1 in the same year. Being “major releases” means that the entire release process is followed and that the release is created from a completely new CVS branch. Generally, this is enough work that it should not be attempted very often, and each “major” release should include a large number of new features. In practice, the project has not managed to maintain the schedule necessary to complete two major releases in a year, and a more typical period is eight months per release. A typical release process will take about two months to execute, so this can be thought of as six months of major code development plus two months of release activity.

10.12.2 Release planning meeting

The first step in executing a formal release is to hold a release planning meeting. It may be necessary to have more than one. The planning meeting has a checklist which is documented in section 10.10.2, and this checklist can be used as a loose guideline for the goals of the meeting. Primarily, the meeting should address, features for the release, supported platforms, updated library builds, and setting various release dates.

In general, the earlier you can have this meeting, relative to when the release is supposed to happen, the better. Almost every step of the release will take longer than expected, so plan for it.

Feature, Issue and Bugfix Selection

Major features of a release should be agreed upon in a release planning meeting. Once they are agreed upon, all non-essentially bugzilla issues should be deferred to target later releases than the current one. Often, it will be necessary to create a new target milesone in bugzilla, to accomodate deferred bugs. So, for example, if the current release is going to be version 5.1, it may be necessary to create a new target milestone for version 6.0, to which low-priority bugs will be deferred.

Sometimes, on the **Xyce** project, we have gone through every bug in bugzilla during the meeting and decided as a team which ones to defer. Other times, we have left it up to the individual developers to make that choice. Most **Xyce** developers work with specific customers of the project, and as such each developer essentially represents the needs of their customers to the group. So, customer schedules and customer feature requests should be taken into account. It is sometimes not possible to accomodate every feature that every customer wants, so some compromises may need to be made.

Build/Platform/Library Support

Platforms and library versions to support should be agreed upon in a release planning meeting. At first glance, this may seem to be a trivial issue, but it can result in a non-trivial amount of work.

Build decisions include, but are not limited to, the following issues:

- Hardware platforms (intel, mac, windows etc.)
- OS Versions. Strive to support the corporate operating environment (COE), when possible.
- MPI implementation (mpich, OpenMPI, MPILAM, etc.)
- Compilers. Examples include Intel, gcc, Portland Group, etc.
- Libraries, especially Trilinos version. Trilinos typically has major releases once a year, in the fall. If possible, **Xyce** releases should use the most recent Trilinos released version.
- Special builds. For a variety of reasons, it may be desirable to support builds with special features enabled or disabled. Example include builds with and without OpenMP, with and without radiation models, with and without the Dakota library, and with and without the reaction parser enabled.

Ultimately, the agreed-upon library and platform support for a release will result in at least 10 or more unique builds. Each unique build needs to be tested separately for release, and the number of platforms that can be supported will be practically constrained by testing resources. It can also take weeks to months to get all the builds and all the automated testing to stabilize. As such, the earlier the platform decisions are made, the better.

Setting Activity Dates

Set dates for the following. Note, before the CVS Tag and Branch phase, you'll want to have all platforms in good shape. That means that all libraries and builds need to be installed and well established in regression testing. If you are upgrading libraries, or operating systems (to suit the current COE), it can take weeks for all test platforms to stabilize, so allow at least a month for this activity.

- Set a date for all test platorms to be upgraded to the current corporate operating environment (COE). This may require purchasing software licenses (for both OS and compilers), so allow for this. Not all test platforms are covered under site licenses, even for the COE. For example, some platforms require the server version of the OS, and server versions are often not covered by site licenses.
- Set a date for all library and platform updates to be complete. This means that libraries (such as Trilinos) are built and installed on a network drive for every supported platform.
- Set CVS Tag and Branch date. This should probably be a month after the library/platform date.
- Set QA cycle dates. Assume at least two QA cycles, and a week per cycle. Assume a day in between each one, for miscellaneous bookkeeping, etc.
- Set a date for final paperwork, including website updates.
- Set a date for the release announcement.

It is likely that some of these dates will slip. Ideally, most schedule slippage should happen in the first phase, prior to the branch date. It is much better to be doing bugfixes prior to branching than after.

Platform and Testing upgrades

Make a list of all the new libraries that need to be built, for every supported platform. Divide up the work by assigning a subset of platforms to each developer. It may be necessary to upgrade the OS and/or compilers on test platforms, if they have not been updated already.

10.12.3 Release Tag and Branch

10.12.4 QA process

10.12.5 Documentation

10.12.6 Final Paperwork

10.12.7 Website Updates

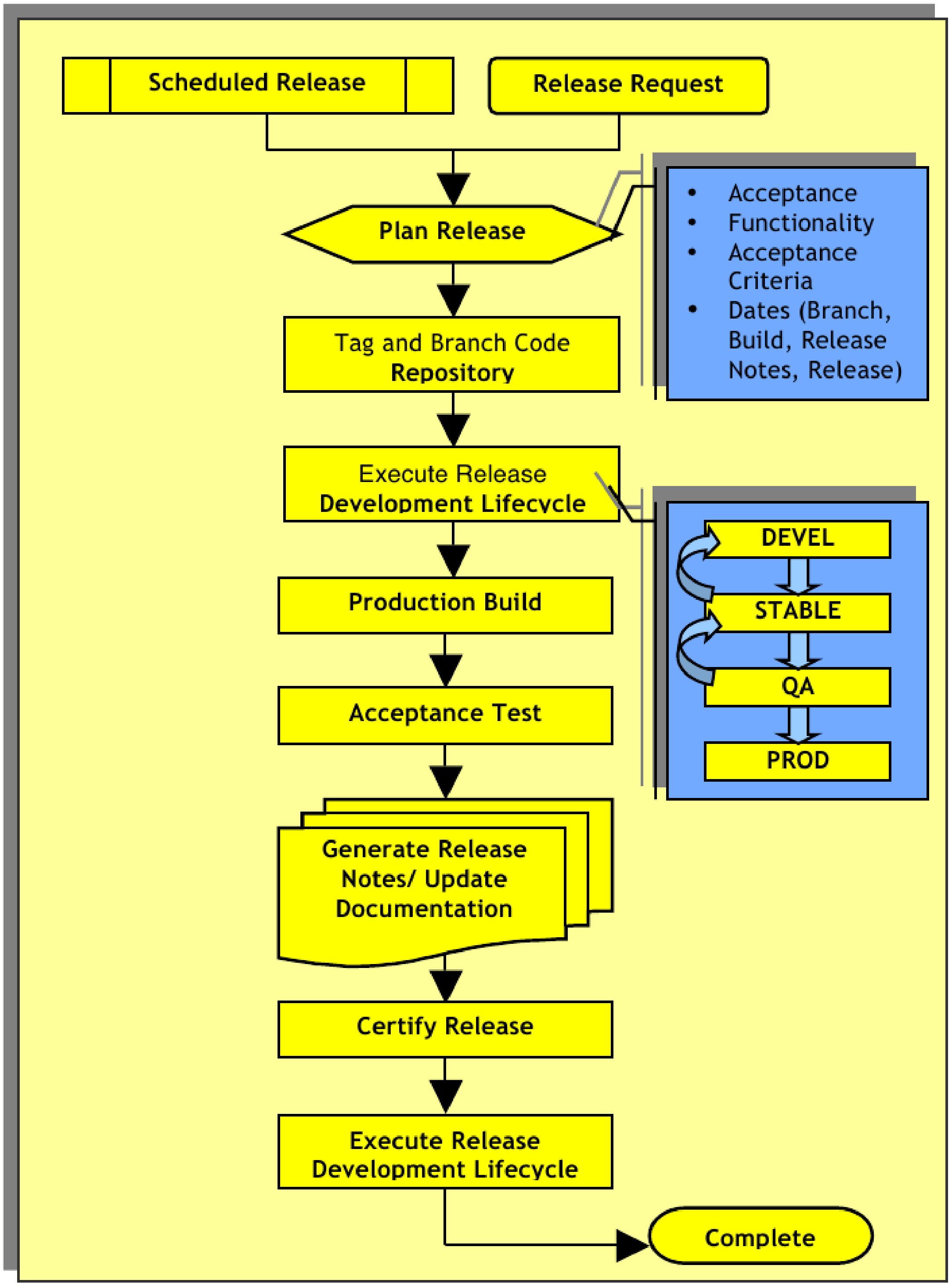
Scheduled Release
Release Request
Plan Release
Acceptance
Functionality
Acceptance Criteria
Dates (Branch, Build, Release Notes, Release)
Tag and Branch Code Repository
Execute Release Development Lifecycle
DEVEL
STABLE
QA
PROD
Production Build
Acceptance Test
Generate Release Notes/ Update Documentation
Certify Release
Execute Release Development Lifecycle
Complete

Figure 10.2. Release Process.

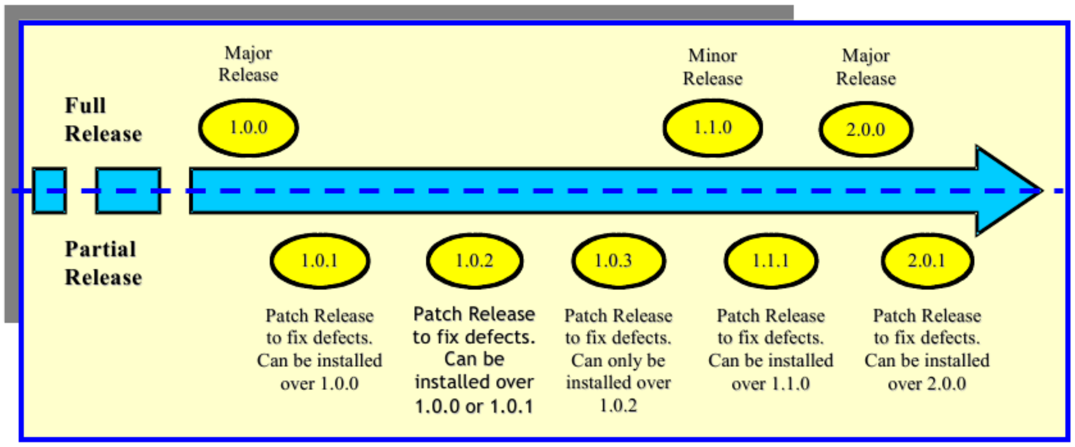
Major Release
1.0.0
Minor Release
1.1.0
Major Release
2.0.0
Full Release
Partial Release
1.0.1
Patch Release to fix defects. Can be installed over 1.0.0
1.0.2
Patch Release to fix defects. Can be installed over 1.0.0 or 1.0.1
1.0.3
Patch Release to fix defects. Can only be installed over 1.0.2
1.1.1
Patch Release to fix defects. Can be installed over 1.1.0
2.0.1
Patch Release to fix defects. Can be installed over 2.0.0

Figure 10.3. Example of a typical product release progression.

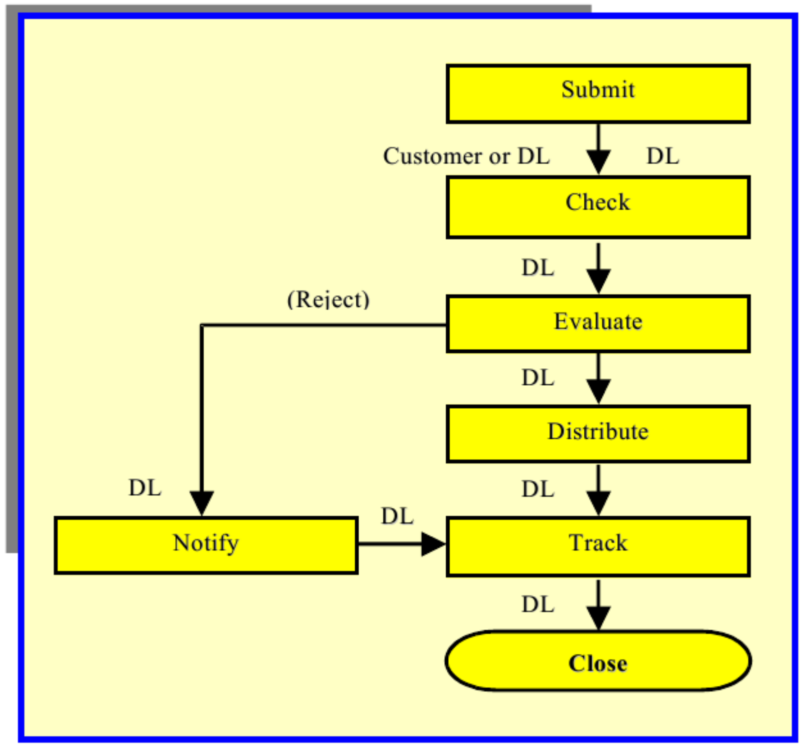

Figure 10.4. Distribution process.

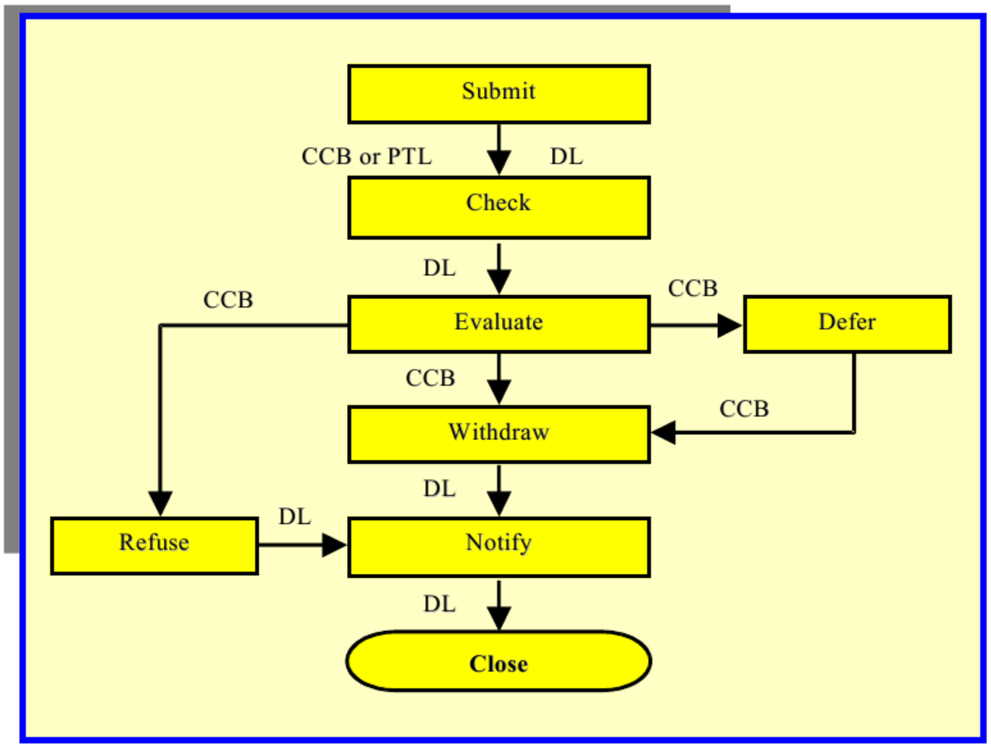

Figure 10.5. Illustration of the Withdraw process.

11. Third Party Software Management

Chapter Overview

This chapter presents a high-level description of the **Xyce**™ Third Party Software Configuration Management Plan. The purpose of this plan is to standardize the manner in which all third party software products required by **Xyce**™ maintained and made available for developers and others to build **Xyce**™.

11.1 Preface

This chapter presents a high-level description of the **Xyce**™ Third Party Software Configuration Management Plan. The purpose of this plan is to standardize the manner in which all third party software products required by **Xyce**™ maintained and made available for developers and others to build **Xyce**™. [23].

11.2 Preliminaries

11.2.1 Purpose

The purpose of this chapter is to provide a high-level description of the Xyce third-party software configuration management plan (TPSCM). This chapter describes the process elements of TPSCM, but it does not describe how they are being implemented. No specific tools are recommended or discussed in detail. There are several implementations that would successfully carry out the process described in this chapter. Therefore, even if the implementation or the toolset changes in the future, this process remains stable.

This chapter explains the acceptance/integration of related or dependent software developed by non-Xyce team members, herein called ”third party software.” Examples of third party software (TPS) are listed in the table in section 11.4. Many of these software groups or libraries are developed at Sandia. Many of these software sets are developed by other government or academic laboratories, by commercial vendors or by Sandia commercial or university partners.

11.2.2 Scope

The Xyce TPSCM process described in this chapter is required for each Xyce product. The main sections of this chapter focus on the tasks and activities that are required to manage the TPS.

11.2.3 Process Ownership

This chapter and associated procedure are owned by the Xyce Technical Leader and the Xyce Third Party Software owner who must approve any changes. Previous versions of this document were stand-alone documents, unlike this one which is a chapter of a larger document. This chapter is Version 1.1 of the third-party release plan, and supercedes all previous versions.

11.3 Third-party software management

11.3.1 Introduction

The TPS required to build Xyce is typically utilized in the form of statically linked libraries that can be categorized in the following manner:

- Unmodified third party software (UTPS)
- Modified third party software (MTPS)
- Xyce specific external software (XSES)

Depending on which category of which a given library is a member, the maintenance of the library is handled slightly differently as will be discussed. Currently, the issues regarding the overall maintenance of these libraries fall into two areas:

- The manner in which the source code for the libraries is handled, etc. (e.g., stored in a repository).
- The location of the built library files on the Xyce project file server

This is a simplification of the overall process and a more complete management system will be implemented in the near future. However, given the limited resources available, this chapter accurately represents the process currently in place.

11.3.2 General Third Party Software Practices

CM: Artifacts Supplied by Responsible Third Party Organization

All product artifacts received from the supplier for unmodified TPS will be stored in the associated tar file. Supplier-furnished artifacts might include:

- tar files, including source, documentation, MAKE, etc.
- Executables (object code)

- MAKEFILE for creating each library per platform
- Source Code
- Test Suites
- List of viable libraries (4th party) and associated details (e.g., build order dependencies)
- User documentation

Quality Assurance Practices

Supplied Source

Integration of externally supplied artifacts should include QA. If the software is delivered (source, library, executable), the QA should include applicable acceptance testing. This may consist of running supplied unit tests, followed by running any of the teams' own acceptance unit tests, and finally running integrated (regression) tests.

Supplied Documentation

The supplied software package should include documentation that consists of installation instructions, user guide, test cases, and test results. The documentation may guide the QA activities that are carried out.

Supplied Tests

The supplied software must go through the Xyce regression tests; if an error is encountered that points to the library, the Xyce developers will work with the suppliers to resolve the error. Some suppliers provide regression testing results or proof of validation; others do not.

Multi-Platform Support

Supporting MAKE/BUILD on all required platforms will be part of the integration work. The code management team may ask a supplier to deliver artifacts for all platforms but be told that only 80% of these platforms are supported. The team will have to port the TPS to the other 20%.

The Xyce development team will validate that a library runs on each of the required platforms and that test results are consistent across different platforms.

Platforms (both parallel and serial, as appropriate) include:

- Linux (Intel)
- Apple OSX (Intel)
- Apple OSX (Power PC)
- FreeBSD (Intel)
- Microsoft Windows (NT based)
- NWCC (Spirit)
- Thunderbird

Specialized research platforms that most external suppliers do not support will require the code management team support in-house on that particular platform:

- ICC
- ASC Red Storm

Managing Upgrades

Each active release of Xyce may depend upon externally supplied artifacts. All externally supplied artifacts required by an active release must be maintained. When maintenance is no longer needed, or a release has been archived, the dependent external artifacts may also be archived.

The phase-in period for a new version of third party software is defined to be ”time- limited” according to its replacement cycle. The Xyce development team will keep at most two versions current. All other versions will be archived.

The new software version will be announced when it has passed all quality assurance activities that are deemed reasonable by the process owner. These activities shall include, at a minimum, the suppliers unit and regression tests as well as integration testing within Xyce (see Appendix 2). Then the software will be integrated into the Xyce code system and the phase-in cycle for that software will be announced to the development team. This announcement serves to notify the team that the default TPS artifacts for the project have been upgraded and how the previous version, still available, may be accessed.

Build and Archiving Process

As new versions of a given TPS product are received, their respective improvements and enhancements are evaluated by the TPS owner and/or the Technical lead and may be scheduled for integration into the current working version of Xyce. This determination, informally conducted among the team members, is based upon a number of criteria including:

- Timing of the new TPS integration and its impact on other project drivers
- A cost/benefit analysis of the TPS integration that will take into account any interface changes required, etc.

If the outcome of this analysis provides for the integration of the upgraded TPS product, the libraries will be built and tested, with Xyce, against all supported platforms (see Section 2.2 above). The resulting artifacts (typically include files and statically linked library files) will then be archived on the server for access by processes that build Xyce. This is in a directory structure that parallels the supported platforms. As an example, these may be stored in

/Net/Proj/Xyce/arch/platform

where platform designates the appropriate supported platform (e.g., linux).

11.3.3 Practices for Unmodified Third Party Software (UTPS)

Unmodified TPS are those software artifacts used by Xyce as provided by the suppliers. The source code for these libraries is updated per the methods and frequency dictated by the suppliers. It will, however, only integrated into the Xyce build structure after acceptance testing has been performed. Interfaces with Configuration Management (CM) All artifacts associated with unmodified TPS used by Xyce will be stored in simple UNIX tar files on the Xyce fileserver. This is in a directory structure that parallels the supported platforms. As an example, these may be stored in

/Net/Proj/Xyce/arch/OTHER_SOURCES

on the Xyce file server.

11.3.4 Practices for Modified Third Party Software (MTPS)

Modified TPS artifacts are those used by Xyce by external suppliers but which must be modified, to some degree, in order to by used by Xyce. The source code and other artifacts supplied will be updated per the methods and frequency dictated by the suppliers. As above, it will only integrated into the Xyce build structure only after acceptance testing has been performed. Interfaces with Configuration Management All artifacts associated with modified TPS used by Xyce will be placed under internal configuration management/version control (CVS module). Once this is completed, the Xyce developers will modify the supplied source code as necessary, to support its integration with the current version of Xyce. These modifications are then tracked the CM/version control system.

11.3.5 Practices for Xyce-Specific External Software (XSES)

Xyce-Specific External Software (XSES) artifacts are those developed and supported by Xyce development team and are typically used as interfaces to TPS artifacts. They have been developed and/or maintained outside of the main Xyce source for a variety of reasons. Typically, however, it is because the functionality they provide is planned to be integrated into the associated TPS by its supplier. The source code and other artifacts will be updated per the methods and frequency dictated by the either the development team and/or the suppliers of the associated TPS. As above, it will only be integrated into the Xyce build structure after acceptance testing has been performed.

Interfaces with Configuration Management

All artifacts associated with XSES used by Xyce will be placed under internal configuration management/version control (CVS module). Once this is completed, the Xyce developers will modify the supplied source code as necessary, to support its integration with the current version of Xyce. These modifications are then tracked via the CM/version control system.

11.4 Third Party Software List

Software	Description	Owner/Vendo	UTPS /MTPS /XSES	Supported Release
Trilinos (and the individual packages therein) [26]	Parallel Linear Solver Framework	Mike Heroux (01416)	MTPS	7.0, 8.0 and development
Zoltan [27] / ParMETIS	Parallel Partioning and Data-Management Services	Karen Devine (01416)	UTPS	3.0
AMD / UMF-PACK [28]	UMFPACK is a set of routines for solving unsymmetric sparse linear systems, Ax=b, using the Unsymmetric MultiFrontal method. AMD is a set of routines for ordering a sparse matrix.	Tim Davis (U. Florida)	UTPS	4.1
KLU/BTF [29]	KLU is a circuit-specific sparse direct linear solver. BTF is a block triangular form permutation package	Tim Davis (U. Florida)	MTPS	1.0
SuperLU [30]	General purpose sparse direct linear solver	Sherry Li (LBNL)	UTPS	3.0

Table 11.1:

11.5 High-Level Description

Acquisition - TPS packages are acquired as a result of a request from a developer, supplier, or other interested party. The physical acquisition includes receipt of supplier/license and redistribution agreements, and the software distribution package and its associated artifacts.

Configuration - The TPS package is installed on all targeted platforms using the build instructions. Modifications are documented and versioned.

Unit/Regression Testing - If provided, the supplier's unit and regression tests will be run on each targeted platform. If problems occur, the executable may be returned to Configuration for rework. If interfaces to Xyce packages are affected, the package will go to Development & Migration.

Development & Migration - Developers of affected packages will modify these to accommodate interfaces to the TPS. These mods will be tested and migrated for integration testing.

Integration Testing - Each impacted Xyce executable will be rebuilt with the new TPS and tested via regression/integration tests. The results will be verified. Discrepancies may result in returning the TPS to Configuration or in returning the Xyce application to Development & Migration.

QA Review - A team or individual will review the Integration Testing results to determine that all activities are complete, that standards are upheld, and that test results are suitable. More testing or rework may result or the QA Review may result in acceptance of TPS.

Baseline - Now the TPS has been accepted. All associated and necessary agreements, licenses, user documentation, etc. have been resolved and prepared. The TPS, including all delivered artifacts as well as all built artifacts (e.g., libraries) for each targeted platform will be included in the baseline. The TPS is ready for internal use.

Release - The TPS package is installed in the Xyce environment. This will include installing end user documentation and updating the TPS web page information. Developers will be notified that the phase-in period for the new (current) software and will then have an announced period of time to phase out the previous version of the package.

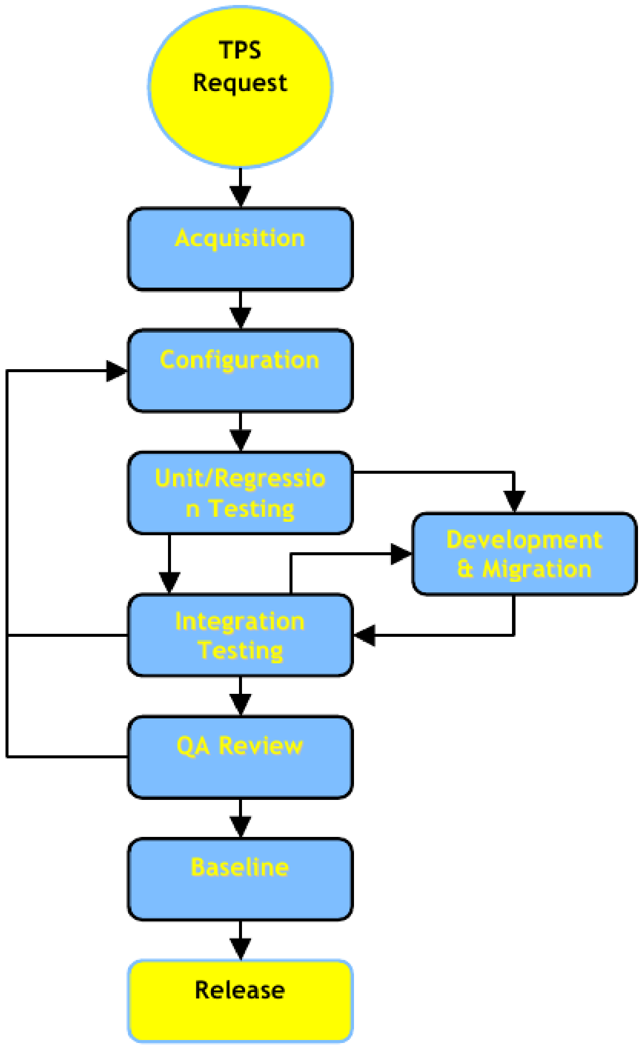

Figure 11.1. **Xyce** third party management.

Bibliography

[1] Charon Device Simulator. `http://micro.sandia.gov/charon.html`.

[2] Michael D. Simpson. Cvs version control & branch management. Dr. Dobb's Journal, pages 108–112, October 2000.

[3] C++ Programming Style Guidelines. `http://geosoft.no/development/cppstyle.html`.

[4] Scott Meyers. Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd Edition. Addison-Wesley Professional, 2005.

[5] Scott Meyers. More Effective C++: 35 New Ways to Improve Your Programs and Designs. Addison-Wesley Professional, 1996.

[6] Scott Meyers. Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library. Addison-Wesley Professional, 2001.

[7] C++ Coding Standard, Todd Hoff. `http://www.possibility.com/Cpp/CppCodingStandard.htm` .

[8] Doxygen documentation system. `http://www.stack.nl/ dimitri/doxygen/index.html`.

[9] Keith Gabryelski. Wildfire C++ Programming Style. `http://www.wildfire.com/ ag/Engineering/Development/C++Style/` .

[10] Steve McConnel. Code Complete.

[11] Marshall Cline. C++ FAQ Lite. `http://www.parashift.com/c++-faq-lite/` .

[12] Marshall Cline. C++ FAQs. Addison-Wesley, 1999.

[13] Steve Oualine. Vi IMproved - Vim, 2001.

[14] Vim Tip 12: Converting tabs to spaces.
`http://www.vim.org/tips/`.

[15] Tom Quarles. Spice3f5 users' guide. Technical report, University of California-Berkeley, Berkeley, California, 1994.

[16] Compact Model Council. `http://www.eigroup.org/cmc/` .

[17] Eric R. Keiter, Scott A. Hutchinson, Robert J. Hoekstra, Lon J. Waters, and Thomas V. Russo. Xyce parallel electronic simulator design: Mathematical formulation, version 2.0. Technical Report SAND2004-2283, Sandia National Laboratories, Albuquerque, NM, June 2004.

[18] Eric R. Keiter, Thomas V. Russo, Eric L. Rankin, Richard L. Schiek, Keith R. Santarelli, Heidi K. Thornquist, , Deborah A. Fixel, Todd S. Coffey, and Roger P. Pawlowski. Xyce parallel electronic simulator: User's guide, version 5.1.2. Technical Report SAND2010-3332, Sandia National Laboratories, Albuquerque, NM, 2010.

[19] Eric R. Keiter, Thomas V. Russo, Eric L. Rankin, Richard L. Schiek, Keith R. Santarelli, Heidi K. Thornquist, , Deborah A. Fixel, Todd S. Coffey, and Roger P. Pawlowski. Xyce parallel electronic simulator: Reference guide, version 5.1.2. Technical Report SAND2010-3331, Sandia National Laboratories, Albuquerque, NM, 2010.

[20] Eric R. Keiter, Thomas V. Russo, Eric L. Rankin, and Roger P. Pawlowski. Xyce parallel electronic simulator: Radiation models reference guide, version 5.1. Technical Report SAND2009-7324, Sandia National Laboratories, Albuquerque, NM, 2009.

[21] ADMS Model Compiler. `http://mot-adms.sourceforge.net/`.

[22] Kenneth S. Kundert. The Designer's Guide to SPICE and Spectre. Kluwer Academic Publishers, 1995.

[23] Asci apps software development guide. Technical report, May 2001.

[24] Xyce issue tracking. Technical report.

[25] Xyce requirements management. Technical report.

[26] The Trilinos Project. `http://www.cs.sandia.gov/Trilinos/`, 2002.

[27] Erik Boman and Karen Devine. Zoltan: Parallel Partitioning, Load Balancing and Data-Management Services. `http://www.cs.sandia.gov/Zoltan/`, 2007.

[28] Tim Davis. UMFPACK. `http://www.cise.ufl.edu/research/sparse/umfpack/` , 2007.

[29] Tim Davis. KLU. `http://www.cise.ufl.edu/research/sparse/klu/` , 2007.

[30] James W. Demmel, John R. Gilbert, and Xiaoye Li. SuperLU Users' Guide. `http://www.nersc.gov/ xiaoye/SuperLU/`, 1999.

Index

DISTRIBUTION:

0 none

Unless otherwise noted, all of the following copies were distributed electronically

1 MS 1323
Eric R. Keiter, 1445

1 MS 1138
Biliana Paskaleva, 06923

1 MS 1323
Eric L. Rankin, 1445

1 MS 1323
Thomas V. Russo, 1445

1 MS 1323
Richard Schiek, 1445

1 MS 1323
Heidi K. Thornquist, 1445

1 MS 1323
Scott A. Hutchinson, 1445

1 MS 1188
Christina E. Warrender, 6343

1 MS 1322
Ting Mei, 1445

1 MS 0899
Technical Library, 9536