Compartmentalized Continuous Integration: Enabling rapid flexible collaboration for complex software projects

David Neto*, Devin Sundaram†
Altera Corporation

May 19, 2011

Abstract

We describe a strategy for supporting rapid flexible collaboration for large complex software projects developed by hundreds of engineers at multiple sites around the world.

Existing strategies tradeoff development velocity or flexibility against build reliability, or they sharply degrade as the project increases in size and complexity. Using the Mainline development model as the foundation, we extend the Continuous Integration methodology,[1] making it practical for large, complex software projects, and avoid most of the undesirable tradeoffs forced by previous strategies.

*David Neto earned a PhD in Computer Science from the University of Toronto. He joined Altera in 2000, working on CAD for FPGAs, and was instrumental in migrating all Altera software development to Perforce. More recently he led Altera's software build process reengineering and unification effort.

†Devin Sundaram earned a BS in Engineering Physics from Cornell University. He joined Altera in 1997 and has worked on many areas of software infrastructure. He is a key Perforce and software build architect in Altera, and is responsible for the continuing evolution of Altera's software build and development environment.

[1]Unless otherwise stated, we use "integrate" and "integration" in the usual software engineering sense of "putting together the pieces", rather than the Perforce feature for propagating file contents from one depot path to another. In other words we use "integration" as in Continuous Integration rather than the `p4 integ` command.

We use a set of *gatekeeper builds* to validate new code changes before they are accepted into the project-wide integration build. A flexible and dynamic classification scheme is used to assign new changes to gatekeepers, based on what files are touched and sometimes what development site submitted the change. We use a second "sister" Perforce repository to track each change's integration status, allowing us to fully automate submission-time classification, selection of code for builds, and validation marking.

We compare our approach to other known strategies for managing feature integration including Microsoft's Virtual Build Labs, and various types of branching patterns.

Our Perforce-based policy layer has been in use at Altera Corporation since 2007, when we adopted Perforce for all software development work. In three years it has managed over 175 000 code changes from hundreds of developers at five development sites around the world, enabling the rapid development of our software offerings.

Contents

1 Prelude

After you submit a code change, how quickly will your fellow developers be able use it?

The delay is a function of the raw build time, and the probability that the build is broken. Build time increases linearly with code size and complexity. But the probability of a broken build increases *exponentially* with the rate of code change, i.e. the number of changes since the most recent good build. Each code change in isolation will break the build with a small probability, but the probabilities compound because *all* changes must be good for the build to be good.

Even worse, there is a reinforcing feedback loop. The longer you go between good builds, the greater the risk to the next build: even more changes will be included, and each will have been developed on an increasingly out of date build.

How do you cope if your project is complex, and the code changes very rapidly?

2 Introduction

Complex software projects with aggressive timelines face competing priorities. On one hand you want to support rapid and flexible collaboration among developers in various feature groups and sites around the world. On the other hand, you want to protect the integrity of builds, with a high probability of succesfully producing a complete product build at least every day.

The well known Continuous Integration methodology [3][1] tries to support both goals and aims to reduce overall project risk by allowing teams to catch integration issues early in the development cycle when they are easy and cheap to fix. But Continuous Integration can become impractical for large projects. For example, standard advice is to keep your build time as fast as possible, no more than 30 minutes or so. For complex projects this is simply not practical.

Many projects work around the integration problem by setting hard boundaries. For example, development of a component is often contained to a single team at one site. Or the project bans cross-component development, resulting in lost flexibility or development velocity. Hard boundaries can delay integration: the work of one feature group can take days or weeks to reach the rest of the team.

This paper describes Altera's solution to these challenges, which builds on both the Mainline development model [4] and Continuous Integration. First, we use a set of *gatekeeper builds* to validate most new code submissions before their inclusion in the product-wide integration build. Second, we apply a policy layer over Perforce to track the integration status of each change, and to enforce a lightweight checkin policy to ensure build stability. The policy applies to a single codeline at a time, and is implemented as a set of Perforce *form* and *change* triggers, and a client-side script. Integration status of each revision is tracked in a separate "sister" Perforce repository. This technique is generally applicable, and within limits can manage any user-level metadata not supported directly by Perforce. The policy can be viewed as a step toward supporting nested changelist submission.

Another benefit of our methodology is that it is easy to move a project or functional responsibility from one site to another. Our infrastructure nimbly supports changing business imperatives.

This paper is organized as follows. Section 3 summarizes Continuous Integration, and Section 4 shows how Continuous Integration can break down

for rapidly developed complex projects. Previous approaches to managing feature integration are described in Section 5. Altera's compartmentalization strategy is presented in Section 6 and the underlying mechanics are presented in Section 7. Sections 9 and 8 explore other uses for the ability of the backend to store metadata for file revisions. Finally, Altera's real world experiences with the system are summarized in Section 10.

3 Continuous Integration

In a large project, the trickiest and costliest problems are found only when you put together all the pieces. It pays to find and fix these integration problems as early and often as possible.

Continuous Integration is a set of best practices in software development that supports project integration on a rapid, repeated basis [3][1]. They are:

- Maintain a code repository
- Automate the build
- Make the build self-testing
- Everyone commits to the mainline every day
- Every commit (to the mainline) should be built
- Keep the build fast
- Test in a clone of the production environment
- Make it easy to get the latest deliverables
- Everyone can see the results of the latest build
- Automate deployment

The goal of is to perform a project-wide integration as often as possible. Striving to achieve this shapes your infrastructure, development practices, and attitudes.

Attitude is important. The entire team must commit to achieving successful integration at all stages of the project. For instance, a development task is not considered to be "done" until the feature appears in the integration

build and is proven to work there. That shared commitment should make developers uneasy when they risk divergence by working in isolation for any lengthy period, e.g. when they are using an old build as a stable development base, or when they commit their changes their only infrequently.

We cannot emphasize enough the importance of frequent integration. It really does reduce project risk.

4 The challenge of size and complexity

Continuous Integration is a laudable ideal, but there will always be conditions under which it will become impractical.

Despite your best efforts, "keep the build fast" may not be achievable for complex projects. Often "the build" is more than just compiling code: it also delivers data, documentation, and other non-executable material. Furthermore, in a self-testing build of a layered system, a higher-layer component will have to wait for the build of lower-layer component, even if only to run its validating tests. This reduces the opportunity to use parallelism to speed up the build. Section 4.1 describes key aspects of Altera's software build, providing a sense of scale to what we mean when we say "large" and "complex".

4.1 What is large? Altera's build by the numbers

Altera's primary software product is ACDS, the Altera Complete Design Suite, a CAD system for our programmable logic devices (usually known as Field Programmable Gate Arrays or FPGAs).

The largest FPGAs have billions of transistors, and the lowest levels in the CAD system must accurately model all those silicon details for logic function, connectivity, timing, power, etc. Higher levels of the CAD system provide programming support including compilers for multiple languages (Verilog, VHDL, C), software-based simulation and hardware debug facilities, configurable design fragments ("IP cores"), and component-based system integration tools.

Here are a few key facts about the ACDS build:

- It uses 255K source files, takes up 45GB on disk, and the source takes 27 minutes to sync into an empty workspace.

- The build system has over 1000 makefiles.
- The product-wide integration build always builds into an empty workspace. This is more robust because it minimizes the impact of incorrect or incomplete dependency information.
- It has many layers and therefore functional dependencies. All stages have smoke tests.
- The end-to-end build time is 14 hours on a multi-processor Linux machine. The Windows build is slower.
- The majority of the C/C++ code is compiled in under 30 minutes.
- Processing device data takes about 3 hours, even in parallel.
- ACDS is delivered on four platforms: Linux and Windows, both 32 and 64 bits.
- It produces about 50GB of output.
- Hundreds of engineers develop ACDS, at five sites around the world.
- Hundreds of source changes are committed per day.
- Often, wideranging changes must quickly be propagated throughout the product. For example, the development of a new device family occurs under tight timelines, and will impact most of the CAD system.

Altera has already invested a lot in trying to keep builds fast. For example, we exploit parallelism as much as possible, e.g. when compiling code and building device information. But much of the delay is induced by the functional dependencies between layers of the system. Most components require device information, and so must be built after devices are ready.

As a development project, ACDS certainly counts as "large and complex", even with ongoing investment in making its build faster. It cannot intrinsically meet the Continuous Integration ideal of a "fast" build, so instead we must adapt the Continuous Integration strategy for ACDS.

4.2 What is fast enough?

What is fast enough? The typical guidance is that the complete build process should take up between 10 minutes to an hour [3].

In the Prelude we argued that you have to pay attention to both the duration of a build, and how quickly the code changes. The key metric is *the number of changes between successful integrations.*

For example, if you have 100 changes a day, and every change is good with a probability of 99%, then the chance of a clean daily build is just under 37%. If you have 200 changes a day, it's only 17%. This is simply not good enough.

With a complex build, demanding even 99% reliability from your developers can be too expensive. It is too much to ask each developer to rebuild everything, or to rebuild a small change on all platforms. You are better off accepting some risk in your build, and assigning staff to monitor and repair the integration build mid-stream as needed.

If you have enough broken builds in a row, then you can get into a vicious divergent spiral: Your developers base their work on an old build (a.k.a. stable base), and that just increases the chance of introducing yet more integration problems the next time they check in code or data.

4.3 How Continuous Integration breaks down with a complex build

The primary goal of Continuous Integration is to provide rapid feedback [3].

Effective *negative* feedback has these properties:

- It says that a problem exists.
- It says the problem is *new*, i.e. did not exist until recently.
- It is actionable, i.e. someone has a reasonable chance of knowing a problem is theirs to fix.

But this breaks down if your code changes too quickly relative to build time.

- The build is too big to be fast. Bigness can be measured in sync time, compile time, and layering complexity.

In Altera's case, Moore's Law does not help! Our device data scales up with Moore's Law even while CPU vendors can only give us more cores with those transistors.

- Change velocity makes it hard to tell who broke the build: Many changes may have been committed since the previous build. A standard answer is to increase the build frequency and even run pipelined builds. But this can overwhelm your CPU or disk resources. It would be very expensive to buy enough hardware to run many more builds.
- Semantic complexity makes it hard to tell who broke the build: Cause and effect may be widely separated.
  - It might take three hours between compiling the faulty change, and the failure of the smoke test that catches the bug.
  - In some cases better small-scale testing can help, but it's no silver bullet!

The result is that plain Continuous Integration is ineffective with a complex build undergoing rapid development.

So you must keep the build stable. But how? We resort to the very general *Compartmentalization* design pattern: limit the damage to the whole by separating the parts in some way. That is, you must limit the number of unstable changes accepted into the build. The next section describes other previous approaches for doing so. Section 6 describes our approach for limiting instability while also supporting rapid incorporation of good changes.

5 Other approaches

Two simple techniques can go a long way to reducing the pain of a long-duration build, but do not fully address integration challenges for complex builds. They are described in Sections 5.1 and 5.2

Many other techniques require more codelines and manual merges between them. (See Sections 5.3 and 5.4.) We reject those techniques because of the extra confusion they cause, their complexity, and the resulting inability to make a coordinated change across multiple components.

Finally, Section 5.5 describes the Promotion model, which is undesirable in its conventional form, but which we adapt to track integration status over short timescales.

5.1 Staged builds

Fowler describes "staged builds", where the end-to-end build process is broken into a pipeline of stages [3]. The first stage performs code compilation and only very minimal testing. Typically, later stages consist mainly of more elaborate testing, e.g. with larger test data and more complex cases. Developers mostly work off the results of the first build, and therefore only pay attention to it, thus tightening the integration cycle.

We think this works well as far as it goes. But it does not fully address the development requirements of layered systems. Events like device rollout require rapid dissemination of device modeling updates, and development in upper layers (like IP cores) require updated baseline support in lower level support (like compilation and simulation support). Intrinsically, staged builds do not address truly project-wide integration concerns.

5.2 Incremental rebuild bots

Another technique for addressing long build times is the use of incremental compilation tests to catch simple breaks.

For example, you can set up an incremental build bot which kicks off an incremental rebuild of every newly submitted change, notifying developers of pass/fail status of the incremental build. It always builds upon the latest internal release of the build, and accumulates new changes in its workspace as they pass its own incremental test.

An incremental build bot can rapidly catch a large fraction of naive errors such as portability problems. However, the feedback tends to be one-sided because a "pass" is necessary but not sufficient for successful project-wide integration. Matters are complicated further because in large projects dependency information may be incomplete or out-of-date.

Also, some code changes induce large changes in automatically generated code and data. The incremental rebuild does not have time to propogate such a change through the build, because it may require a full product rebuild. For such changes, a perfect test is really the product-wide integration test, which is what you're trying to speed up in the first place!

Overall, incremental rebuild bots provide high value, effectively improving the apparent reliability of developers. But their caveats must be clearly understood by the development team, and they do not fully address the integration challenges of complex builds.

5.3 Virtual Build Labs

Maraia [5] describes Virtual Build Labs (VBL). A VBL is a build of a portion of the product based on code that is segregated on its own branch. Each component team works only from their own branch, with their own private build. From time to time their work is propagated to the back into the mainline of the entire project, and vice versa.

This technique suffers from several major problems:

- Development occurs on a separate branch, isolated from other teams. This greatly increases the chances of divergence between teams, and therefore increases the difficulty of reintegration.
- Integration between codelines is manual. The larger the project, the more challenging this becomes. In effect you are expecting some poor soul to expertly resolve the toughest integration challenges, and to do so rapidly.
- It takes a long time for updates to propagate to everyone. Imagine how a feature moves from team A to team B: It is developed by team A, must pass their own build, then eventually be manually integrated to the mainline, pass there, and then eventually be manually integrated into team B's tree. This is not a recipe for rapid collaboration.
- To make a pervasive change to the project, you have to break the model. A pervasive change would require a private branch of the entire code base, or a blind change that would be outside the model.

These difficulties are common to any system that requires multiple codelines.

5.4 Other strategies using multiple codelines

The difficulties with Virtual Build Labs are common to any system in which development occurs on multiple codelines. Such systems isolate one body of work from another, require manual intervention to push changes across all teams, and therefore strongly deter frequent integration. In essence, we are reminded of why the mainline model provides so much value in the first place!

A selection of these other techniques include [2]:

Remote Development Line This model uses a separate codeline for each development site. Changes are manually propagated between them.

Inside/Outside Codelines This model uses different codelines to segregate work done by an internal tightly coupled team from work done by an external entity. Occasionally changes are manually propagated in one or both directions.

Change Propagation Queues A propagation queue is used to enforce a strict ordering of changes being migrated from one codeline to another. Maintaining a strict order can reduce the difficulty of performing the integration, but in general it still requires manual intervention.

Virtual Codeline This pattern is conceptually the same as using multiple codelines for separate development as for Virtual Build Labs, but it minimizes the amount branching actually performed in the underyling SCM. It primarily uses a single *physical* codeline for all files, and a floating label to describe the contents of the *virtual* codeline. Update conflicts on the same file are resolved by just-in-time branching: all but the first are committed to branched versions of the file. The label for each virtual codeline is tagged with the associated branched version of the file. Eventually a new stable base is generated by manually merging the virtual codelines back into the primary codeline.

Of these, Virtual Codelines is the closest to our approach. However, we never use branching to represent integration status, and we simply ban conflicting updates in the short term.

5.5 Promotion levels

Finally, in the Promotion Model (e.g. see its characterization in [4]), the trunk codeline is used simultaneously for different aspects of release management, stretching over long periods of time.

Each file is tagged with its lifecycle status (e.g. development, QA, released, or retired) and its status is updated over time. A user can request the latest set of files at a given promotion level, e.g. for acceptance testing a user might ask for all files in the QA state.

However, complex mechanisms are required when you need to update a file for a given purpose but the head revision is not tagged with the corresponding

tag. For example, the head revision may be tagged for "development", but QA may need to check in an update based on an older revision having the "QA" tag. In this case you must either resort to per-file branching as in Virtual Codelines, or resort to checking in the new change at the head revision and adding a weaving-type mechanism on top of the base system. Either choice leads to undesirable complexity and confusion [4].

In our approach, we use separate codelines for release management as the in the reference Mainline model. However, like the Promotion model we track metadata for file revisions on a codeline. But there are three key differences. First, the "integration status" we track on each file revision is short-lived. A file revision is not-yet-fully-integrated for only a short time, i.e. until the next good build, and that *should be* a short time. (After all that's the whole point of continuous integration!) Second we avoid update conflicts via a change submission exclusion rule. Finally, integration status does not restrict visibility or access to file revisions. For almost all uses, we retain the simplicity of a single codeline as in the standard Mainline model.

6 Compartmentalized Continuous Integration

This section describes how Altera has adapted Continuous Integration so it is effective in a rapidly evolving complex software project. Our goals are:

- Rapidly provide effective integration feedback
- Reliably produce a product-wide integration build
- Quickly propagate good changes across the entire team
- Support changes that cut across many components, when needed

We have argued that the enemy of an integration build is the amount of untested change it accepts. We can't reduce the number of changes submitted, so instead we need to limit the impact of a bad change. We use a set of smaller *gatekeeper builds* to validate new changes before they are permitted into the integration build. The trick is to retain flexibility, automation, and simplicity from the user's perspective.

The key features and requirements of our strategy are:

- The policy should be deployable on a codeline-by-codeline basis.

- The policy should be deployed only with server-side configuration, so that it works with all clients transparently.
- The policy should be reconfigurable as conditions change. Module definitions change, sites change.
- No manual intervention should be essential. No manual merging should ever be required.
- Use the mainline model; don't overbranch. The codeline checkin policy is the same as it would otherwise be.
- Keep views simple. Developers should see all the code in one place.
- The common case should be simple, following the 80-20 rule. Most developers should never have to learn the details of the system.
- Support rapid development, e.g. multiple changes to a file between integration builds.
- Support flexible collaboration. A developer should be able to make any type of change, even one spanning the entire product if necessary.
- Support rapid dissemination of good changes.

At a high level our strategy is as follows:

- All new code changes are guilty until proven innocent. This implies that any point in time each file revision is in one of two *integration states*: *fresh* (guilty) or *verified* (innocent).
- A code change proves its innocence by being successfully incorporated into a build of some type. This induces a transition from *fresh* to *verified* for the file revisions in the change.
- We limit the number of *unverified* changes permitted in any build. This improves the ability to find what change caused the break, and keeps breaks close to the breaker.
- We support error containtment by major component. Most updates are contained within one component. Modularity helps.

- We support error containtment by site. This can be more a natural cutline when a component is developed simultaneously at multiple locations. But you want a build break to be limited to where the break was submitted. (You break it, you own it.)

More precisely, "integration status" has these properties:

- A newly submitted revision always starts out in the *fresh* state.
- A file may have more than one revision in the *fresh* state.
- If a file has a *verified* revision, then all previous revisions are also *verified*.
- At some point in time a revision will transition from *fresh* to *verified*; at that time all previous revisions will be marked as *verified* if they aren't already.
- By design, a file revision will be in the *fresh* state for only a short period of time. Almost all file revisions will be in *verified* state.

6.1 Zones, Domains, and Gatekeepers

The agent of error containment is a *gatekeeper build*. Its job is to prove the innocence or guilt of a subset of recently submitted changes by running a portion of the overall build. (Some kinds of gatekeepers run the *full* integration build flow, but with carefully selected file revisions.)

You should have a gatekeeper for each active major component in your overall build. In other words, divide along the modular boundaries. Also, if a large component is developed at multiple sites *you should have a set of gatekeepers to contain build breaks by site.* To support changes crossing multiple components, we can also use a *combination gateekeper*, or simply fast track those rare changes directly into the project-wide integration build.

The remainder of this section provides more precise descriptions of these concepts.

Definition: A *zone* is a named set of files under the codeline in question. A zone is a *code zone* if it contains source files used by the product-wide integration build. All other zones are *non-code zones*. A non-code zone will include things like regression test files which are not used during the build, but which may be used after delivery of the build.

Create a zone for each major component in the product. For example, ACDS has:

- Four primary code zones: Quartus, SOPC Builder, DSP Builder, and IP, with associated names `quartus`, `sopc`, `dsp` and `ip`. Each corresponds to a major component or layer in the product.
- Two incidental code zones: Version, and Common, respectively `version` and `common`. The Version zone is tiny, encapsulating build numbering. The Common zone contains build infrastructure such as makefile fragments and build scripts.
- A non-code zone: `therest`. It covers the regression test suite.

Definition: A *site* is a location in your organization that runs its own product-wide integration build and delivers that build to local users for further development, test, or general use.

ACDS is developed in five locations: San Jose, Santa Cruz, Toronto, Penang, and the High Wycombe, with four corresponding sites `SJ`, `TO`, `PN`, and `UK`. Santa Cruz is lumped together with San Jose because they both share the integration build created in San Jose.

Definition: A zone can be *site-sensitive*. Build errors for a site-specific zone should be contained to the originating site.

For example, in ACDS the `quartus` zone is site sensitive because its component build is rather large and its build errors should be contained to the originating site.

Definition: A *domain* is either a zone-site pair (if the zone is site-sensitive), or otherwise just a zone. When a zone is site-sensitive, its domains will be written as *zone-name*:*site-name*.

In ACDS, the `quartus` zone is site-sensitive, so it has associated domains `quartus:SJ`, `quartus:TO`, `quartus:PN`, and `quartus:UK`. The other ACDS zones are not site-sensitive, so their associated domains are just the zones themselves.

Each change is assigned to a domain at change submission time. The zone is determined by what set of *code zones* cover all the code files the change (non-code zones don't count). If the change includes files from more than one code zone, then it is assigned to the "COMBO" meta-zone; see Section 6.2. A new change is always assigned to a site, regardless of whether the assigned zone is site-sensitive.

For every change, *some* build must be responsible for validating it, i.e. updating the integration status of its file revisions from *fresh* to *verified*. For simplicity's sake, and to keep errors contained, there is usually only one responsible build per domain, but the infrastructure supports there being more than one. It can even be the product-wide integration build.

Definition: A *gatekeeper build* is associated with one or more domains, and:

- Runs a portion or all of the product-wide build. If it runs only portion of the full build, then it is rebuilt on top of a recent stable base of the product-wide build. It could rebuild just one component of the product build, or alternately a small set of logically related components.
- Is responsible for proving or disproving that the set of *fresh* changes from its associated domains are stable, i.e. they compile and pass the smoke tests.
- Uses a mixture of source file revisions. The set of *files* must be sufficient to support the rebuild process; this is expressed as a list of code zones. For each file the *revision* used is the later of:
  - A revision proven stable by a previous gatekeeper build (for the same or a different domain) or by a previous product-wide integration build. That is, a *verified* revision.
  - A revision from a recent code change that has not yet been proven stable, but which has been assigned to (one of) this gatekeeper's associated domain(s). That is, a *fresh* revision assigned to an error containment domain associated with this gatekeeper.
- If the gatekeeper build is successful, it marks the *fresh* revisions it used as *verified*. Integration builds can also do this.
- If the gatekeeper build is not succesful, the build breaks are advertised, and the likely culprits are notified.

Summary: The job of a gatekeeper build is to take *fresh* changes from some domain and validate their stability. If they are stable, then mark those changes as *verified* and therefore ready for other gatekeepers and the product-wide integration build. Otherwise, the gatekeeper build advertises the break, and notifies the likely culprits.

6.2 The COMBO domain, and COMBO changes

There is always a special meta-domain known as "COMBO". Any change that includes files from more than one code zone is assigned to the COMBO domain, and is called a COMBO change.

COMBO changes allow you to *safely* make a sweeping product-wide change when needed. With a COMBO change you won't *automatically* induce potential build breaks among the component builds. This is a fundamental advantage over any integration methodology that requires multiple codelines or repositories, for example the approaches described in Section 5.

COMBO changes should be rare. If they are not, you should refactor your zone definitions. If you have a large build but have a hard time breaking it up into zones, then you should seriously consider refactoring your code.

If you have the spare hardware, you can set up a COMBO gatekeeper. The COMBO gatekeeper requires code from all code zones, but for each file will take the later of the last *verified* revision and the last *fresh* revision assigned to the COMBO zone. Otherwise COMBO changes go directly into the product-wide integration build, and you accept the risk that implies. This can often be acceptable because COMBO changes occur infrequently.

6.3 Submission exclusion rule

We apply this submission exclusion rule: *A file may have fresh revisions from at most one domain.* (For the purposes of exclusion, a fresh COMBO change conflicts with fresh changes from all regular domains.) Section 7 describes how to use Perforce triggers to enforce this rule.

The rule still permits rapid development in the common case because a file may have many *fresh* revisions from a single domain. That is, several updates may be made by the same team to the same file between successful gatekeepers or integration builds.

The exclusion rule is used to avoid many types of integration breaks in a gatekeeper. Because each gatekeeper only sees its "own" fresh changes, you want to prevent any fresh change from one domain from *requiring* a fresh change from a different domain (i.e. is destined for a different gatekeeper).

This example shows how a file update could be assigned to more than one domain, and hence be destined for different gatekeepers:

- Suppose each of your gatekeepers is responsible for validating fresh changes from exactly one domain.

- Suppose source file `foo.c` is in zone A.
- When `foo.c` is updated, the updates could be assigned to several different domains:
  - If zone A is not site-sensitive, then the update could be assigned to domain A or `COMBO`, which are processed by different gatekeepers.
  - If zone A is site-sensitive, then the update could be assigned to domain A:X for any site X or to the `COMBO` domain. Each of those domains is processed by a different gatekeeper.

Section 7.2 describes the domain assignment algorithm in more detail.

6.3.1 Motivating example for exclusion

Here's a concrete example: Suppose you have sites Toronto (`TO`) and San Jose (`SJ`) and zones `A` and `B`. Furthermore, zone `A` is site-specific. Your domains are `A:SJ`, `A:TO`, and `B`. Suppose you also have a gatekeeper for each domain.

- Alice works in `TO`, and commits a change to files `foo.c` and `foo.h`, both of which are in zone `A`. Then this change is assigned to domain `A:TO`, and will be validated by the `A:TO` gatekeeper build.
- Bob works in `SJ`, and tries to submit a change to files `foo.c` and `bar.c`, and both files are in zone `A`.
- Perforce forces Bob to resolve his change with the updates Alice made to `foo.c`. Bob resolves by syncing both of Alice's `foo.c` and `foo.h`, and merges his change for `foo.c`.
- Bob tries to submit again. Bob's change would normally be validated by the gatekeeper for domain `A:SJ`. But the `A:SJ` gatekeeper will not see Alice's *fresh* change to `foo.h` because Alice's change is in the `A:TO` domain. If we allow Bob's change, then we pretty much guarantee an integration break in the `A:SJ` gatekeeper build.

This is a contrived example, but the problem illustrated is real and often difficult to work around.

We find that it is much better for Bob's submission to be rejected, at least temporarily. Bob must wait until Alice's change has been validated by

the `A:TO` gatekeeper or the product-wide integration build. Then Bob can move his work to the stable base incorporating Alice's change, and retest his change (including the effect of Alice's work), and resubmit.

Bob can be aggressive and cheat a bit. He can submit his change immediately after Alice's change has been marked as *verified*. That is, he doesn't have to wait until the next full product-wide build. The next iteration of the `A:SJ` gatekeeper build will include both his and Alices's change, because they are both from the same *zone* and Alice's change has alredy been verified. Remember that gatekeeper builds accept all *verified* changes (in addition to *fresh* ones) from the zones they are gating.

Note that if `foo.h` describes a fundamental API, then updating it can imply rebuilding nearly the entire product. In case of a complex build, Bob is better off waiting for the next stable base anyway.

6.3.2 Relaxing the exclusion rule when the build frequency drops

Toward the end of the release stabilization period, development slows and you often reduce the frequency of builds. Each release candidate build is tested thoroughly, and you wait a longer time between each build. Very few changes are permitted, and each is carefully reviewed.

The chance of incurring integration breaks during this period is very low, and the exclusion rule is only a nuisance. So we turn it off with an additional flag to the triggers applied to that codeline.

7 Mechanics

Multiple coordinating pieces are required to apply the policy on a particular codeline:

Integration status tracking We maintain a database of the integration status and assigned domain for each committed file revision.

Domain assignment A newly submitted change must be assigned to exactly one domain.

Submission exclusion Once assigned to a domain, a new change is rejected if it would break the exclusion rule.

Commit tracking When a change is committed, we add it to the integration status database, with status *fresh*.

Code selection for builds Each build needs to know what file revisions to use. It queries the integration status database to determine the latest *fresh* revisions for the domains for which it is responsible, and the latest *verified* revisions it also needs to support its compilation steps.

Validation marking Successful builds update the integration status database, promoting file revisions from *fresh* to *verified* status.

These processes are supported by the following artefacts, comprising a configuration file, a second Perforce repository, and a script that implements both the server-side triggers, and client-side functionality:

A policy configuration file A configuration file specifies the zone definitions for a codeline. It is checked in at the top directory of the codeline, so that the zone definitions can naturally be carried over into any release branches made from the codeline.

Sister repository The integration status database for a particular codeline is just a flat file under revision control in a separate "sister" Perforce repository.

Site groups We use Perforce user groups to define which users are developing at which sites. For example, all users in Toronto are listed in the `p4sip.to.users` group.

Form-out trigger The form-out trigger is used to suggest a site assignment for a new change, by inserting a site tag string into the beginning of the change description. For example, a user in Toronto will be prompted with "`[TO] <enter description here>`" for the change description. They have the option to update it to another site tag if they want to temporarily masquerade as being from another site. Also, they can force a change to be a COMBO change by changing the site string to be "`[COMBO]`".

Form-in trigger The form-in trigger is used to validate the site string (or COMBO), and to ensure that the user really has provided a description other than "`<enter description here>`".

Change-submit trigger A submit trigger assigns a new change to a domain, and then consults the sister repository for a conflict to determine whether to reject the change, or to let it pass through. This trigger is applied on a codeline by codeline basis.

Change-commit trigger A commit trigger executes after the point-of-no-return for the change submission. It records the new file revisions as *fresh* in the sister repository. Like the change-submit trigger, this is configured on a codeline by codeline basis.

Code selection client script This client side script is used to create a label containing the set of file revisions to be used by a build.

Validationg marking client script This is run at the end of every successful build. The success of the build proves that the set of file revisions used by the build are "good", and so all *fresh* revisions listed in the build label are promoted to the *verified* status.

We also apply the rules from Section 6. For example, suppose a file has *fresh* revisions `foo.c#100`, `foo.c#101`, and `foo.c#102`, and a successful build used `foo.c#101`. Then we mark both revision `foo.c#100` and `foo.c#101` as *verified*, and revision `foo.c#102` remains in the *fresh* state.

This script follows the same sync/edit/update discipline used by the commit trigger, so it is not described further.

The system requires no client customization, and so it works with any Peforce client side software, e.g. `p4` or P4V. Regular users never interact directly with the mechanics of the system.

The key components are the codeline configuration file, the domain assignment algorithm, the contents of the sister repository, and the build code selection algorithm. These will be described in the following sections. The other pieces are relatively straightforward once you know the details of the key parts of the system.

7.1 Runtime configuration via the zone file

We store a policy configuration file, named `p4sip.zone`, in the top directory of the codeline.[2] On every execution, the triggers and the client side script read the configuration file via `p4 print`.

It is advantageous to include the policy configuration the codeline itself, because it travels along with any official branches made for release stabilization. (You could also apply the policy to private development branches, but they are lower on the tofu scale [6], and so usually don't require the extra level of protection from integration breaks.) In order for this to work, the data *inside* the file should only describe files and paths relative to the codeline root, and not mention the codeline root itself.

The main job of the configuration file is to define named zones as sets of paths underneath the codeline. This is just like a Perforce label view, except that we only allow positive views, we only specify paths relative to the codeline root, and the only pattern permitted is the triple-dot at the of the path spec. (In fact, the triple dot is mandatory.)

For example, here is a configuration file fragment which defines four named zones, `quartus`, `sopc`, `ip`, and `therest`:

```
files    quartus  quartus/...
files    sopc     ip/sopc/...
files    sopc     ip/infrastructure/...
files    ip       ip/...
files    therest  ...
```

To assign a given file depot path to a zone, scan the `files` definitions list from top to bottom, and return the zone for the first definition that matches.

The configuration file is completed by using weights to say which zones are code zones, to define which zones are site-specific, and to list the names of the sites themselves. This fragment continues the example:

```
weight   quartus  1     # A ‘‘code zone’’
weight   sopc     1
weight   ip       1
weight   therest  0     # Not a ‘‘code zone’’
```

[2]The `p4sip` name is historical, meaning "Perforce site isolation policy", for an early incarnation of the mechanism.

```
exclusion quartus site  # The quartus zone is site-specific.
sites    SJ,TO,UK,PN
```

7.2 Domain assignment

Both the change-submit and change-commit triggers need to associate a changelist with a domain. First, we use `p4 describe -s` to get the change description and the list of depot paths of affected files.

The site string in the change description has been inserted by the form-out trigger, possibly updated by the user, and finally validated by the form-in trigger. It will be one of the valid sites, or it will be COMBO. If it is COMBO, then the change is in the COMBO domain, and we stop. Otherwise we have the originating site.

We now look at the list of files in the change. We use the matching algorithm described earlier to map each file into a *zone*. After matching all the files, we look at resulting zones as a set.

If more than one code zone is covered by the change, then we assign the change to the COMBO domain, and stop.

If zero code zones are covered, then no domain is assigned. Such a change does not interact with the system at all: we don't reject it, and we don't record it. This is common for changes that only affect test files that are not used by the integration build process.

If exactly one code zone is covered, the resulting domain depends on whether the zone is site-specific. If the zone is site-specific, then the assigned domain is the combination of the zone and site; otherwise it is the domain with same name as the zone itself.

(Examples?)

7.3 Integration status held in a control file in a sister repository

The integration status must be valid at all times, so it must be maintained by submit and commit triggers. The easy answer would be to use a floating label to remember the last *verified* revision for each file in a code zone. But Perforce triggers can't update Perforce metadata!

One solution is to use an external database of some kind. But we want a simple client and server setup. The client side is used by the build process to

generate the initial build label, and to mark file revisions as verified. So it's highly advantageous to avoid third party database software, and to require only the Perforce client software.

Altera's solution is to use a "sister" Perforce repository alongside the main repository. This simplifies the client and server deployment: you already have Perforce on the host. There is no need for SQL, etc. It provides concurrent access, atomic updates, low latency, compact data representation, and reuses server side administration and backup infrastructure.

All integration status for a single codeline is held in one "control file" Submit/commit triggers on the main server use edit/update/submit on the control file in a retry loop with backoff.

Updates to integration status are done a concurrent environment. If two submissions cover different sets of files, the server may interleave the execution of their submit/commit triggers. Over and above the edit/update/submit concurrency control, the submit and commit triggers must enforce their own serialization, e.g. by locking a server-side file specific to the codeline. We learned that the hard way!

7.4 Control file contents

The control file encodes the integration status for each file revision in the codeline. For simplicity, it is a text file with a sequence of lines, with one line per file revision. Each line contains:

- Change number
- Integration status, either *fresh* or *verified*
- Submitter's user id
- Originating site
- Zone
- Full depot path, including revision number

We always record the originating site, regardless of whether the zone is site-specific: the domain is always constructed on the fly, using the policy configuration to say whether a zone is site-specific. The COMBO domain is recorded in the zone field.

The file is log-structured: the lines are in chronological order, with oldest revisions coming first.

We use two strategies to avoid a data explosion.

First, we use a purging algorithm based on the observation only very recent revisions will be *fresh*. All earlier changes are *verified*, by definition. You can eliminate those earlier changes from the control file. For example, if entry is for a revision in change 1000000, then all file revisions for changes 999999 and below are known to be in *verified* state. In Altera's environment, purging keeps the control file to at most a few thousand lines long.

Second, the control file is stored with file type `text+S512`. This tells the server to use reverse delta storage for the 512 most recent revisions, and discard the contents for anything older. (Depending on development velocity, 512 revisions is a few days worth of changes.) This keeps the underlying RCS file small. (The system only ever uses the head revision, but occasionally we have found it useful to manually examine recent integration history.)

7.5 Code selection for builds

Each build, whether gatekeeper or product-wide integration, begins by creating a label containing the set of file revisions it will use.

If a product-wide integration build succeeds then it becomes a stable baseline, i.e. developers use it as the basis for further incremental development, and later gatekeepers can use it as the basis for a later build. In these cases we keep the build label for those other uses.

If the build is manually monitored and repaired, then the build label is updated accordingly so it always reflects the source files used. In particular, you may only repair a build using file revisions that are committed to the repository. You may cherry pick specific files from a committed change. This is why integration state is a property of a file revision rather of than a whole change: you might pull in only part of an already-committed change.

In repairing a build, you may either roll the build label *forward* (to include revisions that are more recent than those originally selected for the build) or you may roll it *backward*. In the backward case you are typically pushing a bad recent change out of the current build, deferring its inclusion until the next edition of that build. This gives the developer extra time to develop and submit a proper fix. We call this technique a "soft rollback" because we avoid having to do a "hard rollback" to repair the build. (A "hard rollback" commits a new change to the repository to invert the effect

of the bad change.) Soft rollbacks are handy because they save people from having to remember how to perform hard rollbacks, and they result in much cleaner file histories. Soft rollbacks can be used whenever the set of source file revisions used for a build is defined by a label, as opposed to a timestamp.

To create a build label we need to know what code zones will be required to support compilation. We always need at least the *verified* revisions from the corresponing domains. We also need to know what domains, if any from which we should accept *fresh* changes. Any time we take a fresh revision from a domain, we should always also take all *verified* revisions from that domain. Also, any time we take *verified* revisions from a domain associated with a site-specific zone, then we take revisions from all (site-specific) domains associated with that zone.

The build label is constructed from the codeline configuration, the list of *verified* and *fresh* domains, and from the head revision of the control file.

The label construction algorithm for the product-wide integration build is the easier case, and is as follows:

1. Empty the label.
2. Read the codeline configuration, to get the list of zones and their views.
3. Read the head revision of the control file for the codeline, and cache its contents.
4. Note the change number of the first revision in the file. It is the "implied verification" mark, i.e. all changes older than it consist only of *verified* revisions.
5. For each selected domain *A* or *A:SITE*, do the following:
   (a) Change the label's view to match only the set of files in zone *A*. This is done by creating one mapping line for each `files` definition line in the codeline configuration file, but listing them in reverse order. Use a positive mapping for trees associated with *A*, and a negative mapping (with a minus sign) for trees associated with other zones. (If *A* is COMBO, then the view positively maps all the code zones.)
   (b) Add all file revisions older than the implied verification mark. This gets us the old *verified* revisions in zone *A*.

(c) Scan the cached contents of the control file. Add any revision in the selected domain which is marked as *verified*. If you also want *fresh* revisions from this domain, then add those as well. If more than one revision matches these criteria, take the later one. (It is convenient to scan the control file contents from top to bottom.)

6. Finally, change the label view to include the entire codeline.

The label construction algorithm for a generic gatekeeper is similar. The build process knows what code zones contain the source files required for a gatekeeper build, and so the gatekeeper's label will cover only those zones. The issue is what revisions to use. Because you won't build the entire product, you have to rebuild on top of a developer image of a previous stable base build. So for a given file in a requested domain, the gatekeeper will use the last of: the revision in the stable base's build label, the last *verified* revision for the associated zone, or the last *fresh* revision. So we modify the algorithm as follows:

- Insert a step after 5a to add revisions from the label for the stable base.
- In steps 5b and 5c, you must be careful to only add a revision if it is later than the revision already in the label you are constructing.

In both cases you must be careful avoid overwhelming the server, either in running time or journal file traffic. This takes some thought when you have 255K files in your build, and you have to optimize trans-global network traffic! We have tuned the algorithm so it runs in about 90s for the ACDS product-wide integration build. The key tuning ideas are:

- Cache the set of *verified* files older than the implied verification mark.
- Use a three step "add newer revisions from `X` to label `Y`" algorithm:

    1. Store the contents of `@>X,@Y` in a temporary label. This will only include revisions in `Y` that are newer than in `X`. In particular it only includes revisions for files that are in *both* `X` and `Y` (like an inner join in SQL).
    2. Tag revisions from `@X` into `Y`. This will catch cases when `X` has files that `Y` does not.
    3. Finally, tag revisions from the temporary label into `Y`.

8 Maintaining other metadata

The control file can also be used to store other metadata with a lifetime similar to the recent integration status of a file revision. The updates for other metadata can occur either in the change-commit trigger, or from an external client program that uses the same edit/update/submit algorithm.

Examples include:

- Integrate with an external defect tracking system. During release stabilization, we can require a defect tracking number in the change description, and record that number in the control file.

- Track "rebuilt" info. For example: What gatekeeper marked this change as *verified*? What product-wide integration build was the first to fully include this change? Given the possible obstacles to getting a change into the build, this is often valuable information. The "rebuilt" details can also feed back into a defect tracking system.

  When gatekeepers and product-wide integration builds succeed, they add their "rebuilt" info to the affected changes in the control file. A separate daemon polls for these updates, and copies the "rebuilt" info into the descriptions of the changes themselves. That way users can see the rebuilt information directly in P4V or via `p4 describe`.

9 Toward nested transactions

Compartmentalized Continuous Integration presents a set-of-changes abstraction over and above Perforce itself. (That is, a *set* of multi-file atomic changes.)

Changes are propagated to the integration build in bunches, i.e. after they have been validated by gatekeepers. From the perspective of the integration build, changes appear in bursts, and are grouped by domain. When zones are site-specific, you also get the effect that one site will see bursts of new changes coming from other sites.

The reason is the gatekeeper mechanism collects and holds back fresh changes in its domain until they are *all* good and consistent *as a group*, and only then does it pass them for use in other builds.

Furthermore, the exclusion rule acts at the set-of-changes level in a way which is analogous to how Perforce's resolution mechanism acts on a single-

changelist level. That is, if someone else has slipped in a fresh change to a file you are editing, but to a different domain, then you have to do some work to resolve to that other change before you can submit your change. The exclusion rule is conservative in that it *assumes* you must move to a new stable base incorporating that earlier conflicting change. So it prevents you from submitting your change until that new stable base is available. (More precisely, it prevents you from submitting until the gatekeeper for that other domain passes. See the example from Section 6.3.1.)

But just like Perforce itself, the exclusion rule and the triggers that enforce it are an optimistic concurrency control mechanism. That is, coordination takes place *late* in the submission cycle, and only when an apparent conflict exists. Otherwise, anybody can modify anything anywhere. Generally this works well because there is a natural breakdown of work between people, groups, and sites. But the system is flexible enough to support ad hoc changes of any shape.

It may be possible to extend the system to create a completely general ad hoc nested change transaction system. But we are not sure whether whether it would be worth the effort, because any corresponding exclusion rule would cancel out any benefits.

10 Real world experiences

Overall integration success Every day, Altera peforms product-wide integration builds at three of our development sites:

- The integration build in Toronto takes *verified* revisions from all domains (of course), and accepts *fresh* revisions only from the COMBO domain and the site-specific `quartus:TO` domain. This gives us a good balance between safety and short turnaround for local Quartus changes. (COMBO changes are rare.) Typically with very little work, a good integration build is almost always available the next day.
- The integration build in Penang is similar to Toronto, except that it takes *fresh* revisions from COMBO and `quartus:PN`.
- In San Jose we take advantage of the flexibility of the code selection algorithm: The San Jose build is usually configured to get the newest Quartus code from both Toronto and San Jose,

by selecting *fresh* domains COMBO, `quartus:SJ`, `quartus:TO`, and `quartus:UK`. Any added risk in accepting unverified Quartus code from Toronto is typically offset by the fact that San Jose runs three hours behind Toronto. So any repairs required to the Toronto build can be discovered in time to be also applied in San Jose. (The UK site makes very few changes to Quartus.)

Altera runs gatekeeper builds for the smaller components every hour or three hours, depending on typical development velocity for those domains. We never repair a broken gatekeeper. Instead, the system notifies relevant users of the break, and then waits for the next scheduled gatekeeper to run. Overwhelmingly, a change checked in to one of those smaller components is available for use in that night's integration build, anywhere around the world.

In any case, we can adapt to changing conditions by reconfiguring what gatekeepers are run at what frequency, what domains those builds are responsible for, and what fresh domains are accepted in any particular integration build. For example, during the last stages of release stabilization, development has settled down enough that release candidate integration builds accept *fresh* revisions from *all* domains.

Feature velocity If builds succeed, and they usually do, then a submitted change is usually delivered the next day to everyone at the submitting site as part of the product-wide integration build, despite the 14 hour build time. If the submission is to a non-site-specific zone, the update is usually available worldwide within 24 to 36 hours, depending on timezone differences and how submission time correlates with the local build start time. If the submission is to a site-specific zone, then the update is usually available worldwide within 24 to 48 hours, depending again on timezone differences.

Performance In three years the system has processed over 175 000 changes with no noticeable effect in performance. The sister repository is tiny, using only a few megabytes per codeline.

Usability Overall, most people work within one zone, and most code is owned by people in one development site. So most people never need to know the details of the system: they are not slowed down by the ex-

clusion rule because a file may have multiple outstanding *fresh* revisions from the same domain.

But the system is flexible enough to dynamically adapt as projects move around the world, even for code in site-specific zones. Sometimes there is a ping-pong effect as an update at one site has to wait for the passage of a build at a different site.

The main usability issue is one of "temporal build mechanics". That is, the gating action and build breaks raise the following questions in users minds:

- When will my change go in?
- When will that person's change get to me?
- When *did* my change go in? This is the "rebuilt" information alluded to earlier.
- Relativity effects: Two changes often appear in different orders in different builds. This is another reason to post "rebuilt" info change descriptions.

Operational / Semantic issues Unfortunately, sometimes users *do* need to know the details of the system, for example to resolve the following issues:

- Why can't I submit my change? This is related to the operation of the exclusion rule. It's important for the change-submit trigger to issue a good error message. When a change is rejected, the message should should say why, listing what file revisions in previous changes block the current change, and who is responsible for them. Even better, the message should tell the user what their next steps should be, including possible workarounds such as breaking up their change to avoid conflicts.
- When the exclusion rule rejects a change, the files in the change are automatically locked, just like with any validation rejection from a trigger. Sometimes this gets in the way, but users can manually unlock them, e.g. with `p4 unlock`.
- Will I break the build if I submit a change dependent on that other person's change? The answer is as follows (but most people's eyes glaze over when you tell them!):

- If the proposed change would end up in the same domain, then answer is almost certainly "No": it is safe to submit the proposed change.
- The proposed change could be in a the same zone but a different domain because the zone is site-specific and the two changes originate in different sites. If the earlier change is still *fresh* then the proposed change would be blocked anyway. But if the earlier change has just been *verified* then it's usually ok to submit the proposed change: the next gatekeeper will use both changes together.
- The proposed change could cover files from a different zone. In this case the file sets are disjoint, and the gatekeepers are likely disjoint. So it's purely a logical dependence, and you'll likely break a gatekeeper that only builds a portion of the product wide build. So you either accept that or just wait until the next product-wide stable base has been released. Then the *subsequent* gatekeeper will have that earlier change, and your new change can safely depend on the earlier change.

Tolerance for broken gatekeepers How long can you afford to let a gatekeeper remain broken? When do you start getting concerned? It depends on how tightly that component is coupled with the rest of the product. Ideally a gatekeeper for a domain should pass at least daily. For some components, three days without a passing gatekeeper is considered a major problem; for others a week can be absorbed. In any case the product-wide integration build should always succeed, but may use somewhat old file revisions for some domains.

11 Conclusion

Altera has used Perforce-based Compartmentalized Continuous Integration since it adopted Perforce for all software development. With the policy, we achieve our goals of supporting flexible collaboration, rapid dissemination of good changes, and a high success rate for producing integration builds.

Is it right for you? That depends on the structure of your code, the complexity of your build, the development velocity of your project, and how quickly you need to push new features out to your entire team. It does incur

some conceptual and infrastructural costs, but these are well worth it for Altera's development environment.

Acknowledgements

This work is a team effort. Many people have contributed in various ways, including the developers and builders who live with the system every day. But we want to specifically thank the following people for their ideas, requirements, feedback, encouragement, exhortations, and elbow grease: Rob Romano, Jeff DaSilva, Mun Soon Yap, Addy Yeow, David Karchmer, and Alan Herrmann.

References

[1] Continuous integration. `http://en.wikipedia.org/wiki/Continuous_integration`, 2011.

[2] Brad Appleton, Stephen P. Berczuk, Ralph Cabrera, and Robert Orenstein. Streamed lines: Branching patterns for parallel software development. `http://www.cmcrossroads.com/bradapp/acme/branching/branch-creation.html`, 1998.

[3] Martin Fowler. Continuous integration. `http://martinfowler.com/articles/continuousIntegration.html`, 2006.

[4] Perforce Inc. Software life-cycle modeling. `http://www.perforce.com/perforce/papers/life.html`.

[5] Vincent Maraia. *The Build Master: Microsoft's Software Configuration Management Best Practices*. Addison Wesley, 2006.

[6] Laura Wingerd. *Practical Perforce*. O'Reilly Media, 2005.