Backtracking Incremental Continuous Integration

Tijs van der Storm
Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB
Amsterdam, The Netherlands
storm@cwi.nl

Abstract

Failing integration builds are show stoppers. Development activity is stalled because developers have to wait with integrating new changes until the problem is fixed and a successful build has been run. We show how backtracking can be used to mitigate the impact of build failures in the context of component-based software development. This way, even in the face of failure, development may continue and a working version is always available.

Index terms: software configuration management, build management, software maintenance.

1. Introduction

Continuous integration [8] has been heralded as a best practice of software development. After every change to the sources the complete system is built from scratch and the tests are run. If any of the tests fail, all effort is directed at fixing the problem in order to obtain a working version of the system. If the build fails, development is stalled. Continuous integration has therefore been called the "heartbeat of software". If it stops, you can't ship.

In this paper, I describe a continuous integration scheme in component-based development settings. In this scheme I assume that integration is defined as building the source of a component against the (build artifacts of) its dependencies. Integrating the whole application then means building the topmost component in the dependency hierarchy.

The scheme employs two features to improve the feedback obtained from it. First, instead of building the complete system on every change, only the components that have affecting changes are rebuilt, and previous build results are reused otherwise [19]. Components are integrated in an incremental fashion, similar to the way the Unix tool MAKE can be used to selectively recompile files [7]. It turns out that due to the amount of build sharing, the feedback is much quicker on average. As a result developers are can respond more quickly to problems encountered during integration.

The second feature, the primary focus of this paper, is backtracking. If the build of a component has failed, it would make no sense to build any client components. Normally this would stall integration until the problem is fixed and the breaking component has been rebuilt. To prevent this from occurring, components that normally would depend on a broken component build, are built using *earlier* build results of the very same component. This way some measure of being completely up-to-date is traded for increased build feedback. In the end, *any* build is better than no build at all.

Contributions The contributions of this paper can be summarized as follows:

1. I present a formalization of incremental continuous integration in the context of component-based development.
2. The formalization of incremental continuous integration is extended with two forms of backtracking, dubbed "simple backtracking" and "true backtracking"; both approaches are compared and I present and efficient algorithm for the latter.
3. Simple backtracking has been validated in practice; this has resulted in empirical data supporting its viability to improve continous integration.

Both build sharing and backtracking have been implemented as part of the continous integration and release system Sisyphus [20]. Sisyphus was used to validate the scheme in the setting of the ASF+SDF Meta-Environment [17], which is a language engineering workbench consisting of around 60 heterogeneous components. The results in this paper derive from that case study.

2. Background

2.1. Component-Based Development

In component-based software configuration management (SCM) the sources of a software system are divided over in-

dividual components in the version control system (VCS). That is, the system is composed of different source trees that have independent evolution histories. The prime example of this approach to SCM is the Unified Change Management as implemented in IBM Rational's ClearCase [3].

Independent versioning of components promotes parallelism in development activity. Development on a component is more or less isolated from the rest of the system. Having a good architecture thus creates opportunities for reduced time to market. At the same time the traditional advantages of component-based software development apply: complexity is reduced, reuse and variation is stimulated.

Whereas most component models (e.g., COM [15]) separate the notions of interface and implementation, in this paper I assume a more liberal notion: a component is considered to be just a logically coupled set of source files that has its own version history. Practically this means that a component is often represented as a directory entry in a VCS such as, for example, Subversion [5].

Components are often inter-related through dependency relations. Such dependencies are specified without version identifier because that would introduce strong coupling between the client and the dependency component. As soon as the latter changes, the former is out of date. Keeping such dependency relations synchronized can be a true maintenance nightmare. We therefore let components reference their dependencies by name without version information.

However, it now becomes increasingly difficult to select configurations of components that make up consistent versions of a system. Any version of a component is a suitable candidate to satisfy the requirements of the client component that declares a dependency on its name. The configuration space has become very large since we have *complete* decoupling between the evolution histories of components.

This is where continuous integration comes in. Instead of explicitly searching the configuration space for the "right" configuration, we let an automated build system construct "bleeding edge" configurations as frequent and quick as possible. This means that always a working version is available without additional maintenance of selecting the right versions of the right components and doing the integration by hand.

2.2. Continuous Integration

Continuous integration proper originates from the Extreme Programming (XP) software development methodology [2]. There, the process of continuous integration also includes the continuous checking in of changes, however small they may be. Current usage of the term, however, most often refers to the process of building the complete system every time changes have been committed, whichever the frequency they occur in. As such it can be seen as a heavier instance of daily or nightly integration builds [13].

The goal of continuous integration is to know the *global* effects of *local* changes *as soon as possible*. Integration bugs are hard to track down because they originate from the interaction between (changes to) different subsystems, so they are hard to test for on a subsystem level. Post-poning integration makes things even worse: the interaction between changes increases very fast making integration bugs exponentially harder to find. It is therefore important that integration builds are executed quickly enough. As Martin Fowler states: "The whole point of Continuous Integration is to provide rapid feedback." [8] Failing builds, of course, are the main impediment to such rapid feedback if they are not fixed timely.

In component-based software development the sources of a product are partitioned in independently versioned components. This means that "change integration" (check in) can be highly parallelized since every component has its own development line. This increased parallelism poses even higher demands on continuous integration. Furthermore, integration builds not only test the new changes, but also the *composition* of the different components (possibly containing those changes) to obtain a complete system.

2.3. Motivation: Continuous Release

The goal of backtracking continuous integration is to mitigate the effect of build failures in order to have a working version at all times and at the same time increase feedback for developers. Always having a working version of the system is a key requirement for continuous *release*, which entails making the software available after every change.

Releasing in this context means making the software available to a certain group of users. For instance, it might not be desirable to continuously release to actual end-users. However, it may be very beneficial to release the software after every change to beta-testers or to the developers themselves (who want to see the effect of their changes).

Continuous release is motivated along the same line as the reason for continuous integration: early discovery of defects and a shortened feedback loop. In order to extend the process of continuous integration to continuous release, I distinguish the following goals:

- Feedback: *a* build (successful or not) is always better than no build. If there are changes to a component, the system should find a way to integrate them, even if builds of certain dependencies may have failed.
- Currency: the continuous integration system should always attempt to build the latest version of the software. Builds should be maximally up-to-date.
- Traceability: accurate *bills of materials* (BOMs) [12] should be maintained for the sake of tracing releases to the sources that were used to build them.

- Purity: the integration of components should be *pure*, i.e. the set of builds that transitively participate in an integration build should not involve multiple versions of the same component; this is a requirement for the derivation of release packages.
- Efficiency: the continuous integration system should perform no duplicate work. This means that previous build results should be reused if possible.

Not all of these goals can be achieved at once. For instance, we will see that there is a trade-off between maximal up-to-dateness and maximal feedback.

3. Overview

3.1. Introduction

Before I a describe incremental continuous integration and the two kinds of backtracking, I first introduce some preliminary assumptions. First of all, it is assumed that the dependencies of a component can be derived from the sources, for instance by analyzing a specific file that lists them explicitly. Since the specification of dependencies thus is part of the sources of a component, a change in the dependencies induces a change of the component. This allows for smooth evolution of a system's architecture.

In the context of our case-study the dependencies are specified in `pkgconfig` files [9]. For instance, the pretty-print subsystem of the ASF+SDF Meta-Environment [17] corresponds to the `pandora` component. Its `pkgconfig` file is shown below (slightly abridged for clarity):

```
Name: pandora
Requires: asc-support,aterm, \
  pt-support,toolbuslib
```

The first line in the file declares the identity of this component, in this case `pandora`. The second line lists the required dependencies. Note that these dependencies do not have version identifiers attached. Requiring this would surely introduce a large maintenance penalty: on every change to one of the dependencies this files would have to be updated. Instead, at a certain moment in time the continuous integration system will take a snapshot of the repository and bind each unversioned dependency to the versions of those required components at that very moment in time. In other words, for all components the *latest* revision is taken.

Taking a snapshot of the repository results in a set of *source trees* which capture the state of every component at the moment of the snapshot. These source trees are related in a dependency graph that results from the requirements as specified within those source trees at the moment of the snapshot. The snapshot is now input to the continuous integration process.

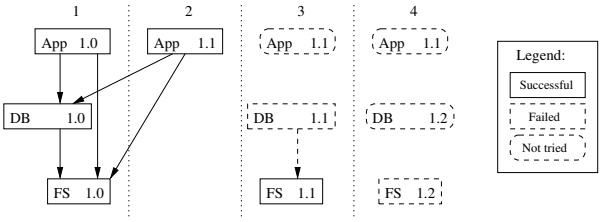

Figure 1. Incremental integration example

3.2. Build Sharing

Now it is time to describe incremental continuous integration based on the snapshots introduced above. As an example, consider a small component-based application consisting of three components: App (the application) , DB (a database server) and FS (a file system library). The App component requires both DB and FS, whereas DB only requires FS; FS has no dependencies whatsoever. Of course, every build of a component has an implicit dependency on the build environment (e.g. compilers, build tools etc.). This dependency however, we assume, is managed by the continuous integration system itself.

Figure 1 shows four integration cycles, corresponding to each column. In the first iteration, all components have been successfully built (indicated by solid boxes and arrows). The arrows indicate the dependency relation in the snapshot at the time of integration.

In the second iteration, the App component has changed since the first iteration, but there are no changes in DB and FS. Instead of building all components from scratch—which would mean a waste of valuable resources—the incremental continuous integration system reuses the build results (e.g., binaries, libraries etc.) from earlier integrations for the dependencies of App. This is indicated by the arrows going from App 1.1 to DB 1.0 and FS 1.0. In other words, the builds of DB 1.0 and FS 1.0 are shared between the consecutive builds App, versions 1.0 and 1.1 respectively.

However, suppose that a build fails. This is shown in integration 3. Changes have been committed to both FS and DB, so all components require a build. In the case of App a *re*build is required of version 1.1 in order to take the changes of DB and FS into account. So it is very well possible that a single component source tree will be built many times because of changes in dependencies.

Suppose that the build of DB 1.1 fails, however. This has been indicated by the dashed box around DB 1.1. This state of affairs prohibits a build of App 1.1 because one cannot build against failed dependencies. Builds that will not be attempted because of this reason are called "not tried". In the figure this is indicated by dashed boxes with rounded corners. Clearly, "not tried" builds are to be avoided since we lose feedback. In the example no feedback is generated, for

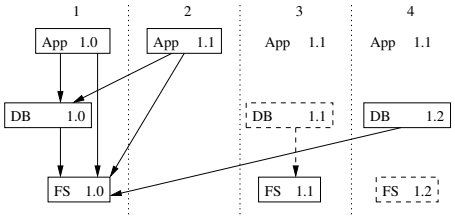

Figure 2. Incremental integration with simple backtracking

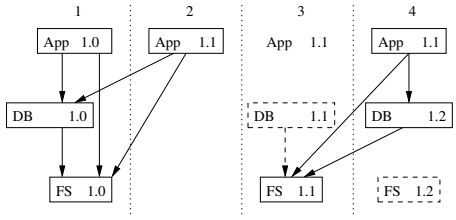

Figure 3. Incremental integration with true backtracking

instance, on how the changes in FS affect App 1.1.

Finally, in the fourth cycle (column 4), again there are changes in DB (hopefully to fix the previous build) and in FS. However, now the build of FS 1.2 fails. As a consequence there is neither feedback on the changes in DB itself nor on the integration of changes in DB 1.2 with App 1.1. Again, feedback is less than optimal and, moreover, we still can only release App 1.1 with DB 1.0 and FS 1.0, and we can release FS 1.1 as a stand-alone component. I will now describe how a simple form of backtracking improves this situation slightly.

3.3. Simple Backtracking

In incremental continuous integration builds to satisfy component dependencies are always searched for within the current snapshot. For instance, in Figure 1, during the second integration the continuous integration system find builds for DB 1.0 and FS 1.0 to satisfy the dependencies of App 1.1, since both DB and FS have not changed since; both DB 1.0 and FS 1.0 are in the snapshot of cycle 2. In the next two cycles the current snapshot contains DB 1.1 and FS 1.1, and DB 1.2 and FS 1.2 respectively. However, in cycle 3 the build of DB 1.1 has failed, and in cycle 4, the build of FS 1.2 has failed. Hence it is not possible to build App 1.1 in either of the two cycles.

Figure 2 shows the application of simple backtracking. This means that, if there is a failed build in any of the dependencies of a component, say App, in the current cycle (with source trees in the current snapshot), the continuous integration goes back in time to find the first successful build of the component in question (in this case App), checks if the requirements are still the same—does App still require both DB and FS?—and if so, uses the set of builds that were used back then.

To illustrate this, consider Figure 2. In cycle 3, there are failed dependencies for App 1.1. The most recent successful build of App with the same requirements, is the build of cycle 2. However, using that set of dependencies (DB 1.0 and FS 1.0) does not achieve anything: we would be merely *duplicating* the build of cycle 2 because App has not changed in between cycles 2 and 3. This is indicated by the absence of a box around App 1.1 in cycle 3. Note that this is a different outcome than "not tried", since with "not tried" builds we always lose something, either feedback or currency, and this is not the case here.

Another important detail here is that we cannot just use DB 1.0 and FS 1.1 for building App 1.1 in cycle 3, since that would lead to an impure build: DB 1.0 uses FS 1.0 whereas App 1.1 would have used FS 1.1. This means there are two versions (1.0 and 1.1) of the same component (FS) in the closure of the build graph of App 1.1.

Simple backtracking shows its value in the fourth cycle: there is a change in DB, and there is a successful most recent build, the build of DB 1.0 in the first cycle. Using simple backtracking, at least DB 1.2 can be built. We do not get feedback on the integration of DB 1.2 and FS 1.1 but it is better than nothing at all. Although in this case, it seems trivial to just use the build of FS 1.1 for building DB 1.2, this is deceiving. When the dependency graph is more complex one cannot just take the most recent successful builds of dependencies without ensuring the result will be pure. This is exactly what true backtracking achieves, which I will discuss next.

3.4. True Backtracking

Simple backtracking involves searching for earlier successful builds of the component that should be built now. True backtracking adapts this search by search for sets of successfully built required components such that a purity of integration is ensured. Figure 3 shows the example scenario with true backtracking enabled.

The figure only differs from Figure 2 in the fourth cycle. Cycle 3 remains the same because using the most recent set of successful dependency builds that maintain purity (DB 1.0 and FS 1.0) again would entail duplicating the App build of cycle 2. It is still impossible to use DB 1.0 and FS 1.1 because it would violate purity.

In the cycle 4 however, the new version DB (1.2) can now be built. The set of most recent successful dependency builds is $\{$FS 1.1$\}$ and this set does not violate purity from the perspective of DB 1.2. Furthermore, App 1.1 can now also be built: $\{$DB 1.2, FS 1.1$\}$ maintains purity.

Note that in both Figure 2 and Figure 3 all "not tried" have disappeared. However, true backtracking presents the following advantages over simple backtracking:

- We were able to build DB 1.2 against FS 1.1 instead of FS 1.0, hence with true backtracking the build of DB 1.2 is more on the bleeding edge.
- It was possible to build App 1.1 against DB 1.2 and FS 1.1, hence we obtain one additional release opportunity for component App.

In addition, the simple backtracking suffers from the fact that it only works if dependencies have not changed in between integrations. In that case, the probability of finding an earlier build with the same set of requirements is rather low.

In the following section I will present a light-weight formalization of incremental continuous integration and the two forms of backtracking. The formalization is based on the formalization presented in [19], slightly reformulated for presentation purposes.

4. Formalization

4.1. Preliminaries

In order to reason about integration I introduce a lightweight formal model of components, revisions and builds in this section. It is instructive to see the relations and sets that make up the model as a *software knowledge base* (SKB) that can be queried and updated. The SKB is required for implementing build sharing, backtracking and to automatically derive pure releases.

To be able to build a component, a source tree is needed for every component. To reflect this relation explicitly I introduce the relation *state* that bijectively maps components (names) to source trees (revisions) according to some criterion (for instance, by taking the *current* revision of each component). It has the following type: $State \subseteq Components \times Revisions$.

In practice, a revision $r \in Revisions$ is often represented as a tuple of a source location (e.g. a path or URL) together with a version identifier.

Source trees may declare dependencies on components. This is modeled by the relation *Requires*: $Requires \subseteq Revisions \times Components$. Note that the domain and range of this relation are not the same. The idea is that dependencies on components may change inbetween revisions.

For instance, source tree T may require the components A and B, but the following revision T' might only require A. Moreover, the range of the relation is unversioned for another reason. If *Requires* would have been a relation from *Revisions* to *Revisions* this would mean that it would have to be updated on every change to a component that is depended on by T. In the current situation, the requires relation can be maintained *within* the source trees themselves.

A snapshot maps every path to a tree according to some criterion, that is, it fixates a certain version for each component. The definition makes use of the *State* relation introduced above: $Snapshot = Requires \circ State$. This relation represents a dependency graph between *Revisions* and is the starting point for integration. This means that subsets of carrier(*Snapshot*) will be built in an appropriate order. The results are stored in a relation $Builds \subseteq State \times \mathbb{N}$.

This *Builds* set is partitioned in two subsets *Success* and *Failure*, resp. containing the successful builds and the failed ones. A build relates a tree to a build number, because a single tree can be built many times, possibly using different dependencies. Which builds of dependencies actually were used is recorded in the *Integration* relation: $Integration \subseteq Builds \times Builds$. Again, *Integration* is a dependency graph but this time between builds.

For any successful integration we require that the set of builds that participated in the integration is *pure*. Purity of a set of builds B is defined as: $\text{pure?}(B) \equiv |B| = |\text{domain}(\text{domain}(B))|$, with $\text{domain}(R) = \{x \mid \langle x, y\rangle \in R\}$ Note that a set B contains tuples of the form $\langle\langle c, r\rangle, i\rangle$, so that $\text{domain}(B) \subseteq State$, and hence $\text{domain}(\text{domain}(B)) \subseteq Components$. In other words, a set of builds B is pure if there are no two builds and/or revisions for the same component contained in B.

The algorithms presented here ensure that *Integration* is pure for every build. Formally this means $\forall b \in Builds : \text{pure?}(Integration^*[b])$. This invariant was introduced in [19]. It ensures that the *Integration* relation can be used to derive *compositions* for every build which is what is delivered to users. If *Integration* would not have been pure composition would be ambiguous: it could occur that two builds used different versions for the same dependency,—which one should be in the composition? This invariant is used in Subsection 4.4 where backtracking is added to integration.

4.2. Schematic Integration Algorithm

Now that I have introduced the preliminary definitions, I present a schematic version of an algorithm for continuous integration in component-based development setting; it is shown in pseudo-code in Algorithm 1. The input to the build algorithm is the *Snapshot*, i.e. a relation between *Revisions* and a number that identifies the build cycle. Since snapshots are directed, acyclic graphs (DAGs) they have a topological order. The topological order consist of a list of vertices in the DAG, such that every dependency of vertex N comes *before* N. The topological order of the snapshot is stored in variable *order* on line 2.

Then, for each revision/source tree t in *order* (line 3) we obtain a *workingset* for t (line 4). Workingsets consist of

Algorithm 1 Template for component-based integration

```
1:  procedure INTEGRATE(i, Snapshot)
2:      order ← topological-sort(Snapshot)
3:      for t ∈ order do
4:          w ← workingset(t)
5:          if w is undefined then                ▷ "Not tried"
6:              Failed ← Failed ∪ {⟨t, i⟩}
7:              continue
8:          if build?(t, w) then
9:              b ← execute-build(i, t, w)
10:             Builds ← Builds ∪ {b}
11:             Integration ← Integration ∪ ({b} × w)
```

builds ($\in$ *Success*) that will be used to satisfy the requirements of t (i.e. $Requires[t]$). The definition of workingset is a parameter of this algorithm, as it captures the nature of backtracking. For now, we just assume that it returns a valid (i.e. pure) set of builds compatible to the requirements of t, if any. Below I will present three versions of the function, corresponding to the cases of no backtracking, simple backtracking and finally true backtracking.

If the function workingset is undefined (i.e. there are no valid workingsets) the algorithm continues with the next source tree in *order*. In this case the build of t is "not tried". Otherwise, w will be used in build i of t.

As Figure 2 and 3 showed, rebuilding a component using earlier dependencies occasionally amounts to duplicating earlier builds of that same component. The *build criterion* prevents this from occurring:

$$\text{build?}(t,w) \equiv \neg\exists b \in Builds : \\ \text{tree}(b) = t \wedge Integration^{+}[b] = Integration^{*}[w]$$

This function takes a tree t and a workingset w and searches *Builds* for an earlier build of t. If such a build is found, the *Integration* relation is used to check whether the same (transitive) dependencies were used the algorithm is about to use now via w. If the two closures are the same, building t against w would mean duplicating an earlier build, and no build is required.

If, on the other hand, a build *is* required according to the build criterion, t is built against workingset w by the function execute-build on line 9. This function returns a new build entity b which is either failed or successful. The following lines update the software knowledge-base. First, b is added to *Builds*[1], then *Integration* is extended with tuples linking b to each build in w since this captures how t has been built.

4.3. Incremental Continuous Integration

In this subsection I explain how the model just introduced, can be used to do continuous integration in an incremental fashion. This is done by presenting an implementation of the workingset function referenced in Algorithm 1. Without backtracking, this function can specified as follows:

$$\text{workingset}(t) = w$$
where
$$w = \{\langle t', i\rangle \in Builds \mid t' \in T, \neg\exists\langle t', j\rangle \in Builds : j > i\}$$
$$w \subseteq Success$$

For every revision (required by t) in the current snapshot, the working set contains the most recent build that has been successful. So, the set of workingsets is defined as the edge of the set of dependencies of t in the current snapshot if all builds are successful.

A valid workingset should contain successful builds for *every* component in the set $Requires[t]$. Because of topological order, every dependency of t has an element in *Builds*. This means, in turn, that the workingset contains the (globally) most recent build for those dependencies. However, if it contains a failed build, it makes no sense to proceed with building t. In that case t is added to the *Failure* part of *Builds* (see line 6 of Algorithm 1). This kind of failure—failure because dependencies have failed—is labeled with "not tried". It is precisely these kinds of failures that backtracking is designed to mitigate.

If we turn our attention to Figure 1, we observe that the only valid working sets in each cycle (indicated by subscripts) are as follows:

$$\begin{aligned}
\text{workingset}_1(\langle \text{FS}, 1.0\rangle) &= \{\} \\
\text{workingset}_1(\langle \text{DB}, 1.0\rangle) &= \{\langle\langle \text{FS}, 1.0\rangle, 1\rangle\} \\
\text{workingset}_1(\langle \text{App}, 1.0\rangle) &= \{\langle\langle \text{DB}, 1.0\rangle, 1\rangle, \langle\langle \text{FS}, 1.0\rangle, 1\rangle\} \\
\text{workingset}_2(\langle \text{App}, 1.1\rangle) &= \{\langle\langle \text{DB}, 1.0\rangle, 1\rangle, \langle\langle \text{FS}, 1.0\rangle, 1\rangle\} \\
\text{workingset}_3(\langle \text{FS}, 1.1\rangle) &= \{\} \\
\text{workingset}_3(\langle \text{DB}, 1.1\rangle) &= \{\langle\langle \text{FS}, 1.1\rangle, 2\rangle\}
\end{aligned}$$

The working sets are presented in the order of building, as follows from the the topological order between component revisions and the integration cycles.

4.4. Backtracking Incremental Continuous Integration

In the previous section dependencies were resolved by taken the latests builds out of *Builds* whether they had failed or not. In this section I change the dependency resolution algorithm in order to find the latest *successful* set of dependencies that lead to consistent (i.e. pure) integration. In the following I discuss two ways of backtracking: simple backtracking and true backtracking.

4.4.1. Formalization of Simple Backtracking

The simplest approach to find such a set is to look at earlier builds of the same component we are resolving the dependencies for. If an earlier successful build exists, then that

[1] This entails that b is either added to *Success* or *Failed*.

build used successful dependencies. Since all built artifacts can be reproduced at all times, the dependencies of that earlier build could be used. In this case the workingset is computed as follows:

$$\begin{aligned}&\text{workingset}(t) = w\\&\text{where}\\&\quad t = \langle c, v\rangle, t' = \langle c, v'\rangle, \langle t', i\rangle \in \mathit{Success},\\&\quad \neg\exists\langle\langle c, v''\rangle, j\rangle \in \mathit{Success} : j > i,\\&\quad w = \mathit{Integration}[t'], \mathit{Requires}[t'] = \mathit{Requires}[t]\end{aligned}$$

In other words, the set of successful builds is searched for the most recent build t' of the component of t (i.e. c). For this build the working set is retrieved from the *Integration* relation. Because the requirements of t may have changed since t'—requirements declarations are part of the source tree—we explicitly require that t and t' have the same requirements.

By induction on the sequencing of builds (i.e. in time and topological ordering of build) we know that the workingset w is pure because build $\langle t', i\rangle$ is, and therefore w can be used to build t. As a consequence purity of *Integration* is maintained. If no workingset w is found, the build of t still fails with "not tried".

Simple backtracking has been implemented as part of the Sisyphus continuous integration system [20]. How actual continuous integration performance is affected by this strategy is discussed in Section 5.

4.4.2. Formalization of True Backtracking

Let's state the problem more precisely. Assume we are building a source tree t. The objective is to find the most recent set of successful builds D for resolving the declared dependencies of t. Normally the dependencies used will be the builds for $\mathit{Snapshot}[t]$, as they have been built already due to the topological order. But since these builds may have failed this requirement is weakened, that is, we are looking for *any* most recent set of successful builds for each component in $\mathit{Requires}[t]$ such that building t against D is pure.

If the builds for the dependent trees in the current snapshot *did* actually succeed, the following algorithm will select these builds as D nevertheless. Thus, if all builds succeed, no currency is lost with respect to the normal dependency resolution algorithm.

Next I will present a formal version of selecting the most recent set D that can be used to build a tree t. It operates by computing all combinations of successful builds for each of the components in $\mathit{Requires}[t]$ and then selecting the newest combination. Formally, this reads:

$$\text{workingsets}(t) = \prod_{c \in \mathit{Requires}[t]} \{\langle\langle c, v\rangle, i\rangle \in \mathit{Success}\}$$

The function workingsets returns all workingset candidates that *could* be used for building source tree t. However, this could contain invalid permutations that would cause the build of t to become impure.

If we consider Figure 3 again, it can be seen that this algorithm returns the following sets of workingsets in the fourth integration cycle for component revision App 1.1:

$$\begin{aligned}\text{workingsets}_4(\langle\text{App},1.1\rangle) = \{&\\&\{\langle\langle\text{DB},1.0\rangle,1\rangle,\langle\langle\text{FS},1.0\rangle,1\rangle\},\\&\{\langle\langle\text{DB},1.0\rangle,1\rangle,\langle\langle\text{FS},1.1\rangle,3\rangle\},\\&\{\langle\langle\text{DB},1.2\rangle,4\rangle,\langle\langle\text{FS},1.0\rangle,1\rangle\},\\&\{\langle\langle\text{DB},1.2\rangle,4\rangle,\langle\langle\text{FS},1.1\rangle,3\rangle\}\\\}\end{aligned}$$

The second and third workingsets lead to impure integrations of App 1.1. This is a consequence of the fact that the FS version in those workingets (resp. 1.1 and 1.0) are not the versions that have been used in the builds of DB (cf. Figure 3). Therefore, App 1.1 cannot be built using those workingsets. To fix the problem, the *Integration* relation is used to filter out the workingsets leading to impurity. This leads to the final version of workingset which implements true backtracking:

$$\begin{aligned}&\text{workingset}(t) = w\\&\text{where}\\&\quad w \in \text{workingsets}(t),\\&\quad \text{pure?}(\mathit{Integration}^*[w]),\\&\quad w \text{ is most recent}\end{aligned}$$

Now it is clear that, the third workingset leads to impure integration:

$$\begin{aligned}&\text{pure?}(\mathit{Integration}^*[\{\langle\langle\text{DB},1.2\rangle,4\rangle,\langle\langle\text{FS},1.0\rangle,1\rangle\}])\\&\quad\equiv \text{pure?}(\{\langle\langle\text{DB},1.2\rangle,4\rangle,\langle\langle\text{FS},1.0\rangle,1\rangle,\langle\text{FS},1.1\rangle,3\rangle\}\\&\qquad\qquad\equiv 4 = 3 \equiv \bot\end{aligned}$$

Whether one workingset is more recent than another can be determined as follows. Since builds in *Builds* are totally ordered (in time), subsets $w \subseteq \mathit{Builds}$ can be sorted such that builds that are more recent come up front. Whether one working set is newer than another is determined by defining a lexicographic order on the sorted workingsets. In the example above it then follows that workingset $\{\langle\langle\text{DB},1.2\rangle,4\rangle,\langle\langle\text{FS},1.1\rangle,3\rangle\}$ is the most recent one.

4.4.3. Efficient Implementation

The generalized product used to find all permutations that could serve as a working set is very expensive. The number of workingsets increases very fast so this is no feasible way of implementation. In this subsection I describe an algorithm to generate all workingsets incrementally. By ordering builds

Algorithm 2 Incremental search for the latest working set

```
1:  function WORKINGSET(t)
2:      cursor ← |Success| − 1
3:      loop
4:          w ← ∅; i ← 0; todo ← Requires[t]
5:          while todo ≠ ∅ do
6:              if cursor − i < 0 then return nil
7:              b ← Success_{cursor−i}
8:              c ← component(tree(b))
9:              if c ∈ todo then
10:                 w ← w ∪ {b}
11:                 todo ← todo \ {c}
12:             i ← i + 1
13:         if pure?(Integration*[w]) then
14:             return w
15:         cursor ← cursor − 1
```

in decreasing temporal order, only the workingsets have to be generated that come before the one that will be used.

The algorithm is displayed in Algorithm 2. The function WORKINGSET takes a source tree t and incrementally searches for a valid working set in order to build it. It does this by maintaining a *cursor* that indicates the start of a search window over the set of successful builds (*Success*). Builds are ordered in time so that $Success_0$ is the first build maintained by the system, and $Success_{|Success|-1}$ is the last build. The cursor is moved from the last build downwards towards the first one in the outer loop (line 3) of the algorithm.

In the body of the loop the variable *todo* is initialized with the components we have to find builds for, i.e., the components required by t (line 4). Additionally, the current workingset w is initialized to be empty and a cursor *within* the window will be 0 (i).

The inner loop (line 5) iterates over every successful build in the current search window as long as *todo* is non-empty. A successful build is retrieved by indexing *Success* on $cursor - i$. If this index is below zero, however, we have exhaustively searched through *Success* without finding a suitable workingset, so we fail by returning the empty set (line 6). Otherwise, b will contain the ith build in the current search window (starting at *cursor*). If the component of this build b (c) is in *todo*, it is added to the current workingset w and c is removed from *todo*. Upon normal loop exit, *todo* is empty and w represents a workingset candidate. If the extent of w through *Integration* is pure, the workingset candidate is valid and w is returned as the result of WORKINGSET (line 14). Because the search window is moved downwards, we postulate that *if* a workingset is found, it will be the most recent one. Otherwise, if w is not valid, the search window is moved down one position and the outer loop starts anew.

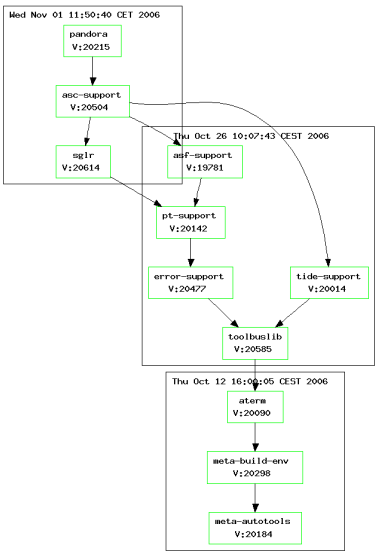

Figure 4. Automatically generated integration graph of the `pandora` subsystem of the ASF+SDF Meta-Environment

5. Evaluation

I have validated the simple form of backtracking in the context of the Sisyphus continuous integration and release system [20]. True backtracking has been implemented just recently and as a consequence no interesting data is available yet. However, the implementation of simple backtracking has delivered fruitful results.

To evaluate simple backtracking, I have compared build statistics derived from the database maintained by Sisyphus over two consecutive periods of 32 weeks. During this period Sisyphus continuously integrated the ASF+SDF Meta-Environment [17]. The Meta-Environment is an integrated development environment for developing source code analysis and transformation tools. It consists of around 60 components and is implemented in Java, C, and several domain specific languages. Figure 4 shows an example integration graph of a subsystem of the Meta-Environment, called `pandora`. The nodes represent successful builds, and the edges represent dependencies between those builds. The clusterering of nodes indicate build cycles. The figure shows that certain builds are used across build cycles.

General statistics about the two periods of time are collected in Table 1. In this table I have counted the number of revisions, the number of successful builds, the number of failed build and the number "not tried" builds. The total number of builds is shown as well. Although in the period that simple backtracking was enabled, the number of component revisions was one third fewer than in the previous

	No backtracking	Simple backtracking
#Revisions	1497	1025
#Success	11565	9391
#Failure	1074	507
#"Not tried"	4499	1163
#Builds	17138	11061

Table 1. Build statistics over two consecutive periods of time of 32 weeks

period, the number of failed builds has decreased by roughly 43% and the number of "not tried" builds has decreased even stronger, by around 74%. These absolute numbers suggest that on average the build feedback has improved considerably.

The amount of not tried builds per component is dependent on the component architecture of the system and how the architecture evolves. For instance, a component that has no dependencies will never be "not tried". The consequence of this is that "not tried"-ness has a cumulative effect if there is no backtracking. If a component build fails, *every* component requiring that build, transitively, will be assigned the status of "not tried". Clearly there is considerable gain if this can be avoided.

To show the propagating effect of build failures I have plotted the number of revisions, successes, failures, "not trieds" and builds per component to see how these numbers relate to the position of a component in the dependency graph. The plot for the first period—no backtracking—is shown in Figure 5. The X-axis of this plot represents the different components of the Meta-Environment. They are sorted according to the total number of builds. This way of ordering components is a raw estimate of position in the dependency graph. If this dependency graph would have been stable, this ordering corresponds to the topological sort of the graph. However, components are added and removed, and dependency relations may change in between integration cycles. It would therefore make no sense to use the topological sort. A similar plot for the second period of time, with simple backtracking enabled, is displayed in Figure 6.

If we compare the two figures, what stands out the most is that the number of builds (total, success, failed and "not tried") in the period without backtracking grows much steeper than in the period of backtracking if the components are higher up in the dependency graph. A second observation is the relative large *and growing* distance between the number of total and successful builds. This is precisely the cumulative effect of build failures.

In the period with simple backtracking, however, as displayed in Figure 6, the build figures grow much slower and the distance between the total number of builds and the number of successful builds is more or less constant. In addition, the line indicated "not tried" builds is almost flat. This means that even the simple form of backtracking almost completely eliminates the problem of build failure as an impediment to delivery. These results do not say anything about whether the actual integrations are optimal with respect to up-to-dateness. Still, changes could be missed in an integration.

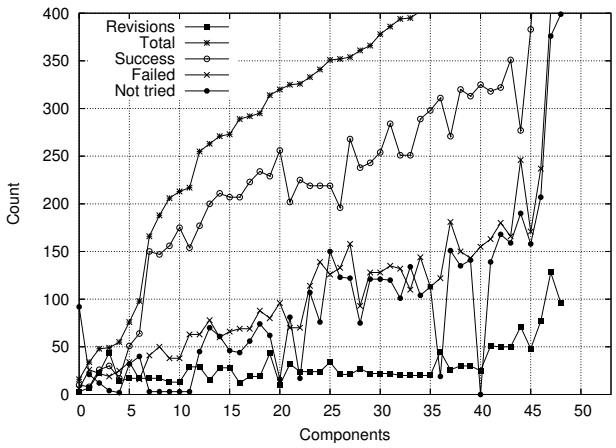

Figure 5. Build results without backtracking

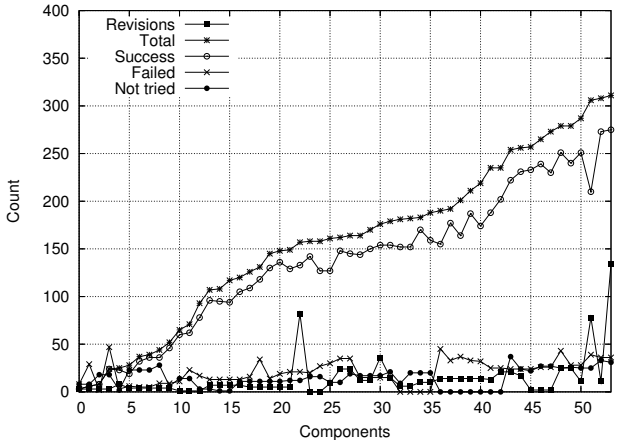

Figure 6. Build results with simple backtracking

6. Related Work & Conclusion

6.1. Related Work

Continuous integration has received very little attention from the research community; we only know of Dolstra [6], who describes the use of the deployment system Nix as a continuous integration system with similar goals as Sisyphus. Additionally, Lippert *et al.* [11] describe the implementation

of a continuous integration system as means for realizing the *practice* of continuous integration. The lack of attention is surprising since there exists a host of continuous integration systems, both commercial and freely available. Unfortunately no empirical data on the assumed merit of continuous integration seems to be available as of today although it is widely considered to be a best practice [2, 8, 13].

Incremental building, or selective recompilation, goes back to Make [7] and has been researched quite vigorously; see e.g. [4, 10]. This work, however, mostly considers dependencies on the level of files. Determining whether a file requires recompilation mostly involves checking timestamps of cryptographic hashes. In this work, however, we compare actual revisions of version control system (VCS) to a database storing accurate bills of materials of all past builds.

Build optimization is another area of related work. Caching builds [16], distributing builds [14] and build paralellization [1], header restructuring [21] and precompilation [22] mostly optimize towards minimizing resource consumption. In this paper I try to optimize towards improved feedback and maximum release opportunity. Of course, both goals are not mutually exclusive.

6.2. Conclusion

The subject of this paper is to improve automatic continuous integration in component-based development settings in order to maximize feedback and maximize release opportunity. I introduced an algorithm for incremental continuous integration and subsequently extended it with "simple backtracking" and "true backtracking" to make the integration process more resilient with respect to build failures. Finally I discussed some empirical results that were obtained from a running implementation of simple backtracking. These results show that even the simple backtracking algorithm almost completely neutralizes the cumulative effect of build failures. Future work will have to show how true backtracking improves this situation. The true backtracking algorithm is still highly experimental and will require further study in order to positively claim that it behaves as expected. Additionally, it is not clear what the worst-case complexity of the algorithm is.

References

[1] E. H. Baalbergen. Design and implementation of parallel make. *Computing Systems*, 1(2):135–158, 1988.

[2] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley, 2004.

[3] D. E. Bellagio and T. J. Milligan. *Software Configuration Management Strategies and IBM Rational ClearCase. A Practical Introduction*. IBM Press, 2nd edition, 2005.

[4] E. Borison. A model of software manufacture. In *Proceedings of the IFIP International Workshop on Advanced Programming Environments*, pages 197–220, Trondheim, Norway, June 1987.

[5] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion*. O'Reilly Media, 2004. Online: `http://svnbook.red-bean.com/`.

[6] E. Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Faculty of Science, University of Utrecht, 2006.

[7] S. I. Feldman. Make – A program for maintaining computer programs. *Software – Practice and Experience*, 9(3):255–265, Mar. 1979.

[8] M. Fowler and M. Foemmel. Continuous integration. Online: http://martinfowler.com/articles/continuousIntegration.html, February 2007.

[9] J. Henstridge and H. Pennington. Pkgconfig. Online: `http://pkgconfig.freedesktop.org` (May 2007).

[10] D. A. Lamb. Relations in software manufacture. Technical report, Department of Computing and Information Science, Queen's University, Kingston, Ontario K7L 3N6, october 1994.

[11] M. Lippert, S. Roock, R. Tunkel, and H. Wolf. *Extreme Programming Perspectives*, chapter Stabilizing the XP Process Using Specialized Tools. XP Series. Addison-Wesley, 2002.

[12] H. Mather. *Bills of Materials*. Dow Jones-Irwin, 1987.

[13] S. McConnell. Daily build and smoke test. *IEEE Software*, 13(4), July 1996.

[14] M. Pool. DistCC, a fast free distributed compiler. In *Proceedings of `linux.conf.au`*, 2004.

[15] D. E. Rogerson. *Inside COM. Microsoft's Component Object Model*. Microsoft Press, 1997.

[16] E. Thiele. CompilerCache. `http://www.erikyyy.de/compilercache`.

[17] M. van den Brand, M. Bruntink, G. Economopoulos, H. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J. Vinju. Using The Meta-environment for Maintenance and Renovation. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE Computer Society Press, 2007.

[18] M. van den Brand, A. Kooiker, J. Vinju, and N. Veerman. A Language Independent Framework for Context-sensitive Formatting. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 103–112, Washington, DC, USA, 2006. IEEE Computer Society Press.

[19] T. van der Storm. Continuous release and upgrade of component-based software. In E. J. Whitehead, Jr. and A. P. Dahlqvist, editors, *Proceedings of the 12th International Workshop on Software Configuration Management (SCM-12)*, pages 41–57, 2005.

[20] T. van der Storm. The Sisyphus continuous integration system. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 335–336. IEEE Computer Society Press, 2007.

[21] Y. Yu, H. Dayani-Fard, and J. Mylopoulos. Removing false code dependencies to speedup software build processes. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 343–352, 2003.

[22] Y. Yu, H. Dayani-Fard, J. Mylopoulos, and P. Andritsos. Reducing build time through precompilations for evolving large software. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 59–68, 2005.