FACHHOCHSCHULE AACHEN, CAMPUS JÜLICH

FACHBEREICH 09 – MEDIZINTECHNIK UND TECHNOMATHEMATIK

STUDIENGANG ANGEWANDTE MATHEMATIK UND INFORMATIK

SEMINARARBEIT

Aufspaltung von Validierungs- und Testprozessen in separate Repositories

Autor:
Denis Wolter, 3568604

Betreuer:
Prof. Dr. Mehdi Behbahani
Tobias Bongartz, M. Sc.

Aachen, 20. Dezember 2024

Inhaltsverzeichnis

1. Einleitung

Die vorliegende Arbeit beschäftigt sich mit der Optimierung und Modularisierung des Repository Managements für die Simulationssoftware *xns*. Ziel ist es, durch den Split des bestehenden *xns-testing* Repositories die Wartbarkeit, Erweiterbarkeit und Flexibilität des Systems zu erhöhen. Dabei wird ein neuer Workflow eingeführt, der die Integration von Testcases und Validierungsskripten erleichtert und eine effizientere Nutzung des CI-Systems ermöglicht.

xns und xnx-testing

xns ist die lehrstuhlinterne CATS-Software zur Simulation von Strömungen mithilfe der Finite-Elemente-Methode – ein bei unterschiedlichen physikalischen Aufgabenstellungen angewendetes numerisches Verfahren, das unter anderem komplexe Festkörper auf Festigkeit und Verformung untersucht. Der Fokus der Software liegt aufgrund des hohen Rechenaufwands und der Komplexität auf der Stabilität und auf der Performance des Codes. Der Source Code von xns wird am CATS über Gitolite zur Verfügung gestellt. Gitolite ist ein verwaltungsorientiertes Open-Source-Tool, das es ermöglicht, Git-Repositories auf einem zentralen Server zu verwalten und den Zugriff auf diese Repos zu steuern. Der Zugriff auf Gitolite Repositories wird mit entsprechenden Anträgen der Mitarbeiter protokolliert und von der IT verwaltet.

Das CI-System am CATS testet *xns* automatisch, indem es die Software kompiliert und vordefinierte Test-Cases durchläuft. Immer wenn ein Differential von *xns* oder *xns-testing* erstellt wird, wird das CI-System getriggert. Außerdem werden nachts automatisch weitere, komplexere Tests durchgeführt, die während des laufenden Betriebs zu zeitlich bedingten Verzögerungen führen würden. Die Kompilierung von *xns* lässt sich recht zuverlässig und einfach validieren. Zu testen, ob Simulationsergebnisse nach den Änderungen noch problemlos laufen ist etwas komplizierter.

Dazu werden reference states genutzt, die jeden morgen (nach den nächtlichen, komplexeren Tests) aktualisiert werden.
Der *xns-testing*/Validierungs- Prozess ist also in zwei separate Schritte aufgeteilt:

- Das Bauen der Software xns mit allen Libraries und Packages, die dafür nötig sind
- Das Ausführen der Testcases um sicherzustellen, dass nach dem veränderten Differential die Ergebnisse dieselben sind. [1]

[1] xns Overview – siehe Anhang S.28 Verfügbar unter:
https://cats.pages.rwth-aachen.de/xns-documentation/html/index.html
(letzter Zugriff: 20.12.2024)

Struktur und Intention der Arbeit

Ziel der Arbeit ist es, das Repository *xns-testing* unseres Lehrstuhls zu spalten und nahtlos in unser bestehendes CI-System zu integrieren.
Validierungsskripte und Testcases sind bisher eng miteinander verknüpft. Die Trennung soll dazu beitragen, die Wartbarkeit, Modularität und Erweiterbarkeit der Inhalte zu verbessern.

Fast alle Repositories starten als monolithische Strukturen, da dies Anfangs oft effizient erscheint. Mit zunehmendem Umfang und zunehmender Komplexität ergeben sich jedoch Herausforderungen. Die enge Kopplung zwischen Komponenten erschwert Refactoring, die Versionskontrolle und eine klare Trennung der Verantwortlichkeiten. Das Spalten des Repos erleichtert die Weiterentwicklung beider Module und schafft eine Grundlage für neue Nutzungsmöglichkeiten. Durch die klare Verantwortlichkeitszuweisung wird außerdem die Pflege und die Weiterentwicklung des Codes vereinfacht. Die Arbeit hat zum Ziel, diese Trennung technisch und organisatorisch umzusetzen, die Änderungen in das bestehende CI-System zu integrieren und dadurch eine nachhaltige und flexible Weiterentwicklung zu ermöglichen.

2. Grundlagen und Hintergründe des Repository-Splits

In diesem Abschnitt werden die grundlegenden Konzepte und die Struktur der verwendeten Repositories erläutert. Der Aufbau und die Organisation der Repositories werden erläutert, um ein umfassendes Verständnis der zugrunde liegenden Struktur und ihrer Bedeutung für den Entwicklungsprozess sowie das CI-System zu schaffen.

2.1 Motivation und Zielsetzung

Der Split des Repositories *xns-testing* in zwei separate Repositories verfolgt das Ziel, die Organisation und den Workflow der Tests klarer zu strukturieren und effizienter zu gestalten. Dabei wurden folgende Bereiche aufgeteilt:

validierungsbezogene Skripte und Tools

- Dieses Repository umfasst alle Skripte, Konfigurationsdateien und Tools, die zur Validierung der Testläufe erforderlich sind. Es fokussiert sich auf die Methodik und die Automatisierung des Testings, z.B. den Vergleich der Ergebnisse vor- und nach den Codeänderungen und die Verwaltung der Referenzlösungen.

Testcases

- Dieses Repository enthält ausschließlich die konkreten Testcases und die dazugehörigen Eingabedaten.

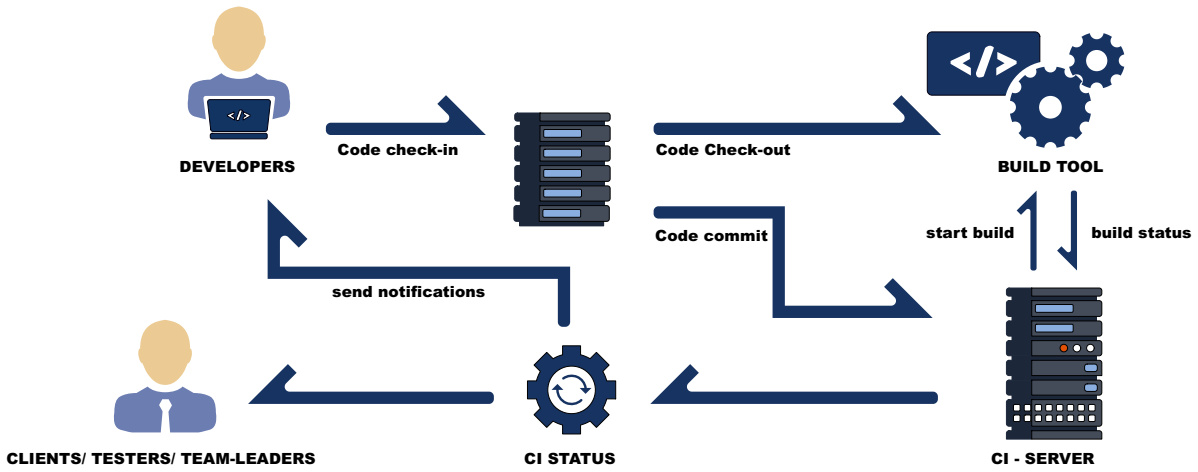
Ziele

- verbesserte Wartbarkeit: Durch die Trennung der Testcases und der Validierungsskripte können Änderungen in einem Bereich vorgenommen werden, ohne den anderen zu beeinflussen. Das reduziert Komplexität und erleichtert die Weiterentwicklung.
- klarer Workflow: Der Testing-Worklfow wird übersichtlicher, da der Build- und Validierungsprozess von den eigentlichen Testcases entkoppelt ist.
- Skalierbarkeit: Die Anzahl der Testcases und die Komplexität der Validierung können unabhängig voneinander wachsen, ohne das Repository zu überladen oder an Übersichtlichkeit einzubüßen.
- Potential für Open Source und Wiederverwertbarkeit: Das Repo für Validierungs-Skripte und Tools kann als eigenständiges Framework genutzt werden, sodass andere Projekte oder Strukturen die Verwendung der Validierungsmethodik nutzen können.

2.2 Der Nutzen von Repositories und Continuous Integration

Repositories dienen als zentrale Ablage für den Quellcode, ermöglichen eine Versionskontrolle und erleichtern die Zusammenarbeit im Team.
Alle Änderungen können über Git nachverfolgt, dokumentiert und bei Bedarf rückgängig gemacht werden. Git spielt dabei eine entscheidene Rolle, da es Entwicklern Werkzeuge bietet, um Änderungen zu verwalten und mit anderen Entwicklern zu teilen.

Continuous Integration (CI) automatisiert diesen Entwicklungsprozess, indem es alle übermittelten Änderungen regelmäßig integriert und auf Fehler überprüft.
Dadurch werden Probleme frühzeitig erkannt, die Codequalität gesteigert und eine schnelle Bereitstellung neuer Features ermöglicht. CI kann sowohl lokal, als auch in zentralen Systemen genutzt werden, um Änderungen am Code noch vor, oder nach einem Push zu überprüfen.



CI-Workflow

Entwickler ändern den Code lokal und committen ihn anschließend in das Repository (Code-check-in). Nach einem Code-commit startet der CI-Server automatisch (mithilfe eines Build-Tools) einen neuen Build, der die Änderungen integriert und überprüft.
Nach Abschluss des Builds übermittelt das Build-Tool den Build-Status an den CI-Server. Dieser Server sendet anschließend die Build-Logs, Testergebnisse und andere relevante Informationen an die Entwickler, Tester und/oder Teamleiter, um sie über den Status zu informieren und potentielle Fehler transparent zu machen. [2]

² Die Abbildung basiert auf folgendem Piktogramm:
https://mcuoneclipse.com/wp-content/uploads/2023/10/continuous_integration.jpg
(letzter Zugriff: 20.12.2024)

2.3 Einführung in Continuous Integration und die verwendeten Tools

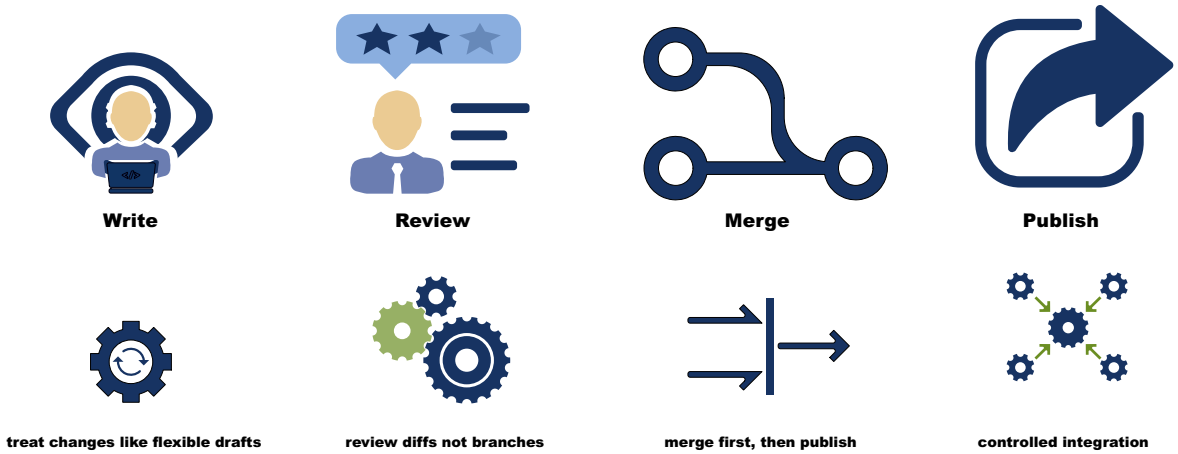
Dieser Abschnitt gibt einen Überblick über das Continuous Integration (CI) System und die verwendeten Tools, die zur Automatisierung der Build- und Testprozesse eingesetzt werden. Diese Werkzeuge unterstützen den Entwicklungszyklus und tragen maßgeblich zur Sicherstellung der Codequalität bei.

2.3.1 Phabricator

Phabricator ist eine Open-Source-Entwicklungsplattform für Softwareprojekte, das besonders in großen Teams genutzt wird, um Qualität und Transparanz des Codes zu fördern.

Zu den Hauptfunktionen von Phabricator gehören Code-Review (Differentials), Aufgaben und Projektmanagement (Maniphest), Repository-Verwaltung, Wiki und CI-Integration.

Das Konzept basiert darauf, dass Änderungen nicht endgültig sind, sondern flexibel überprüft und angepasst werden können. Reviewer können gezielt die exakten Änderungen im angepassten Code prüfen, ohne den gesamten Branch analysieren zu müssen. Auch Entwicklungsfortschritte werden getrackt, um den gesamten Entwicklungsprozess übersichtlich und nachvollziehbar zu gestalten. [3]



[3] Die Abbildung basiert auf folgendem Piktogramm:
https://graphite.dev/blog/phabricator-code-review
(letzter Zugriff: 20.12.2024)

2.3.2 Arcanist

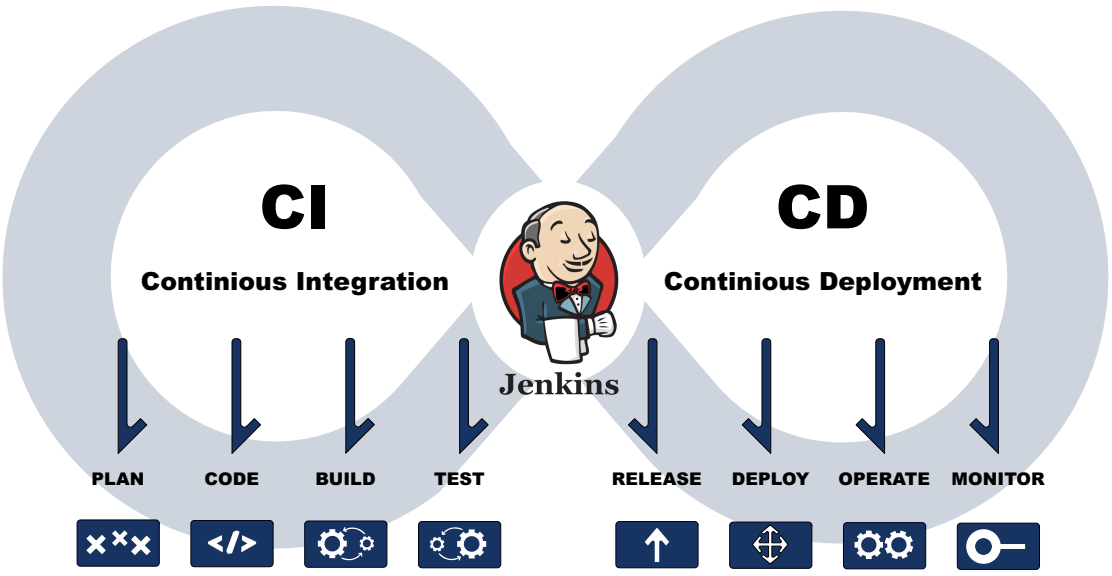
Arcanist ist ein Kommandozeilen-Tool, das als Schnittstelle zwischen Entwicklern und Phabricator dient. Arcanist erstellt und verwaltet Revisions (Änderungsvorschläge) für Code-Reviews in Phabricator. Befehle wie `arc diff` oder `arc land` erlauben das Hochladen von Änderungen für Reviews und das Zusammenführen von Änderungen nach Freigabe. Es ermöglicht eine nahtlose Integration in die Workflows von Phabricator, z.B. für Aufgabenverfolgung und Build-Prozesse.

Typischer Workflow

- Entwickler ändern Code in ihrem lokalen Branch
- Mit `arc diff` wird eine Revision erstellt, die Änderungen zusammenfasst und an Phabricator sendet.
- Reviewer geben Feedback in Phabricator und Entwickler können entsprechende Änderungen vornehmen und diese mit weiteren `arc diff`-Uploads aktualisieren.
- Nach der Freigabe wird der Code mit *arc land* in den Hauptbranch integriert.

2.3.3 Jenkins

Jenkins ist ein Open-Source-Automatisierungsserver, der für Continious Integration (CI) und Continuous Delivery (CD) verwendet wird.
Jenkins automatisiert das Bauen, Testen und Bereitstellen von Codeänderungen, sodass Entwickler schneller, häufiger und effizienter Änderungen einbringen können.



Jenkins führt in vorher definierten Abständen Tests durch, erstellt entsprechende Berichte über den Build Status und die Testergebnisse. So wird sichergestellt, dass Code regelmäßig gebaut und getestet wird und Probleme frühzeitig erkannt und behoben werden können. [4]

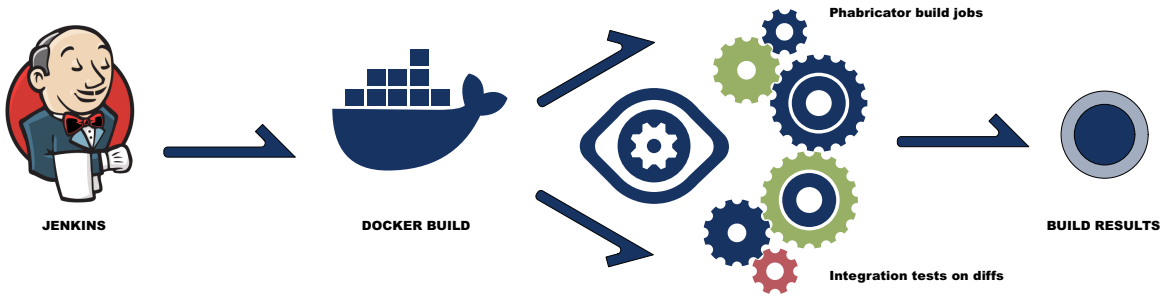
Jenkins übernimmt am CATS die Funktionalität des CI-Servers. Test werden (nach Änderungen am Code) beim Update eines Differentials auf einer virtuellen Maschine (VM) ausgeführt.

Jenkins unterstützt verschiedene Plugins und kann mit vielen Tools und Plattformen integriert werden. Am CATS ist das wichtigste Jenkinks Plugin das Phabricator Plugin.
Phabricator wird als ergänzende Plattform genutzt, um die Änderungen und Testergebnisse nach einem Build in Jenkins zentral zu dokumentieren und die entsprechenden Code-Änderungen zu verwalten.

---

[4] Die Abbildung basiert auf folgendem Piktogramm:
https://de.fiverr.com/azuredev/setup-ci-cd-pipelines-using-jenkins
(letzter Zugriff: 20.12.2024)

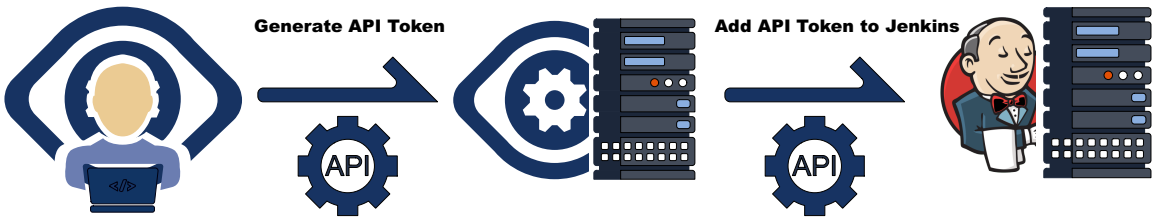
2.3.4 Das Phabricator Plugin:



Das Phabricator Plugin dient dazu, Integration zwischen Jenkins und Phabricator zu ermöglichen. Automatische Builds und Prüfungen aus Jenkins werden in Phabricator gemeldet und die Ergebnisse in der Phabricator Benutzeroberfläche angezeigt.

Das CI-System baut und führt alle Tests in einem virtuellen Ubuntu-System innerhalb eines Docker-Containers aus.
Docker stellt Container bereit, die immer das gleiche System, dieselben Umgebungsvariablen etc. enthalten – unabhängig davon, auf welchem Computer sie laufen. Das sorgt dafür, dass Tests unter denselben Bedingungen ausgeführt werden, egal auf welchem Endgerät Docker ausgeführt wird.

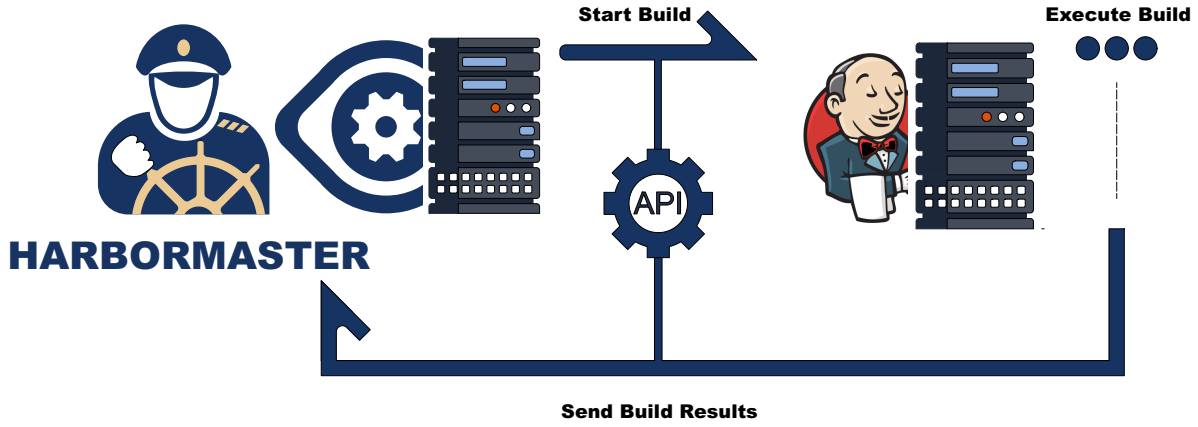
Das Plugin verbindet Jenkins mit Phabricator durch das Einrichten eines Bot-Accounts in Phabricator und das Konfigurieren eines API-Tokens in Jenkins (phabricatortesttoken). Dieses Token wird verwendet, um mit der Phabricator-API zu kommunizieren. [5]



[5] API Tokenverwaltung sinngemäß aus der folgenden Quelle entnommen:
https://github.com/jenkinsci/phabricator-plugin
(letzter Zugriff: 20.12.2024)

Build-Trigger

Ein Build wird ausgelöst, wenn eine Differential Revision eingereicht wird.
Nach Abschluss des Builds sendet Jenkins den Status des Builds (Build succeeded/build failed) zurück an Phabricator. Der entsprechende Status wird dann im Build-Protokoll angezeigt.



Das Phabricator Plugin ermöglicht u.a. mithilfe des Harbormaster Tools eine nahtlose Integration von CI-Prozessen in einen Phabricator basierten Workflow. Harbormaster ermöglicht das Erstellen grundlegender Build-Pläne, das Auslösen von Builds über Herald-Regeln (die automatisierte Aktionen auf bestimmte Ereignisse oder Änderungen in Phabricator definieren) und das Anzeigen von Ergebnissen für Code-Reviews und Commits.

2.3.5 Lokales Bauen des CI-Systems

Das CATS CI-System kann auch lokal gebaut werden um die Funktionalität von geändertem Code mit einem lokalen Build zu überprüfen und zu verhindern, dass Fehler eingeführt werden.
Um längere Warteschlangen auf Jenkins zu verhindern und wegen dem simpleren Workflow, ist es oft nützlich, Änderungen am Code lokal zu überprüfen, bevor Sie über Phabricator/Arcanist gesteuert werden.

In Jenkins können Tests längere Wartezeiten verursachen, weil mehrere Tests gleichzeitig ausgeführt werden müssen. Aus diesem Grund ist es oft sinnvoll, die Tests lokal auszuführen, bevor man Änderungen in Jenkins hochlädt (dies geschieht durch das Aktualisieren des

Differentials). Ein Differential ist ein Code-Änderungsvorschlag, der zur Überprüfung in Phabricator eingereicht wird.

Das lokale CI-System lässt sich im selben Docker-Container ausführen, den Jenkins nutzt. Docker stellt sicher, dass die Umgebung identisch ist, um Fehler auch außerhalb des CI-Systems nachvollziehen zu können. Wenn man Tests außerhalb dieses Container (z.B. auf dem Cluster) ausführt, würde es u.a. anhand den Unterschieden in der Architektur zu unterschiedlichen Ergebnissen kommen.

Das lokale Bauen des CI-Systems war besonders bei der Umsetzung des Repository-Splits hilfreich, um Tests durchzuführen und Änderungen am Code auf Stabilität zu überprüfen. [6]

Nötige Schritte um das CI-System lokal zu reproduzieren:

- Ein entsprechendes directory (z.B. LocalCI) erstellen und in das Repo navigieren
- ci-tools klonen
  ```
  git clone git@git.cats.rwth-aachen.de
  ```
- Setzten der Umgebungsvariable:
  ```
  export CI_TOOLS=./ci-tools
  ```
- CI-System vorbereiten:
  ```
  ./ci-tools/build/runJenkinsBuild.sh ci-tools –local
  ```
- CI-System im Docker Container ausführen:
  ```
  ./ci-tools/build/runDockerLocally.sh
  ```

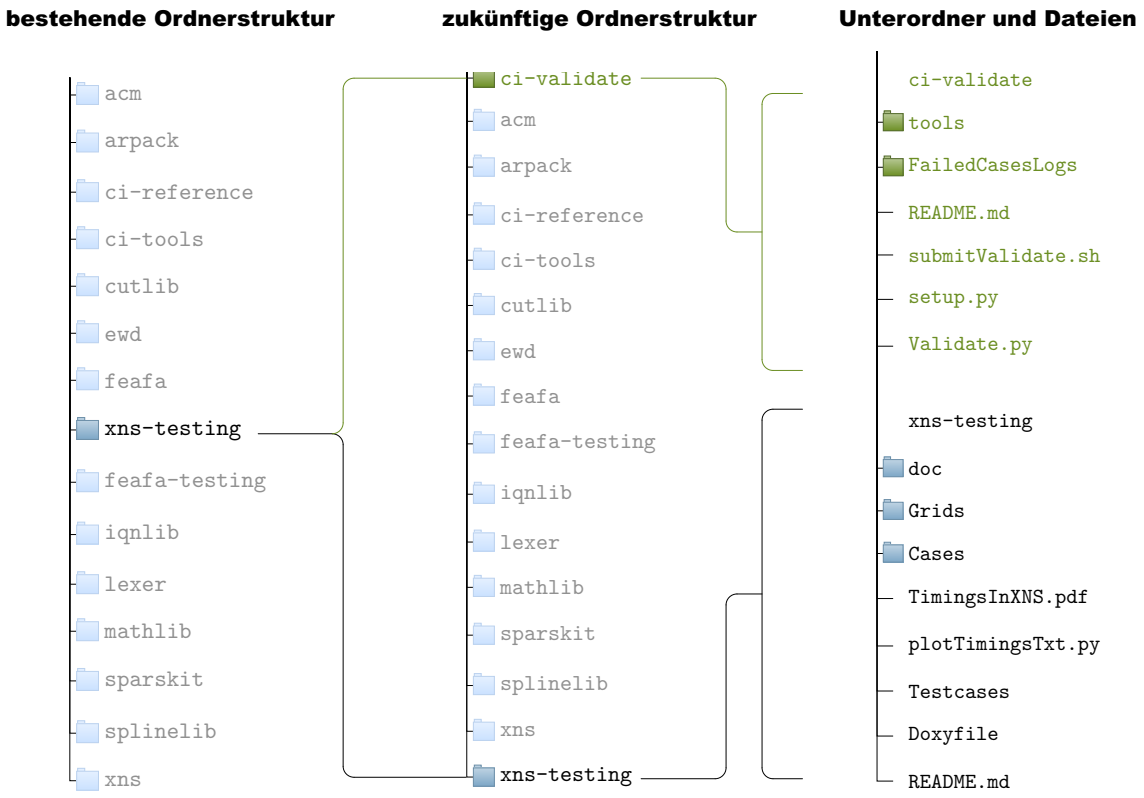
3. Implementierung des Repository-Splits und neuer Workflow

Im Folgenden wird die technische Umsetzung des Repository Splits erläutert, sowie die Integration in den bestehenden Workflow beschrieben. Ziel ist es, die Trennung der Inhalte ohne Unterbrechungen für das CI-System oder den Entwicklungsprozess zu realisieren und gleichzeitig eine klare Struktur für zukünftige Anpassungen und Erweiterungen zu schaffen.

[6] sinngemäß entnommen aus einem cats.pages Tutorial siehe Anhang S. 29-31. Verfügbar unter: https://cats.pages.rwth-aachen.de/xns-documentation/html/localCiGuide.html (letzter Zugriff: 20.12.2024)

3.1 Detaillierte Beschreibung des Repository-Splits

Die zukünftige Ordnerstruktur gliedert sich wie folgt:



Um das neue Repository ci-validate zu erstellen, musste das Repository in unserer Verwaltungsdatei des Gitolite Servers */conf/gitolite.conf* hinzugefügt- und die Berechtigungen (Schreib- und Leserechte) vergeben werden.

```
repo ci-validate
        RW+ = @manager wolter
        RW  = @staff
        RW = jenkins
```

Das einfache Kopieren der Dateien und der Ordner aus dem xns-testing Repository in das neue ci-validate Repository, beispielsweise per Drag-and-Drop oder mittels scp in der Kommandozeile, hätte den Verlust der Commit-Hostorie zur Folge gehabt. Um die History beizubehalten, waren daher zusätzliche Schritte erforderlich:

1. Klonen des repos xns-testing:

```
git clone git@git.cats.rwth-aachen.de:xns-testing
cd xns-testing
```

2. Erstellen eines neuen Branches mit den gewünschten Dateien/Verzeichnissen

```
git checkout -b ci-validate-branch
```

Erstellt einen neuen Branch (ci-validate-branch).

```
git rm -rf *
```

Löscht alle Dateien, die nicht benötigt werden.

```
git checkout main -- FailedCasesLogs tools Validate.py
setup.py submitValidate.sh
```

Kopiert die gewünschten Dateien und Ordner aus der History von *xns-testing* in den neuen Branch.

3. Committen der Änderungen

```
git add .
git commit -m "added selected files and folders from xns-
testing with full history"
```

Fügt die Dateien und Ordner hinzu und erstellt einen entsprechenden Commit.

4. Klonen von ci-validate

```
cd ..
git clone git@git.cats.rwth-aachen.de:ci-validate
```

wechselt den Ordner ins Oberverzeichnis und klont das Repo *ci-validate*.

```
cd ci-validate
git init
```

Wechselt in den Ordner *ci-validate* und initialisiert das leere Repository.

```
git pull ../xns-testing ci-validate-branch --allow-unrelated-
histories --no-rebase
```

Die Option `–allow-unrelated-histories` erlaubt es mir, zwei Repositories zu kombinieren, die unterschiedliche Historien haben.

5. Testen ob die History übernommen wurde (Anhand von Validate.py)

```
git log -- Validate.py
```

Hier sehe ich in der Konsole anhand von Validate.py, welche History gespeichert ist.

6. Pushen der Änderungen

```
git push -u origin main
```

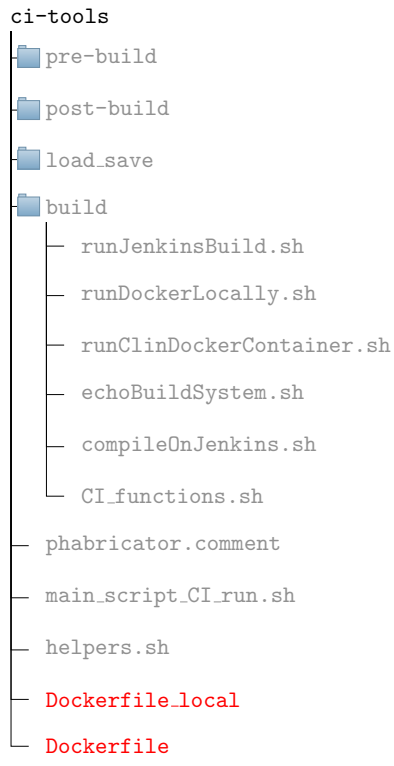
Wenn die Commit-Histoy korrekt übernommen wurde, kann das Repo auf dem Gitolite Server mit meiner lokalen Version aktualisiert werden. [7]

[7] Struktur für einen Repository-Split mit Erhalt der History sinngemäß entnommen aus: https://alexey-anufriev.com/blog/split-git-repository/ (letzter Zugriff: 20.12.2024)

Blackbox Tests und Änderungen im Code

Nachdem das neue Repo ci-validate erstellt wurde, habe ich das lokale CI-System über den Docker Container gestartet und Blackbox Tests durchgeführt. Dabei wurden die Fehlermeldungen in der Konsole analysiert und die entsprechenden Stellen in den Skripten angepasst.
Die erste Fehlermeldung wies auf fehlerhafte Verweise in unseren Dockerfiles hin, da das Skript setup.py nach dem Split in unserem neuen Repo *ci-validate* verschoben wurde.

```
[+] Building 35.1s (18/21)                                                                    docker:desktop-linux
 => [internal] load .dockerignore                                                                          0.0s
 => => transferring context: 2B                                                                            0.0s
 => [internal] load build definition from Dockerfile                                                       0.0s
 => => transferring dockerfile: 1.32kB                                                                     0.0st
 => [internal] load metadata for docker.io/library/ubuntu:20.04                                            5.0sa
 => [internal] load build context                                                                         22.3sr
 => => transferring context: 3.57GB                                                                       22.2sc
 => [ 1/17] FROM docker.io/library/ubuntu:20.04@sha256:8e5c4f0285ecbb4ead070431d29b576a530d3166df73ec44affc1cd27555141b      0.0s
 => CACHED [ 2/17] RUN set -x; apt-get -y update && apt-get install -y cmake build-essential lsb-release git vim python3.8   0.0s
 => CACHED [ 3/17] RUN set -x; apt-get -y update && apt-get install -y gawk flex byacc libblas-dev liblapack-dev doxygen gra  0.0s
 => CACHED [ 4/17] RUN set -x; update-alternatives --install /usr/bin/gfortran gfortran /usr/bin/gfortran-10 100           0.0s
 => CACHED [ 5/17] RUN set -x; update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-10 100                         0.0s
 => CACHED [ 6/17] RUN set -x; update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-10 100                         0.0s
 => CACHED [ 7/17] RUN set -x; update-alternatives --set gfortran /usr/bin/gfortran-10                                    0.0s
 => CACHED [ 8/17] RUN set -x; update-alternatives --set gcc /usr/bin/gcc-10                                              0.0s
 => CACHED [ 9/17] RUN set -x; update-alternatives --set g++ /usr/bin/g++-10                                              0.0s
 => CACHED [10/17] RUN set -x; apt-get install -y python3-pip                                                             0.0s
 => CACHED [11/17] RUN pip3 install numpy                                                                                 0.0s
 => [12/17] ADD . /code                                                                                                   7.7s
 => [13/17] WORKDIR /code/xns-testing                                                                                     0.0s
 => ERROR [14/17] RUN pip3 install .                                                                                      0.8s
------
 > [14/17] RUN pip3 install .:
0.766 ERROR: Directory '.' is not installable. Neither 'setup.py' nor 'pyproject.toml' found.
------
Dockerfile:17
--------------------
```



Dockerfile

```
11 #RUN set -x; update-alternatives --set gcc /usr/bin/gcc-10
12 #RUN set -x; update-alternatives --set g++ /usr/bin/g++-10
13 #RUN set -x; apt-get install -y python3-pip
14 #RUN pip3 install numpy
15 ADD . /code
   WORKDIR /code/xns-testing
16 WORKDIR /code/ci-validate
17 RUN pip3 install .
18 WORKDIR /code/ci-reference
19 RUN pip3 install .
20 WORKDIR /code
21 ENV CC=/usr/bin/mpicc
```

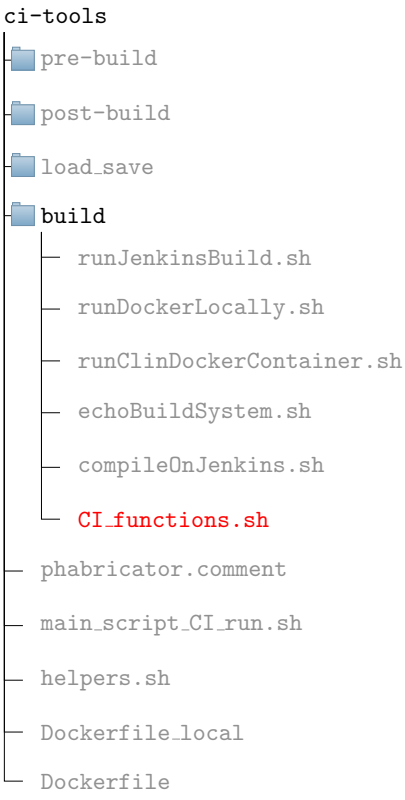
Dockerfile_local

```
11 RUN set -x; update-alternatives --set gcc /usr/bin/gcc-10
12 RUN set -x; update-alternatives --set g++ /usr/bin/g++-10
13 RUN set -x; apt-get install -y python3-pip
14 RUN pip3 install numpy
15 ADD . /code
   WORKDIR /code/xns-testing
16 WORKDIR /code/ci-validate
17 RUN pip3 install .
18 WORKDIR /code/ci-reference
19 RUN pip3 install .
20 WORKDIR /code
21 ENV CC=/usr/bin/mpicc
```

Die zweite Fehlermeldung wies auf falsche Verweise im Skript */code/ci-tools/CI_functions.sh* hin, wodurch die xns-validation-cases nicht korrekt ausgeführt werden konnten. Der Grund dafür war ebenfalls die geänderte Ordnerstruktur.

Es waren zwei Anpassungen nötig: Die erste Änderung betraf den Pfad der Datei *Testcases* (ein Textdokument, das die absoluten Pfade aller zu durchlaufenden Testcases enthält). Der angepasst Pfad wird mit dem Befehl `--input` aufgerufen, einem bereits bestehenden Argument im Skript Validate.py.
Die zweite Änderung betrifft das Oberverzeichnis *ci-validate*.

```
+++++ TIMINGS: Compiling software took 674 seconds +++++
*********************************  Done compiling code  ****************************************

**************************************************************************************************
**********************   Running XNS-Testing Validation Cases   *********************************
**************************************************************************************************
Launching xns-testing
Testing all features (in Release mode)
/code/ci-tools/build/CI_functions.sh: line 177: ./Validate.py: No such file or directory
================================================= Deleting local docker container ================================================
==
6859e7c556a0
6859e7c556a0
Untagged: localdocker:latest
Deleted: sha256:68403b0ef918e1c0a2eae63f8a3eb0714178a012d4c1adac46fce2ad8082129f
==================================================================================================================================
==
deniswolter@monitor-it-004 SeminararbeitLocalCI %
```



CI_functions

```
143   # Set time stamp for starting of compile process
144   xnsTestingTimerStart=$(date +%s)
145
146   # Set arguments of Validate.py
147   if [ "$forceUpdateMode" = "true" ] ; then
        validationArguments="--no-clean --update --case-timings"
148     validationArguments="--no-clean --update --case-timings
          --input /code/xns-testing/Testcases"
149     logmsg "Launching xns-testing - update mode!"
150   else
        validationArguments="--no-clean --case-timings
            --allowed_to_tolViolate_input
            ${MAIN_DIR}/TestcasesWithNewResults"
151     validationArguments="--no-clean --case-timings
            --allowed_to_tolViolate_input
            ${MAIN_DIR}/TestcasesWithNewResults --input
            /code/xns-testing/Testcases"
152     logmsg "Launching xns-testing"
153   fi
154
      # Enter xns-testing
      cd $MAIN_DIR/xns-testing
155   # Enter ci-validate
156   cd $MAIN_DIR/ci-validate
157
158   # In daily build, various combinations of OPTIONS are
        tested.
159   if [ "$daily" = "true" ] ; then
160
161     logmsg "PLAIN XNS (in $buildType mode)"
```

Die dritte Fehlermeldung wies ebenfalls auf einen falschen Pfad der Testcases hin. Die Funktion zur Validierung der Testcases im Repo *ci-validate* verweist noch auf den alten, ungültigen Pfad. Auch hier war es erforderlich den Pfad zum Testcase-Verzeichnis anzupassen.

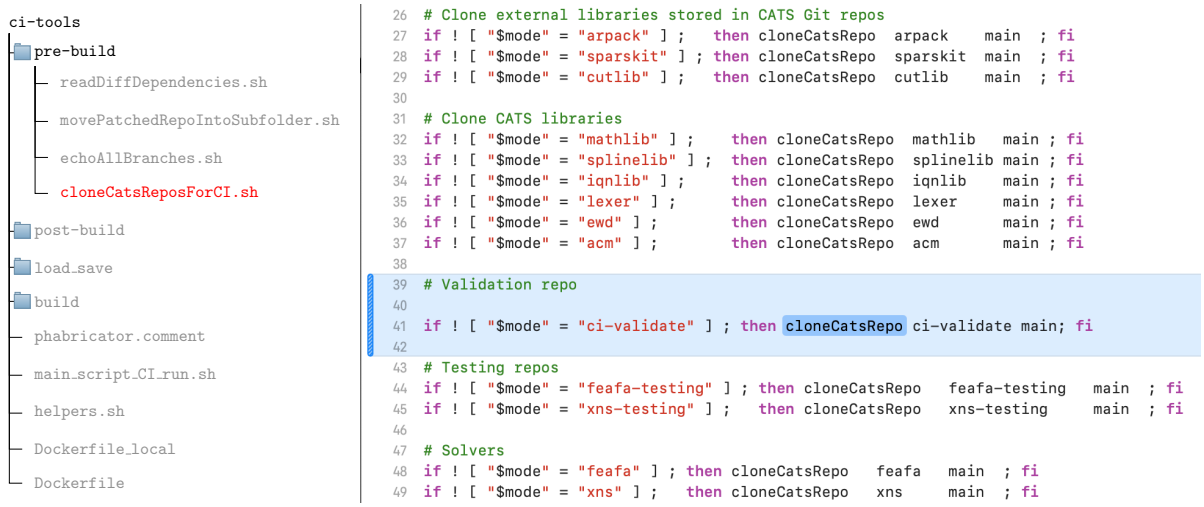
```
Traceback (most recent call last):
  File "./Validate.py", line 125, in <module>
    main()
  File "./Validate.py", line 121, in main
    status = ValidateBash.validate(args)
  File "/code/ci-validate/./tools/BASH_FRONTEND/ValidateBash.py", line 120, in validate
    testCase = tc.TestCaseBash(testCaseDir, args)
  File "/code/ci-validate/./tools/BASH_FRONTEND/TestCaseBash.py", line 27, in __init__
    parent.__init__(self,testCaseDir, arguments.xns[0], arguments)
  File "/code/ci-validate/./tools/TestCase.py", line 21, in __init__
    with open(osp.join(self.testCaseDir, 'DEPENDENCIES')) as f:
FileNotFoundError: [Errno 2] No such file or directory: '/code/ci-validate/Cases/1stLevel/AdvectionDiffusion/DEPENDENCIES'
================================================ Deleting local docker container ================================================
==
96e46c6f2fa6
96e46c6f2fa6
Untagged: localdocker:latest
Deleted: sha256:4988bf8d47ad4b4c6c96cda4f18f860cff615e97121c3246f1806ebced011e6e
================================================================================================================================
==
deniswolter@monitor-it-004 SeminararbeitLocalCI %
```

ci-validate after repo-split

- tools
  - testCoverage
  - BASH_FRONTEND
  - yaml2slurm.py
  - timings.py
  - test.py
  - defaulted_conf.yml
  - TestReport.py
  - TestCase.py
  - SingleValueInputRelativeDiff.py
  - README.md
  - L2ErrorCheck.py
  - ForceDiff2D.py
  - FluxDiff.py
  - FileCmp.py
  - Doxyfile
  - CheckOutputSurfaceIntegral.py
  - CheckError.py
  - Check.py
  - BinDiff.py
- FailedCasesLogs
- README.md
- submitValidate.sh
- setup.py
- Validate.py

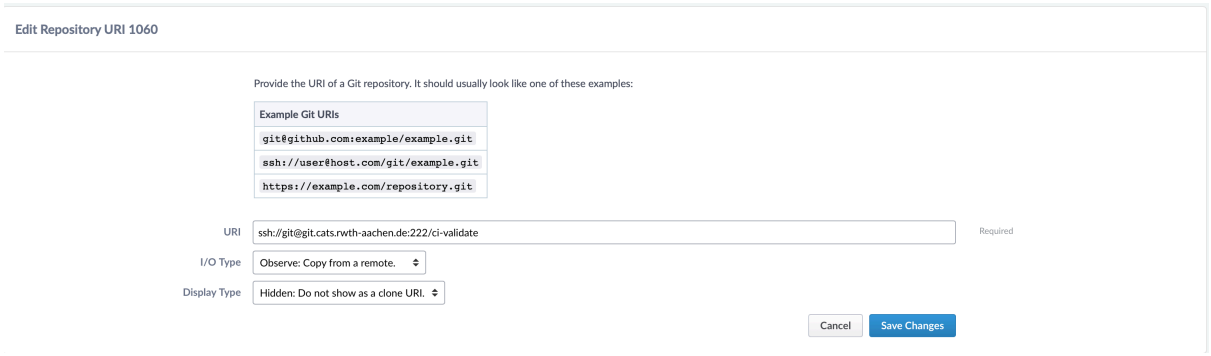
```python
import  os.path as osp
from    os import getcwd
import  subprocess as sp
import  timings as t


#This is the base class for TestCases. The BASH cases inherit from it.

class TestCase:

    _initialized = False

    def __init__(self, testCaseDir, xnsPath, args):
        cwd = getcwd()
        self.relativeDir = testCaseDir
        self.testCaseDir = osp.join("..", "xns-testing", testCaseDir)
        self.passed      = False
        self.active      = True
        self.xnsPath     = xnsPath
        self.allowedToTolViolate = False
        with open(osp.join(self.testCaseDir, 'DEPENDENCIES')) as f:
            self.dependencies   = f.read().splitlines()
        with open(osp.join(self.testCaseDir, 'TAGS')) as f:
            self.tags           = f.read().splitlines()
        with open(osp.join(self.testCaseDir, 'ARCHITECTURES')) as f:
            self.architectures  = f.read().splitlines()
        if args is not None:
            excludedDueDep      = self.hasDependencies(args.exclude)
            excludedDueTag      = self.hasTags(args.exclude)
            includedDueDep      = self.hasDependencies(args.include)
            includedDueTag      = self.hasTags(args.include)
            runDueArch          = ((args.architecture is None) or
                (self.hasArchitecture(args.architecture)))
            allowed = not (excludedDueDep or excludedDueTag)
            if len(args.include) > 0:
                allowed = allowed and (includedDueDep or includedDueTag)
            if not (allowed and runDueArch):
                self.active = False
            if args.time:
                path, case_name = osp.split(self.testCaseDir[:-1])
                self.timers = t.case_timer(case_name)
```

Während den Blackbox Tests fiel auch auf, dass das Lokale CI-System noch nicht auf Apple Geräten mit ARM-Prozessoren kompiliert werden konnte. Die Lösung bestand darin, eine Standardplattform für den Docker Container festzulegen. Wenn das Host-System ARM64 nutzt, aber Images ausgeführt werden, die nur für amd64 verfügbar sind, sorgt der Befehl: `export DOCKER_DEFAULT_PLATFORM=linux/amd64`
dafür, dass Docker automatisch die richtige Architektur verwendet. Da in Zukunft das Klonen des Repositories ci-validate für das CI-System unerlässlig ist, muss noch eine weitere Anpassung im Repository ci-tools erfolgen. Das CI-System wird vorbereitet, indem alle nötigen Repositories auf dem aktuellen Stand des Servers geklont werden. In diesem Fall muss das Klonen um das Repositoriy *ci-validate* erweitert werden.

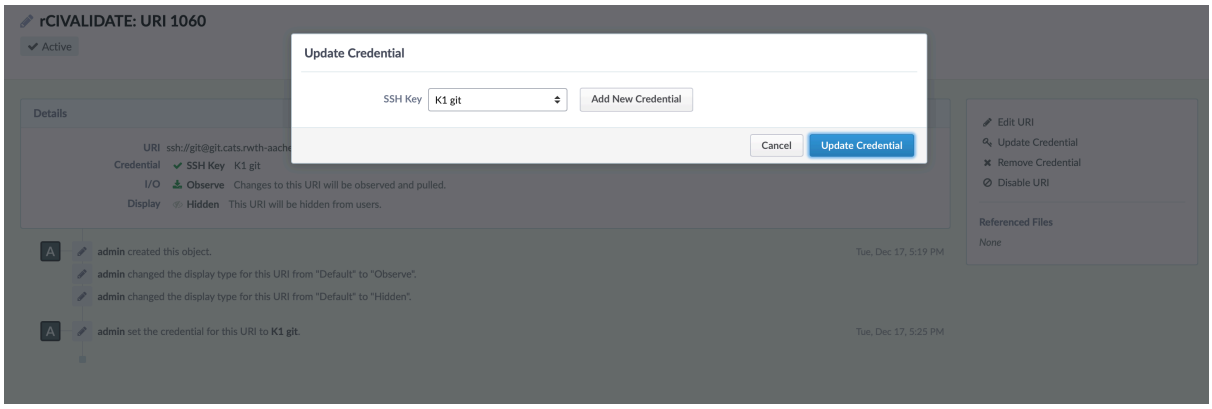


3.2 Erstellen eines Phabricator repos für ci-validate

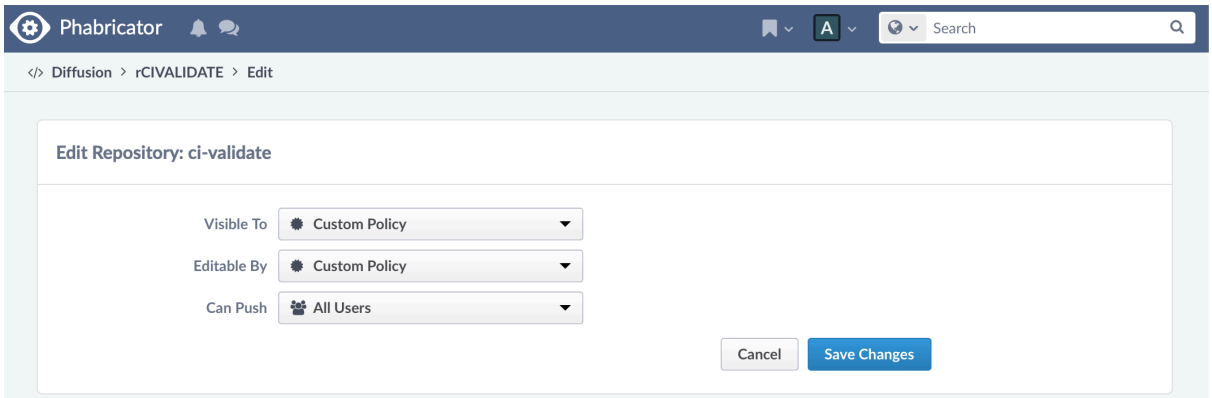
Damit für das Repository ci-validate in Zukunft auch eine Dokumentation in Phabricator möglich ist, muss das Repo dort zunächst aktiviert werden. Anschließend muss die URI, die auf das Gitolite Repository verweist, ergänzt werden, um die Verbindung zwischen Phabricator und dem Repository herzustellen.



Im Anschluss muss auch der ssh-key mit den entsprechenden Berechtigungen zum Klonen des Repositories hinzugefügt werden.



Der letzte Schritt in Phabricator besteht darin, die bisherigen Berechtigungen der CATS-Mitarbeiter aus Gitolite zu übertragen und die Accounts hinzuzufügen. Das stellt sicher, dass alle betroffenen Benutzer nach dem Landen des Differentials ihre Rechte beibehalten.



3.3 Einreichen der Revision in Phabricator

Die Revisions müssen voneinander abhängen. Zuerst wird eine Revision für ci-tools erstellt. Diese Revision hängt von den Änderungen in *xns-testing* ab und umgekehrt. Um diese Logik korrekt auf Phabricator zu übertragen wird der Test-Plan entsprechend angepasst.

Für die beiden Repositories werden als ersten Schritt separate commits erstellt, die oben genannten geänderte Strukturen beschreiben. Danach wird eine Revision von *xns-testing* mit `arc diff` erstellt.
Die Beschreibung der Revision beinhaltet dann alle relevanten Informationen, dazu gehören:

- eine Erklärung der Änderungen.
- ein Test-Plan, der beschreibt, wie die Änderungen validiert wurden.
- die relevanten Reviewer, die den Code prüfen sollen werden definiert.
- eine Abhängigkeit von anderen Diffs wird vorerst übersprungen.

Danach wird das Repository *ci-tools* aufgerufen. Mit `arc diff` wird hier ebenfalls eine neue Revision eingereicht. In der Beschreibung dieser Revision wird dann die Abhängigkeit im Text definiert: `Depends on D<Revision(ID der Revision) von xns-testing>` Zu den weiteren Beschreibungen gehören - wie in der zuvor erstellten Revision von *xns-testing* auch eine Erklärung der Änderungen sowie der Test-Plan. [8]

Verknüpfung der Abhängigkeiten in Phabricator prüfen
Sobald die beiden Revisiosn eingereicht wurden, melde ich mich in der Weboberfläche von Phabricator an. Dort kann ich meine eingereichten Revisions sehen und die Abhängigkeiten prüfen, bzw. auch die Abhängigkeit von *xns-testing* ergänzen.

Nach der Genehmigung der Revision kann *xns-testing* in den Hauptzweig gemerged werden. Danach folgt dann das Repo ci-tools.

Mit diesem Workflow wird sichergestellt, dass die Änderungen der Repos korrekt miteinander verbunden sind und keine Inkonsitenzen auftreten.

[8] sinngemäß entnommen aus einem cats.pages Tutorial – siehe Anhang S.32. Verfügbar unter: https://cats.pages.rwth-aachen.de/xns-documentation/html/workflow.html (letzter Zugriff: 20.12.2024)

3.4 Workflow nach dem Repo-Split

Der Split der Repositories wurde so umgesetzt, dass der Workflow und das CI-System ohne Unterbrechungen weiterfunktionieren. Die grundlegende Struktur, die in dieser Arbeit beschrieben wird, blieb dabei erhalten. Alle relevanten Repositories, die vom CI-System für die Testdurchläufe geklont werden, wurden um das Repository *ci-validate* ergänzt. Alle anderen Änderungen beschränken sich auf Pfade in den Skripten. Das Repository *ci-validate* wurde in Phabricator eingerichtet und mit denselben Zugriffsrechten wie *xns-testing* versehen. Dadurch ist ein nahtloser Übergang in das neue System möglich, ohne dass zusätzliche Nutzeranträge erforderlich sind.

Änderungen in einem Differential lösen weiterhin dieselben Test-Builds aus wie zuvor. Die Laufzeiten des CI-Systems haben sich durch die Einführung des zusätzlichen Repositories nur geringfügtig verlängert, während der Workflow insgesamt klarer und strukturierter wird. Entwickler, die neue Testcases hinzufügen möchten, können das jetzt im dedizierten Repository *xns-testing* vornehmen. Für Anpassungen in den Validierungsskripten steht hingegen das Repository *ci-validate* zur Verfügung.

Diese Aufteilung verbessert insbesondere die Lesbarkeit der Repository-Historie. Änderungen an Validierungsskripten und Änderungen an Testcases sind jetzt voneinander getrennt und nachvollziehbarer, was die Wartung und Weiterentwicklung deutlich erleichtert. Zudem ist das lokale Ausführen des CI-Systems nun auch auf Apple-Geräten mit ARM-Prozessor möglich.

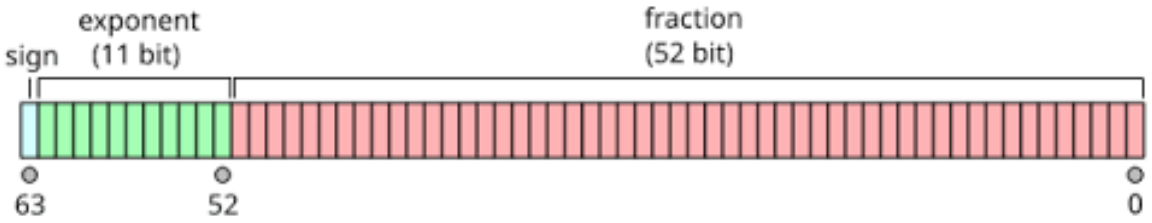
3.5 Analyse und Vergleich von Binärdateien: Einsatz und Nutzen von bindiff.py

Das Skript bindiff.py im Repo *ci-validate* wird verwendet, um Binärdateien zu vergleichen, die im MIXD-Output Format vorliegen. Es dient dazu sicherzustellen, dass nach den Änderungen am Code keine unzulässigen Abweichungen in den Ergebnissen auftreten, die außerhalb des Toleranzbereichs liegen. Das ist besonders für Testcases essentiell, die regelmäßig im CI-Prozess nach den Änderungen eines Diffs überprüft werden, um Fehler frühzeitig zu erkennen.
Neben den Toleranzabweichungen existieren auch Timer die ablaufen, wenn die Ausführung der Testcases einen vordefinierten Zeitschritt überschreiten. Dieser Timer wird in den einzelnen Testcases separat festgelegt und führt bei Überschreitung zu Timeouts.

Es werden explizit zwei Dateiformate für die Variablen in bindiff.py definiert:

i4 für 4-Byte-Integer (int32)
Dieser Dateityp wird für Ganzzahlen verwendet, ist also ideal für Operationen, bei denen die Genauigkeit der Ganzzahlen entscheidend ist und keine höhere Präzision erforderlich ist.

r8 für 8-Byte-Gleitkommazahlen (float64)
Hohe Präzision, aber längere Rechenaufwand. Gleitkommazahlen werden verwendet, um wissenschaftliche Berechnungen durchzuführen. R8 ist zwar nicht das genaueste Dateiformat für Berechnungen (das ist r16) aber bietet dafür eine gute Balance zwischen Präzision und Leistung. [9]



1 Bit für das Vorzeichen– gibt an, ob die Zahl positiv oder negativ ist.
11 Bit für den Exponenten – gibt an, wie stark die Zahl skaliert werden muss
52 Bit für die Mantisse – die eigentliche Zahlendarstellung

Die Mantisse besteht aus 52 Bit, plus der implizierten führenden 1 = 53 Bit.
53 Bit können eine Zahl im Binärsystem bis zu etwa 2^53 präzise darstellen, was in etwa 16 Dezimalstellen entspricht.

[9] Details zu floating points sinngemäß- und Abbildung entnommen aus:
https://en.wikipedia.org/wiki/Double-precision_floating-point_format
(letzter Zugriff 20.12.2024)

Der Vergleich der Reference-states erfolgt nach dem Einreichen des Differentials, indem die Differenz zwischen den beiden Arrays berechnet wird, die aus den Binärdateien geladen werden. Dazu wird die numpy Bibliothek verwendet, um die Dateien als Arrays zu interpretieren und die Differenz zwischen den beiden Arrays zu berechnen. Anschließend wird die Norm der Differenz ermittelt, was den Betrag der Abweichungen zwischen den beiden Dateien darstellt. Dieser Wert wird durch die Gesamtheit der Elemente in der Datei geteilt, um einen Durchschnittswert der Differenz zu erhalten.

$$res = \frac{||data1 - data2||}{data1.size}$$

```
50      def perform(self):
51
52          if not (exists(self.file1) and exists(self.file2)):
53              raise FileNotFoundError
54
55          data1 =  np.fromfile(self.file1, dtype=self.dt)
56          data2 =  np.fromfile(self.file2, dtype=self.dt)
57          try:
58              self.res = np.linalg.norm((data1 - data2))/data1.size
59              #raise ValueError
60          except ValueError as e:
61              self.ReportValueError(e)
62
63          if self.res > self.tol:
64              self.info += '\n Difference       : ' + str('{0:4e}'.format(self.res))
65              self.info += '\n Allowed tolerance: ' + str('{0:4e}'.format(self.tol))
66              self.info += '\n Failed'
67              raise ToleranceError('Tolerance Violation')
68          self.info = 'Passed'
```

Der resultierende Wert res wird dann mit dem vorgegebenen Toleranzwert (tol) vergleichen, der standardmäßig bei 0.0 liegt. Falls der berechnete Unterschied den Toleranzwert überschreitet, wird eine Fehlerbehandlung ausgelöst.

Es gibt mehrere potentielle Fehler, die während der Ausführung des CI-Systems auftauchen können:

- FileNotFoundError – wird ausgelöst, wenn einer der beiden zu vergleichenden Dateien nicht existiert.
- ValueError – wird ausgelöst, wenn die beiden Arrays aus den Binärdateien nicht kompatibel sind, z.B. unterschiedliche Größen haben.
- ToleranceError – wird ausgelöst, wenn die Differenz der beiden Dateien den festgelegten Toleranzwert überschreiten.

4. Zukunftsvision und Verbesserungsmöglichkeiten

Die Trennung der Repositories legt den Grundstein für eine flexiblere und skalierbare Test- und Validierungsstruktur. Künftige Entwicklungen können sowohl die Erweiterung der Testfallbibliothek als auch die Anwendung der Struktur auf weitere Simulationssoftware umfassen, um Effizienz und Flexibilität weiter zu steigern.

4.1 Erfahrungen und Herausforderungen nach der Umsetzung des Repo-Splits

Nach der Trennung der Repositories ist es nun deutlich einfacher, das Test-Repository unabhängig zu erweitern und neue Testcases hinzuzufügen. Dies ermöglicht eine systematische und gezielte Weiterentwicklung der Testfallbibliothek sowie die Erstellung neuer Testszenarien, ohne die Validierungsskripte zu beeinflussen.

Herausforderungen

- Änderungen in einem Repository können Abhängigkeiten im Anderen auslösen. Dies erfordert eine enge Zusammenarbeit und klare Kommunikation zwischen den beteiligten Teams, um Kompatibilitätsprobleme frühzeitig zu erkennen und zu beheben.
- Zwei getrennte Repositories erfordern mehr organisatorischen Aufwand bei der Wartung und regelmäßige Sicherstellung der Kompatibilität.
- Die Aktualisierungen von Jenkins sind nur erschwert möglich, da einige veraltete Addons im Einsatz sind. Auch Bugs in der Benutzeroberfläche treten in unregelmäßigen Abständen auf, die unter anderem das Einfügen oder Ausführen von Skripten und Git-Befehlen in den Build-Steps verhindern.
- Die Benutzerverwaltung von Gitolite ist aufwendig, da Änderungen und Berechtigungen manuell durch die IT für jedes Repository zugewiesen werden müssen. Im aktuellen Workflow werden wenige Gruppen genutzt und Berechtigungen werden in der Regel einzeln vergeben.

  Gitolite bietet im Vergleich zu GitLab weniger Flexibilität und erfordert häufigere manuelle Eingriffe.

Lösungsansätze

- schrittweise Migration auf GitLab
  Gitolite kann zunächst parallel zu Gitlab betrieben werden, um eine reibungslose Übergangsphase zu gewährleisten. Ein Jenkins-Job wird eingerichtet, der automatisch Code aus den Gitolite Repositories in Gitlab spiegelt, um Redundanzen zu minimieren und die Migration zu beschleunigen. Nach erfolgreicher Migration aller Repositories kann der Gitolite Server abgeschaltet werden, wodurch die Benutzerverwaltung und die Repository Verwaltung zukünftig effizienter gestaltet wird.
- Nach der Migration könnte das GitLab eigene CI-System Jenkins ersetzen und das weitere Nutzen der veralteten Addons verhindert werden.

4.2 Anwendbarkeit der Test-Struktur auf weitere Simulationssoftware

Die neu etablierte Teststruktur bietet Potential für eine breitere Anwendbarkeit auf weitere Simulationssoftware. Für das Repository feafa-testing, das ebenfalls Testcases und Validierungsskripte umfasst, könnte in Zukunft eine Integration mit den Validierungsskripten aus *ci-validate* erfolgen. Durch die Wiederverwendbarkeit der Skripte wird nicht nur der Entwicklungsaufwand reduziert, sondern auch eine einheitliche und konsistente Validierung über verschiedene Projekte hinweg ermöglicht.

Dies schafft die Grundlage für eine modulare und zentralisierte Testinfrastruktur, die flexibel an unterschiedliche Anforderungen weiterer Simulationssoftware angepasst werden kann. Solche Synergien fördern zudem die Effizienz und Qualitätssicherung im gesamten Entwicklungsprozess. Darüber hinaus besteht nun die Möglichkeit, das Validierungs-Repository in Zukunft als Open-Source-Projekt zu veröffentlichen, um die Entwicklung weiter zu fördern und weitere Entwickler einzubinden.

5. Fazit

Die Umsetzung des Repository-Splits von *xns-testing* hat sich als sinnvoller Schritt erwiesen, um die Modularität und Wartbarkeit des Systems nachhaltig zu verbessern. Durch die Trennung von Validierungsskripten und Testcases konnte eine übersichtliche Struktur geschaffen werden, die zukünftige Erweiterungen und Anpassungen erleichtert. Der neue Workflow integriert sich nahtlos in das bestehende CI-System. Ausblickend bietet die Modularisierung Potential für die Integration weiterer Simulationssoftware und die kontinuierliche Optimierung des Systems.

6. Anhang

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

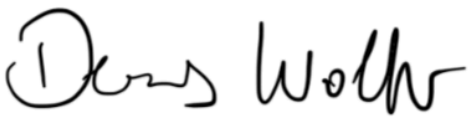
Aufspaltung von Validierungs- und Testprozessen in separate Repositories

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.
Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: Denis Wolter

Aachen, den 20.12.2024

Unterschrift der Studentin / des Studenten



Literaturverzeichnis

https://cats.pages.rwth-aachen.de/xns-documentation/html/index.html [1]
xns Overview siehe Anhang S.28

https://github.com/jenkinsci/phabricator-plugin [5]

https://cats.pages.rwth-aachen.de/xns-documentation/html/localCiGuide.html [6]
Reproducing the CI-System locally siehe Anhang S.29-31

https://alexey-anufriev.com/blog/split-git-repository/ [7]

https://cats.pages.rwth-aachen.de/xns-documentation/html/workflow.html [8]
Advanced Workflow For Differentials siehe Anhang S.32

https://en.wikipedia.org/wiki/Double-precision_floating-point_format [9]

https://www.techtarget.com/searchsoftwarequality/definition/Jenkins#:~:text=Jenkins%20is%20an%20open%20source,CI%2FCD%20workflows%20called%20pipelines.
https://de.wikipedia.org/wiki/Phabricator
https://secure.phabricator.com/book/phabricator/article/introduction/
https://secure.phabricator.com/book/phabricator/article/harbormaster/
https://en.wikipedia.org/wiki/Double-precision_floating-point_format

Abbildungsverzeichnis

https://mcuoneclipse.com/wp-content/uploads/2023/10/continuous_integration.jpg [2]

https://graphite.dev/blog/phabricator-code-review [3]

https://de.fiverr.com/azuredev/setup-ci-cd-pipelines-using-jenkins [4]

https://www.pngegg.com/en/png-wgyyw
https://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg

Skript bindiff.py

```
#! /usr/bin/env python3.6

# At least Python 3.6 is required because of the need fo subprocess.run
method with the timeout.

"""
Check to compare two binary data files (MIXD output format)
"""

from    Check       import Check
from    CheckError  import CheckError, FileNotFoundError, ToleranceError
from    os.path     import exists
import  numpy as np

"""
Extension to base class to implement a binary comparison of two files
"""
class BinDiff(Check):

    file1=''
    file2=''
    tol = 0.0
    res = np.nan
    dt = np.dtype('>f8')
    info = ''

    """
    @brief constructor
    @params file1 Reference or patch-applied results file
    @params file2 Results file to compare against
    @params tol Chosen tolerance
    @params dt Data type. Use numpy syntax, e.g. >i4  for int or >f8 for
dp-real
    """
    def __init__(self, file1, file2, tol=None, dt=None):
        self.name  = 'BinDiff'

        self.file1 = file1
        self.file2 = file2
        self.tol   = float(tol)
        if dt == 'i4':
            self.dt = np.dtype('>i4')
        elif dt == 'r8':
            self.dt = np.dtype('>f8')
        else:
            self.ReportInputError('Invalid data type \'' + dt +'\'.
Expected \'i4\' or \'r8\'')
        self.info  = self.name + ' ' + dt + ' between File 1 and File 2' +
'\n File 1: ' + self.file1 + '\n File 2: ' + self.file2

    """
    @brief Check driver
    """
    def perform(self):

        if not (exists(self.file1) and exists(self.file2)):
```

```
            raise FileNotFoundError

        data1 =  np.fromfile(self.file1, dtype=self.dt)
        data2 =  np.fromfile(self.file2, dtype=self.dt)
        try:
            self.res = np.linalg.norm((data1 - data2))/data1.size
            #raise ValueError
        except ValueError as e:
            self.ReportValueError(e)

        if self.res > self.tol:
            self.info += '\n Difference       : ' +
str('{0:4e}'.format(self.res))
            self.info += '\n Allowed tolerance: ' +
str('{0:4e}'.format(self.tol))
            self.info += '\n Failed'
            raise ToleranceError('Tolerance Violation')
        self.info = 'Passed'

"""
@brief: Stand-alone driver => Execute test if called as stand-alone
script, e.g.
during debugging.
"""
if __name__ == "__main__":
    import argparse
    import numpy
    import sys

    parser = argparse.ArgumentParser(description='Binary diff')
    parser.add_argument('DiffFile', type=argparse.FileType('r'), nargs=2,
help='This are the input files for the diff.')
    parser.add_argument('-t', '--filetype', choices=['r8', 'i4'], type=str,
default='r8', help='The file type.')
    parser.add_argument('-e', '--eps', type=float, default=0.0,
help='Allowed tolerance.')
    args = parser.parse_args()

    bd = BinDiff(args.DiffFile[0].name, args.DiffFile[1].name, args.eps,
args.filetype)

    try:
        print(bd.info)
        print(' Tolerance:  ' + str('{0:4e}'.format(bd.tol)))
        bd.perform()
        print(' Difference: ' + str('{0:4e}'.format(bd.res)))
    except CheckError as e:
        print(' Difference: ' + str('{0:4e}'.format(bd.res)))
        print(str(e))
        print('Failed')
        sys.exit(1)
    else:
        print('Passed')
        sys.exit(0)
```

Overview

[…] XNS is the CATS inhouse **finite element flow simulation** program. Although there are some myths about the naming, the story confirmed by M. Behr is the most plausible. Originally the code was developed for UNIX machines on which no file extensions were used in the early days of XNS development. In order to distinguish between different file types, single letters were used as pre- or postfixes. And the prefix X stood here for executable files. NS stands for Navier-Stokes equations, which were exclusively solved in the original version. The resulting name XNS was not changed in the further development, although meanwhile many other equation systems can be solved. […]

Reproducing the CI System Locally

The goal of the CATS CI system is to protect code functionality by running automatic checks, i.e., compiling and test cases. These tests are executed on a virtual machine, called **jenkins**.

To avoid long queuing times on jenkins and because of the simpler workflow, it is often useful to locally run these tests before updating your differential. On the one hand, however, using the reference state of XNS to update the solutions in `xns-testing` before comparing them to your differential state can be quite cumbersome; on the other hand, due to slight differences in the system environment, the CI system sometimes throws errors that are not reproduceable on the cluster.

Therefore, this guide explains how to locally execute the CI system in one command in the same Docker container jenkins is using. This allows you to reproduce the CI system's behaviour.

NOTE: The new CI system no longer produces the reference solutioons in every run. Instead, the solutions produced in the daily build are stored in the (hidden) Git repository 'ci-reference'. Since these results are computed within the Docker container, the comparison only works inside this container. If you run your cases outside of it, e.g., on the cluster, you are bound to have deviations due to different computer architecture, library states, environment flags, and so on. In that case, you should first checkout the `reference` branch of xns and the libraries, compile it and produce "your own" reference results.

Some comments about Docker

The basic idea of Docker is to provide containers that always contain the exact same system, environment variables etc., no matter on which machine it is used. For example, the CI system builds and runs all test cases on a virtual Ubuntu System inside a Docker container. This way, the test system is independent from software updates etc on the machine it is running on. Instead, every update of the OS, compilers, etc. has to be done explicitly via changing the Docker container's configuration file. Moreover, it means that we can run the same tests on our own machine as long as we have Docker available...

For detailed information about Docker and how it works, please look into the official documentation.

Requirements

Here's what you need:

1. **Docker** has to be installed on your desktop. It can be found for example in the Managed Software Center. Note that as of now, Docker is not available on the RWTH cluster.
2. Access to the CATS software packages on gitolite, including **ci-tools** and **ci-reference**. If you don't have all required access rights, ask IT for it.
3. Obviously, your system will require **git**.

Run CI System

Preparation

- Create a directory you want to execute the CI in, e.g., `LocalCI` and enter it.
- Clone the repository `ci-tools` from the CATS gitolite server.
- Set the `CI_TOOLS` environmental variable by: `export CI_TOOLS=./ci-tools`

Execution

When locally running the CI system, the execution procedure is divided into two parts:

1. Clone Libraries and Apply Your Changes

Run `./ci-tools/build/runJenkinsBuild.sh ci-tools --local` to prepare the CI system. This is the same script jenkins calls when you create or update a differential. It starts by cloning all required repositories from the CATS Gitolite server, like xns, xns-testing, ewd, splinelib, etc. They will all be on the latest state, i.e., `origin/main`.

In case of local execution, the script will stop here. This is to give you the opportunity to apply all changes you want to test in the CI run. For example, you may want to apply some differential to XNS, deactivate some test cases in xns-testing etc.

2. Run CI System in Docker Container

Once you applied your changes, you can build and execute the docker container. It will run the CI system for the local state of your repositories, therefore including all changes you applied in step 1. It is best to use the provided script for that:

`./ci-tools/build/runDockerLocally.sh`

Comments, Tipps & Tricks

- Running the Docker container **does not** change the files and folders outside the container. Therefore, you can run the testing in docker multiple times without cloning the repos over and over again.
- For debugging, it can be useful to run the docker container in interactive mode: It just enters the container but does not start the building. You can browse through the files, use custom build commands, run certain test cases etc. If you want to start the standard CI build process, execute `./main_script_CI_run.sh`. To run Docker's interactive mode use: `./ci-tools/build/runDockerLocally.sh –interactive`

Advanced Workflows for Differentials

This page lists workflows that handle testing and landing of differentials for complicated cases, like differentials that depend on each others, differentials that change the solution of some test cases etc...

Case 1: Differential A of Repo X depends on Differential B of Repo Y

Testing

Without any further ado, both builds will be failing, but here's how to handle this:

- In differential A, add a file arcpatch.sh with the content:
- cd <Y>
- arc patch D<B>
- cd -
    - For example, in case of a xns differential depending on D3000 of xns-testing:
    - cd xns-testing
    - arc patch D3000
    - cd -
- With this, differential A will be tested in the CI system against differential B.
- The build in B will still be failing, but you can use the build in A to tell whether the combination A&B is working.

Landing

Most important: arcpatch.sh is only for testing within a differential. **WE NEVER WANT TO LAND IT!**

Also, the two differentials should both be accepted before going on, since they have to be landed together via the following steps:

- After build was green with arcpatch.sh, remove it from the differential A.
    - The build will be red again, but that is fine - you already checked that it will be working if both A and B are landed.
- The next two steps have to be done right after each other, since the first one temporarily breaks the CI system:
    - Land differential B on repo Y.
    - Land differential A on repo X.
- Now that both differentials are landed, the CI system should be green again - since you tested the combination A&B.