HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

DIPLOMARBEIT

Herr
Lars Lechner

Untersuchung der Leistungsfähigkeit des Generiertools Build Forge von IBM in großen Softwareentwicklungsprojekten am Beispiel des Drive Technologie Bereichs der Siemens AG

2010

Fakultät **Informationstechnik & Elektrotechnik**

DIPLOMARBEIT

Untersuchung der Leistungsfähigkeit des Generiertools Build Forge von IBM in großen Softwareentwicklungsprojekten am Beispiel des Drive Technologie Bereichs der Siemens AG

Autor:
Lars Lechner

Studiengang:
Technische Informatik

Seminargruppe:
KT07w1

Erstprüfer:
Prof. Dr.-Ing. Uwe Schneider

Zweitprüfer:
Dipl.-Ing. (FH) Klaus Koller

Mittweida, August 2010

Bibliografische Angaben

Lechner, Lars: Untersuchung der Leistungsfähigkeit des Generiertools Build Forge von IBM in großen Softwareentwicklungsprojekten am Beispiel des Drive Technologie Bereichs der Siemens AG, 107 Seiten, 36 Abbildungen, 7 Tabellen, Hochschule Mittweida (FH), Fakultät Informationstechnik & Elektrotechnik

Diplomarbeit, 2010

Satz: LaTeX

Referat

Ziel der Diplomarbeit ist die Untersuchung von Build Forge für die Automatisierung von Generierungen durch das Konfigurationsmanagement (KM). Zu diesem Zweck werden in dieser Arbeit zuerst Grundkenntnisse über das Softwarekonfigurationsmanagement und dessen Hauptaufgaben in Softwareprojekten vermittelt. Mit Hilfe dieser Kenntnisse und dem bei Siemens DT im SINAMICS-Projekt eingesetzten Generierautomaten werden die Anforderungen an ein Automatisierungswerkzeug für Generierungen herausgearbeitet. Um die Tauglichkeit von Build Forge in großen Softwareprojekten zu verifizieren, werden diese Anforderungen mit Hilfe eines Testprojekts einzeln auf die Unterstützung durch Build Forge geprüft. Neben der technischen Betrachtung der Einsatztauglichkeit von Build Forge in großen Softwareprojekten soll auch die wirtschaftliche Seite betrachtet werden. Dazu erfolgt eine Analyse der Kosten einer Eigenentwicklung eines Generierwerkzeugs und den Kosten eines Zukaufs von Build Forge für Projekte. Mit den Ergebnissen aus der technischen und wirtschaftlichen Betrachtung wird anschließend ein Fazit über den Einsatz von Build Forge in neuen und bereits bestehenden Softwareprojekten gezogen. Ein Einsatz in bestehenden Projekten ist z.B. dann sinnvoll, um eine Vereinheitlichung über mehrere Projekte zu erreichen. Für eine endgültige Entscheidung über den Einsatz von Build Forge in Softwareprojekten müssen jedoch noch weitergehende Untersuchungen, wie z. B. die zeitgesteuerte Generierung, erfolgen. Im Ausblick werden hierzu die wichtigsten noch zu untersuchenden Themen kurz vorgestellt und auch Alternativen zu Build Forge aufgezeigt.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

IV. Verzeichnis der cmdata-Dateien

V. Abkürzungsverzeichnis

BF	Build Forge
CC	ClearCase
CF	Compact Flash
CM	Configuration Management / Konfigurationsmanagement (Methode)
DB	Datenbank
DT	Drive Technology
GA	Generierautomat
IDE	Integrated Development Environment / integrierte Entwicklungsumgebung
IEEE	Institute of Electrical and Electronics Engineers
KM	Konfigurationsmanagement (Abteilung)
Komp	Komponente
MC	Motion Control
MM	Mannmonat
MS	Mannstunde
MT	Manntag
PKM	Produktkonfigurationsmanagement (Abteilung)
SCM	Software Configuration Management (Methode)
SKM	Softwarekonfigurationsmanagement (Abteilung)
STZ	Systemtestzentrum
tcsh	Tenex-C-Shell
VOB	Versioned Object Database

VI. Danksagung

Die vorliegende Diplomarbeit wurde von Februar 2010 bis August 2010 innerhalb der Siemens AG geschrieben.

Ich möchte mich hiermit ausdrücklich bei allen bedanken, die mich bei der Erstellung der Diplomarbeit unterstützt haben. Hier ist besonders Herr G. Ströbel zu nennen, ohne dessen Hilfe die Installation von Build Forge sicher nicht möglich gewesen wäre. Danke auch an Herrn P. Nikitka, der mir bei Fragen zu unserer Entwicklungsumgebung jederzeit Rede und Antwort gestanden hat. Ein herzlicher Dank geht auch an Frau H. Hofbauer, die mir mehrmals die notwendigen Testlizenzen für den Betrieb von Build Forge zur Verfügung gestellt hat. Dann möchte ich mich noch bei der Siemens AG und der evosoft GmbH bedanken, die mir Zeit für das Erstellen der Diplomarbeit zur Verfügung gestellt haben. Ein großer Dank geht auch an Herrn Prof. Dr. U. Schneider und Herrn K. Koller für die Betreuung, Durchsicht und konstruktiven Vorschläge bei der Ausarbeitung der Diplomarbeit. Abschließend möchte ich mich noch bei allen Korrekturlesern der Diplomarbeit bedanken.

1 Einleitung

1.1 Motivation

Da in jedem großen Softwareprojekt früher oder später die Notwendigkeit nach der Automatisierung der Erzeugung von Software besteht, entwickelt sich in diesen Projekten mit der Zeit auch eine selbstgeschriebene Lösung. An dieser Stelle setzt Build Forge an. So bietet Build Forge bereits Lösungen für Fragestellungen, die sich in großen Softwareprojekten im Laufe der Zeit ergeben. Hierunter fällt z. B. die parallele Generierung mehrerer Softwarekomponenten. Um jedoch den Nutzen und die Tauglichkeit von Build Forge in großen Softwareprojekten beurteilen zu können, ist es notwendig, die von IBM getroffenen Aussagen unter realen Bedingungen zu verifizieren. Neben der Untersuchung der Einsatztauglichkeit von Build Forge in neuen Softwareprojekten gibt es innerhalb von SIEMENS MC RD Gedanken die Generierabläufe der bestehenden großen Softwareprojekte zu harmonisieren. Dies wiederum würde zu frei werdenden Ressourcen führen, da nicht mehr „n“ Generierskripts, sondern nur noch Build Forge als Werkzeug gepflegt werden müsste. Ein weiterer Vorteil dieser Homogenisierung liegt darin, dass Mitarbeiter des Konfigurationsmanagements eines Projekts schneller in ein anderes Projekt eingearbeitet werden können, da dieses analog arbeitet. Wie für den Einsatz in neuen Projekte muss auch bei der Homogenisierung existierender Projekte Build Forge bestimmte Kriterien erfüllen, die ebenso im Rahmen dieser Diplomarbeit untersucht werden.
Neben dem Wunsch der Harmonisierung bereits bestehender Projekte der Siemens AG gibt es auch immer wieder Gedanken darüber, ob bei neuen Softwareprojekten eine Eigenlösung oder doch ein bereits fertiges Produkt verwendet werden soll. Die Frage nach dem Einsatz eines fertigen Produkts ist auch insbesondere für Firmen, wie die evosoft GmbH, interessant, die Softwarekonfigurationsmanagement als eigenes Produkt anbieten wollen. So können diese mit dem Einsatz eines Frameworks, wie z.B. Build Forge, für das Softwarekonfigurationsmanagement die eigenen Kosten gering halten, da die Anschaffungskosten nur ein einziges mal anfallen und neue Kunden über den Zukauf von Lizenzen bedient werden können. Somit muss diese Firma nicht für jeden einzelnen Kunden mit viel Zeitaufwand eine eigene Lösung entwickeln, sondern kann mit eigenen darin gut geschulten Mitarbeitern schnell eine funktionierende Lösung auf Basis von Build Forge anbieten.

Die Untersuchungen von Build Forge erfolgten im Rahmen des SINAMICS[1]-Projekts der Siemens AG, wobei die Untersuchungsergebnisse für den Einsatz von Build Forge auf neue Softwareprojekte übertragbar sind. Daher können die Ergebnisse dieser Diplomarbeit zum einen als allgemeine Entscheidungshilfe für die Ablösung existierender

[1] Softwareprojekt bei der Siemens AG, in dessen Rahmen Motorsteuerungen entwickelt werden

Systeme und ebenso für den Einsatz als Generierwerkzeug in neuen Projekten verwendet werden.

1.2 Kapitelübersicht

Kapitel 2 gibt einen Überblick über das Softwarekonfigurationsmanagement und erklärt dessen wichtigsten Begriffe anhand des quasi-Standards ClearCase als Versionsverwaltungswerkzeug.

Kapitel 3 stellt das zu untersuchende Werkzeug Build Forge von IBM vor. So werden dessen Fähigkeiten anhand von Informationen, die IBM veröffentlicht hat, vorgestellt und noch einmal darauf eingegangen, warum Build Forge untersucht wird. Da Build Forge als möglicher Ersatz für bereits bestehende Generierskripts verwendet werden soll, werden in diesem Kapitel auch die Unterschiede zwischen Build Forge und den aktuell eingesetzten Werkzeugen herausgearbeitet.

Kapitel 4 stellt die Kriterien vor, die Build Forge für eine erfolgreiche Verwendung auf jeden Fall erfüllen muss. Da diese Diplomarbeit innerhalb von SIEMENS MC RD erstellt wird, gibt es neben den allgemein gültigen Kriterien, wie z. B. der Parallelgenerierung mehrerer Softwarekomponenten, auch spezielle Projektanforderung wie die Unterstützung mehrerer Generierebenen.

Kapitel 5 geht zum einen auf die Installation von Build Forge ein. Zum anderen wird das erstellte Testprojekt vorgestellt. Weiterhin wird kurz erläutert, welche Punkte bei einem Einsatz von Build Forge berücksichtigt werden sollten.

Kapitel 6 beschreibt die ausgeführten Tests und geht auf deren Ergebnisse ein.

Kapitel 7 zeigt den Preis von Build Forge und vergleicht dessen Kosten mit denen einer Eigenentwicklung.

Kapitel 8 fasst die Ergebnisse der Diplomarbeit aus den durchgeführten Tests und der Kostenabschätzung zusammen und gibt eine Empfehlung bzgl. des Einsatzes von Build Forge für den betrachteten Fall ab. Auch gibt dieses Kapitel einen Ausblick über noch durchzuführende Tests und einen Hinweis auf andere Generierwerkzeuge.

Anhang A stellt die in Kapitel 3.3.1.1 vorgestellten cmdata-Dateien dar und erläutert die Bedeutung der wichtigsten Felder innerhalb der Code-Fragmente.

Anhang B zeigt ausschnittsweise die Laufzeit einer SINAMICS-Generierung mit dem aktuellen Generierautomaten.

Anhang C zeigt die Laufzeit einer Generierung des Testprojekts mit dem aktuellen Generierautomaten. Diese Tabelle wird automatisch durch den Generierautomaten erzeugt.

Anhang D ist ein mit Build Forge erstelltes Diagramm, welches die Dauer einer Generierung mit der Fähigkeit der Parallelgenerierung von Build Forge aufzeigt. Daneben werden auch die Startzeiten der einzelnen Projekte dargestellt.

Anhang E zeigt ein mit Build Forge erzeugtes Balkendiagramm, das die Dauer der einzelnen Build Forge Projekte darstellt.

Anhang F zeigt die parallele Abarbeitung und Synchronisation der Build Forge Archivierungsschritte.

Anhang G zeigt die parallele Abarbeitung und Synchronisation der Build Forge Erzeugungsschritte.

2 Softwarekonfigurationsmanagement

Um das Softwarekonfigurationsmanagement besser zu verstehen, wird zuerst einmal die Bedeutung von Software und Konfigurationsmanagement einzeln betrachtet. Durch diese Aufteilung wird deutlich, dass Konfigurationsmanagement nicht nur in der Softwareentwicklung seinen Platz hat, sondern auch in anderen Bereichen eingesetzt wird. Es stellt sich also die Frage was ist Konfigurationsmanagement?

2.1 Begriffsdefinition Konfigurationsmanagement

Auf oben gestellte Frage gibt Christoph Reichenberger folgende Antwort:

> „Unter Konfigurationsmanagement versteht man im allgemeinen die Disziplin, die sich mit Methoden und Werkzeugen zur Unterstützung der Entwicklung komplexer Systeme beschäftigt.“ [Rei94, S. 2]

Beim Softwarekonfigurationsmanagement handelt es sich um einen Spezialfall des Konfigurationsmanagements. Es hilft und unterstützt den Entwickler durch definierte Prozesse in der Softwareentwicklung. Dieser definierte Prozess wird mit seinem Vorgehen, den zu verwendenden Werkzeugen und Abhängigkeiten zu anderen Abteilungen durch den Softwarekonfigurationsmanagementplan beschrieben. Der Aufbau und Inhalt eines Konfigurationsmanagementplans wird durch den IEEE-Standard (IEEE Std 828-2005) festgelegt. In dessen Vorwort wird folgendes über das Softwarekonfigurationsmanagement (SCM) geschrieben:

> „SCM constitutes good engineering practice for all software projects, whether phased development, rapid prototyping, or ongoing maintenance. It enhances the reliability and qualitiy of software by
>
> - Providing a structure for identifying and controlling documentation, code, interfaces, and databases to support all life cycles phases
> - Supporting a chosen develepoment/maintenance methodology that fits the requirements, standards, policies, organization, and management philosophy
> - Producing management and product information concerning the status of baselines, change control, tests, release, audits, etc.“
>
> [iee05, S. iii]

Durch einen standardisierten Ablauf und den Einsatz geeigneter Werkzeuge bei der Softwareentwicklung wird sichergestellt, dass Software eindeutig identifizierbar und reproduzierbar ist. So wird durch die Einhaltung des im Softwarekonfigurationsmanage-

mentplan definierten Prozesses auf Dauer die Qualität der Software sichergestellt und gesteigert.
Ist im weiteren Verlauf dieser Diplomarbeit von CM die Rede, ist damit die Methode des Configuration Managements gemeint, während KM als Abkürzung für das Konfigurationsmanagement als Abteilung verwendet wird. Da nun bereits der Begriff des Softwarekonfigurationsmanagements gefallen ist, wird dieser im nächsten Abschnitt ausführlich erläutert.

2.2 Softwarekonfigurationsmanagement

Was ist die Hauptaufgabe des Softwarekonfigurationsmanagement? Einfach gesagt, geht es um die Unterstützung des Softwareentwicklungsprozesses durch Methoden aus dem Konfigurationsmanagement. Diese Unterstützung der Entwicklung besteht aus mehreren wichtigen Aufgaben, die im folgenden Abschnitt vorgestellt werden.

2.2.1 Aufgaben

Das Softwarekonfigurationsmanagement muss als Hauptaufgabe sicher stellen, dass die **Reproduzierbarkeit** und **Nachvollziehbarkeit** von Softwareständen gewährleistet ist. Des Weiteren besteht in großen und komplexen Softwareprojekten noch die Anforderung, dass mehrere Entwickler die gleichen Daten bearbeiten können und auch Dateien anderer Entwickler sehen. Zu den weiteren Aufgaben des Softwarekonfigurationsmanagements gehört das Bereitstellen lauffähiger und reproduzierbarer Softwarestände an

- die Entwicklung
- andere Softwareprojekte
- den Integrationstest für die Sicherstellung der Funktionalität
- die Hardwareentwicklung für Baugruppentests
- die Produktion

Neben der funktionalen Weiterentwicklung der Software erfolgt innerhalb der Projekte auch die Wartung und die Fehlerbehebung bereits freigegebener Software. Somit muss das Softwarekonfigurationsmanagement in der Lage sein, unterschiedliche Freigabeversionen zu verwalten und die Entwicklung mehrerer Softwareversionen zu unterstützen.

Für all diese oben genannten Probleme und Aufgaben muss das vom Softwarekonfigurationsmanagement verwendete Werkzeug eine Lösung bieten. Neben dem bereitstellenden Teil arbeitet das Softwarekonfigurationsmanagement auch eng mit der Infrastruktur zusammen, die wichtige Dienste, wie z. B. die Pflege der Generierserver, anbietet.
Um die Reproduzierbarkeit und Nachvollziehbarkeit von Softwareständen zu gewährleisten, wird bei großen und komplexen Softwareprojekten die Methode des Softwarekonfigurationsmanagements eingesetzt.

2.2.2 Definition

Unter dem Begriff Softwarekonfigurationsmanagement werden zwei Bereiche zusammengefasst. Zum einen handelt es sich um das Softwarekonfigurationsmanagement selbst sowie das Softwareversionsmanagement.
Unter **Softwarekonfigurationsmanagement** im eigentlichen Sinn versteht man die Bereitstellung von Informationen über die Erzeugung einer Software. Dazu gehört die Kenntnis über den verwendeten Compiler und mit welchen Optionen ein bestimmter Softwarestand erzeugt wurde. Hierunter fallen auch Abhängigkeiten der Komponenten untereinander. Als alternative Bezeichnung von Softwarekonfigurationsmanagement wird häufig der Begriff Buildmanagement verwendet. Mit Hilfe des Softwarekonfigurationsmanagements wird das Problem der Reproduzierbarkeit von Softwareständen gelöst.
Softwareversionsmanagement bedeutet die Bereitstellung jeder erzeugten und archivierten Version für die Entwickler zu jedem Zeitpunkt der Entwicklung. Dies ermöglicht die Weiterentwicklung der Software, wie auch die Fehlerbehebung auf alten Ständen.
Zum Softwareversionsmanagement gehört auch die Weitergabe und das Bereitstellen von abgeschlossenen Softwareständen an andere Projekte oder an das Produkt-KM.
Es wird deutlich, dass KM neben der Zusammenarbeit mit der Entwicklung im eigenen Projekt auch Schnittstellen zu anderen Bereichen besitzt, welche in der folgenden Abb. 2.1 auf der nächsten Seite dargestellt werden.

Nachfolgend werden potentielle Abhängigkeiten einer Konfigurationsmanagement-Abteilung zu anderen Bereichen erläutert.

Mögliche Schnittstellen von KM sind:

SKM: Ist es erforderlich, dass ein anderes Projekt Komponenten aus einem Softwarestand benötigt, erfolgt die Übergabe an das jeweils projektspezifische Software-KM.

PKM: Produkt-KM erhält von SKM eine freigegebene Softwareversion und stellt diese der Fertigung zur Verfügung. PKM muss nicht als eigenständige Abteilung existieren, sondern diese Rolle kann z. B. auch von SKM ausgefüllt werden.

Administratoren: Die Administration stellt die Infrastruktur für KM zur Verfügung. Darunter fällt unter anderem die Beschaffung und Pflege von Generierservern.

Entwickler: Die Softwareentwickler erfahren durch KM Betreuung und Unterstützung bei Problemen mit der Versionsverwaltungssoftware. Weiterhin gehört zu dieser Schnittstelle auch die Abstimmung von Terminen der Softwareversionen zwischen dem Softwaremanagement und KM.

STZ: Das Systemtestzentrum[2] erhält reproduzierbare Softwareversionen zum Testen. Das Ziel der Abgabe an das STZ ist das Erfüllen der Anforderung des STZ an die Software, so dass diese freigegeben werden kann. Dem STZ ist es möglich Fehler gegen einen abgegeben Softwarestand zu melden, der dann in einem neuen Stand behoben wird.

Fertigung: Nach Absprache der Projektleitung mit dem STZ wird die Software freigeben und über PKM der Fertigung in einer Datenbank zur Verfügung gestellt. Daraufhin wird die freigegeben Software im Fertigungsprozess verwendet.

Um zu verdeutlichen, dass kein festgelegtes Schema für den Aufbau innerhalb eines Projektes existiert, wird in Projekt 2 der Abb. 2.1 die Aufgabe des Produkt- und Softwarekonfigurationsmanagement vom gleichen Personenkreis ausgeführt.

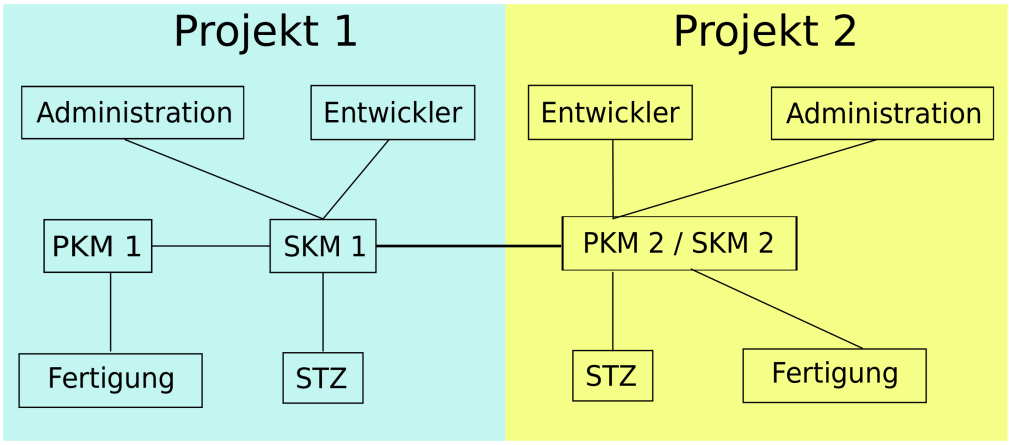


Abbildung 2.1: Schnittstellen von KM

Neben der Softwareversionsverwaltung und dem Softwarekonfigurationsmanagement hat KM noch andere Aufgaben zu erfüllen. Dazu gehört das Erstellen von Metriken, um dem Management Kennzahlen über den Fortgang der Entwicklung im Projekt und dessen Komplexität bereitzustellen. Nachdem nun die beiden Teile des Softwarekonfigurationsmanagements vorgestellt wurden, muss noch die Frage beantwortet werden,

[2] der Systemtest testet das Zusammenspiel mit der Hardware

für welche Probleme ein CM-System eine Lösung anbieten muss. Neben den bereits erwähnten Hauptfunktionen Reproduzierbarkeit und Nachvollziehbarkeit, muss ein Softwarekonfigurationsmanagementsystem Lösungen für folgende Probleme anbieten:

Sicherheit: Es muss ein Zugriffsschutz für nicht autorisierte Anwender und die Möglichkeit der Wiederherstellung einer funktionalen Umgebung nach einer kritischen Fehlerbehebung bestehen.

Stabilität: Es muss eine stabile Entwicklungsumgebung bereitgestellt werden, in der jeder Entwickler Änderungen einbringen kann, ohne Einfluss auf andere Entwickler auszuüben.

Parallelentwicklung: Mehrere Entwickler müssen die gleichen Dateien bearbeiten können, ohne sich gegenseitig zu beeinflussen.

Nachvollziehbarkeit: Das System muss Informationen vorhalten, zu welchem Zeitpunkt was für eine Änderung von wem vorgenommen wurde.

Reproduzierbarkeit: Alte Konfigurationen müssen jederzeit wieder hergestellt und generiert werden können.

Skalierbarkeit: Da ein Softwareprojekt in der Regel klein anfängt und mit der Zeit wächst, muss das SCM-System auch die Fähigkeit mitbringen, mit einem wachsenden Projekt ohne zusätzliche Mehrkosten mitzuwachsen. Falls sich die Entwicklung, wie z. B. bei SINAMICS, nicht nur auf einen Standort beschränkt, muss das SCM-System die Synchronisation der Daten unterschiedlicher Standorte beherrschen.

In der Siemens AG wird ClearCase (CC) als Werkzeug für das SCM eingesetzt, da ClearCase Unterstützung für das Softwarekonfigurationsmanagement und das Softwareversionsmanagement bietet.

2.2.3 ClearCase

Bei ClearCase handelt es sich um eine Sammlung von Programmen, die Unterstützung für das Konfigurationsmanagement bieten. ClearCase erfüllt alle oben genannten Anforderungen, die an ein Softwareversionsmanagementsystem gestellt werden. Für das weitere Verständnis von CM wird nun die grundlegende Funktionsweise und spezielle Begriffe am Beispiel von ClearCase erläutert, da es sich bei ClearCase quasi um das Standardwerkzeug in der kommerziellen Softwareentwicklung handelt.

2.2.3.1 Allgemeine Handhabung

In großen Softwareprojekten gibt es eine Vielzahl von Daten, die der Übersichtlichkeit halber strukturiert in **Versioned Object Databases**, auch VOBs genannt, abgelegt werden. Um mit diesen Daten arbeiten zu können, muss sich jeder Entwickler eine **View** auf diese Daten anlegen. Durch die sogenannte **ConfigSpec** ist es möglich, die View so zu konfigurieren, dass in jeder View eine bestimmte Version der Daten gesehen werden kann. Mit Hilfe der Views kann die Parallelentwicklung innerhalb eines Releases als auch die Wartung bereits freigegebener Releases umgesetzt werden. Für die Bearbeitung von Daten werden diese durch einen **Check-out** lokal in die View kopiert und in dieser lokal durch den Entwickler bearbeitet. Damit jeder Entwickler seine Änderungen einbringen kann, ohne die Arbeit anderer zu beeinflussen, erfolgt der Check-out auf Entwicklerbranches. Um die bearbeiteten Daten im VOB zu speichern, erfolgt auf dem Entwicklerbranch ein **Check-in**. Sollen die Änderungen einem anderen Entwickler oder KM zur Verfügung gestellt werden, werden die Änderungen des Entwicklerbranches auf dem Zielbranch zusammengeführt. Diesen Vorgang des Zusammenführens nennt man **Merge**. Nachdem die Änderungen durch den Integrator[3] getestet wurden, erfolgt der Check-in auf offiziellen KM-Branches und KM erzeugt mit diesen Daten regelmäßig neue Versionen. Jede in der Integrationsview sichtbare Version eines VOB-Elements wird vor der Erzeugung eines Standes mit einem **Stempel**, auch **Label** genannt, versehen. Solange die Integration noch nicht abgeschlossen wurde, kann dieser Stempel durch KM auf neuere Versionen der Elemente verschoben werden. Nach Abschluss einer Integration dient dieser Stempel zur Reproduzierbarkeit des abgeschlossenes Standes, aber auch als Abzweigpunkt für Entwicklerbranches. Bei den Versionen gibt es **interne** und **externe**. Die interne Version gewährleistet die Unterscheidbarkeit offiziell generierter Stände, da bei jedem neuen Softwarestand die interne Version inkrementiert wird. Die externe Version kennzeichnet einen **Release** und wird daher erst nach einer Freigabe erhöht. Interessiert man sich für die Historie eines VOB-Elements, so kann man sich diese im **Versionsbaum** anschauen.
Da nun das prinzipielle Vorgehen bei der Entwicklung beschrieben wurde, werden nun die erwähnten Begriffe näher erläutert.

2.2.3.2 Begriffserläuterungen

Da bei Siemens DT ClearCase für das Softwarekonfigurationsmanagement eingesetzt wird, sind diese Erläuterungen auf ClearCase bezogen. Jedoch finden diese oder ähnliche Begriffe auch in anderen Versionsverwaltungswerkzeugen, wie z. B. Subversion[4], Verwendung.

[3] Rolle in der Softwareentwicklung. Dieser fügt die Änderunges der Entwicklungsbranches auf einem Zielbranches zusammen und startet Tests
[4] freies Softwareverwaltungssystem

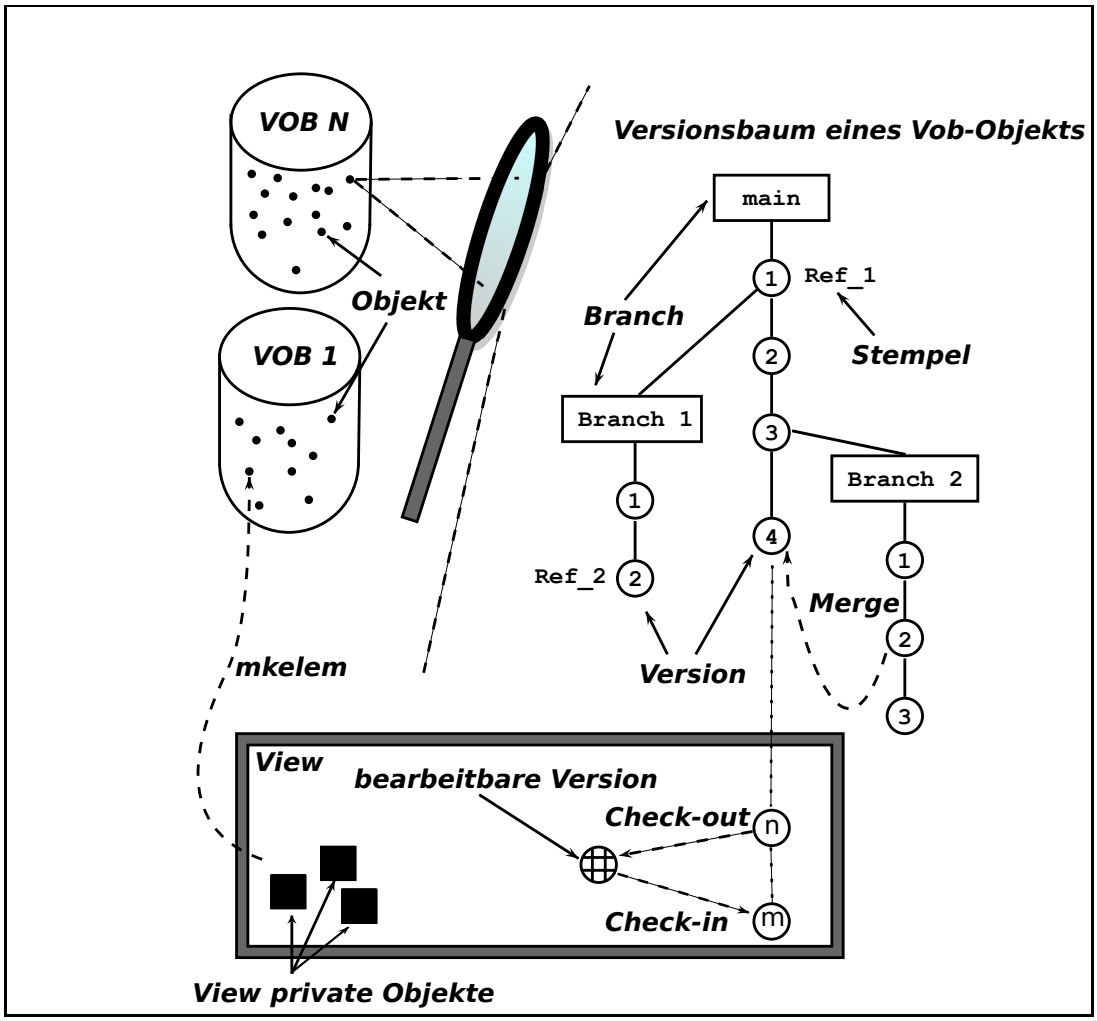


Abbildung 2.2: CM-Begriffe

VOB: Ein VOB beinhaltet die Daten, die unter Versionsverwaltung liegen. Allgemein nennt man den VOB auch Repository. Zugriffe auf VOBs sind nur mit ClearCase-Befehlen möglich. Bei ClearCase besteht die Möglichkeit, VOBs für bestimmte Zwecke anzulegen. Der Zugriff des Anwenders auf den VOB erfolgt über einen VOB-Tag[5]. Mit Hilfe des VOB-Tags kann der VOB wie ein Verzeichnis unter Windows/Unix verwendet werden.
Beispiele für mögliche VOBs sind:

Stage-VOBs enthalten die Generierergebnisse

Source-VOBs enthalten den Quellcode und Sourcen für die Generierung

Dokumenten-VOBs enthalten z. B. Designdokumente.

Tool-VOBs enthalten Programme und Skripts, die für die Generierung notwendig

[5] Name des VOBs

sind.

Diese Aufteilung erhöht die Übersichtlichkeit der Daten im Softwareprojekt, da Objekte gleichen Typs in einem VOB zusammengefasst sind. So hat sich diese Aufteilung z. B. auch im SINAMICS-Projekt durchgesetzt. Die Entscheidung über die Anzahl der VOBs innerhalb eines Projekts hängt unter anderem von der Projektgröße ab. So kann es auch durchaus sinnvoll sein, nur einen VOB anzulegen, der durch Verzeichnisse weiter unterteilt wird.

Objekt: Ein Objekt ist ein, in einem Repository, verwalteter Datentyp. Um Objekte unter ClearCase-Verwaltung zu stellen, müssen diese mit dem Befehl „mkelem“ wie in Abb. 2.2 auf der vorherigen Seite dargestellt, einmalig im Repository angelegt werden. Typische Objekte für einen VOB sind:

- Dateien (SourceCode, Designdokumente)
- Buildskripts
- Testskripts
- Generierergebnisse

Version: Eine Version entsteht durch Änderungen an einem VOB-Objekt, die in den VOB eingespielt wurden. So wird in Abb. 2.2 auf der vorherigen Seite durch den Check-in die neue Version „m“erzeugt. Betrachtet man alle Versionen eines Objektes erhält man den sogenannten Versionsbaum. Auf jede Version eines Objektes kann über ClearCase zugegriffen werden.

Stempel: Ein Stempel kennzeichnet exakt die Versionen mehrerer Objekte, die zum Erzeugen einer Konfiguration zu einem bestimmten Zeitpunkt verwendet wurden. Dadurch wird die Reproduzierbarkeit eines von KM generierten und freigegebenen Standes gewährleistet. Außerdem kann der Entwickler den Stempel als Basis für die Parallelentwicklung verwenden, was in Abb. 2.2 auf der vorherigen Seite bei Stempel „Ref_1“dargestellt wird.

View: Eine View ermöglicht die Sicht in den VOB. Durch die ConfigSpec kann die Sichtweise einer View beeinflusst werden. Dies erlaubt die Sicht auf unterschiedliche Versionen eines Objektes. Das Bearbeiten von VOB-Objekten ist nur innerhalb von Views möglich. Um ein VOB-Objekt zu bearbeiten, ist es notwendig dieses aus dem VOB in die Sicht zu kopieren. Dieses Vorgehen nennt man unter ClearCase „Check-out“. Sobald das Objekt in der View als „CHECKEDOUT“-Version liegt, kann es bearbeitet werden. Sind die erforderlichen Änderungen ausgeführt, wird die bearbeitete Version mit einem „Check-in“ wieder in das Repository eingespielt und eine neue Version des Objekts erzeugt. Eine View kann auch Elemente enthalten, die nur in der gerade gesetzten View sichtbar sind. Diese View-Private-Dateien, in Abb. 2.2 auf der vorherigen Seite als bearbeitbare Versi-

on gekennzeichnet, können über „mkelem“ zu VOB-Objekten umgewandelt und somit für andere Entwickler sichtbar gemacht werden. Anstelle des Begriffs View wird bei anderen Konfigurationsmanagementsystemen oft der Begriff *privater Arbeitsbereich* verwendet

ConfigSpec: Die ConfigSpec konfiguriert die Sichtweise einer View. Die Zeilen einer ConfigSpec sind Regeln, nach denen die Objekte der VOBs angezeigt werden. Die Objekte werden der ersten treffenden Zeile nach angezeigt. Die ConfigSpec für die View aus Abb. 2.2 auf Seite 11 sieht folgendermaßen aus:

```
element * CHECKEDOUT     # Zeige alle ausgecheckten Elemente
element * /main/LATEST   # Zeige main-LATEST, wenn Element
                         # nicht ausgecheckt ist
```

Branch: Der Branch ist ein Zweig auf dem Entwickler Änderungen an Objekten vornehmen können ohne andere Entwickler in ihrer Tätigkeit zu beeinflussen. Durch die Möglichkeit der ungestörten Entwicklung ist es möglich, dass sich von einem Objekt auf unterschiedlichen Branches die unterschiedlichsten Versionen entwickeln. Jeder Branch hat als Basis einen Stempel und nur die auf dem Branch bearbeiteten Elemente liegen dort.
Die ConfigSpec bestimmt auf welchem Branch neue Versionen eines Elements angelegt werden. **Mergen** nennt man das Einbringen der Änderungen eines Branches auf eine andere Entwicklungslinie. So zeigt Abb. 2.2 auf Seite 11 das Einbringen von *Branch 1* auf die Hauptentwicklungslinie auf *main*. Dabei werden die Änderungen des Branches auf dem Zielbranch in das Objekt eingetragen und eine neue Version des Objektes auf dem Zielbranch erzeugt. Stellt sich eine Entwicklung auf einem Branch als Irrweg heraus, wird diese Entwicklungslinie nicht weiterverfolgt. Neben diesen Entwicklerbranches existieren bei SINAMICS offizielle KM-Branches, die nach dem Entwicklungsende auf main angelegt werden. Auf diesen werden vor einer Freigabe noch kleine Fehlerbehebungen durchgeführt. Nach der Freigabe wird dieser Branch für die Wartung und Pflege durch Service Packs und als Basis für Hotfixstände genutzt.

Komponente: Eine Komponente ist eine einzelne Softwareeinheit, die für sich alleine generierbar ist. Sie enthält Informationen über ihre Generieraufrufe und darüber welche Generierergebnisse für sie aufgehoben werden müssen. Diese werden im sogenannten Stage-VOB als zip-Datei archiviert und mit dem Stempel der Generierung versehen. Komponenten können nicht nur als eine atomare Einheit vorliegen, sondern eine Wiederverwendung bereits generierter Komponenten als Bestandteile einer anderen ist möglich.

Softwarestand: Ein Softwarestand ist eine Zusammenstellung mehrerer Komponenten, die mit der gleichen internen Version erzeugt werden.

interne Version: Jeder erzeugte Softwarestand erhält eine eindeutige interne Versionsnummer, anhand derer dieser Stand identifiziert werden kann. Diese wird in die Software mit einkompiliert.

externe Version: In einem Entwicklungszeitraum wird eine externe Version erstellt. Bis eine externe Version fertiggestellt wird, kann sie mehrere interne Versionen verbrauchen. Ein Beispiel für eine externe Version ist V2.3. Unter dieser externen Version wird die Software dem Kunden als Release zum Kauf angeboten. Nach Freigabe einer externen Version, wird im nächsten Entwicklungszeitraum eine neue externe Version erzeugt.

Neben diesen hier vorgestellten Begriffen existieren noch weitere, wie z. B. Trigger[6] und Attribute[7], die im Zusammenhang mit ClearCase auch verwendet werden, aber für das allgemeine Verständnis von Softwarekonfigurationsmanagement nicht erforderlich sind. Da neben ClearCase als Werkzeug für das Softwarekonfigurationsmanagement auch noch andere Arbeitsmittel benötigt werden, sollen diese hier kurz vorgestellt werden.

2.3 Eingesetzte Werkzeuge bei Siemens DT

Die wichtigsten Werkzeuge des Softwarekonfigurationsmanagements bei Siemens DT werden in diesem Abschnitt kurz vorgestellt.

ClearCase: ClearCase ist eine Sammlung von Programmen die Unterstützung für das Konfigurationsmanagement bieten. Die in Abschnitt 2.2 geforderten Eigenschaften eines Softwarekonfigurationsmanagementsystems werden von ClearCase umgesetzt.

- Die Stabilität wird durch das View und Branchkonzept ermöglicht.
- Die Sicherheit wird durch Repositories und das Mergen gewährleistet.
- Die Reproduzierbarkeit wird durch das Stempeln sichergestellt.
- Die Nachvollziehbarkeit wird durch Abspeichern der Historie der Objekte sichergestellt.
- Die Skalierbarkeit, da ClearCase sich sehr gut für den Einsatz in sehr großen und sehr kleinen Projekten, z. B. durch die variable Anzahl der eingesetzten VOBs, eignet.

[6] Skripts die bei bestimmten ClearCase-Operationen ausgeführt werden können
[7] Mit Attributen kann ein Objekt mit Zusatzinformationen versehen werden

Perl: Perl ist eine Programmiersprache mit der Skripts geschrieben werden, die den Entwicklern das Arbeiten mit ClearCase erleichtern sollen.Darunter fallen zum Beispiel Skripts, die Views anlegen oder mit denen Entwickler eigene Stempel vergeben können. Es existiert auch eine Sammlung von Funktionen in Perl-Modulen, die den Zugriff auf die KM-Datenhaltung vereinheitlicht und somit vereinfacht.

tcsh: Die tcsh[8] ist als Standardshell an der SUN eingerichtet. Sie dient der Interpretation von Kommandos, die der Anwender eingibt. Neben der direkten Interpretation von Kommandos ist es möglich, Kommandos als Skript zu speichern und jederzeit wieder aufrufen zu können.

cmdata-Struktur: Ist ein von KM entwickeltes Datenformat, in dem KM alle relevanten Daten beschreibt. So wird jede von KM verwaltete Komponente in einer cmdata-Datei beschrieben. Auf den Aufbau des cmdata-Formats wird später genauer eingegangen.

Generierautomat: Der Generierautomat (GA) ist eine Sammlung von Perl- und tcsh-Skripts, mit deren Hilfe eine automatisierte Erzeugung von Software im SINAMICS-Projekt ermöglicht wird. Durch diesen Automatismus wird der KM-Mitarbeiter entlastet, da die Befehle der zu erzeugenden Software generisch aus den zu erzeugenden Komponenten gelesen werden. Eine Beschreibung der wichtigen Skripts folgt in Abschnitt 3.3.1.

make und Makefiles: Mit **Makefiles** lässt sich z. B. die Übersetzung von Software von Quellcode über Objektdateien[9] bis hin zu fertig gelinkten[10] ausführbaren Dateien oder Libraries beschreiben. Der Befehl **make** interpretiert die Anweisungen in den Makefiles.

Solaris: Solaris ist ein Betriebssystem für die SUN und bei Siemens DT innerhalb der SINAMICS-Welt die Hauptentwicklungsumgebung.

Windows: Windows ist ein Betriebssystem für den PC. Für einige zu generierende Komponenten wird Windows XP benötigt. Der Hauptteil der Entwicklung findet allerdings unter Solaris statt.

Beim SINAMICS Softwareprojekt handelt es sich um einen Softwarepool, aus dem mehrere Softwareprodukte erzeugt werden. Dieser Softwarepool besteht aus einer Vielzahl von Komponenten, die durch entsprechende Kombinationen die Softwareprodukte ergeben. Für die Erzeugung und zu Dokumentationszwecken wurde ein Generierautomat entwickelt. Dessen ausführliche Beschreibung folgt in Abschnitt 3.3.1.2.

[8] Tenex-C-Shell ist eine Weiterentwicklung der C-Shell
[9] Ergebnisse eines Übersetzunglaufes
[10] Linken: Verbinden einzelner Programmteile zu einem fertigen Programm

3 Build Forge

Build Forge ist ein flexibles Framework[11] für das Konfigurationsmanagement. Es ermöglicht die Automatisierung sich wiederholender Aufgaben, die Einführung von Standards und den Informationsaustausch innerhalb der Entwicklung. Durch eine integrierte Weboberfläche lassen sich Entwickler an unterschiedlichen Standorten problemlos betreuen und die Entwicklungstätigkeiten synchronisieren. Ganz allgemein kann man sagen, dass die Untersuchung eines Frameworks für das Konfigurationsmanagement eine immer größer werdende Bedeutung annimmt, da für Softwareprojekte die Entscheidung getroffen werden muss, ob in die Entwicklung eigener Werkzeuge investiert wird oder ob der Zukauf und Einsatz bereits bestehender Lösungen nicht sinnvoller ist. Ist sich die Projektleitung bereits in einer frühen Planungsphase über diesen Schritt im Klaren, kann die weitere Planung bei der Einrichtung der Projekte auch auf die Fähigkeiten der zugekauften Werkzeuge optimiert werden.

3.1 Fähigkeiten und Einsatzmöglichkeiten

Um eine Basis für die weitere Untersuchung von Build Forge (BF), zu schaffen, werden in diesem Abschnitt die von IBM beworbenen Fähigkeiten von Build Forge angesprochen.
Im folgenden werden die wichtigsten Punkte aus [Cor10c,] und [Cor10b] aufgezählt:

1. Build Forge ist in die eingesetzte Entwicklungsumgebung integrierbar und ist auf unterschiedlichen Plattformen lauffähig.
2. Mit Build Forge können Stücklisten erstellt werden, die Informationen bzgl. Änderungen und den Inhalt einer Generierung enthält.
3. Durch Threading wird die parallele Ausführung unabhängiger Prozesse gestattet. Durch sogenannte Threadblöcke, wird die Generierung so optimiert, dass nach einem Generierfehler nicht der gesamte Generierlauf erneut ausgeführt werden muss, sondern nur der fehlerhafte Block.
4. Für jedes Projekt können Metriken erstellt werden. So kann z. B. die Ausführungszeit eines Projekts über mehrere Generierungen verfolgt werden. Dies erlaubt die Überprüfung erfolgter Optimierungen auf deren Wirksamkeit.
5. Build Forge bietet die Möglichkeit Server in Gruppen zusammenzufassen und diese intelligent anzusteuern. Dadurch wird eine optimale Ressourcennutzung ermöglicht und die Fehlertoleranz erhöht, da nicht funktionierende Server beim Erzeugungsprozess automatisch ausgelassen werden.
6. ClearCase und ClearQuest[12] können sehr gut in Build Forge integriert werden.

[11] ein Rahmen, innerhalb dessen eine Entwicklung erfolgen kann
[12] Ein Tool zur Fehlerverfolgung

7. Build Forge ist in der Lage bei der Automatisierung des kompletten Entwicklungsprozesses von der Entwicklung bis zur Freigabe zu helfen.

Durch das „BF-Adaptor Toolkit“ können neben den IBM eigenen Programmen zum Softwareversionsmanagment und der Fehlerverfolgung auch Programme anderer Hersteller integriert werden.

3.2 Build Forge in großen Softwareprojekten

Nun stellt sich die Frage, warum Build Forge im Rahmen dieser Diplomarbeit überhaupt untersucht wird, da in bestehenden großen Softwareprojekten bereits gut funktionierende Skripts für das Erzeugen von Software und z. B. für das Erfassen von Metriken existieren bzw. diese Skripts für neue Softwareprojekte selber entwickelt werden können.
Als Antwort auf diese Fragen lassen sich mehrere Gründe anführen. So stellt sich bei der Erstellung von Softwareprojekten immer die Frage, welche Werkzeuge werden von externen Anbietern eingekauft und welche werden im eigenen Projekt entwickelt. Um eine Entscheidung treffen zu können, ist es wichtig über die Fähigkeiten der zu kaufenden Werkzeuge Bescheid zu wissen. So bietet der Kauf von Fremdwerkzeugen, in diesem Fall Build Forge als Framework für das Konfigurationsmanagement, zum einen den Vorteil, dass bei der Erstellung mehrerer Softwareprojekte eine Basis vorgegeben wird, an die sich die einzelnen Projektleiter halten müssen. Durch dieses Vorgehen ist es möglich Synergieeffekte zu nutzen, wenn weitere neu anzulegende Projekte mit Build Forge als Basis angelegt werden. So können z. B. Skripts, die in einem Projekt entwickelt werden, mit geringem Aufwand auch in anderen Projekten eingesetzt werden. Dies bringt nebenbei noch den Vorteil mit, dass sich mögliche projektspezifische Ausprägungen schon von Beginn an reduzieren lassen. Neben der Neuerstellung von Softwareprojekten spielt auch die Umstellung und Vereinheitlichung von großen Softwareprojekten einer Firma bei der Untersuchung von Build Forge eine entscheidende Rolle. So ist dies einer der Gründe für die Untersuchung von Build Forge im Rahmen dieser Diplomarbeit. Mit der Zeit treten in unterschiedlichen Softwareprojekten die verschiedensten Ausprägungen und Lösungen ähnlich gelagerter Probleme auf, wodurch Kapazität für die mehrfache Entwicklung gebunden wird, anstatt diese in die Optimierung des Generierprozesses zu investieren. Für die Umstellung muss, wie bei der Neuerstellung, die Einsatztauglichkeit von Build Forge überprüft werden. Allerdings sind in dem Fall der Umstellung die Kriterien strenger zu sehen, da durch einen zu großen Aufwand bei der Umstellung möglicherweise die komplette Weiterentwicklung der Software behindert wird.
Demgegenüber steht bei der Einrichtung neuer Softwareprojekte erst einmal nur die Zeit für die Einrichtung, die jedoch auch bei einer Eigenentwicklung durch KM auftritt. Außerdem lassen sich bei neu zu erstellenden Projekte mögliche Probleme oder Konflikte, die mit der Arbeit durch Build Forge auftreten können, schon durch eine Anpassung der Anforderungen vermeiden, während bei bestehenden Projekten mit der eingesetzten

Umgebung gearbeitet werden muss. Für den Fall der Umstellung bereits vorhandener Softwareprojekte muss jedoch sichergestellt werden, dass Build Forge zumindest die Grundanforderungen dieser Projekte abdeckt. Denn ist Build Forge nicht in der Lage auch nur die Grundanforderungen abzudecken und wird dies erst im Laufe der Umstellung festgestellt, so werden ebenfalls Ressourcen verschwendet, da das Ziel des Einsatzes von Build Forge in alten Softwareprojekten nicht erreicht werden kann.
Sowohl beim Anlegen als auch bei der Umstellung von Softwareprojekten ist von entscheidender Bedeutung wie schnell mit Build Forge eine tragfähige Lösung erreicht werden kann. Die Projektleitung wird sich mit Sicherheit für die schnellere Lösung entscheiden und darauf bestehen, dass noch nicht vorhandene Probleme bei ihrem ersten Auftreten schnell gelöst werden. Hier gilt jedoch, dass im Entscheidungsprozess auch berücksichtigt werden sollte, für welche Probleme Build Forge bereits Lösungen anbietet. Dies reduziert nachträglich erheblich den Aufwand, der für die Erweiterung selbstentwickelter Werkzeuge betrieben werden muss. Ein Beispiel hierfür findet sich im Projekt SINAMICS, das aktuell versucht die Generierzeit durch die Parallelgenerierung mehrerer Komponenten zu optimieren. Dies ist mit Mehraufwand verbunden, der beim Einsatz von Build Forge nicht aufgetreten wäre, da dort für die parallele Generierung, durch das Konzept des Threadings, bereits eine Lösung existiert.
Warum wurde Build Forge für Siemens relevant und wird dort im Projekt SINAMICS untersucht? Die Frage nach den Fähigkeiten von Build Forge stellt sich bei Siemens DT aus dem Grund, da der Gedanke aufgekommen ist, die großen Softwareprojekte SIMOTION, SINAMICS und SINUMERIK auf eine gemeinsame Basis zu bringen und die historisch gewachsenen unterschiedlichen Projektausprägungen zu reduzieren. So wird der Generierautomat, der in Abschnitt 2.3 bereits kurz angesprochen wurde, auch nur im Projekt SINAMICS und zwei kleineren Projekten verwendet. Allerdings hat die Umstellung des Generierautomaten auf die kleineren Projekte erhebliche Ressourcen gebunden, da dieser anfangs nur auf das SINAMICS-Projekt ausgelegt war ausgelegt war.
Neben dem Wunsch nach Vereinheitlichung der einzelnen Projekte innerhalb der Siemens AG, besteht auf Seite der evosoft GmbH auch der Gedanke KM als Produkt für neue Softwareprojekte anzubieten. Für diese Dienstleistung ist der Einsatz eines fertigen Werkzeugs sinnvoll, da sich so unterschiedliche Projekte analog bearbeiten lassen. Dies führt durch Synergieeffekte zu einer Kostensenkung, da nicht für jedes neue Projekt eine eigene Lösung entwickelt werden müsste. Jedoch müssen vor einer Festlegung auf Build Forge erst einmal Kenntnis über die realen Fähigkeiten gewonnen werden.

3.3 Unterschiede zwischen Build Forge und dem Generierautomaten

Um die Unterschiede zwischen Build Forge und dem aktuell bei SINAMICS verwendeten Generierautomaten zu verstehen, wird dessen Arbeitsweise im nächsten Abschnitt vorgestellt.

3.3.1 Der eingesetzte Generierautomat

In diesem Abschnitt wird der aktuell bei Siemens DT im Projekt SINAMICS[13] verwendete Generierautomat beschrieben, um die momentane Vorgehensweise des Generierablaufs innerhalb des Projekts besser zu verstehen.

Warum wurde ein Generierautomat nötig?
Bei der Entwicklung der Software für SINAMICS wurde auf ein modulares Konzept gesetzt. Das bedeutet, dass nicht eine allumfassende Softwarekomponente existiert, die mit einem Generieraufruf erzeugt werden kann, sondern die Software aus vielen einzelnen Komponenten besteht. Dies ermöglicht eine sehr hohe Kombinationsmöglichkeit der einzelnen Komponenten. Diese Anforderung hat zur Folge, dass jede Komponente für sich generierbar sein muss. Durch diese Modularität und Kombinierbarkeit können bereits erzeugte Komponenten auch in anderen Komponenten weiterverwendet werden. Da Komponenten existieren, die auf anderen Komponenten aufbauen, ist es nötig die Generierung levelbasiert zu gestalten, da nur so gewährleistet wird, dass eine wiederverwendete Komponente vor einer von ihr abhängigen Komponente erzeugt wird. So entstanden im Laufe des Projekts einzelne Funktionsblöcke von teilweise komplexen, immer wiederkehrenden Aktionen, die anfangs bei kleiner Projektgröße noch per Hand abgearbeitet werden konnten.

Für das Erzeugen einer Komponente sind in der Regel immer folgende Schritte notwendig:

- Eintragen der internen Versionsnummer in den Datensatz der Komponente und deren Quellcode
- Holen der benötigten Komponenten
- Generieraufruf
- Kopieren und Packen der Generierergebnisse
- Sichern der gepackten Generierergebnisse in den Stage-VOBs
- temporäres Bereitstellen der Generierergebnisse für den Test
- Überprüfung auf Generierfehler
- Archivieren der Generierergebnisse in den Stage-VOBs

[13] Bezeichnung für eine Antriebsfamilie bei Siemens

Für die Steuerung des Generierautomaten wurde von KM ein eigenes Datenformat entwickelt, welches auf ASCII[14]-Dateien beruht. Dieses so genannte cmdata-Format wird in Abschnitt 3.3.1.1 ausführlich erklärt. In einigen dieser cmdata-Dateien, den cmdata_versions, werden für jede durchzuführende Integration die zur Erzeugung benötigten Schritte, wie z. B. die Generieraufrufe, beschrieben. Aus diesen Beschreibungsdateien entwickelte sich mit der Zeit ein System mehrerer Dateien, mit denen die Vorgänge im KM-Umfeld beschrieben werden.
Da bei steigender Anzahl der Komponenten und Branches eine manuelle Bearbeitung immer aufwendiger und fehleranfälliger wurde, bestand der Wunsch und die Notwendigkeit diese Schritte automatisiert auszuführen und die Ausgabe in Dateien zu protokollieren. Diese Protokolle werden für das automatisierte Erkennen von Fehlern und deren Analyse genutzt. So entstand ein Skript, dem eine per Hand erzeugte Liste zu generierender Komponenten übergeben wurde, das die einzelnen Schritte nacheinander abarbeitet. Damit war die Basis für den Generierautomaten gelegt. Neben der Möglichkeit diese Dateien für die Automatisierung der Softwareerzeugung zu verwenden wird durch dieses Datenformat auch die Reproduzierbarkeit der Komponenten zu einem späteren Zeitpunkt gewährleistet.
Mit der Zeit wurde der Generierautomat immer wieder verbessert und an neue Anforderungen angepasst, so dass er heutzutage aus mehreren Skripts und einer eigenen Umgebung im cmdata-Format besteht.
Den Ablauf zur Erzeugung der Komponenten von der Planung bis hin zum Abschluss zeigt Abb. 3.1 auf der nächsten Seite. In diesem Bild erkennt man auch, dass neben dem reinen Ablauf der Generierung mit Hilfe der Skripts des Generierautomaten auch cmdata-Dateien und Datenbereiche verwendet werden. So beziehen beispielsweise die im Ablauf verwendeten Skripts Informationen über die aktuelle Generierung aus den im Bild links dargestellten cmdata-Dateien. Innerhalb der Datenbereiche auf der rechten Seite im Bild finden sich z. B. Verzeichnisse, die als temporärer Ablageort der Generierergebnisse dienen (*generation tmp*).
Im noch folgenden Abschnitt 3.3.1.2 werden die in Bild 3.1 auf der nächsten Seite unter Ablauf aufgeführten Punkte und deren Abhängigkeit zu den Daten und Datenbereichen ausführlich beschrieben.

[14] American Standard Code for Information Interchange

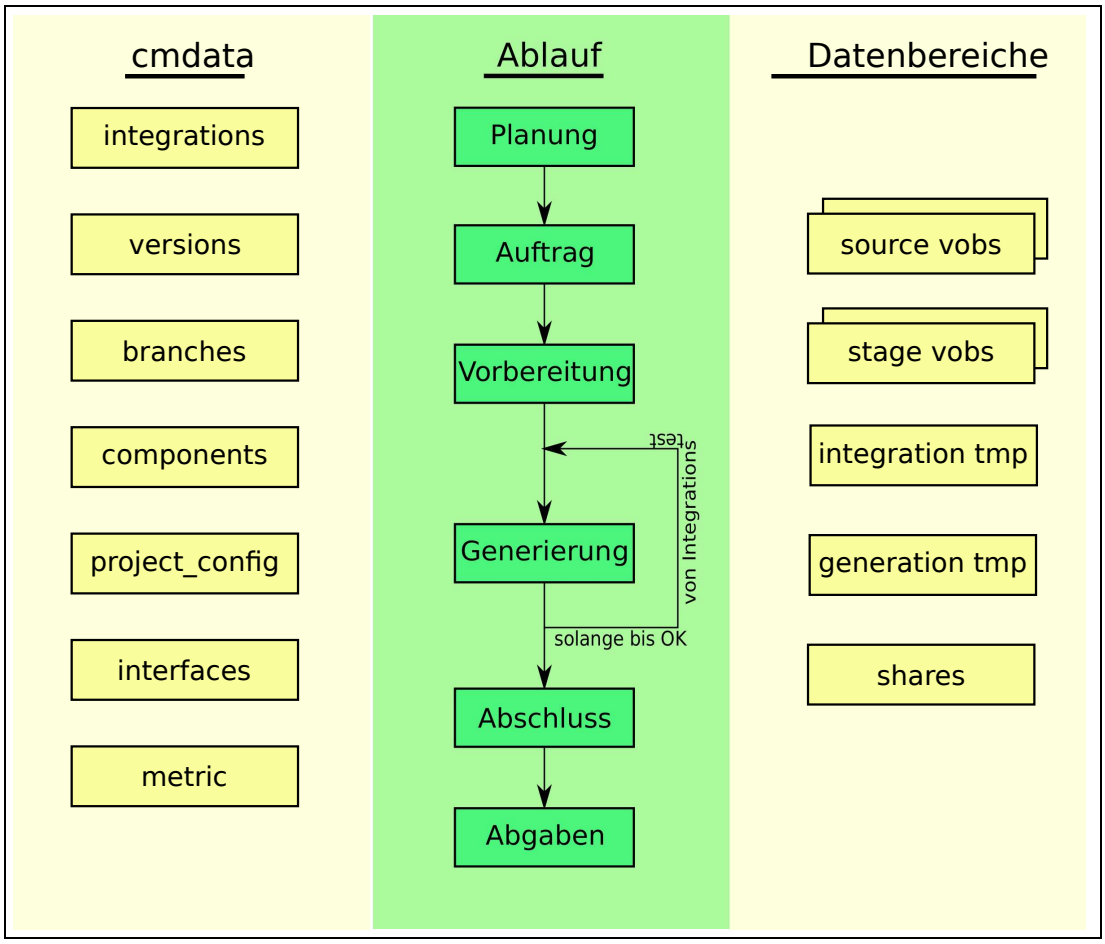


Abbildung 3.1: Darstellung des Generierablaufs bei KM

3.3.1.1 cmdata-Format

Für die Datenhaltung verwendet KM Textdateien, die in einem selbst entwickelten Format beschrieben werden. Jede cmdata-Datei ist nach folgendem Schema aufgebaut. Mit „KEYWORD“fängt in einer cmdata-Datei jeweils ein neuer Datensatz an. Um die

```
KEYWORD: key1
{
    Schluessel1: Wert1
    Schluessel2:
    {
        Ein Eintrag in einer cmdata-Datei kann auch mehrere
            Zeilen enthalten
    }
    ...
}
...
KEYWORD: keyN
{
   Schluessel1: Hier steht der Inhalt von Schluessel1
   Schluessel2: Kann auch nur aus einer Zeile bestehen
   ...
}
```

cmdata 3.1: *Aufbau einer cmdata-Datei*

Algorithmen zu vereinfachen besteht die Vereinbarung, dass jeder Datensatz eines bestimmten cmdata-Typs alle Schlüssel enthalten muss. Besitzt ein Schlüssel in einem Datensatz keine Relevanz so ist der Wert auf “NOT_DEFINED„ zu setzen.
Die im folgenden erklärten cmdata-Dateien sind in Anhang A mit einer Erklärung der wichtigsten Felder aufgeführt.

cmdata_paths
Jede bei KM verwendete cmdata-Datei wird in cmdata_paths beschrieben. Dies ermöglicht den Zugriff auf cmdata-Dateien nur über den Dateinamen.

cmdata_versions und cmdata_components
Jede Komponente, die von KM generiert wird, existiert bei KM als cmdata-Datei. Die Konvention für die Namensgabe bei Komponenten ist als Präfix das cmdata_versions. Jede dieser cmdata_versions-Dateien enthält versionsspezifisch sämtliche Informationen, die für den Ablauf der Generierung notwendig sind. Für jede erzeugte Version

wird ein neuer Datensatz zu dieser Datei hinzugefügt. Um die Übersichtlichkeit zu bewahren, werden nur die für den Generierablauf relevanten Felder eines Datensatzes angezeigt.

In cmdata_components werden versionsunabhängige Informationen einer Komponente gespeichert. Ein Beispiel für solch eine Information ist der Verantwortliche der Komponente, der bei Fragen und Problemen mit dieser Komponente anzusprechen ist. Auch wird vermerkt für welche Schnittstellen welche Komponente relevant ist.

cmdata_integrations
cmdata_integrations ist die Basisdatei für die Planung und den Start der Generierung. Hier werden die zu generierenden Versionen mit Stempeln und weiteren nötigen Daten eingetragen.

cmdata_branches
cmdata_branches enthält Informationen über alle offiziellen KM-Branches. So wird hier z. B. gekennzeichnet von welchem Stempel der Branch abzweigt und ob es sich um einen noch in Benutzung befindlichen Branch handelt.

cmdata_interfaces
In cmdata_interfaces werden sämtliche Schnittstellen beschrieben, die von KM bedient werden. Zu diesen Informationen gehört z. B. ein Mailverteiler, der über neue Versionen der zu liefernden Komponenten unterrichtet wird.

cmdata_project
Da der eingesetzte Generierautomat in mehreren Projekten eingesetzt werden kann, besitzt jedes Projekt eine cmdata_project-Datei, in der projektspezifische Einstellungen vermerkt sind. So gibt es hier beispielsweise einen Schlüssel, der alle für die Generierung benötigten Pfade enthält.

3.3.1.2 Die Skripts des Generierautomaten

In Abschnitt 3.3.1.1 wurde die Datenbasis der Komponenten beschrieben. Für die Auswertung und Abarbeitung der Generieraufrufe wird ein Generierautomat verwendet. Dieser besteht aus mehreren tcsh- und Perlskripts.
Bild 3.1 auf Seite 22 zeigt den Ablauf einer Integration von der Planung über das Vorbereiten eines Standes bis hin zu den Abgaben. Zudem werden die verwendeten cmdata-Dateien und die verwendeten Datenbereiche dargestellt.

Vorbereitung: Während der Vorbereitung werden die Datensätze der zu generierenden Komponenten für den Branch in ein temporäres Verzeichnis exportiert und der Branch als geöffnet gekennzeichnet. Es erfolgt der Check-out und ein anschließendes Löschen und Neuanlegen der Stage-VOB-Dateien mit der Größe Null für die relevanten Komponenten.

Generierung: In dieser Phase werden die in Abschnitt 3.3.1.1 beschriebenen Felder GENERATE, COPY, PACKAGE, ARCHIVE in einer Schleife über die Generierlevel ausgeführt. Am PC wird das in Abschnitt 3.3.1.1 beschriebene Feld GENERATE_PC für die Generierlevel ausgeführt. Es erfolgt eine Ablaufsynchronisation mit der Generierung an der SUN, da der PC nur für die den Aufruf des Inhalts von GENERATE_PC benötigt wird. Kopieren, Packen und das Archivieren in den Stage-VOBs erfolgt nur an der SUN.

Abschluss: Es erfolgt der Check-in, der in der View liegenden Stage-VOB-Dateien und der Import der exportierten Datensätze aus dem temporären Verzeichnis. Der importierte Datensatz wird als branch-Latest gekennzeichnet.

Abgaben: Verschicken von Mails über die generierten Komponenten mit Information über den Stempel, die interne Version und den Stage-VOB-pfaden an die Schnittstellen.

Eine ausführlichere Beschreibung der einzelnen Schritte findet sich in den folgenden Paragraphen. Dort werden auch die Abhängigkeiten der Schritte zu den verwendeten Datensätzen und den verwendeten Datenbereichen dargestellt.

Vorbereitung

Für die Vorbereitung kommt das Skript **prepcmgen** zum Einsatz.
In Abb. 3.2 auf der nächsten Seite erkennt man, dass bei der Vorbereitung folgende Abhängigkeiten auftreten:

1. In cmdata_integrations wird die aktuelle Integration als geöffnet gekennzeichnet.
2. Für die zu generierenden cmdata_versions-Dateien erfolgt ein Check-out und der aktuelle Branchdatensatz wird als offen gekennzeichnet.
3. Aktualisieren der internen Version im Quellcode und Stempeln aller Dateien mit dem aktuellen Generierstempel.
4. Check-out der zip-Dateien für die spätere Archivierung und Vergabe der Produktstempel und Anlegen des Integrationsstempels.
5. Die aktuellen Datensätze der Komponenten werden hierher exportiert und aktualisiert.
6. Die Datei zum Erfassen der Metriken wird angelegt.
7. Die Integrationsview wird im Unix-Home-Verzeichnis des Nutzers, der die Generierung ausführt, angelegt und für die Verwendung am PC vorbereitet.
   Auch wird die Batchdatei für den Start der PC-Generierung angelegt.

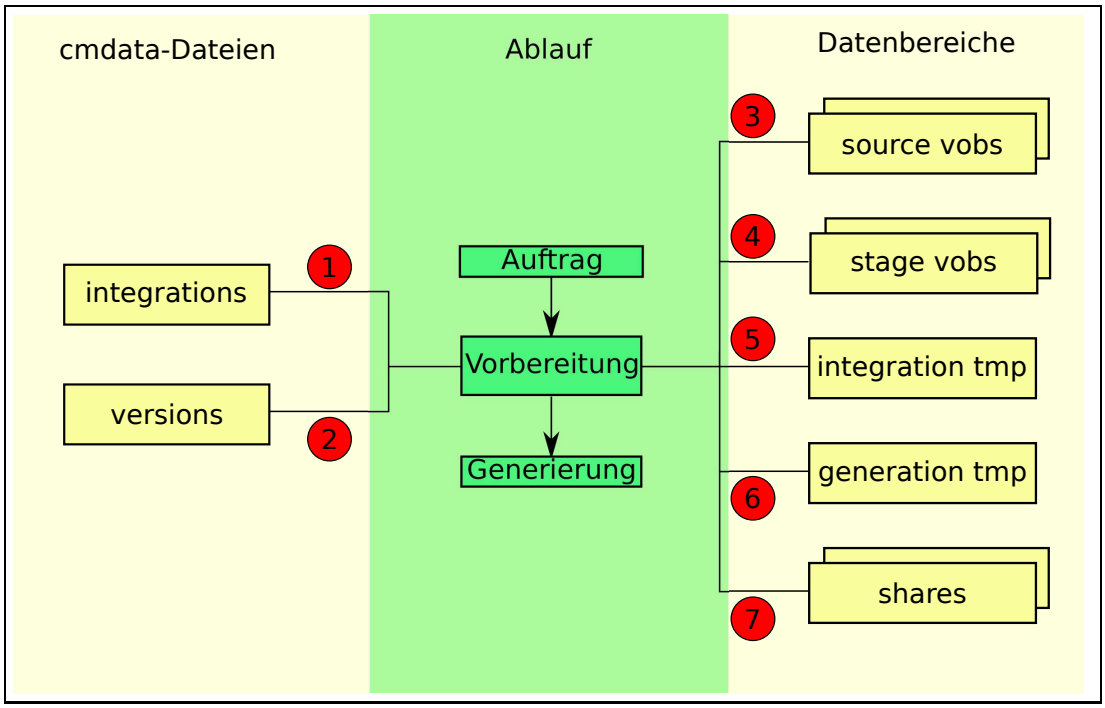


Abbildung 3.2: prepcmgen

Generierung

Die Generierung erfolgt in den beiden Skripts **docmgen** und **docmgen_pc**.
Nun werden die in Abb. 3.3 dargestellten Abängigkeiten erläutert.

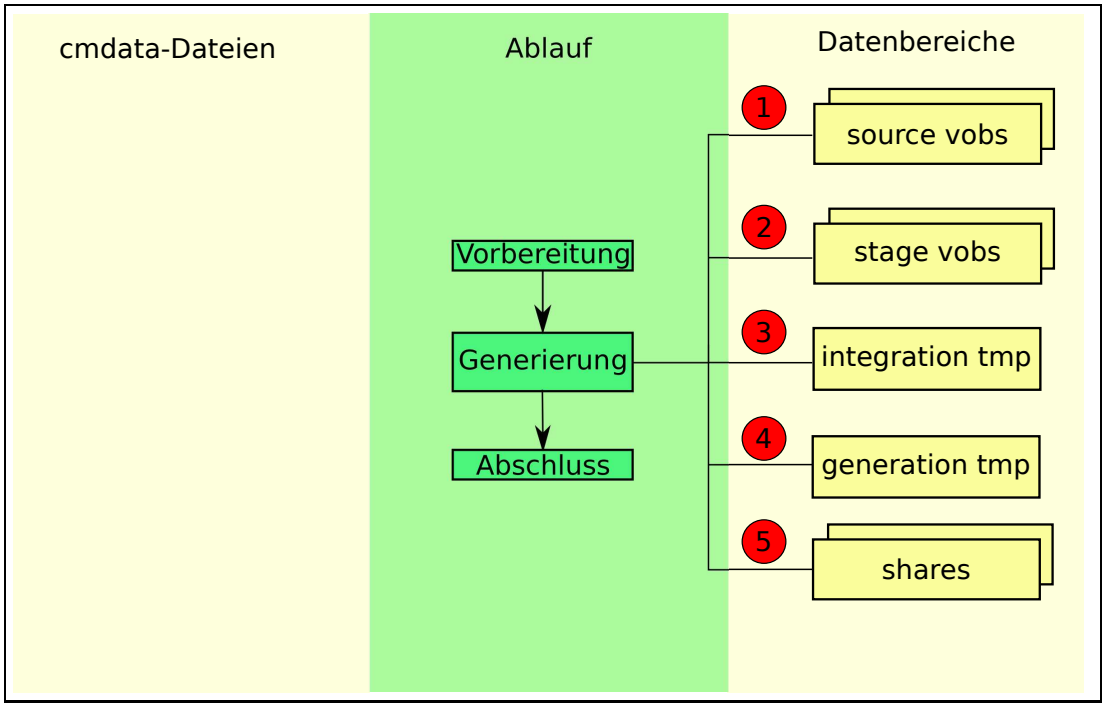


Abbildung 3.3: docmgen

1. In den Source-VOBs wird der Integrationsstempel auf Branch-Latest verschoben.

Danach wird auf den Source-VOBs generiert und der Rückgabewert der einzelnen generierten Komponenten überprüft.

2. die in die View kopierten Stage-VOB-Dateien werden gelöscht und mit der Größe Null erneut angelegt. Anschließend werden die Generierergebnisse der Source-VOBs gepackt und in die zugehörige Stage-VOB-Datei kopiert. Da die Stage-VOB-Dateien vor dem Start der Generierung die Größe Null haben, wird eine erneute Fehlerüberprüfung ermöglicht, indem nach jedem Level kontrolliert wird, ob die Dateigröße, der in diesem Level generierten Komponenten, weiterhin Null beträgt.
3. Aktualisieren der Versionen für die Sicherheitszertifizierung.
4. Anlegen von Verzeichnissen für Log-Dateien und die Zwischenergebnisse der Komponenten. In den Metriken wird die Anzahl der Generierungen erhöht und je nach Status der Generierung noch das Feld für eine erfolgreiche oder fehlerbehaftete Generierung inkrementiert.
5. Die Generierergebnisse werden aus dem Stage-VOB auf ein Netzlaufwerk kopiert. Dies vereinfacht den Test der generierten Komponenten durch den Integrationstest bzw. den für die Komponente verantwortlichen Entwickler. Um eine frühest mögliche Fehlererkennung zu gewährleisten, werden die Komponenten auch temporär an andere Projekte abgeben, die ihrerseits einen temporären Stand mit der erfolgten Zulieferung für ihren Integrationstest generieren.

Da es sich beim docmgen um das zentrale Skript des Generierautomaten handelt, soll dieses noch genauer betrachtet werden. Dazu zeigt Abb. 3.4 auf der nächsten Seite einen vereinfachten Ablauf einer KM-Generierung. Da sowohl an der SUN und am PC generiert wird, muss an der SUN das Skript docmgen und am PC docmgen_pc aufgerufen werden.
Wie in der Abbildung zu erkennen ist, muss die Generierung zwischen SUN und PC synchronisiert werden, da alle anderen Aktionen, wie z. B. das Archivieren der Generierergebnisse, nur an der SUN ausgeführt werden. Das bedeutet, dass die SUN in einem Level erst fortfahren darf, wenn die PC-Komponenten dieses Levels generiert wurden. Andererseits muss der PC mit der Generierung des nächstens Levels natürlich solange warten, bis die SUN das Feld CMGET_COMPLIST ausgeführt hat, da erst danach alle Abhängigkeiten der Komponenten für dieses Level gelöst sind. Diese Synchronisation erfolgt über ein selbstentwickeltes Token-Verfahren, bei dem in eine freigegebene Datei das jeweilige Token für den Generierlevel (${gen_level}[15] geschrieben wird). So teilt das windowsspezifische docmgen_pc über das Token GEN_LEVEL${gen_level}_DONE mit, dass für die levelspezifischen Komponenten alle GENERATE-Teile ausgeführt wurden, und die Weiterverarbeitung an der SUN erfolgen kann. Mit dem Token GETSTAGE_LEVEL${gen_level}_DONE teilt das unter Solaris aufgerufene docmgen dem PC mit, dass dieser mit der Generierung von ${gen_level} beginnen kann.

[15] Variable, die auf den aktuellen Generierlevel gesetzt ist

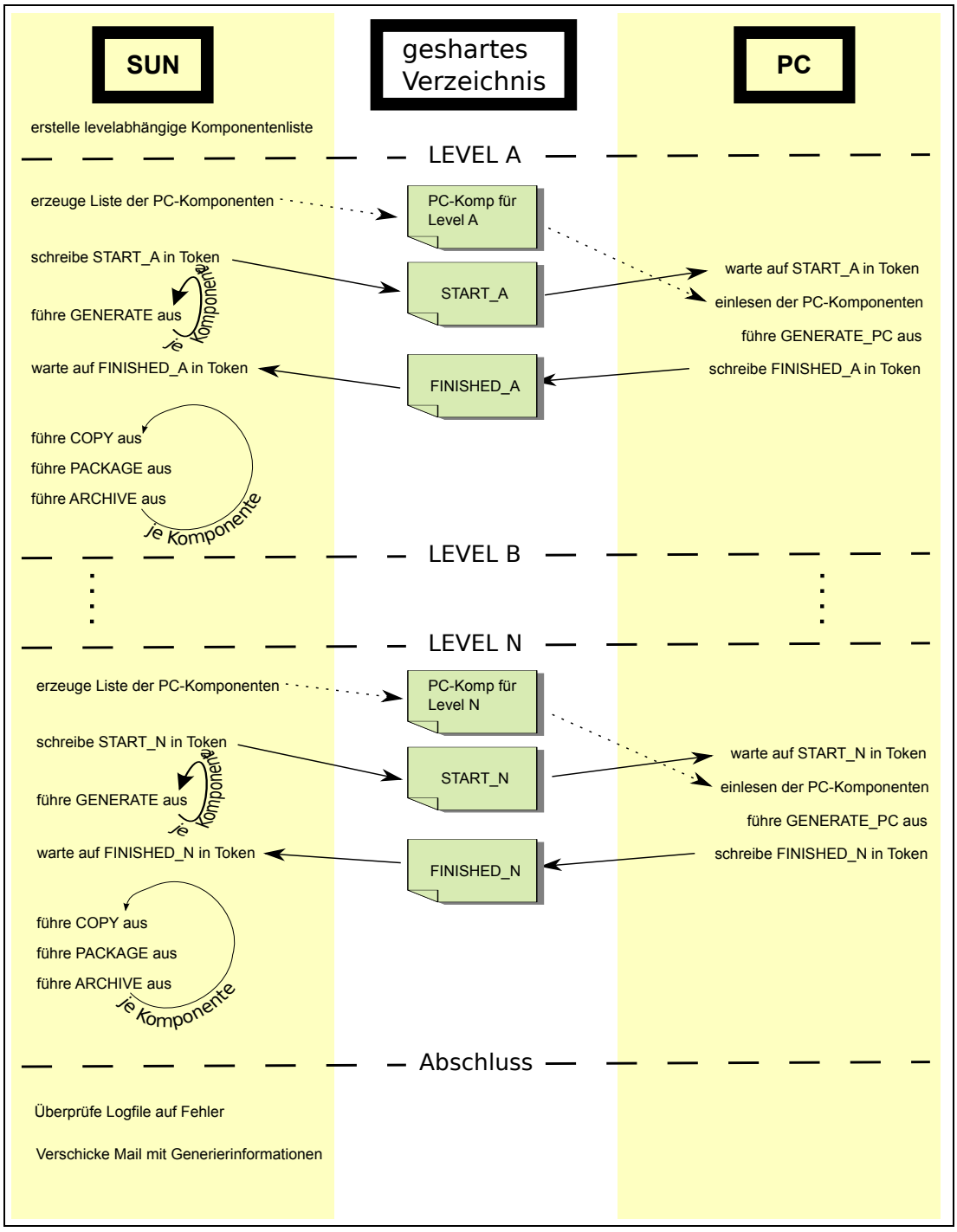


Abbildung 3.4: docmgen (detailliert)

Abschluss

Für das Abschließen einer Integration wird das Skript **fincmgen** verwendet.

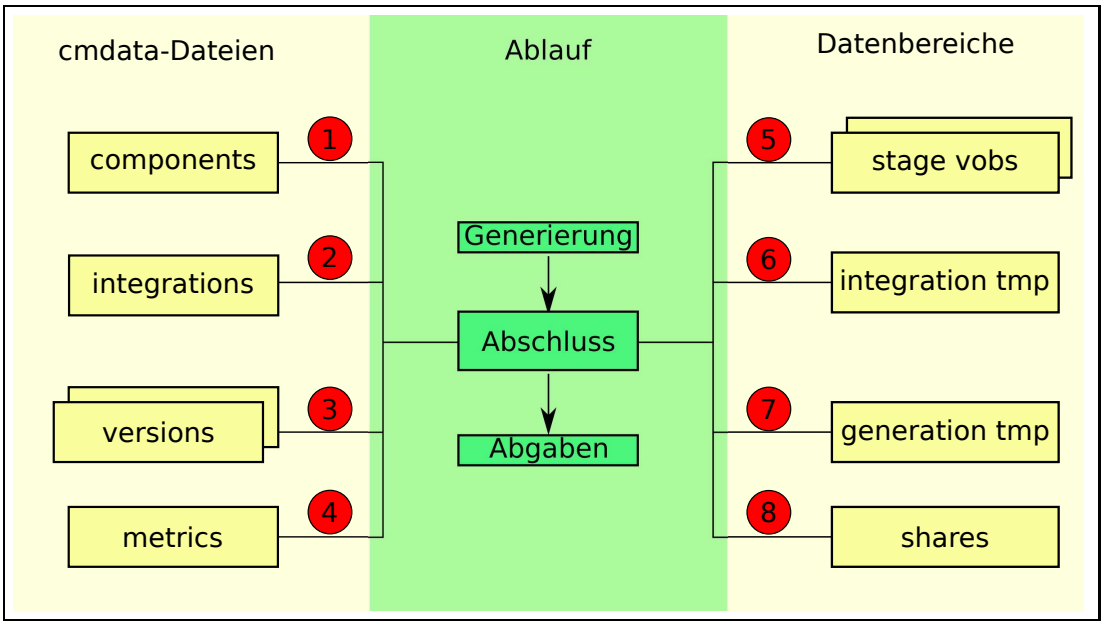


Abbildung 3.5: fincmgen

1. Die versionsunabhängigen Informationen in cmdata_components werden aktualisiert.
2. Der Status der aktuellen Generierung wird im zugehörigen Datensatz von offen auf Branch-Latest gesetzt. Dadurch wird der offizielle Generierstempel als ein möglicher Abzweigpunkt für Entwicklerbranches angeboten.
3. Die temp. Datensätze der Generierung werden in die jeweiligen cmdata_versions-Dateien ihrer Komponenten importiert. Dadurch wird die Reproduzierbarkeit sichergestellt.
4. Nachdem die geänderten Dateien zwischen der letzten und der aktuell abgeschlossenen Integration gezählt sind, wird diese Summe in die Datei der Metriken eingetragen. Hier wird auch die Anzahl der erfolgreichen und fehlerhaften Generierungen vermerkt.
5. Die in der Integrationsview liegenden Zip-Dateien werden im VOB durch den „Check-in“ gesichert.
6. Die temporären Datensätze werden importiert und gelöscht.
7. Eine nach Generierleveln sortierte Liste der Komponenten wird angelegt.
8. Bereitstellen der Komponenten auf einem Netzwerklaufwerk für den schnellen Zugriff durch Entwickler. Die bereitzustellenden Komponenten werden aus cmdata_components bestimmt.

Abgaben

Für die Abgaben kommt das Skript **postcmgen** zum Einsatz. Mit diesem Skript werden alle noch anfallenden Aktionen erledigt, nachdem eine Integration mit fincmgen beendet wurde. Jedoch ist der Hauptzweck, das automatische Bedienen aller Schnittstellen, die Interesse an dieser Integration haben. Zu diesem Zweck werden Mails mit Information über den Integrationsstempel und dem Ablageort in den Stage-VOBs verschickt.

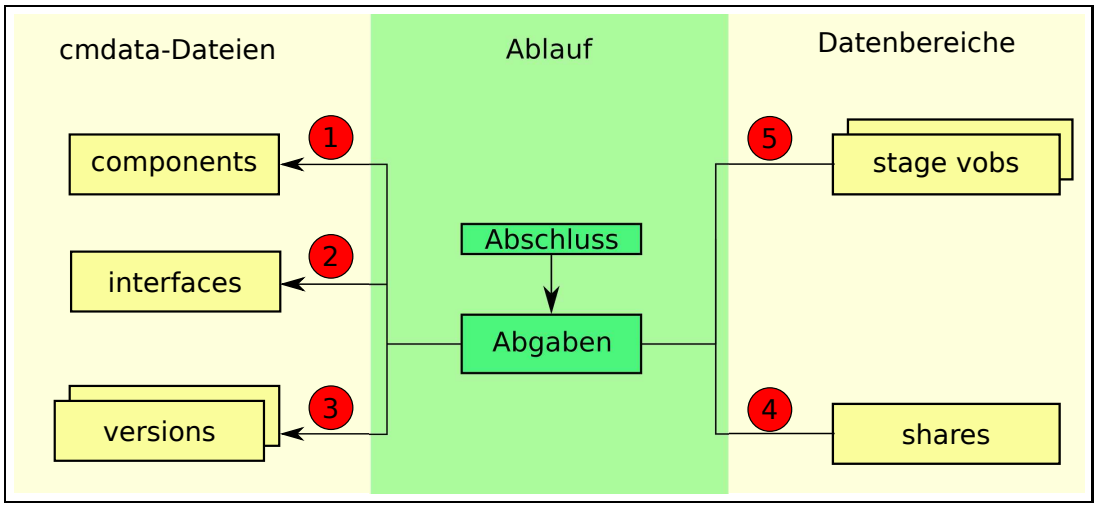


Abbildung 3.6: postcmgen

1. Heraussuchen der für die Abgabe möglichen Komponenten für diese Integration anhand des Feldes ACTIVE.
2. In cmdata_components ist für die Komponenten vermerkt, für welche Schnittstellen diese relevant sind. Für diese Schnittstellen werden aus cmdata_interfaces die notwendigen Informationen gelesen.
3. Einlesen der abzugebenden Komponenten mit Hilfe des Integrationsstempels und nach dem Versenden der Abgabemail an den Mailverteiler der Schnittstelle wird das Feld GIVEN_TO mit den bedienten Schnittstellen beschrieben.
4. Die Stage-VOB-Pfade der Komponenten wird in 1) ausgelesen und an die Schnittstellen verteilt.
5. In Ausnahmefällen werden die Komponenten für die Bedienung der Schnittstellen auch auf ein Netzwerklaufwerk kopiert.

Da in diesem Abschnitt der Generierautomat vorgestellt wurde, sind noch die Unterschiede zwischen dem Generierautomaten und Build Forge von Interesse.

3.3.2 Architekturunterschiede zwischen Build Forge und dem Generierautomaten

Wie sich in Bild 3.1 auf Seite 22 erkennen lässt, besteht der Generierautomat aus mehreren Skripts, die größtenteils unter Solaris ausgeführt werden. Da neben den SUN-Komponenten auch Windows-Komponenten erzeugt werden müssen, existiert ein Skript, welches die Abarbeitung des Feldes GENERATE_PC der cmdata-Dateien (Anhang A.2) übernimmt. Durch das Vorgehen mit docmgen und docmgen_pc für die Erzeugung der Komponenten ist es möglich eine Parallelisierung bei der Generierung der Komponenten beider Plattformen innerhalb eines Levels umzusetzen. Somit besteht beim SINAMICS Generierautomaten eine Beschränkung bei der Umsetzung einer parallelen Generierung auf zwei Komponenten unterschiedlicher Entwicklungsplattformen. Innerhalb der gleichen Entwicklungsplattform ist eine Parallelisierung von Komponenten daher nicht möglich. Zwar kann durch Ausnutzen der clearmake[16]-Fähigkeiten die Generierung einer Komponenten noch auf weitere SUNs verteilt werden, jedoch wird die Liste der Komponenten bei der Erzeugung trotzdem sequentiell abgearbeitet.
Demgegenüber verwendet Build Forge ein Server-Client-basiertes Konzept, welches in Abb. 3.7 auf der nächsten Seite (nach [BH09, S. 4]) dargestellt ist. Dort sieht man zum einen das Server-Client-Konzept, als auch den Aufbau von Build Forge aus 3 logischen Einheiten. Diese sind:

Zugriff: Der Zugriff auf den Build Forge Server wird über die Management Console oder Plugins in den Entwicklungsprogrammen, wie z.B Eclipse, ermöglicht.

Umsetzung: Enthält die für den Build Forge Betrieb notwendigen Programme. Das sind zum einen der Build Forge Server und zum anderen eine Datenbank (DB).

Ausführung: Befehle die der Build Forge Server verteilt, werden von den den Clients ausgeführt. Diese heißen im Build Forge Kontext *Agenten*.

Im Build Forge Server werden Einstellungen, Generierserver und die Entwickler eines Projekts verwaltet. Jeder Nutzer wird auch mindestens einem Rollenmodell zugeordnet, wodurch unterschiedliche Rechte in der Build Forge Benutzung abgebildet werden. Sämtliche Einstellungen, die in Build Forge vorgenommen werden, werden in einer separaten Datenbank abgespeichert. Der Build Forge Server ist auf unterschiedlichen Plattformen, z. B. SUN oder Windows, lauffähig und kann mit verschiedenen Datenbanksystemen unterschiedlicher Hersteller betrieben werden. Sollte man sich für Windows als Basis für den Build Forge Server entscheiden, enthält das Installationspaket für Build Forge auch eine geeignete Datenbank. Da es sich bei Siemens DT um eine gemischte Umgebung handelt und alle bis jetzt verwendeten Server unter Solaris laufen, wurde der Build Forge Server im Rahmen der Diplomarbeit auf einer SUN-Workstation installiert. Als Datenbank wird das frei verfügbare MySQL verwendet.
Der BF-Server und die Datenbank dienen der Verwaltung, während auf den sogenann-

[16] make Programm von ClearCase

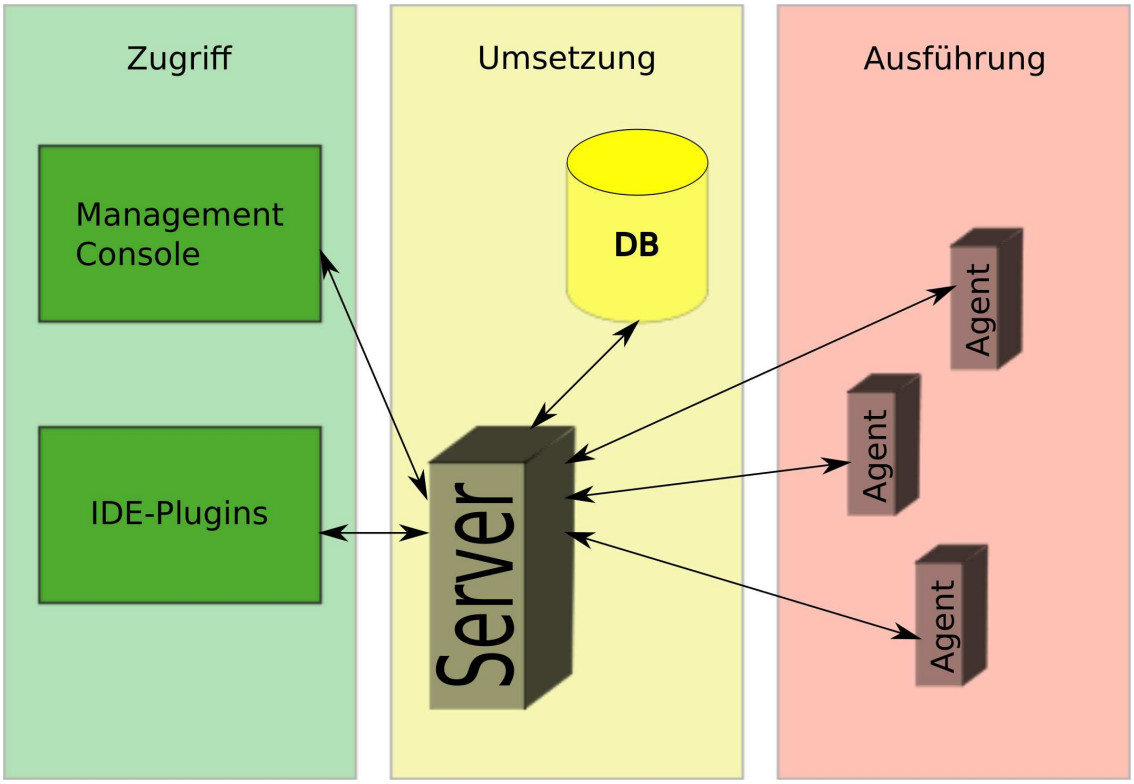


Abbildung 3.7: Architektur von Build Forge

ten *Agenten* die durch den BF-Server verteilten Aufträge abgearbeitet werden. Diese laufen als Serverprozess auf den Generiermaschinen und warten auf einem durch den Anwender festzulegenden Port auf Aufträge des BF-Servers. Nach Beendigung eines Auftrags kann dessen Exit-Status oder dessen Generierergebnis für einer weitere Verarbeitung an den BF-Server zurückgeliefert werden. Die Agenten sind auf unterschiedlichen Entwicklungsplattformen installierbar und unterstützen laut IBM eine Vielzahl von Betriebssystemen. Als Beispiel für unterstützte Plattformen seien Windows, Solaris und MAX OSX 10 genannt.
Da bei SINAMICS als Entwicklungsplattformen Windows und Solaris zum Einsatz kommen, wurde je ein Agent auf einer SUN-Workstation und einem „Windows XP“-PC installiert. Der Zugriff und die Konfiguration des Servers erfolgt über die *Management Console*. Das ist eine Webbasierte Oberfläche, die den Zugriff auf den BF-Server von beliebigen Klienten mit unterschiedlichen Rechten aus erlaubt. Zudem gibt es noch Plugins[17] für IDEs[18] die ebenfalls Zugriff auf bestimmte Funktionen bieten. Da sich die Untersuchung auf die Einsatztauglichkeit für KM beschränkt, werden die Plugins in dieser Untersuchung nicht behandelt.

Durch das serverbasierte Modell und den Betrieb mehrerer Agenten und Generierserver kann die Parallelisierung nicht nur innerhalb der Erzeugung einer Komponente erfolgen, sondern dies ermöglicht auch die parallele Erzeugung von mehr als zwei Komponenten

[17] Erweiterungen
[18] Integrierte Entwicklungsumgebungen

auf der gleichen Entwicklungsplattform. Die Fähigkeit zur parallelen Generierung mehrerer Schritte wird unter Build Forge *Threading* genannt.
Ein weiterer Unterschied zwischen dem Generierautomaten und Build Forge im Hinblick auf die Parallelisierung ist die Verwaltung der zur Verfügung stehenden Generierserver. Zwar muss bei beiden eine Liste der Generierserver gepflegt werden, allerdings besteht bei Build Forge die Möglichkeit ausführlichere Informationen über den Generierserver abzuspeichern. So existiert für den Generierautomaten eine Liste von SUN-Workstations, die anhand einer Mindestanforderung an die Hardware gepflegt wird. Aus diesem Grund stehen in der Liste des GA nur Server, die ein Mindestmaß an Arbeitsspeicher besitzen. Dadurch kann es vorkommen, dass SUNs nie in der verteilten Generierung einer Komponente verwendet werden, während andere dauerhaft ausgelastet werden. Da es sich bei diesen SUNs auch um in der Entwicklung eingesetzte Maschinen handelt, wird vor einer Verteilung auf diese lediglich geprüft, ob die CPU-Auslastung unter einem bestimmten Grenzwert liegt, um den Entwickler nicht spürbar zu behindern.
Betrachtet man nun Build Forge so müssen dort für eine dynamische Serverauswahl ebenfalls die zur Verfügung stehenden Generierserver eingepflegt werden. Im Unterschied zum Generierautomaten kann diese Liste jedoch alle zur Verfügung stehenden Maschinen enthalten, da die Serverauswahl bei Build Forge mit Hilfe der **Kollektoren** und **Selektoren** erfolgt. Diese werden nach Anlegen der Serverliste durch den Anwender erstellt und dienen der Erfassung und Auswertung serverspezifischer Daten, wie z. B. der Serverarchitektur, und enthalten die Server jeder Hardwareplattform. Neben den Kollektoren und Selektoren sind **Manifeste** ein wichtiger Bestandteil bei der dynamischen Serverauswahl, die in [Cor09, S. 24] folgendermaßen beschrieben wird:

> „Drei verschiedene Datenobjekte ermöglichen dem System die dynamische Serverauswahl:
>
> - Ein *Collector* ist ein Objekt, das die Menge der Eigenschaften definiert, die das System von einer Serverressource erfasst oder ihr zuweist. Ein Collector wird bei der Überprüfung der Eigenschaften einer Serverressource ausgeführt. Die erfassten Eigenschaftswerte werden in einem Manifest gespeichert.
> - Ein *Manifest* ist eine Liste der Eigenschaften für einen bestimmten Server. Es enthält die Ergebnisse, die beim Ausführen des Collectors erzielt wurden.
> - Ein *Selektor* ist eine Liste mit Eigenschaften und Vergleichen wie MEM_TOTAL = 512. Die Eigenschaften eines Selektors können vom System mit einem Manifest verglichen werden, um festzustellen, ob ein Server die Anforderungen für einen bestimmten Selektor als eine Eigenschaft definiert. Beim Ausführen des Projekts oder Schritts wird der Selektor mit dem Manifest aller definierten Serverressourcen verglichen, um die für die Ausführung zu verwendende Serverressource zu ermitteln.“

Durch die Kollektoren, Manifeste und Selektoren kann eine optimale Verteilung der einzelnen Projekte und Schritte erreicht werden, da jedes Projekt oder jeder Schritt mit einem für diesen passenden Selektor versehen werden kann. So werden Erzeugungsschritte, die keine sonderlichen Anforderungen an die Hardware stellen, mit Selektoren versehen, die weniger leistungsfähige Server auswählen, während gut ausgestattete Maschinen für Schritte mit langer Laufzeit oder hohem Speicherbedarf zur Verfügung stehen.

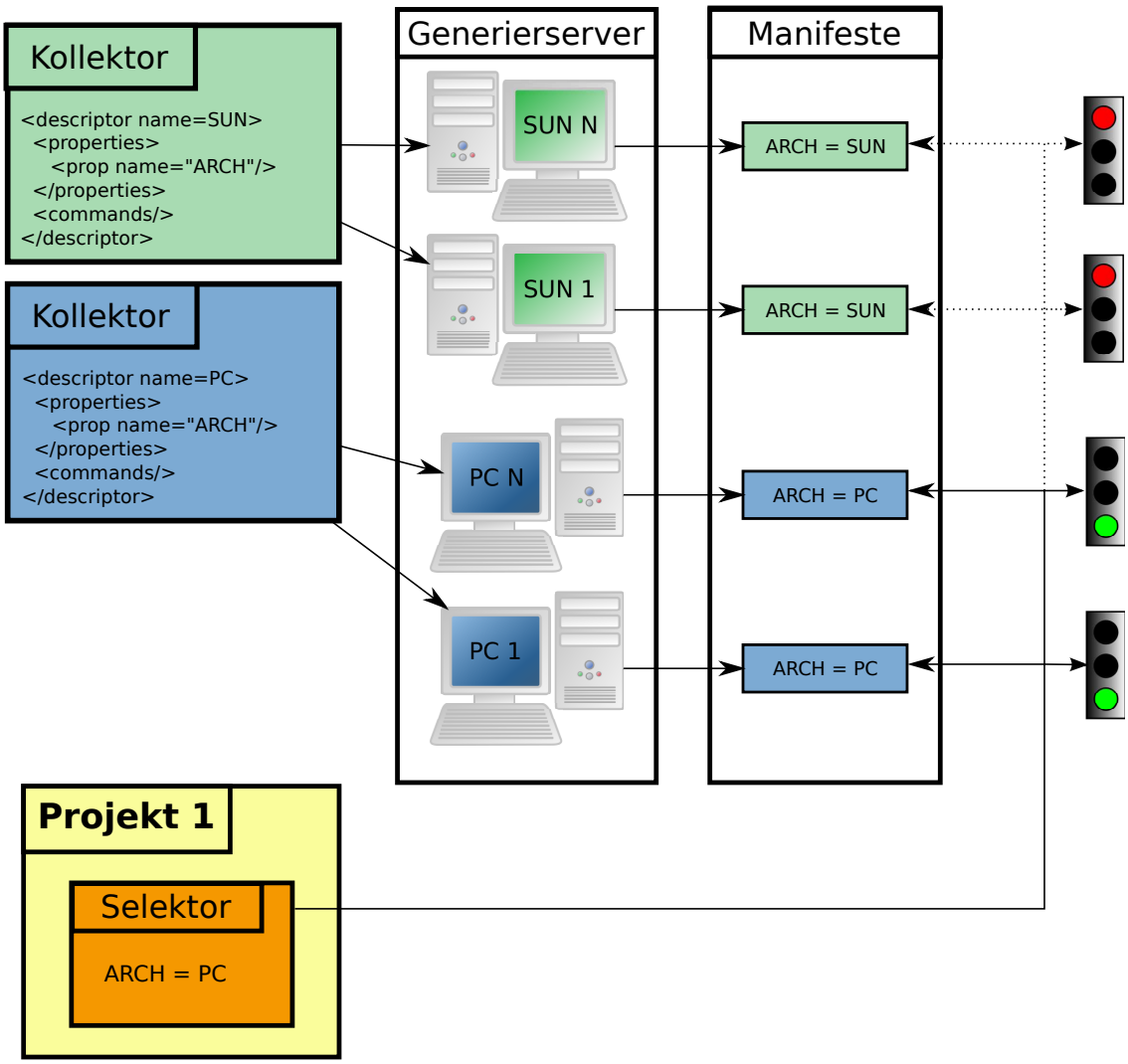


Abbildung 3.8: Generierserverauswahl bei Build Forge

In Abb. 3.8 (übernommen aus [BH09, S. 7]) wird das Zusammenspiel zwischen Kollektoren, Selektoren und Manifesten noch einmal beispielhaft dargestellt. Hier wird noch einmal deutlich, dass die Kollektoren die Daten, der zur Verfügung stehenden Server erfassen und dann im Manifest abgespeichert werden. Durch den Selektor eines Projekts werden die Manifeste der zur Verfügung stehenden Server abgefragt und dadurch die zu verwendenden Server ausgewählt. So wird in Abb. 3.8 durch den Selektor des Projekts nach einem PC-Architektur-Server gefragt. Der Selektor wählt anschließend einen Server aus, dessen Manifest als Architekturtyp „PC“besitzt, und gibt den Auftrag

an diesen weiter.

Wie man auch aus Abb. 3.7 auf Seite 32 erkennt, müssen für Build Forge noch zusätzliche Dienste, wie z. B. eine Datenbank, installiert werden. Dies ist ein weiterer Unterschied zum Generierautomaten, der ohne Installation von Zusatztools zurecht kommt, sondern nur einige cmdata-Dateien für den Einstieg benötigt.

3.3.3 Vorbedingungen

Für den SINAMICS-Generierautomaten müssen die in den Abschnitten 2.3 und 3.3.1 vorgestellten Programme und cmdata-Dateien bereitgestellt werden. Wie in Abschnitt 3.3.2 bereits erkennbar ist, müssen für den Betrieb von Build Forge zusätzliche Dienste installiert werden.

- Webserver
- Datenbank
- Agenten auf den Generiermaschinen

Durch diese benötigten Zusatzdienste fallen bei KM neben der kontinuierlichen Aufgaben der Integration und Optimierung der Generierabläufe auch Pflegeaufwand für diese zusätzlich Programme an. Alleine an diesen erforderlichen Grundlagen erkennt man, dass schon für die Installation von Build Forge mehr Aufwand in der Infrastruktur betrieben werden muss. Als ein weiteres Problem bei der Verwendung von Build Forge kann sich die Notwendigkeit des Betriebs eines Webservers erweisen, da dessen Einsatz in manchen Firmen von zentraler Stelle aus verboten wird. So muss man sich vor der Anschaffung und dem Einsatz von Build Forge auf jeden Fall über die zusätzlich benötigten Programme und deren Erlaubnis zum Betrieb informieren bzw. für den Einsatz die Betriebserlaubnis einholen. Dies erfordert eine intensivere Zusammenarbeit mit der Infrastruktur als es beim Einsatz des Generierautomaten in anderen Projekten notwendig wäre. Für die durchzuführende Untersuchung von Build Forge wurde auf einer SUN-Workstation die Build Forge Enterprise Edition 7.1 installiert. In diesem Installationspaket ist neben dem Build Forge Server auch Apache als Webserver enthalten. Das ermöglicht eine relativ problemlose Installation, im Vergleich zu der Vorgängerversion, in der jedes weitere benötigte Softwarepaket manuell installiert werden musste. Nur der MySQL-Server musste als eigenständiges Paket unter Solaris installiert werden, da die bei Build Forge zur Installation mitgelieferte Datenbank nur unter Windows funktionsfähig ist.

In diesem und dem vorigen Abschnitt 3.3.2 wurden die technischen Unterschiede zum Generierautomaten herausgearbeitet. Neben diesen sind auch noch die Unterschiede in der Implementierung und die Umsetzung des Wegs zum Softwareprodukt von Interesse.

3.3.4 Philosophie

In Abschnitt 3.3.1 wurde bereits beschrieben, dass im SINAMICS-Projekt ein komponenten- und levelbasiertes Vorgehen für die Erzeugung der Softwareprodukte gewählt wurde. Das bedeutet, dass es mehrere Level gibt in denen Komponenten generiert werden, um dann in Komponenten höher liegender Level, z. B. bei der Erzeugung einer CF[19]-Karte, wiederverwendet zu werden.
So erzeugt der Generierautomat aus einer kompletten Komponentenliste levelspezifische Teillisten, die anschließend sequentiell abgearbeitet werden. Nach der Erzeugung der Generierergebnisse eines Levels werden diese gepackt und in den Stage-VOBs für die weitere Verwendung archiviert.

Bei Build Forge geht der Weg zur Erzeugung eines Produkts nicht über Komponenten und Generierlevel, sondern dort wird bei der Erzeugung eines Softwareprodukts von **Projekten**, **Bibliotheken** und **Schritten** gesprochen. Dabei bestehen Projekte und Bibliotheken aus einem oder mehreren Schritten, die somit die kleinste Einheit bei der Erzeugung eines Produkts durch Build Forge sind. Die Ausführung von Schritten, Projekten und Bibliotheken kann in Build Forge parallel oder sequentiell erfolgen.
Hier stellt sich nun die Frage nach dem Unterschied zwischen einem Projekt und einer Bibliothek, da beide aus Schritten aufgebaut sind. Im Gegensatz zu Projekten besitzen Bibliotheken keinen eigenen Selektor, sondern benutzen den Selektor, mit dessen Projekt sie gerade verknüpft sind. Einzelne Projekte und Bibliotheken können in Build Forge miteinander verknüpft werden, wodurch die nachfolgenden Projekte bei einer Verknüpfung automatisch abgearbeitet werden.

Um mit Hilfe von Build Forge Software erzeugen zu können, müssen als erstes die zur Verfügung stehenden Generierserver in Build Forge als Serverressourcen eingerichtet werden. Dazu gehört auch das Anlegen der Kollektoren und Selektoren. Nach deren Einrichtung ist der nächste Schritt in Build Forge das Erstellen der einzelnen Projekte. Abb. 3.9 auf der nächsten Seite zeigt schematisch den Aufbau von Projekten. Bei Build Forge bestehen Projekte aus Schritten und projektspezifischen Einstellungen. Diese Schritte können so konfiguriert werden, dass sie innerhalb eines Projekts als Thread, d.h. parallel, abgearbeitet werden. Mit dieser Einstellung verteilt der Build Forge Server die Schritte so auf die Klienten, dass diese optimal von den Agenten ausgeführt werden. Neben den einzelnen Schritten können für jedes Projekt noch übergeordnete Projekteigenschaften eingestellt werden. Dazu gehört z. B. die Festlegung der maximalen Anzahl paralleler Prozesse, die ein Projekt starten kann. Zudem kann auch für jedes Projekt eine eigene Umgebung geschaffen werden, wodurch andere Generierumgebungen der Projekte, z. B. durch anders gesetzte Umgebungsvariablen, eingesetzt werden können. Des Weiteren bietet Build Forge die Möglichkeit Projekte untereinander zu verknüpfen. Man spricht dann auch von **Verkettung**. Die Verkettung zu einem Folgeprojekt kann nach der erfolgten Ausführung eines Projekts, von dessen Erfolg oder

[19] Compact Flash

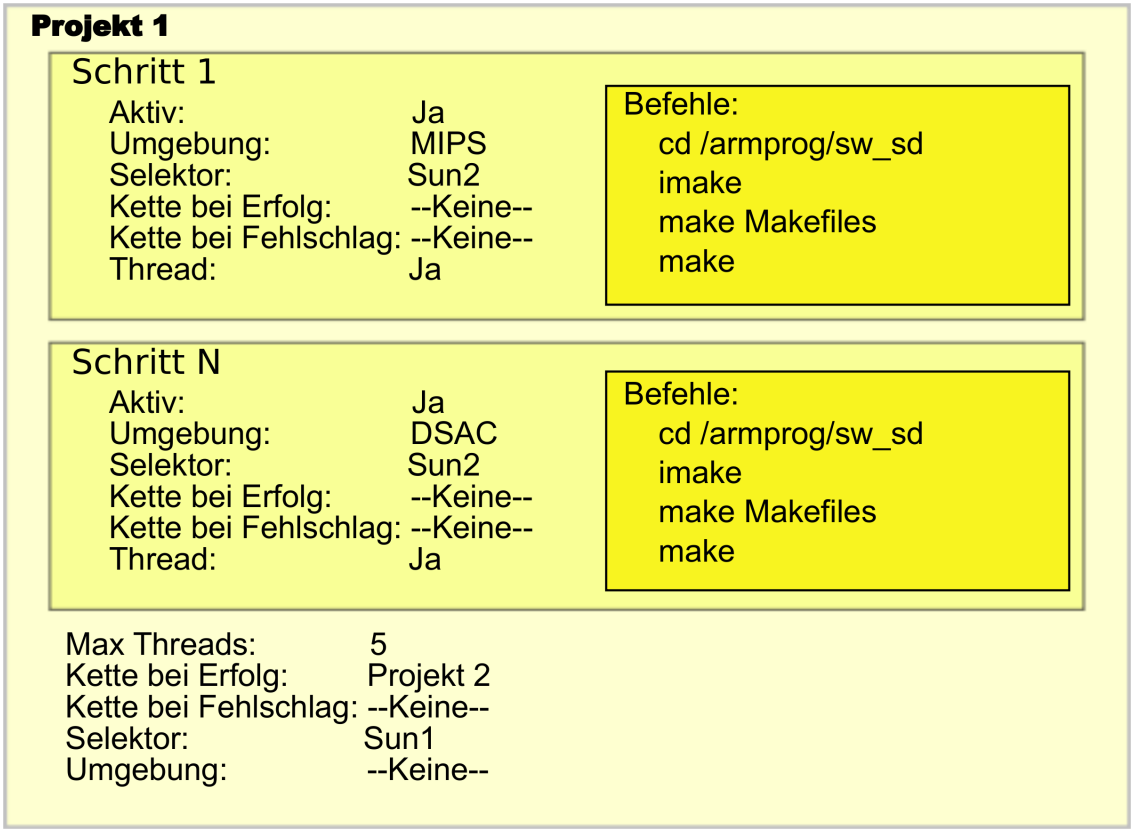


Abbildung 3.9: Aufbau von Projekten

Misserfolg abhängig, ausgeführt werden. Das bedeutet, dass im Fehlerfall ein anderes Projekt ausgeführt werden kann als im Erfolgsfall. Dadurch können immer wiederkehrende projektunabhängige Aufgaben einmal implementiert und nach Abschluss eines Projekts aufgerufen werden. Dies ermöglicht beispielsweise ein standardisiertes Vorgehen im Fehlerfall. Hier kann auch ein möglicher Einsatzort für Bibliotheken, die wie bereits erwähnt wurde, keinen eigenen Selektor besitzen, erkannt werden.

Nachdem nun die Eigenschaften der Projekte erklärt wurden, müssen auch noch die Schritte ausführlich betrachtet werden. Während Abb. 3.9 bereits schematisch der Aufbau eines Projekts aus Schritten und den Einstellungsmöglichkeiten darstellt, zeigt Abbildung 3.10 auf Seite 39 die Seite der Managementkonsole von Build Forge zur Erzeugung von Schritten innerhalb eines Projekts. Hierin sieht man, dass Schritte wieder aus allgemeinen Einstellungen, wie z. B. dem Selektor bestehen. Eine weitere Einstellungsmöglichkeit ist das Aktivieren oder Deaktivieren des aktuell zu bearbeitenden Schritts.
Im Feld *Befehl* wird die, durch einen passenden Agenten, auszuführende Befehlssequenz eingegeben. Wurden alle Einstellungen für den Schritt abgeschlossen, kann dieser abgespeichert werden, womit dieser in der oberen Schrittliste sichtbar wird. Neben der Befehlseingabe sind noch die Felder *Zugriff* und *Thread* von Interesse. Über das Feld *Zugriff* kann nur einer bestimmten Nutzergruppe die Ausführung eines Schrittes gestattet werden. Über das Feld *Thread* wird Build Forge mitgeteilt, ob der aktuelle Schritt parallel zu anderen aus der Schrittliste abgearbeitet werden darf oder ob er an

der Stelle der Schrittliste ausgeführt werden muss.

Durch die Aufteilung in Projekte, Bibliotheken und Schritte bei der Erzeugung von Softwareprodukten ist es möglich flexibel auf die Anforderungen an die Software zu reagieren. So kann einerseits in einem BF-Projekt innerhalb eines Schrittes das benötigte Softwareprodukt erzeugt werden. Andererseits kann auch eine komponenten- und levelbasierte Entwicklung umgesetzt werden.
So besteht z. B. die Möglichkeit eine Komponente durch ein Projekt darzustellen. Eine weitere Möglichkeit ist, die Projekte als Level zu betrachten und jede der zu erzeugenden Softwarekomponenten als einen Schritt im Projekt zu implementieren. Durch den Ansatz von verkettbaren Projekten kann Build Forge sehr flexibel in der Softwareentwicklung eingesetzt werden.

Ein wichtiger Unterschied zwischen dem verwendeten Generierautomaten und Build Forge ist zudem, dass in Build Forge bereits unterschiedliche Rollenmodelle für den Anwender existieren. Der verwendete Generierautomat von SINAMICS kann in seiner aktuellen Ausprägung nur von KM zum Erzeugen offizieller Softwarestände verwendet werden. So verwendet der Entwickler bei SINAMICS für das Erzeugen von CF-Karten für eigene Tests andere Skripts als KM. Da Build Forge von Haus aus die Rollenmodelle von z. B. KM und Entwicklern unterstützt, würde der Entwickler bei Build Forge auch die gleiche Vorgehensweise zum Erzeugen von Software wie KM verwenden.

Da nun ein kurzer Blick auf die unterschiedlichen Philosophien geworfen wurden, behandelt der folgende Abschnitt die Zusatzfunktionen von Build Forge gegenüber dem Generierautomaten.

3.3.5 Zusatzfunktionen

In Abschnitt 3.3.1 wurde der Generierautomat des Projekts SINAMICS vorgestellt. Zu den wichtigen Merkmalen zählt, dass der Generierautomat eine Sammlung mehrerer Skripts und Beschreibungsdateien ist, die nur unter Solaris oder unter Windows funktionsfähig sind. Das bedeutet, dass bei einer neuen Entwicklungsumgebung, wie z. B. Linux, zusätzlicher Aufwand in die Weiterentwicklung investiert werden muss. Als wichtiger Punkt sei hier die Synchronisation zwischen den unterschiedlichen Plattformen erwähnt. In diesem Punkt hat Build Forge einen Vorteil, da durch das in Abschnitt 3.3.2 vorgestellte Server- und Agentenkonzept das Problem der Synchronisierung bereits gelöst ist. So bringt Build Forge bereits Agenten für unterschiedliche Plattformen mit, deren Synchronisation zentral vom Build Forge Server übernommen wird. Dadurch ist die Erweiterung der Entwicklungsumgebung durch neue Hardwareplattformen auf die Installation der Agenten auf der neuen Hardware und dem Anlegen neuer Server, neuer Kollektoren und neuer Selektoren in der Managementkonsole abgeschlossen. Da Build Forge eine Liste aller zur Generierung verfügbaren Maschinen enthält, ist auch die par-

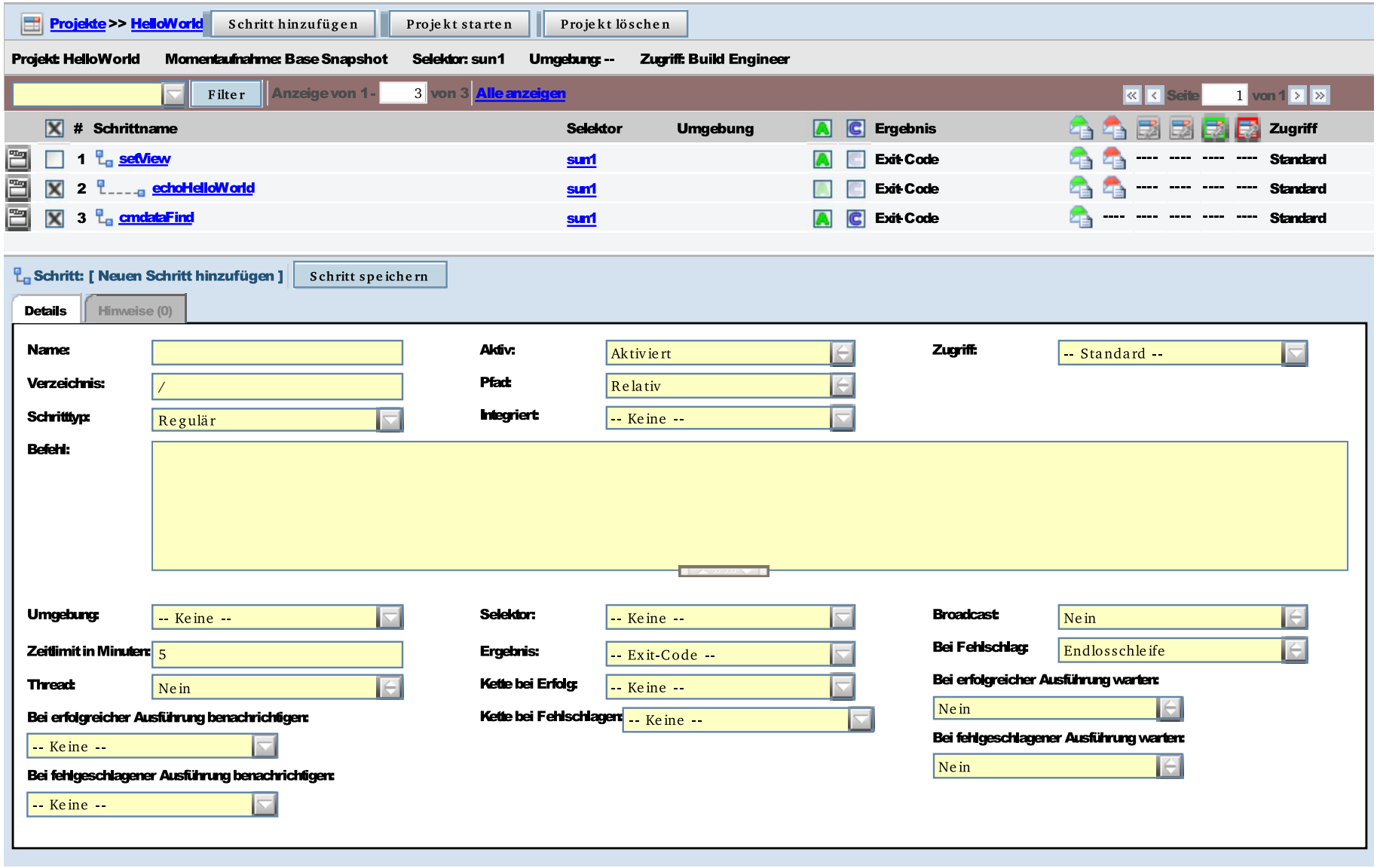


Abbildung 3.10: Aufbau eines BF-Schrittes

allele Erzeugung unter Windows möglich. Diese Fähigkeit besitzt der aktuelle Generierautomat, wie bereits erwähnt, nur innerhalb der Erzeugung einer SUN-Komponenten.
Eine weiterer Vorteil von Build Forge ist die Möglichkeit Umgebungen zu definieren, die für einzelne Schritte gesetzt werden können und diesen unterschiedliche Umgebungsvariablen bereitstellen.
Weiterhin bietet Build Forge eine ausführliche Möglichkeit zur Analyse der Generierläufe. So lässt sich z. B. automatisiert das prozentuale Verhältnis der erfolgreichen und fehlgeschlagenen Generierungen darstellen.

4 Wichtige Kriterien für den Einsatz von Build Forge

Bevor Build Forge in einem Projekt oder sogar projektübergreifend, wie z. B. im Fall von Siemens DT, eingesetzt werden kann, muss sichergestellt werden, dass Build Forge mindestens die in diesem Kapitel vorgestellten Kriterien erfüllt.

Der folgende Abschnitt 4.1 geht auf die wichtigste SINAMICS-spezifische Anforderung, der Unterstützung eines komponenten- und levelbasierten Vorgehens, an Build Forge ein. In den weiteren Abschnitten 4.2 bis 4.5 werden die allgemeingültigen Kriterien für einen projektübergreifenden Einsatz und die Verwendung von Build Forge in neuen Softwareprojekten behandelt.

4.1 Komponenten- und levelbasiertes Vorgehen

Wie bereits in den vorangegangen Kapiteln erläutert wurde, verwendet SINAMICS ein komponenten- und levelbasiertes Vorgehen. Durch dieses Vorgehen ist es möglich eine Optimierung der Laufzeit zu erreichen, da Basiskomponenten eingerichtet werden können, deren Generierergebnisse erst in Komponenten höherer Level direkt weiterverarbeitet werden. So existiert z. B. eine Komponente die sämtliche Linkvarianten der Firmware generiert und diese dann in nachfolgenden Leveln in den zugehörigen Firmwarekomponenten kopiert und archiviert. Hätte KM in Zusammenarbeit mit dem Toolsmith diesen Weg so nicht gewählt, würden durch die Firmwarekomponenten bei der Generierung mehrmals identische Softwareteile erzeugt werden. Weiterhin wird durch dieses Vorgehen neben der direkten Benutzung der Generierergebnisse aus niedrigeren Leveln, wie bei den Linkvarianten der Firmware, auch die Wiederverwendung einzelner Komponenten als Bausteine in Komponenten höherer Level denkbar. Diese Wiederverwendung bereits erzeugter Komponenten wird in den Feldern CARD_COMPLIST oder CMGET_COMPLIST innerhalb der cmdata-Datei jeder Komponente abgebildet. CARD_COMPLIST wird bei der Erzeugung der CF-Karten für die Fertigung verwendet. Hier werden Komponenten niedrigerer Level eingezogen und mit Hilfe bestimmter Erzeugungsvorschriften wird eine Verzeichnisstruktur erstellt. Die zu verwendenden Komponenten werden bei den CF-Karten innerhalb des CARD_COMPLIST-Feldes der zugehörigen cmdata_versions-Datei beschrieben (cmdata 4.2 auf der nächsten Seite). Wie man in Zeile 11 von cmdata 4.2 auf der nächsten Seite auch erkennen kann, muss die eingezogene Komponente nicht zwingend in der aktuellen Integration erzeugt worden sein.

Neben der Möglichkeit die Komponenten über CARD_COMPLIST einzuziehen, gibt es noch das Feld CMGET_COMPLIST, das bei allen Komponenten verwendet wird, die kei-

```
1  ARM_I4402_0592_CF_M_S120: 04.40.06.00
2  {
3     ...
4     INTEG_LABEL:          ARM_I4402_0592
5     CMGET_COMPLIST:       NOT_DEFINED
6     CARD_COMPLIST:
7     {
8        SD_TargetM_S120  : 04.40.06.00  : MASTER
9        SD_DEU_lang      : 04.40.06.00  : INTERN
10       SD_ENG_lang      : 04.40.06.00  : INTERN
11       SAC_PSAXM_FW     : 02.60.41.00  : INTERN
12       AOP_SOC2         : 04.40.32.00  : EXTERN=AOP_SOC2_04.40.32.00
13    }
14    COMPONENT:           CF_M_S120
15     ...
16 }
```

cmdata 4.2: *card_complist*

ne Verzeichnisstruktur als Generierergebnis besitzen. Im Unterschied zu CARD_COMPLIST werden dort die Komponenten nicht über ihre Version, sondern über den Stempel der zu verwendenden Version referenziert und in das angegebene Verzeichnis kopiert (cmdata 4.3). Mit Hilfe von CMGET_COMPLIST können auch Programme, die zur Erzeugung der Komponente notwendig sind, in einer bestimmten Version geholt und verwendet werden.

```
1  ARM_I4402_0592_SAC_PSAXM_BIN: 04.40.06.00
2  {
3     INTEG_LABEL:          ARM_I4402_0592
4     CMGET_COMPLIST:
5     {
6        SAC_PSAXM_FW : ARM_I4402_0592 : $GENTEST_DIR/SAC_PSAXM_FW :
             -tmpdest
7     }
8     CARD_COMPLIST:        NOT_DEFINED
9     COMPONENT:            SAC_PSAXM_BIN
10 }
```

cmdata 4.3: *cmget_complist*

Dieses Verfahren ermöglicht Optimierungen hinsichtlich der Anzahl zu generierender Komponenten, da nur die geänderten Komponenten generiert werden müssen und andere Komponenten in einer bestimmten Version verwendet werden können. Anhand Abb. 4.1 wird verdeutlicht, wie das Arbeiten mit unterschiedlichen Komponenten und Leveln mit Hilfe von CMGET_COMPLIST funktioniert. Deutlich erkennt man in Abb. 4.1,

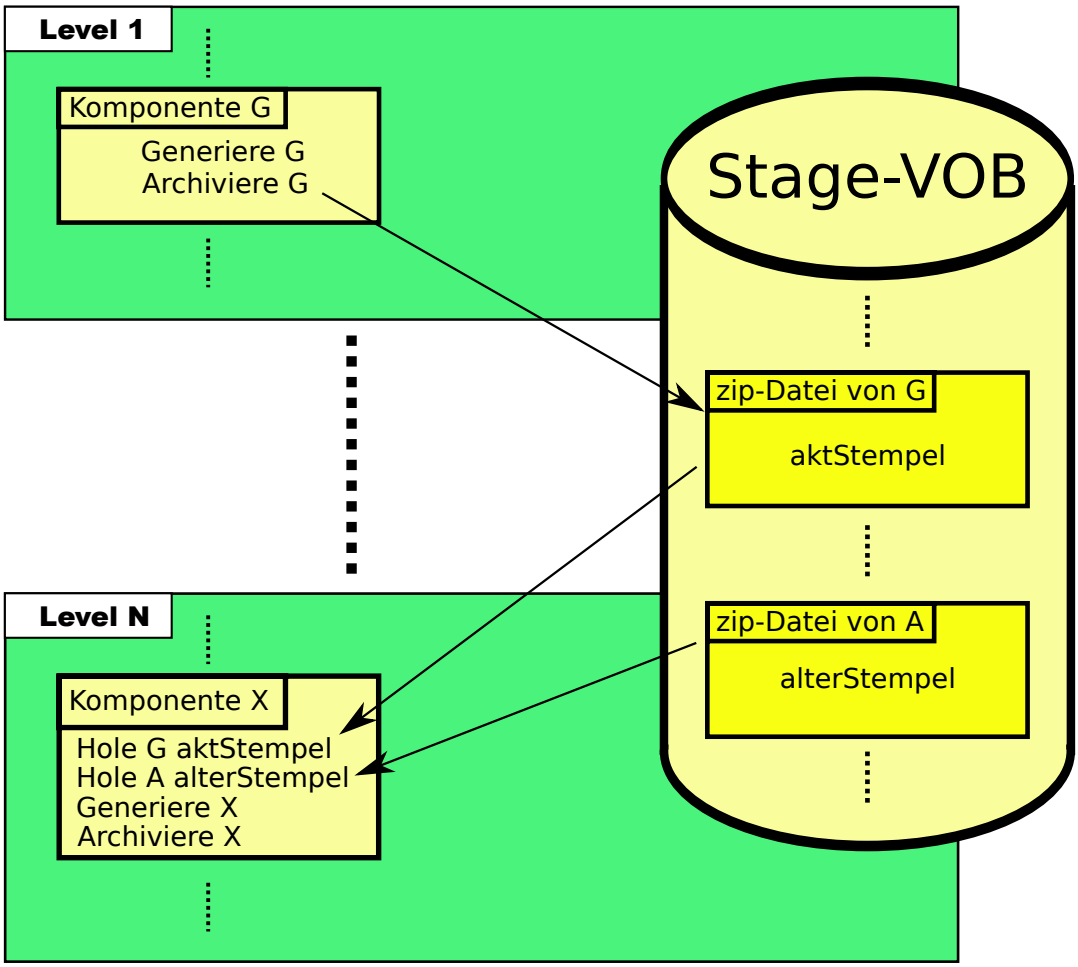


Abbildung 4.1: Aufbau von Projekten

wie in *Generierlevel 1* die *Komponente G* erzeugt und die zugehörige zip-Datei in einem Stage-VOB abgelegt wird. Anschließend wird die zip-Datei im Rahmen der Generierung mit dem Integrationsstempel versehen und kann somit in nachfolgenden Leveln über diesen referenziert werden. Wird nachfolgend in einem *Level N* die *Komponente X* generiert, die auf zwei bereits erzeugten Komponenten, nml. *Komponente G* und *Komponente A* aufsetzt, so werden diese mit Hilfe des Integrationsstempels in ein temporäres Verzeichnis kopiert. In der Abbildung wird auch deutlich, dass die Komponenten unterschiedliche Stempel besitzen können und folglich auch alte Versionen eingezogen werden können. So wird *Komponente A* in einer alten Version wiederverwendet.
Durch diese Generierabläufe wird deutlich, dass Build Forge, um den bei SINAMICS verwendeten Generierautomaten abzulösen, die Fähigkeit besitzen muss ein komponenten- und levelbasiertes Vorgehen abzubilden. Ist dies nicht der Fall, muss der etablierte

Generierablauf aufwendig angepasst werden. Sollte diese Umsetzung nur mit erheblichem personellem und zeitlichem Aufwand möglich sein, wird sich die Projektleitung von SINAMICS mit Sicherheit gegen den Einsatz von Build Forge als Alternative für den verwendeten Generierautomaten aussprechen.

4.2 Einhaltung einer angemessenen Generierzeit

Ein weiterer entscheidender Punkt, der vor einem Einsatz von Build Forge betrachtet werden muss, ist der Blick auf die Generierzeit. Dieser Punkt spielt jedoch nur bei der Umstellung von Softwareprojekten eine Rolle, da nur in diesem Fall ein Vergleich mit bereits erfolgten Generierungen erfolgen kann. Dieser Punkt spielt vor allem bei großen Softwareprojekten eine wichtige Rolle, da bei diesen sehr schnell auf kritische Fehler mit einem Hotfix reagiert werden muss. Da z. B. bei SINAMICS die Generierzeit einer Version bis zur Erzeugung der CF-Karten (Generierlevel 5) auf dem main-Branch ungefähr 14 Stunden dauert, darf die Umstellung auf ein neues Generierwerkzeug die Laufzeit nicht merklich verlängern.

Bei der Erzeugung eines SINAMICS-Standes existieren zwei wichtige Zeitpunkte.
Einer davon ist der Zeitpunkt, ab dem der Integrationstest mit den offiziell generierten CF-Karten testen kann. Durch das levelbasierte Vorgehen ist es möglich, die Komponenten so zu sortieren, dass nur die kartenrelevanten Komponenten vor diesem Zeitpunkt erzeugt werden. Das hat zur Folge, dass die Karten zum frühest möglichen Zeitpunkt bereitstehen und getestet werden können, während die restlichen Komponenten in nachfolgenden Leveln generiert werden.
Der andere entscheidende Zeitpunkt ist das Ende der Komplettgenerierung. Dieser Punkt ist für das Abschließen eines Softwarestandes von entscheidender Bedeutung, da das Skript fincmgen (Abschnitt 3.3.1) für einen Softwarestand erst aufgerufen werden kann, wenn das Skript docmgen (Abschnitt 3.3.1) fehlerfrei beendet wurde.
Um einen realistischen Überblick über die Generierzeit zu bekommen, wird in der Tabelle aus Anhang B ausschnittsweise die Generierzeit einzelner Erzeugungsschritte im docmgen dargestellt. In der Tabelle erkennt man die einzelnen Generierlevel, die nochmals in einzelne Schritte GETSTAGE (STEP 12.LEVEL), GENERATE (STEP 13.LEVEL) und ARCHIVE (STEP 1.LEVEL) unterteilt sind. Deutlich zeigt sich in der Tabelle aus Anhang B auch, das Vorgehen des aktuellen Generierautomaten bei der Verwendung benötigter Komponenten aus anderen Leveln in „STEP 12.LEVEL“. Gefolgt vom Generierteil im „STEP 13.LEVEL“ und dem Anschließenden Archivieren mit „STEP 14.LEVEL“. In der letzten Tabellenzeile wird die Laufzeit der einzelnen Schritte aufsummiert, um einen Überblick über die gesamte Generierzeit zu erhalten. Bei der hier betrachteten Generierung des Hauptentwicklungszweigs ergibt sich eine Laufzeit von 34 Stunden und 53 Minuten für 400 Komponenten. Durch diese ohnehin schon langen Generierzeiten wird es nicht nur schwer schnell auf gemeldete Fehler zu reagieren, sondern auch die Weiterentwicklung der Software wird durch diese Zeiten verzögert. Denn neben dieser

reinen Generierzeit muss bedacht werden, dass die Skripts prepcmgen und fincmgen für einen Softwarestand ausgeführt werden muss, bevor dieser erzeugt werden kann. Hierbei beträgt die Dauer für die Vorbereitung ungefähr 3-4 Stunden und für den Abschluss ungefähr 2-3 Stunden. Abb. 4.2 zeigt noch einmal die vorgestellten Zeiten der Vorbereitung (prepcmgen), der Generierung (docmgen) und des Abschließens (fincmgen). Wie der Abb. 4.2 entnommen werden kann, dauert alleine das Öffnen und Ab-

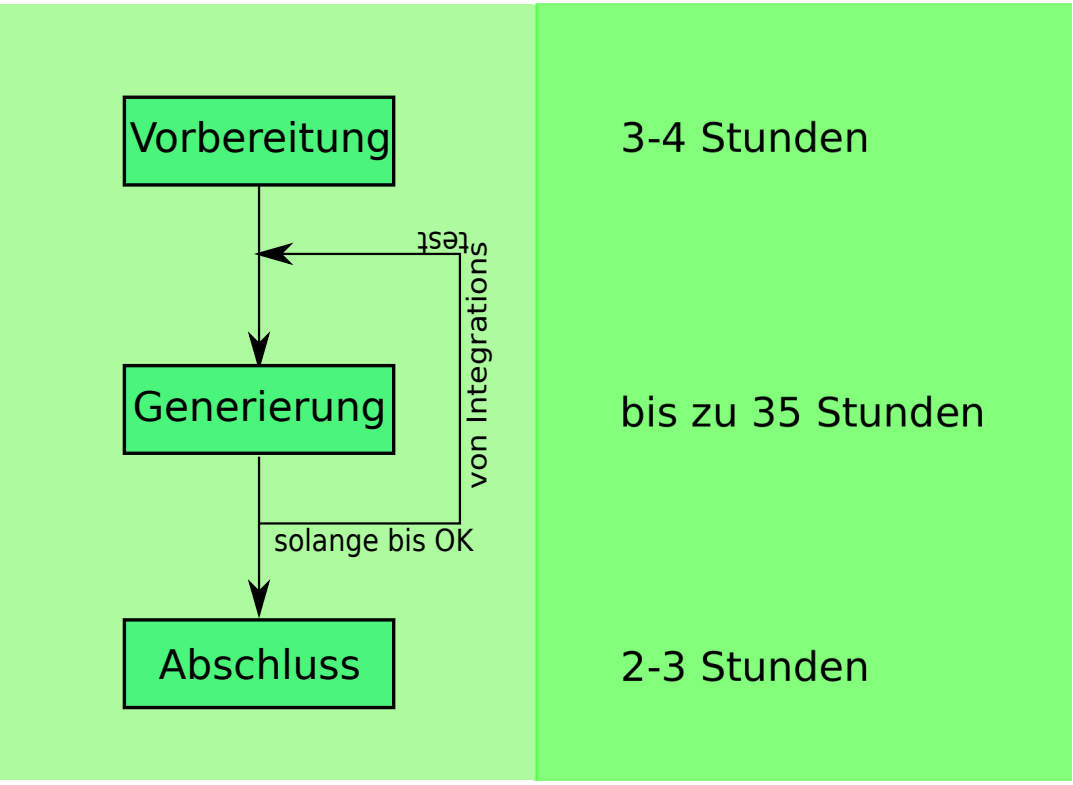


Abbildung 4.2: Laufzeiten der einzelnen Vorgänge

schließen eines Softwarestandes ungefähr einen Arbeitstag. Die langen Generierzeiten haben nicht nur Auswirkungen auf einen aktuell zu generierenden Stand, sondern wirken sich auch direkt auf die Planung weiterer Stände auf diesem Branch und sogar branchübergreifend aus. Jedoch ist die Auswirkung auf den zu generierenden Brach besonders gravierend, da auf einem Branch immer nur eine offizielle Version erzeugt werden kann. Somit schrumpft der Zeitraum, den der Integrator, der festlegt welche Merges für eine Version auf dem Branch benötigt werden, immer weiter zusammen, da der Zeitraum für der Integration normalerweise zwischen fincmgen und prepcmgen liegt. Verlängert alleine der Einsatz eines neuen Generierwerkzeuges die Dauer der Generierung, so wird die Zeit bis zur Freigabe einer kritischen Fehlerbehebung unnötig in die Länge gezogen. Daher darf Build Forge keinen negativen Einfluss auf die Dauer der Generierung haben. Hierzu muss noch gesagt werden, dass Build Forge laut IBM von Haus aus die Fähigkeiten zur Parallelgenerierung auf mehreren Servern mitbringt, wodurch sich die Generierzeit eigentlich verkürzen sollte. Im Vergleich dazu verarbeitet der aktuell verwendete Generierautomat die zu erzeugenden Komponenten sequentiell. Das wiederum bedeutet, dass eine Verlängerung der Generierzeit durch Build Forge inner-

halb einer Komponente durchaus zu verkraften wäre, wenn die Nutzung der möglichen Parallelgenerierung die Gesamtlaufzeit reduziert.

4.3 Lauffähigkeit auf unterschiedlichen Plattformen

Die Lauffähigkeit auf unterschiedlichen Entwicklungsplattformen ist nicht nur für Projekte der Siemens AG, sondern auch für jedes andere Softwareprojekt relevant, da die Anforderungen von Softwareprojekten an die Entwicklungsplattform sehr unterschiedlich sein können. So kann z. B. in einem Projekt die Notwendigkeit bestehen Programme unter MAC OSX zu entwickeln, während ein anderes nur unter Windows arbeitet . Mit der Forderung nach der Lauffähigkeit auf unterschiedlichen Plattformen wird zum einen die gemischte Umgebung, wie z. B. im SINAMICS-Projekt, abgedeckt. Zum anderen werden durch diese Forderung auch Softwareprojekte abgedeckt, die zwar nur eine Entwicklungsplattform verwenden, aber vielleicht eine Plattform verwenden, die von anderen Programmen, wie z. B. dem Microsoft Foundation Server[20], nicht unterstützt werden. Betrachtet man die Softwareentwicklung allgemein, so finden sich auch dort viele unterschiedliche Entwicklungsplattformen, wie z. B. MAC OSX, Windows und Linux, die ein Werkzeug für die Automation von Generierungen unterstützen sollte. Diese Vielfalt der Entwicklungsplattformen findet sich beispielsweise bereits in den einzelnen Projekten bei Siemens DT. So befinden sich z. B. bei Siemens DT in den großen Softwareprojekten SINAMICS, SINUMERIK und SIMOTION schon drei unterschiedliche Plattformen in Verwendung. Die SINAMICS-Entwicklung erfolgt größtenteils unter Solaris, wobei auch Windows bei einigen Komponenten als Entwicklungsplattform zum Einsatz kommt. Demgegenüber liegt der Entwicklungsschwerpunkt bei SIMOTION größtenteils unter Windows und nur bei wenigen Komponenten unter Solaris. Das dritte große Softwareprojekt SINUMERIK besitzt als Entwicklungsplattformen Windows, Linux und Solaris. Das bedeutet, dass ein Generierwerkzeug, wie z. B. Build Forge, das auch projektübergreifend eingesetzt werden soll, auf mindestens diesen drei Plattformen laufen muss.
Betrachtet man nun wieder allgemein den Einsatz von Build Forge in großen Softwareprojekten, deckt die Untersuchung in diesem Rahmen auf jeden Fall schon einmal den Betrieb von Build Forge unter Windows und Solaris ab. Da bei Siemens DT als Server für ClearCase SUN-Server eingesetzt werden, bietet es sich an die bereits vorhandene Serverinfrastruktur für Build Forge zu verwenden. Das bedeutet keinen zusätzliche Mehraufwand an Pflege und Wartung neuer Serverplattformen. In Abschnitt 3.3.2 konnte bereits erkannt werden, dass Build Forge aufgrund der Server/Agenten-Architektur in gemischten Umgebungen eingesetzt werden kann. So kann der BF-Server für das Projekt SINAMICS auf SUN-Servern installiert werden, während plattformabhängige Vorgänge, wie z. B. das Ausführen der Generate-Teile der Komponenten, durch die auf den Entwicklungsplattformen installierten Agenten erfolgt. Für die Untersuchung der Lauffähigkeit auf unterschiedlichen Plattformen wird im SINAMICS-Projekt der Build Forge Server auf einer SUN installiert. Im Installationspaket des Build Forge Server befinden

[20] Konfigurationsmanagement-Werkzeug von Microsoft das nur Windowsbasierende Systeme unterstützt

sich die Installationsroutinen für die Plattformen Windows, Linux und Solaris. Die Agenten wurden für die Untersuchungen bzgl. der Diplomarbeit auf einem Windows-PC und mit Hilfe des UNIX-Administrators auf einer SUN installiert. Da die Diplomarbeit im Rahmen von SINAMICS die Tauglichkeit bzgl. der Unterstützung unterschiedlicher Entwicklungsplattformen betrachtet, müssen vor einem projektübergreifenden Einsatz von Build Forge natürlich noch die weiteren im Einsatz befindlichen Plattformen untersucht werden. Für SIMOTION können die Ergebnisse der Plattformunterstützung übernommen werden, da sich zwischen SINAMICS und SIMOTION nur der Schwerpunkt der Entwicklungsumgebung unterscheidet. Für SINUMERIK jedoch muss auf jeden Fall noch die Verwendbarkeit von Build Forge auf Linux-Rechnern überprüft werden.

Da als allgemeine Anforderung der Betrieb auf unterschiedlichen Plattformen besteht, kann als spezielle Anforderung bei der Softwareentwicklung auch noch die Notwendigkeit der Synchronisation bei der Nutzung unterschiedlicher Plattformen, wie z. B. bei SINAMICS, hinzukommen.

4.4 Synchronisation bei der Nutzung unterschiedlicher Plattformen

Warum muss Build Forge die Synchronisation unterschiedlicher Plattformen unterstützen?
Werden in einem Softwareprojekt Softwareteile auf unterschiedlichen Plattformen entwickelt und ist eine zeitliche Abhängigkeit innerhalb der Generierung gegeben, so müssen diese Generierschritte untereinander synchronisiert werden. Hierunter fällt neben der Synchronisation der Generierung auf unterschiedlichen Plattformen auch die Synchronisation bei der Parallelgenerierung auf gleichen Plattformen. Dies ist insbesondere für das in Abschnitt 3.3.2 vorgestellte „Threading“ von Bedeutung. Ein Beispiel für die Notwendigkeit der Synchronisation findet sich im SINAMICS-Projekt bei der komponenten- und levelbasierten Generierung. Wie bereits in Abb. 3.4 auf Seite 28 aufgezeigt wurde, erfolgt die Erzeugung von Softwarekomponenten bei SINAMICS sowohl unter Windows als auch unter Solaris. Hierbei wird Windows alleine für die Abarbeitung der Generierbefehle verwendet, während alle anderen Aufgaben innerhalb der Erzeugung, wie z. B. das Kopieren benötigter Software als auch das Archivieren, unter Solaris erfolgt. Dies erfordert, dass die Generierung auf den Suns über den Fortgang der Erzeugung der Komponenten am PC informiert wird. Diese Benachrichtigung erfolgt bei SINAMICS durch das in Abschnitt 3.3.1 auf Seite 27 beschriebenen Tokenverfahren. Für den erfolgreichen Einsatz innerhalb von SINAMICS muss Build Forge daher auch die Synchronisation zwischen den eingesetzten Plattformen erlauben. Wegen dem Einsatz des Tokenmechanismuses innerhalb von SINAMICS muss Build Forge eine Synchronisation der unterschiedlichen Plattformen auf jeden Fall gestatten. Wie bereits in Abschnitt 4.3 gesehen wurde, verwenden die anderen Projekte ebenfalls mehr als eine Plattform zur Entwicklung ihrer Software. Daher wird auch für diese Projekte eine Synchronisation

bei der Erzeugung von Komponenten benötigt. Man kann daher davon ausgehen, dass auch die anderen Projekte ein Interesse an der Fähigkeit von Build Forge zur Synchronisation der Generiertätigkeiten auf unterschiedlichen Serverplattformen haben.
Sollte Build Forge diese Anforderung nicht erfüllen können, ist es für den allgemeinen projektübergreifenden Einsatz bei Siemens DT nicht geeignet. Jedoch verhindert diese Aussage nicht generell den Einsatz von Build Forge in Softwareprojekten, da eine Synchronisation ja nicht in jedem Projekt gefordert sein muss.

4.5 Unterstützung von Versionsverwaltung

Da es sich bei großen Softwareprojekten empfiehlt die Entwicklung durch Software zur Versionsverwaltung zu unterstützen, sollte Build Forge natürlich auch mit diesen Werkzeugen, wie z. B. ClearCase oder Subversion, zusammenarbeiten können.

Wegen der durch das Projekt vorgegebenen Rahmenbedingungen der Diplomarbeit und dessen Einsatz von ClearCase als Werkzeug zur Versionsverwaltung muss Build Forge mit ClearCase kompatibel zusammenarbeiten. Mit Hilfe der Adaptoren, die im Rahmen der Diplomarbeit nicht untersucht werden, können laut IBM weitere Programme zur Versionsverwaltung in Build Forge integriert werden.

ClearCase ist bei Siemens DT flächendeckend als Werkzeug zur Softwareversionsverwaltung (siehe Kapitel 2) im Einsatz. Insbesondere muss Build Forge die Arbeit mit unterschiedlichen Views gestatten. Wie in Kapitel 2 dargelegt wurde, wird eine View benötigt, um verschiedene Versionen von Objekten sichtbar zu machen. Erst diese Sichten ermöglichen das Erzeugen von Software auf unterschiedlichen Branches. So wird der aktuelle Generierautomat in einer Integrationsview des zu generierenden Branches aufgerufen und die zu dieser Integration gehörenden cmdata-Datensätze durch den Generierautomaten innerhalb dieser View abgearbeitet.
Da der Build Forge Server und die Agenten jedoch unabhängig von einer gesetzten View als Dienst auf den Generierservern laufen, ist es notwendig, dass bei einer Generierung mit Build Forge, die aktuell zu verwendende View für die Agenten gesetzt werden kann.
Des Weiteren muss neben der Unterstützung der Views auch der Aufruf von ClearCase-Befehlen unterstützt werden. Der Zugriff auf ClearCase-Befehle erfolgt plattformübergreifend über das von ClearCase mitgelieferte Programm *cleartool*. *cleartool* startet eine ClearCase-Shell und erlaubt das Abarbeiten von ClearCase-Befehle, wie z. B. das Ausführen des Check-in-Befehls. Neben dem Arbeiten in einer gesetzten ClearCase-Shell kann der Befehl auch als Parameter übergeben werden, wodurch die ClearCase-Shell nur für diesen Befehl gesetzt wird. Um zu zeigen, dass mit beiden Varianten auch das identische Ergebnis erzielt wird, zeigt Abb. 4.3 auf der nächsten Seite das Arbeiten in der gesetzten ClearCase-Shell, während Abb. 4.4 auf der nächsten Seite das Arbeiten mit der Befehlsübergabe als Parameter zeigt.

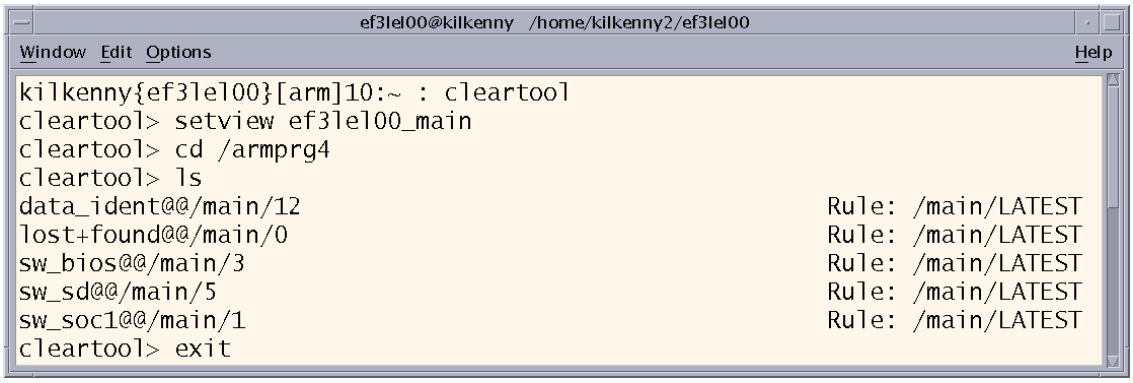


Abbildung 4.3: ClearCase Shell

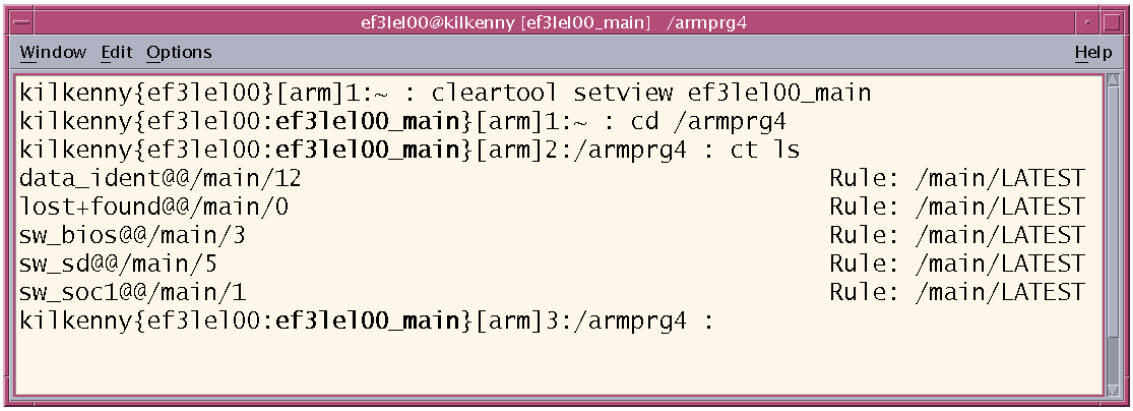


Abbildung 4.4: ClearCase cleartool

In allen Softwareprojekten werden momentan Shell- oder Perlskripts als Generierautomat eingesetzt, die über Systemaufrufe den Zugriff auf Befehle, wie z. B. *cleartool*, erlauben. So kann zum einen mit den Rückgabewerten der Befehle der Status überprüft werden und zum anderen mit den Ergebnissen der Befehle innerhalb der Generierautomaten weitergearbeitet werden.

Von der Fähigkeit der Interaktion zwischen ClearCase und Build Forge hängt die Eignung von Build Forge bei der Siemens DT als Ersatz für einen Generierautomaten innerhalb eines Projekts oder sogar als projektübergreifende Alternative ab. Auch ist der Einsatz von Build Forge als Generierwerkzeug in neuen Softwareprojekten ohne die Unterstützung eines Versionsverwaltungssystems nicht sinnvoll.

5 Testumgebung

Um die Fähigkeiten von Build Forge untersuchen zu können, ist es notwendig eine funktionierende Testumgebung zu schaffen. In diesem Kapitel wird die Vorgehensweise zur Erzeugung einer Testumgebung und die Besonderheiten bei der Erstellung der Testumgebung bei der Arbeit mit Build Forge beschrieben.

5.1 Testkonzept

Da für die Untersuchung von Build Forge auf einer bereits bestehenden Umgebung aufgesetzt wird, muss auf deren Besonderheiten für die Untersuchung Rücksicht genommen werden. Zu diesen Besonderheiten gehört, dass innerhalb von SINAMICS, in dessen Umfeld die Untersuchung stattfindet, ClearCase als Software für die Versionsverwaltung verwendet wird. Das bedeutet, dass sich die Untersuchungen der Unterstützung von Build Forge von Software zur Versionsverwaltung auf ClearCase beschränken. Weiterhin wird durch den Projektrahmen von SINAMICS mit einer gemischten Entwicklungsumgebung Solaris und Windows als zu testende Plattformen für den Einsatz von Build Forge vorgegeben. Jedoch sollten die Untersuchungsergebnisse der Diplomarbeit bzgl. des Einsatzes der Agenten auf anderen Entwicklungsplattformen wie z. B. Linux oder MAC OSX übertragbar sein, da IBM für diese Betriebssysteme Agenten bereitstellt. Neben bereits kompilierten Agenten, kann bei IBM auch der Quellcode der Agenten bezogen werden und somit der Agent für die benötigte Plattform erzeugt werden.
Nach diesem kurzen Überblick der Randbedingungen innerhalb von SINAMICS, folgt nun eine kurze Beschreibung der Installation von Build Forge anhand der aktuellen Entwicklungsumgebung von SINAMICS.

5.1.1 Installation von Build Forge

Abb. 5.1 auf der nächsten Seite (nach [Koc07a, S. 3]) gibt noch einmal einen kurzen Überblick über die Umgebung, in der Build Forge getestet wird. Man erkennt, dass sowohl unter Windows am PC als auch unter Solaris auf der SUN entwickelt wird. Die Datenaustausch zwischen den Plattformen erfolgt über Samba. Die Aufgabe bei der Installation von Build Forge bestand nun darin, den Build Forge Server mit der Managementkonsole und die Agenten in die bestehende Umgebung zu integrieren.

Wie bereits in Abb. 3.7 auf Seite 32 gezeigt wurde, werden für den Betrieb von Build Forge neben dem BF-Server noch eine Datenbank und die plattformspezifischen Agenten benötigt. Nach Rücksprache mit der UNIX-Administration wurde entschieden Build Forge und die Datenbank auf einer im Produktivsystem eingesetzten SUN zu installieren, da auch die VOB-Server von ClearCase auf Solarissystemen installiert sind. Auch

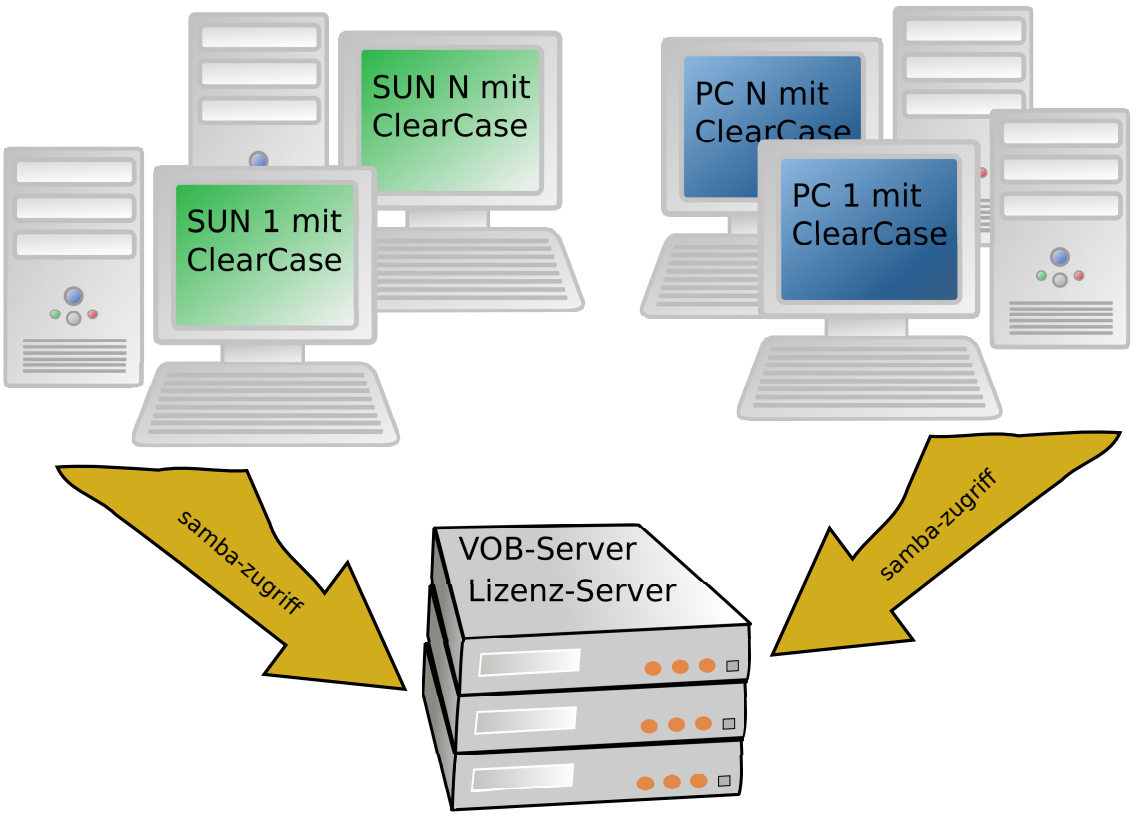


Abbildung 5.1: aktuelle Umgebung

war durch die Installation im Produktivsystem der Zugriff auf die Build Forge Lizenzen über die bereits eingerichteten Lizenzservern möglich. Es mussten lediglich noch die Build Forge Testlizenzen eingespielt werden. Ein weiterer Grund für die Installation von Build Forge unter Solaris war auch, dass nicht noch Server anderer Betriebssysteme in die Infrastruktur gebracht werden sollten, durch die dann möglicherweise unerwünschte Nebenwirkungen auftreten könnten. Durch die Installation mit Hilfe der UNIX-Systemadministration auf einer SUN erfolgt der Test im Produktivsystem unter dem Einsatz realer Daten.

Für Untersuchungs- und Testzwecke ist es ausreichend den BF-Server, die Datenbank und einen Agenten auf einer Maschine zu betreiben. Jedoch wird von diesem Betrieb im realen Einsatz abgeraten. Für den regulären Betrieb empfiehlt IBM diese Komponenten auf unterschiedlichen Servern zu installieren, um die Performance durch häufige Datenbankzugriffe und Zugriffe auf die Managementkonsole nicht negativ zu beeinflussen. In Abb. 5.2 auf der nächsten Seite erkennt man, dass auf einer SUN die MySQL-Datenbank, der BF-Server mit der Management Konsole und ein SUN-Agent installiert wurden, während auf einem Windows-PC nur ein Agent installiert wurde. Für den Betrieb und Test in einer gemischten Umgebung wurde der Agent auf einem Windows-PC installiert, der normalerweise auch in der offiziellen Generierung eingesetzt wird, aber während der Tests sicherheitshalber nicht verwendet wird.

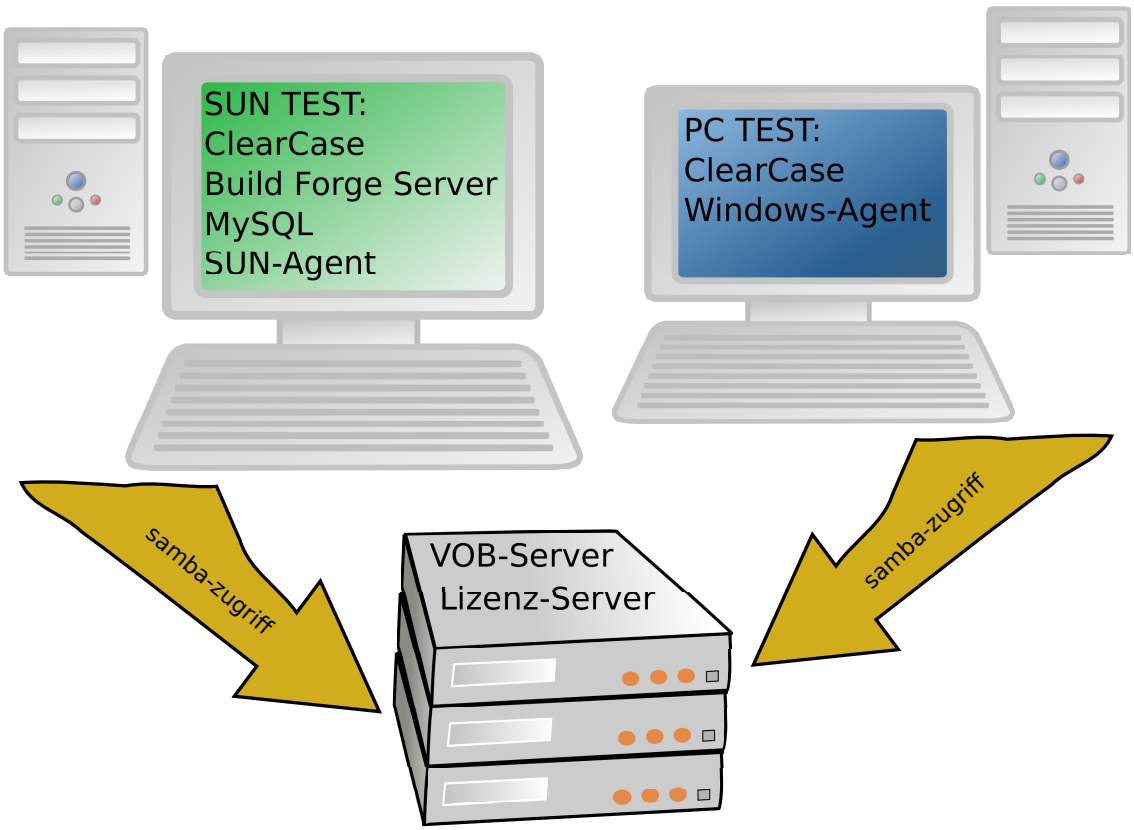


Abbildung 5.2: Umgebung mit Build Forge

Um mögliche Auswirkungen durch die Installation von Build Forge auf offizielle Integrationen auszuschließen, wurde die mit Build Forge installierte SUN ebenfalls aus der verteilten Generierung ausgeschlossen.
Da Build Forge in der offiziellen Umgebung installiert wurde, ist es notwendig sich eine Testumgebung innerhalb der Produktivumgebung aufzubauen.

5.1.2 Aufbau des Testprojekts

Dieser Abschnitt beschreibt die notwendigen Schritte, um innerhalb der Produktivumgebung Build Forge zu testen ohne Einfluss auf das Produktivsystem zu nehmen.

Neben dem großen Softwareprojekt SINAMICS existieren noch zwei kleinere Projekte, die ebenfalls mit dem Generierautomaten aus dem SINAMICS-Projekt erzeugt werden. Da für eine sinnvolle Testgenerierung innerhalb des SINAMICS-Projekts ca. 300 Komponenten benötigt werden, wurde für die Tests eines dieser kleineren Projekte ausgewählt. Dieses umfasst insgesamt 61 zu erzeugende SUN- und PC-Komponenten.

Vor der Untersuchung von Build Forge mit dem ausgewählten Projekt waren noch folgende Schritte notwendig:

1. Export der zu generierenden Datensätze in ein lokales Integrationsverzeichnis. Das entspricht dem Durchführen von Schritt 5 aus Abb. 3.2 auf Seite 26 bei der Vorbereitung.
2. Anlegen der passenden Stage-VOB-Struktur der zu generierenden Komponenten für Tests in einem VOB. Dazu wurde ein ohnehin für Testzwecke vorgesehener VOB verwendet.
3. Nach dem Anlegen der Stage-VOB-Struktur, musste in den cmdata-Datensätzen der Komponenten das Feld STAGE_FILE auf die neue Struktur angepasst werden.
4. Abbilden der Erzeugung der Komponenten durch den Generierautomaten mit Hilfe von Level und Komponenten in Build Forge durch Schritte und Projekte.

Die oben genannten notwendigen Schritte zur Erzeugung der Testumgebung werden in Abb. 5.3 auf der nächsten Seite parallel zu dem Vorgehen bei einer offiziellen Integration dargestellt. Blau dargestellt ist das Aussehen der Testumgebung, während Gelb das Vorgehen einer offiziellen Integration darstellt. Es zeigt sich, dass innerhalb der Datensätze wenige Felder angepasst und die Stage-VOB-Struktur im „driveplay“-VOB nachgebildet werden musste.

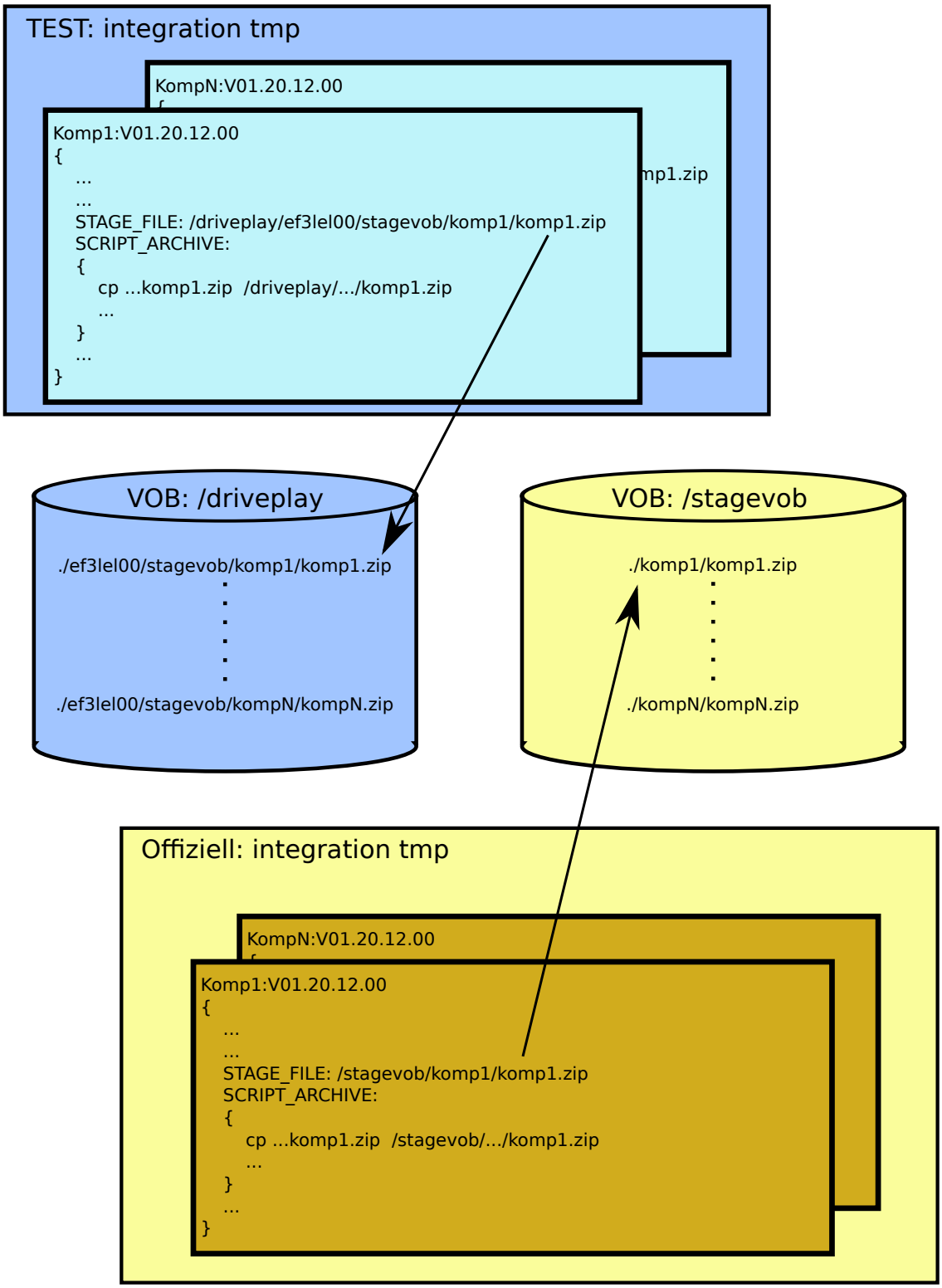


Abbildung 5.3: Gegenüberstellung der Testumgebung und der offiziellen Umgebung

Nach der Vollendung der Anpassung der oben genannten Schritte war es notwendig die einzelnen Generierschritte des GA in Build Forge abzubilden. Dazu fand eine Analyse der Schritte, die innerhalb von docmgen bei der Erzeugung ausgeführt werden, statt und nach dieser Analyse wurden die einzelnen Schritte in Build Forge mit Hilfe der BF-Schritte und BF-Projekte umgesetzt. Folgende Schritte müssen innerhalb von Build Forge für jede Komponente ausgeführt werden.

1. Einziehen der zu verwendenden Komponenten (CMGET_COMPLIST)
2. Ausführen der Generierung (SCRIPT_GENERATE, SCRIPT_GENERATE_PC)
3. Kopieren und Packen der Generierergebnisse in ein temporäres Verzeichnis (SCRIPT_PACKAGE, SCRIPT_COPY)
4. Kopieren der gepackten Generierergebnisse in den Stage-VOB (SCRIPT_ARCHIVE)

Diese herausgearbeiteten Schritte werden in Abb. 5.7 auf Seite 58 auf der rechten Seite im Block Generierautomat nochmals dargestellt.

Die Schritte für die Erzeugung von Komponenten durch den aktuellen GA können in Build Forge auf unterschiedliche Arten abgebildet werden. Aus der möglichen Menge aller Abbildungsvarianten werden nachfolgend die drei offensichtlichsten kurz vorgestellt.

Die erste Variante zeigt Abb. 5.4. Dort werden die einzelnen GA-Schritte zur Erzeugung einer Komponente durch einen Build Forge Schritt innerhalb eines Projekts abgebildet.

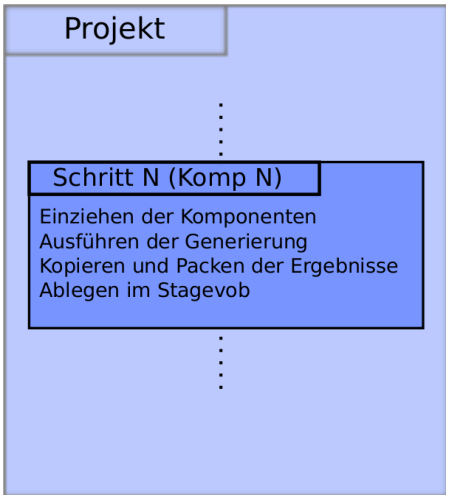


Abbildung 5.4: Ein Projekt

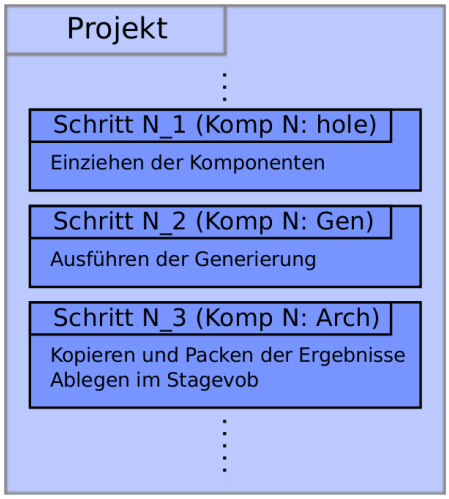


Abbildung 5.5: Ein Projekt, n Schritte

Die zweite Variante zeigt Abb. 5.5. In dieser wird analog zu Abb. 5.4 nur mit einem Projekt gearbeitet, jedoch wird für jeden Erzeugungsschritt einer Komponente ein separater BF-Schritt angelegt. Sowohl in Abb. 5.4 als auch in Abb. 5.5 wird die Levelabhängigkeit

durch eine festgelegte Reihenfolge bei der Schrittabarbeitung abgebildet. Das bedeutet, dass Komponenten niedrigerer Level als BF-Schritte vor Komponenten höherer Level ausgeführt werden müssen. Dies ist möglich, da Build Forge eine Änderung der Reihenfolge der Schrittabarbeitung erlaubt.

Als dritte Variante, dargestellt in Bild 5.6, können die Level als Projekte abgebildet werden und die Schritte des GA werden wie in Bild 5.5 auf der vorherigen Seite aufgeteilt und als einzelne BF-Schritte verwendet. Berücksichtigt man bei der Umsetzung des

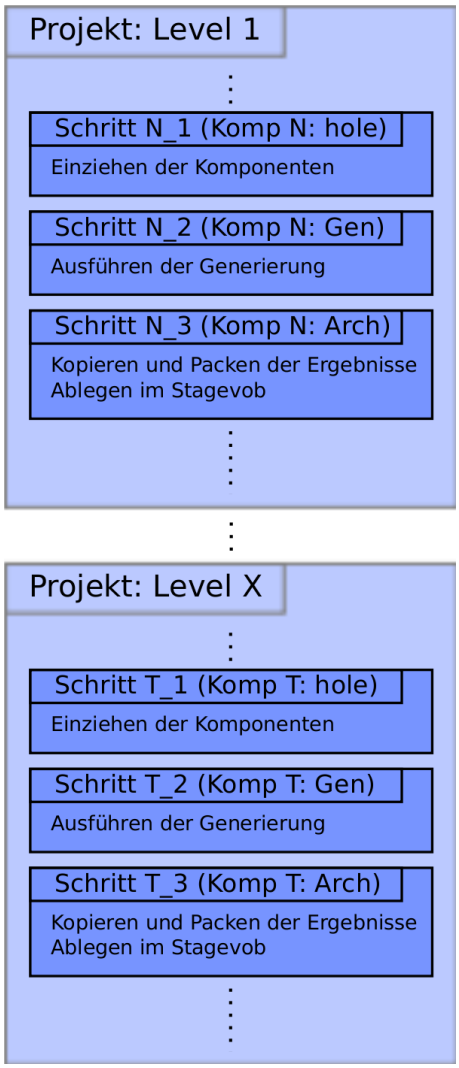


Abbildung 5.6: n Projekt, m Schritte

GA nach Build Forge, dass dieser größtenteils unter Solaris arbeitet, so ist die Variante aus Abb. 5.4 auf der vorherigen Seite nicht die optimale Lösung, da eine plattformspezifische Anpassung der in den BF-Schritten verwendeten Skripts notwendig wird. Daraus kann gefolgert werden, dass jeder einzelne Erzeugungsschritt als einzelner BF-Schritt, wie in Abb. 5.5 auf der vorherigen Seite, abgebildet werden sollte. Hier besteht jedoch die Gefahr die Übersicht über die Levelsortierung der Schritte zu verlieren, da ein Projekt aus sehr vielen Schritten bestehen kann. Dies führt möglicherweise dazu,

dass ein neuer Schritt an einer falschen Stelle eingefügt wird und dies erst in einer offiziellen Generierung auffällt. Aus diesem Grund stellt Abb. 5.5 auf Seite 56 auch nicht die optimale Lösung dar.
Die dritte Variante, dargestellt durch Abb. 5.6, wurde für das Testprojekt umgesetzt. Durch die Abbildung der Generierlevel als Projekte und der Erzeugungsschritte als BF-Schritte ist die parallele Ausführung mehrerer BF-Schritte eines Generierlevels möglich. Mit Hilfe der Verknüpfung können die Projekte sequentiell als Generierlevel abgearbeitet werden. Eine detaillierte Darstellung der Umsetzung des Generierautomaten auf Build

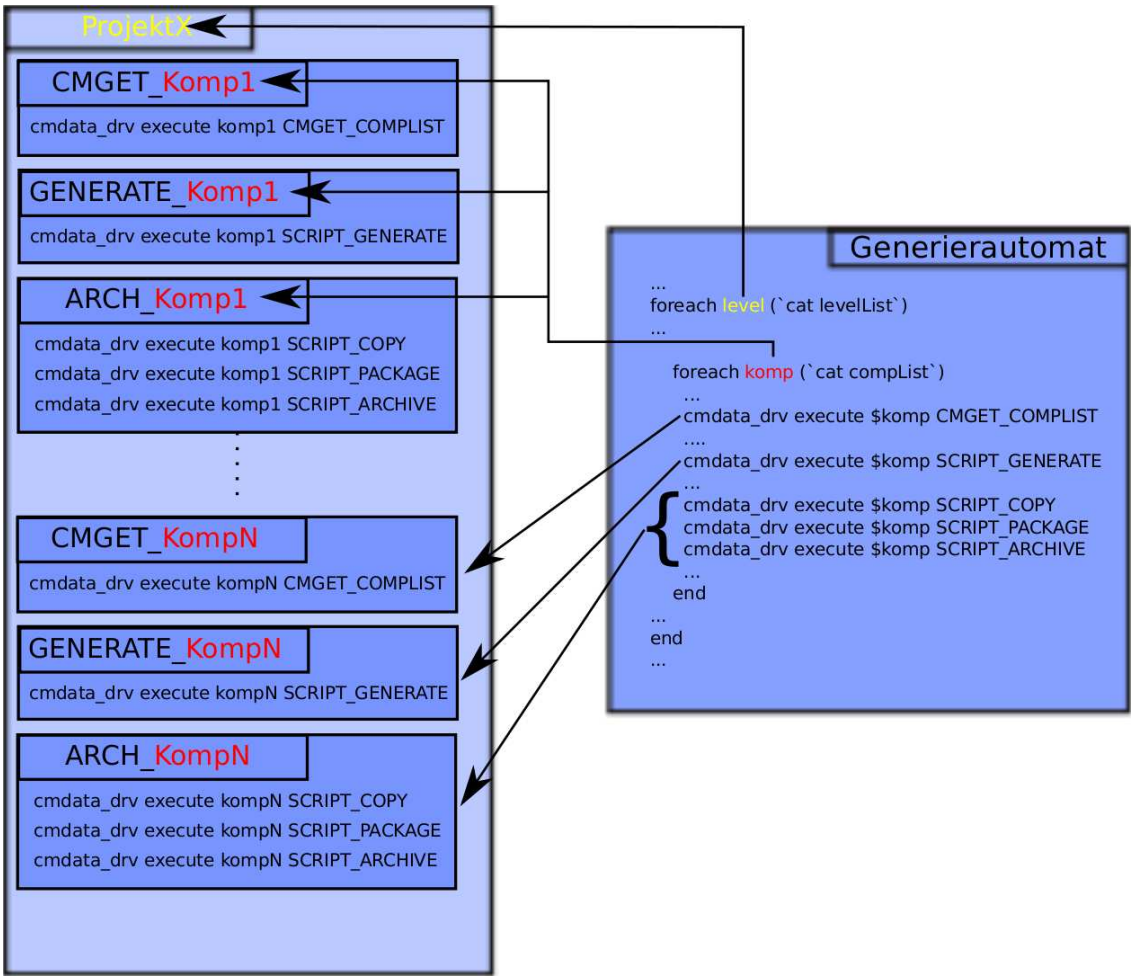


Abbildung 5.7: Umsetzung des Generierautomaten nach Build Forge

Forge zeigt Abb. 5.7. Deutlich erkennt man hier, die Umsetzung der einzelnen Generierlevel als Projekte und die Umsetzung der Erzeugungsschritte von Komponenten aus dem Generierautomaten in BF-Schritte. Dass für das Kopieren, Packen und Archivieren der Generierergebnisse nur ein Schritt verwendet wurde, liegt an der logischen Zusammengehörigkeit dieser Schritte, die jedoch problemlos noch weiter aufgeteilt werden könnten.

5.2 Beachtenswertes bei Build Forge

Da bei den ersten Versuchen mit Build Forge im Rahmen dieser Diplomarbeit Probleme auftraten, wird in diesem Abschnitt kurz auf diese eingegangen und mögliche Lösungsvorschläge für diese vorgestellt.

5.2.1 Installation von Build Forge

Zu der reinen Installation von Build Forge lässt sich sagen, dass auf jeden Fall immer die neueste Version installiert werden sollte. Bei den Installationsversuchen einer älteren Version im Rahmen dieser Diplomarbeit, hätten neben Build Forge auch noch z. B. der Webserver und andere benötigte Perlmodule selbsttätig installiert werden müssen. Dies führte zu mehrmaligen Kontakt mit dem IBM-Support, bis der Hinweis einging, dass doch die neueste Version installiert werden solle. So bieten die neueren Versionen eine graphische Installationshilfe und nahezu alle für den erfolgreichen Einsatz von BF benötigten Softwarekomponenten. Als einzig selbst zu installierendes Programm neben Build Forge blieb die MySQL-Datenbank übrig, da die bei Build Forge mitgelieferte Datenbank nur unter Windows betrieben werden kann. Nach der Installation des Build Forge Servers wurden die Agenten auf den Generierservern installiert.

5.2.2 Installation, Test und Verwendung der Agenten

Bei der Installation der Agenten unter Windows wird dem Anwender die Möglichkeit gegeben den Agenten entweder als Dienst im Hintergrund oder als Vordergrundprozess zu installieren. Die Methode des Hintergrunddienstes ist die von IBM empfohlene Vorgehensweise. Mit der Installation des Agenten als Vordergrundprozess kann der Entwickler während der Ausführung von Schritten durch Build Forge z. B. auch graphische Oberflächen nutzen und mit diesen interagieren. Dies ist bei der Verwendung als Dienst nicht möglich.
Im Gegensatz zu Windows lässt sich unter Solaris bzw. Linux der Agent nur als Dienst installieren. Wie bereits in Kapitel 5 angesprochen wurde, besteht auch die Möglichkeit den Agenten als Quellcode zu erhalten und selbstständig für die benötigte Plattform zu übersetzen. Um zu überprüfen, mit wieviel Aufwand das Generieren eines Agenten verbunden ist, wurde im Rahmen dieser Diplomarbeit ein Agent für eine 64-Bit-Linux-Distribution aus den Quelldateien erzeugt. Die Anleitung dazu findet sich in der README, die mit den Quellen ausgeliefert wird. Innerhalb weniger Minuten konnte der selbstgenerierte Agent installiert und über telnet angesprochen werden. Mit welchem Verfahren ein installierter Agent auf prinzipielle Funktionsfähigkeit geprüft werden kann, folgt noch in diesem Abschnitt.
Da im SINAMICS-Projekt jeder Befehl eines Generierschrittes in einer Kommandozeile absetzbar sein muss, wurde der Agent auf dem Windowstestrechner als Dienst instal-

liert. Wie bei den ersten Versuchen mit Build Forge jedoch festgestellt wurde, ist es anfangs durchaus hilfreich, den Agenten als normales Programm zu installieren, da auf diese Art und Weise einmal festgestellt werden kann, ob Build Forge prinzipiell läuft. Auch ermöglicht dieses Vorgehen eine einfachere Fehlerdiagnose, da im Programmfenster des Agenten Ausgaben, wie z. B. der Start eines neuen Prozesses, abgelesen werden können. Dieses Ablesen der Meldungen kann bei der Analyse fehlgeschlagener Verbindungsversuche helfen, da die Managementkonsole zwar Fehler meldet, aber über diese Fehlermeldungen keine weitergehenden Informationen anbietet. Für die Tests der Agenten muss auch noch erwähnt werden, dass der Agent über telnet standardmäßig auf Port 5555 angesprochen werden kann. Dadurch kann sichergestellt werden, dass der Generierserver vom Build Forge Server aus über den Agenten angesprochen werden. Kann sich der Anwender über telnet mit dem Agenten verbinden, gibt der Build Forge Agent die in Abb. 5.8 gezeigte Antwort auf der Konsole aus. Dadurch kann erst einmal nachgewiesen werden, dass der Agent korrekt installiert wurde und auch über das Netzwerk erreichbar ist. Ist es möglich, den Build Forge Agenten mit Hilfe von telnet

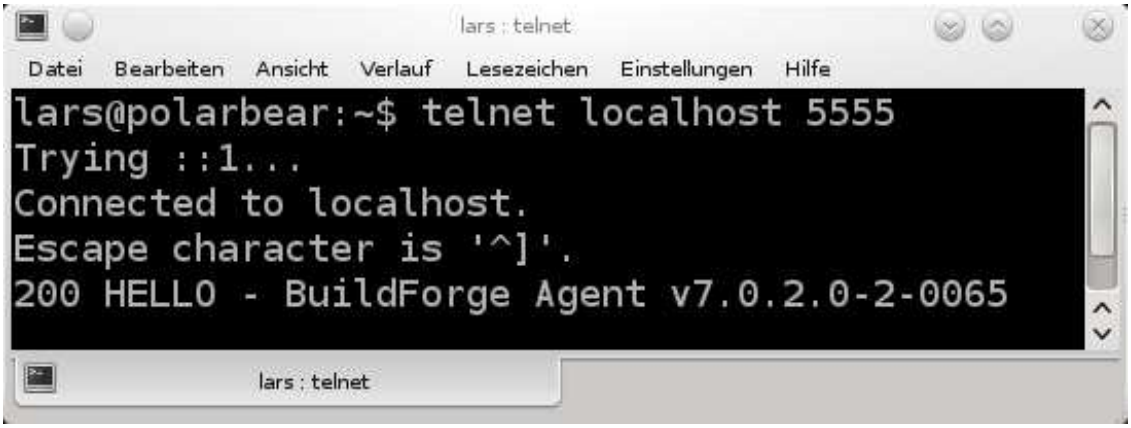


Abbildung 5.8: Agent nach Kontakt mit telnet

zu kontaktieren, aber in der Build Forge Managementkonsole kann immer noch keine Verbindung aufgebaut werden, empfiehlt es sich zum einen nochmals die Einstellungen des eingerichteten Servers, hier insbesondere die korrekte Eingabe des Passworts, zu überprüfen. Eine weitere Möglichkeit wäre, die Neuinstallation einer neueren Agentenversion. Dieses Vorgehen hat bei den durchzuführenden Untersuchungen zum Erfolg geführt. Führt keine der hier vorgestellten Methoden zum Erfolg, so muss man sich an IBM wenden und um Unterstützung bitten.

5.2.3 ClearCase und Build Forge

Anfangs traten bei der Arbeit mit ClearCase und Build Forge kleine Probleme auf. Hier ist besonders zu erwähnen, dass Build Forge nicht auf das Arbeiten in einer gesetzten View, wie in Abb. 4.4 auf Seite 49 gezeigt wird, optimiert ist. Stattdessen findet die Zugriffe auf Views innerhalb von Build Forge hauptsächlich über den kompletten Pfad

einer View, also unter Unix mit „`/view/viewname/vob/`“ statt. Jede in ClearCase gestartete View wird unter Unix unter dem Verzeichnis „`/view`“ und unter Windows unter dem Laufwerk „`M:\`“ eingebunden. Somit kann auf die Inhalte der View auch über „`/view/viewname/`“ bzw. unter Windows mit „`M:\viewname`“zugegriffen werden, ohne die View vorher mit *cleartool setview* gesetzt zu haben.
Um zu zeigen, dass auch mit der eben vorgestellten Variante ein Zugriff auf die View möglich ist, wird diese Methode in Abb. 5.9 angewandt. Wie man sieht, liefert Abb. 5.9 das gleiche Ergebnis wie zuvor schon Abb. 4.3 auf Seite 49 und Abb. 4.4 auf Seite 49. In Abb. 5.7 auf Seite 58 wurde bereits dargestellt, wie der GA in Build Forge abgebildet

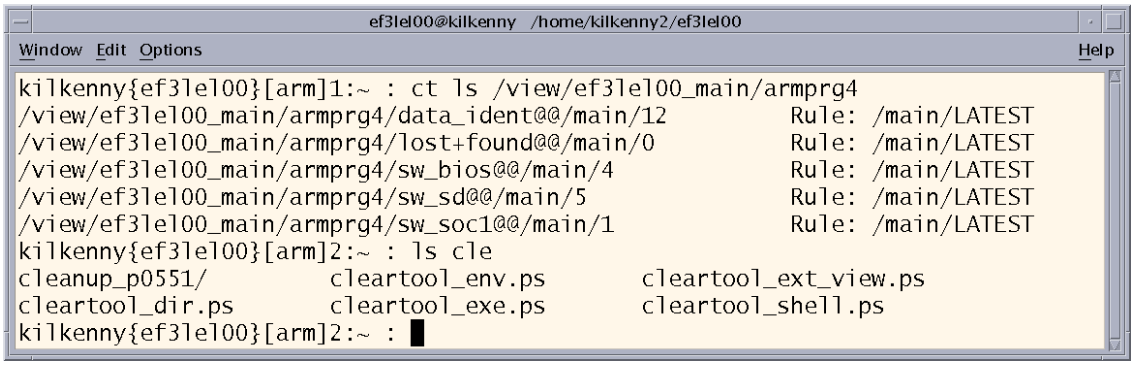


Abbildung 5.9: ClearCase cleartool

wurde. In diesem Bild erkennt man auch, dass die Befehle aus dem GA direkt übernommen wurden. Dies war notwendig, da die aktuelle Generierumgebung von SINAMICS nicht auf die Arbeit mit dem kompletten Pfad der View ausgelegt ist. Daher sollte man bei der Einrichtung eines neuen Projekts mit Build Forge und ClearCase unbedingt darauf achten, dass Generierungen auch ohne das explizite Setzen einer View, sondern nur mit der Arbeit über den Pfad der View ausgeführt werden können. Ansonsten muss in jedem Schritt, der eine gesetzte View benötigt, mit *cleartool setview -exec “Befehl“ viewname* gearbeitet werden. Dies hat wiederum negativen Einfluss auf die Geschwindigkeit. Zwar gibt es in Build Forge eine undokumentierte Umgebungsvariable, mit der automatisch das *cleartool setview* für jeden Schritt erfolgt, jedoch ist es während der Diplomarbeit nicht gelungen, diese nicht unterstützte Eigenschaft auszunutzen. Da es sich, wie bereits erwähnt, auch um ein nicht dokumentiertes und unterstütztes Vorgehen handelt, sollte diese Methode auch nicht angewandt werden, da keine Garantie besteht, dass dies in neueren Build Forge Versionen weiterhin unterstützt wird.

Da nun die Testumgebung und wichtige Punkte bei der Arbeit mit Build Forge vorgestellt wurden, wird im nächsten Kapitel das Hauptaugenmerk auf den Tests der Tauglichkeit von Build Forge in großen Softwareprojekten liegen.

6 Eignung von Build Forge

Im vorigen Kapitel 5 wurde die Testumgebung vorgestellt, die notwendig ist, um die Tauglichkeit von Build Forge innerhalb eines großen Softwareprojekts zu testen. In diesem Kapitel werden nun die einzelnen Tests beschrieben und deren Wichtigkeit für große Softwareprojekte im allgemeinen bewertet. In Abschnitt 4.3 wurde als ein wichtiges Kriterium für den Einsatz von Build Forge der Einsatz auf unterschiedlichen Betriebssystemen genannt.

Im nächsten Abschnitt wird die Tauglichkeit von Build Forge bzgl. dieses Kriteriums untersucht.

6.1 Tests auf unterschiedlichen Plattformen

Da die Untersuchungen von Build Forge während der Arbeit an dieser Diplomarbeit im SINAMICS-Projekt von Siemens DT erfolgen, können nicht alle von IBM angegebenen Plattformen untersucht werden, da in diesem Projekt Windows XP und Solaris 10 als Vorbedingung für die Tests vorgegeben sind. Bei der Untersuchung der Lauffähigkeit von Build Forge auf unterschiedlichen Plattformen muss auch noch zwischen dem Einsatz des Build Forge Servers und der Build Forge Agenten unterschieden werden. Wie bereits in Abschnitt 5.1.1 geschildert, wurde der Build Forge Server aus Projektgründen ebenfalls auf einer Solaris-Maschine installiert. Da Build Forge jedoch eine Installationsunterstützung für Windows mitbringt, sollte dieses auch problemlos auf einer Windowsmaschine installierbar sein. Daher sollten die Ergebnisse auch auf den Einsatz des BF-Servers auf alle laut IBM unterstützten Plattformen, wie z. B. Windows übertragbar sein. Wichtiger für die Lauffähigkeit von Build Forge auf unterschiedlichen Plattformen ist jedoch die Unterstützung der Plattformen durch die Agenten. Diese werden für Solaris und Windows auch bereits als kompilierte Programme geliefert und wurden für die Tests, wie in Abschnitt 5.2.2 beschrieben, auf je einer Maschine installiert. Neben der Installation wurde aus den 61 Komponenten der Testumgebung, deren Erstellung in Kapitel 5 dargestellt wird, jeweils eine Komponente der zu untersuchenden Plattform gewählt.

So zeigt Abb. 6.1 auf der nächsten Seite die Auswahl der PC-Komponente aus dem BF-Projekt „Level 1“ in der Managementkonsole. In diesem Bild wird auch nochmal der Aufbau eines Projekts aus mehreren Generierschritten deutlich. Die Auflistung der einzelnen Schritte des BF-Projekts wird zusätzlich durch die Darstellung der Schrittbefehle und den zu verwendenden Selektor ergänzt. Insbesondere lässt sich hier nochmals die Aufschlüsselung in die Teile „Hole bestehende Komponente“ (CMGET_*), „Generiere die Komponente“ (GENERATE_*) und „Archiviere die Komponente“ (ARCH_*) erken-

nen.

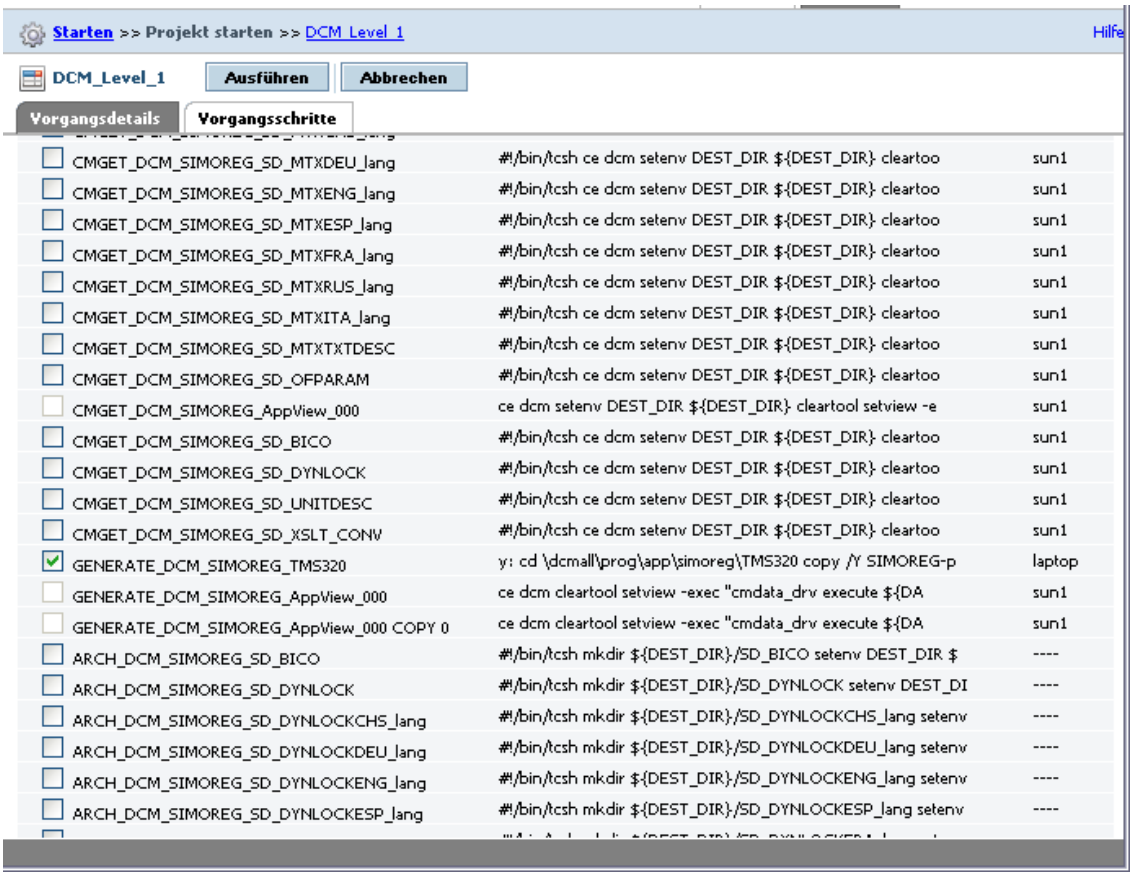


Abbildung 6.1: Auswahl der PC-Komponente

Durch Anwählen des Knopfes „Ausführen“ wird die Generierung der PC-Komponente gestartet, woraufhin über den Selektor ein geeigneter Agent ausgewählt wird und die Abarbeitung des Schrittbefehls durch diesen erfolgt.
Nach Abarbeitung des Schrittes auf dem, durch den Selektor gewählten, Server wird in der Managementkonsole für den Schritt entweder eine Erfolgsmeldung, eine Warnung oder eine Fehlermeldung ausgegeben. Eine Kontrolle der Generierung erfolgt in dem von Build Forge erzeugten Schrittprotokoll, welches, für die zu Testzwecken generierte PC-Komponente, auszugsweise in Abb. 6.2 auf der nächsten Seite dargestellt wird. Für jeden Schritt wird von Build Forge ein analoges Schrittprotokoll erstellt. Durch die Auswahlboxen, wie z. B. „STEP“, am oberen Rand können in den Schrittprotokollen Informationen ein- und ausgeblendet werden. Somit können, z. B. für die Analyse von Fehlern, unnötige Informationen ausgeblendet werden.

Durch die fehlerfreie Erzeugung der ausgewählten PC-Komponente bei der Testgenerierung, siehe Abb. 6.2 auf der nächsten Seite oben rechts, und durch die Kontrolle der erzeugten Dateien in der Testview auf Vollständigkeit wurde nachgewiesen, dass Build Forge unter Windows lauffähig ist. Da mit der Erzeugung der PC-Komponente gezeigt wurde, dass der BF-Agent auf dem PC problemlos funktioniert, muss nun noch gezeigt werden, dass der Agent auch auf anderen Plattformen erfolgreich einsetzbar ist.

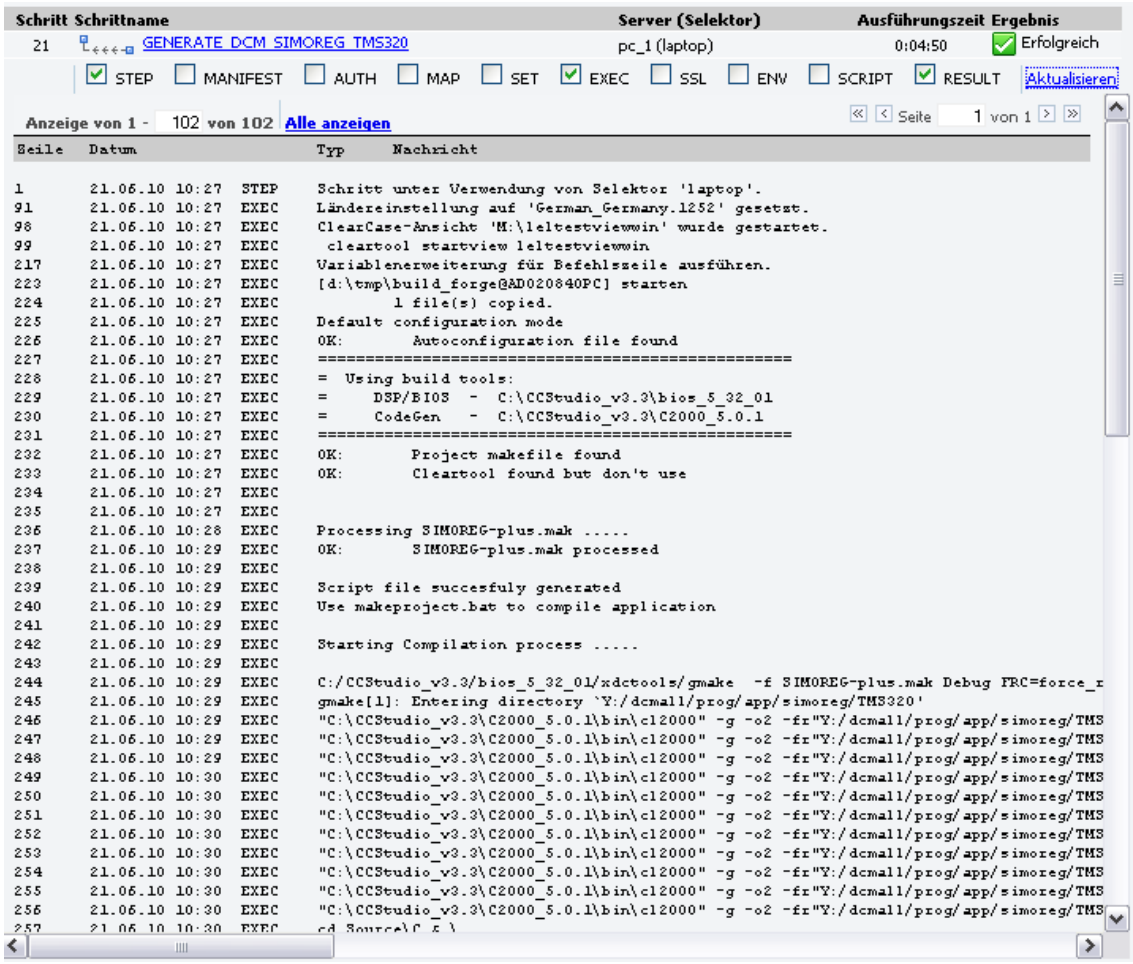


Abbildung 6.2: Schrittprotokoll PC

Für diesen Zweck erfolgt keine Generierung einer SUN-Komponente, sondern das Archivieren der soeben für den PC-Test generierten Komponente. Dies hat den Grund, dass zum einen, wie bereits in Kapitel 3.3.1.2 auf Seite 26 vorgestellt wurde, alle weiteren Schritte an der SUN erfolgen. Als weiterer Grund für dieses Vorgehen ist noch anzuführen, dass somit eine weitere Kontrolle der Vollständigkeit der generierten PC-Komponente gewährleistet werden kann, da bei einer fehlerhaften Generierung auch das Archivieren einen Fehler meldet. Nach der Ausführung des SUN-Teiles der generierten PC-Komponente erhält man das in Abb. 6.3 auf der nächsten Seite gezeigte Schrittübersichtsbild. Auch hier lässt sich ablesen, dass der ausgeführte Schritt an der SUN erfolgreich war. Wird der ausgeführte Schritt angewählt, so erhält man das Schrittprotokoll. Dieses ist analog zu Abb. 6.2. Der in Abb. 6.3 auf der nächsten Seite ebenfalls mit dargestellte weitere Schritt, ist als „übersprungen“ gekennzeichnet, wodurch in der Managementkonsole Schritte der BF-Projekte dargestellt werden, die nicht ausgeführt wurden.

Da, wie in diesem Abschnitt aufgezeigt wurde, sowohl die Befehlsausführung durch die Agenten auf unterschiedlichen Plattformen funktioniert, kann geschlossen werden, dass die BF-Agenten auf unterschiedlichen Plattformen lauffähig sind. Neben der reinen

Abbildung 6.3: Protokoll SUN

Lauffähigkeit zeigt dieser Abschnitt auch, dass die notwendigen Generierergebnisse erzeugt werden, da ansonsten der Testlauf für die Archivierung auf der SUN nicht zum Erfolg geführt hätte.
Da nun die Einsatzfähigkeit auf unterschiedlichen Plattformen gezeigt wurde, erfolgt anschließend die Überprüfung der Synchronisation unter den einzelnen Plattformen.

6.2 Test der Synchronisation

Wie bereits im vorigen Abschnitt 6.1 gezeigt wurde, ist Build Forge auf mehreren Plattformen einsetzbar. Ein weiterer wichtiger Punkt für den Einsatz von Build Forge in großen Softwareprojekten ist die Fähigkeit unterschiedliche Plattformen miteinander zu synchronisieren. Da in Build Forge Projekte aus einzelnen Schritten bestehen, erfolgt die Synchronisation der einzelnen Schritte nicht nur über Plattformgrenzen hinweg, sondern auch innerhalb einer Plattform muss die Ausführung der einzelnen Schritte synchronisiert werden. Diese Synchronisation spielt insbesondere auch beim Einsatz des in Abschnitt 3.3.2 beschriebenen Threadings eine große Rolle, da hierbei evtl. eine mögliche Reihenfolge bei der Abarbeitung der Schritte erfolgen muss.
Daher teilt sich der Test der Synchronisation einmal in den Test für unterschiedliche Plattformen und in den Test bei der Bearbeitung von BF-Schritten innerhalb eines Projekts mit dem Einsatz von Threading auf. Hierfür muss noch erwähnt werden, dass Schritte auf drei unterschiedliche Arten abgearbeitet werden können. Diese Möglichkeiten werden in der folgenden Tabelle 6.1 auf der nächsten Seite dargestellt und kurz erläutert, da die Symbole in den Abbildungen des Abschnitts verwendet werden.

6.2.1 Synchronisation über Plattformen

Der erste Test behandelt die Synchronisation über die Plattformgrenzen hinweg. Hierbei werden wieder die aus Abschnitt 6.1 bekannten Schritte verwendet. Es erfolgt also wieder eine Generierung unter Windows und das Archivieren der Komponenten auf ei-

Ausführung	Symbol	Bedeutung
normal	–	Die Schritte werden nacheinander abgearbeitet
parallel (Threading)	– – – – – –	Diese Schritte können parallel zu anderen Schritten abgearbeitet werden
verbunden	< – < – < – < –	Diese Schritte werden auch parallel abgearbeitet, allerdings wird an diesen auf die vorher gestarteten parallelen Schritte gewartet

Tabelle 6.1: Schrittabarbeitung

ner SUN. Abb. 6.4 zeigt die beiden ausgewählten BF-Schritte. Betrachtet man sich die Spalte „Ergebnis“ in der Abbildung so kann man deutlich erkennen, dass zu diesem Zeitpunkt nur der BF-Schritt GENERATE_DCM_SIMOREG_TMS320 mit der Schrittnummer 21 in der aktiven Ausführung war und der BF-Schritt ARCH_DCM_SIMOREG_TMS320 mit Schrittnummer 42 noch nicht ausgeführt wird. Dies wird auch durch das Pausensymbol vor der Zeile verdeutlicht. Der Schritt 42 wird mit der Ausführung so lange hinausgezögert bis Schritt 21 ausgeführt wurde. Die Reihenfolge der Ausführung wird durch den Einsatz als verbundene BF-Schritte, siehe auch Tabelle 6.1, eingehalten. In der Ab-

Abbildung 6.4: Synchronisation 1

bildung 6.5 wird die weitere zeitliche Abarbeitung dargestellt. Hier erkennt man, dass BF-Schritt 21 nicht mehr aktiv ist, sondern erfolgreich abgeschlossen wurde, woraufhin Schritt 42 in den aktiven Zustand versetzt wurde. Wie in Abb. 6.6 auf der nächsten Seite

Abbildung 6.5: Synchronisation 2

gezeigt wird, wurde auch BF-Schritt 42 erfolgreich beendet. Das bedeutet, dass die Synchronisation der einzelnen BF-Schritte über die Plattformengrenzen hinweg funktioniert,

da BF-Schritt 42 die Generierergebnisse von Schritt 21 für die korrekte Ausführung benötigt. Da nun gezeigt wurde, dass die Plattformübergreifende Synchronisation von

Abbildung 6.6: Synchronisation 3

Schritten funktioniert, wird nun noch die Synchronisation mit mehreren Schritten innerhalb eines Build Forge Projekts untersucht.

6.2.2 Synchronisation innerhalb eines Projekts

Dieser Abschnitt behandelt die Synchronisation mehrerer Schritte in einem BF-Projekt. Wie bereits im vorangegangen Abschnitt 6.2.1 gezeigt wurde, ist die Synchronisation über die Plattformgrenzen hinweg möglich. Dieses Wissen wird auch in diesem Abschnitt weiterverwendet. Für den Test der Synchronisation innerhalb eines Projekts wird der erste Generierlevel, aus der im Rahmen der Diplomarbeit geschaffenen Testumgebung, verwendet. Allerdings wurde für diesen Test die Verkettung zwischen diesem und dem nachfolgenden Projekt ausgeschaltet, da das Ausführen der BF-Folgeprojekte für diesen Test nicht relevant ist. Nach dieser Anpassung wurde die Ausführung des BF-Projekts gestartet. Wie in Abb. 6.7 auf der nächsten Seite zu erkennen ist, werden auch hier anfangs die BF-Schritte mit CMGET_* abgearbeitet, da es sich hierbei auch um die erste Aufgabe des zu ersetzenden Generierautomaten handelt. Sind alle CMGET-Schritte abgearbeitet worden, werden die GENERATE-Schritte der Projekte abgearbeitet. Dies ist in Anhang G deutlich zu erkennen. Wurden die GENERATE-Schritte eines BF-Projekts abgearbeitet, folgen noch die ARCHIVE-Schritte des Projekts. Wie man in Anhang F erkennt, werden diese auch parallel abgearbeitet, nachdem BF-Schritt 21 aus den Abbildungen erfolgreich bearbeitet wurde.

6.2.3 Bedeutung der Synchronisation

Wie in den beiden vorigen Abschnitten aufgezeigt wurde, ist Build Forge in der Lage zum einen die Schritte innerhalb eines Projekts auf einer Entwicklungsplattform als auch plattformübergreifend zu synchronisieren. Durch diese Synchronisation ist ein Softwareprojekt bei der Entwicklung der Software nicht auf eine Entwicklungsplattform beschränkt, sondern es kann eine dynamische Veränderung der vorhandenen Plattformen

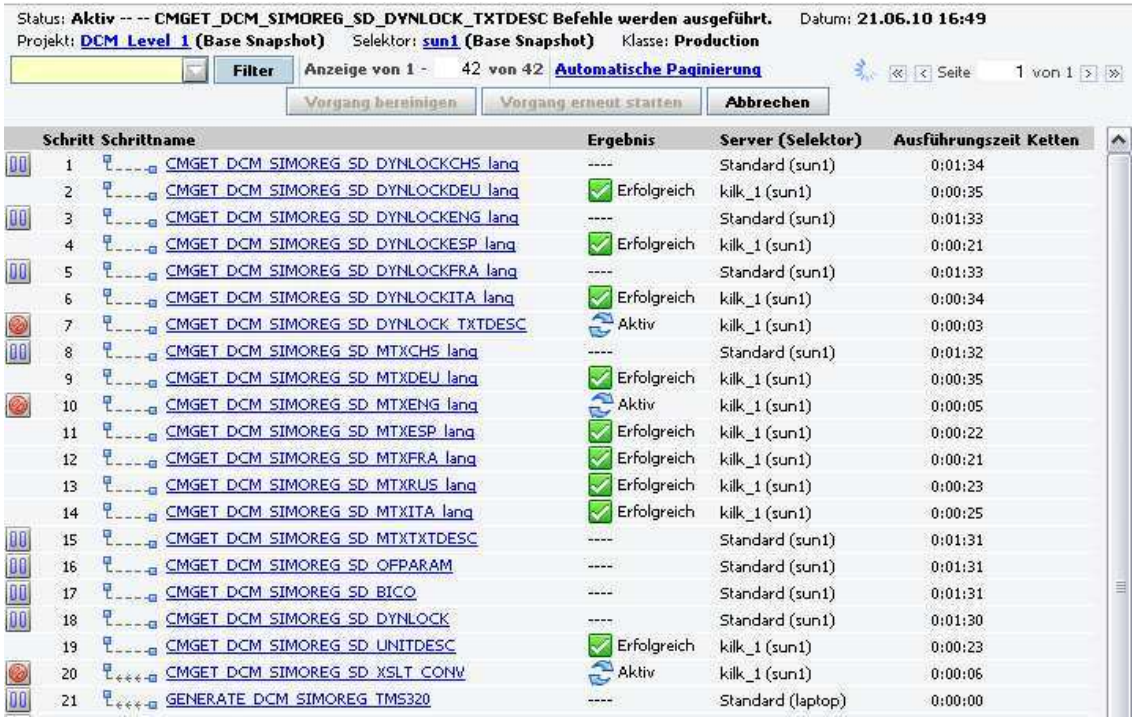


Abbildung 6.7: Synchronisation CMGET_*

erfolgen, ohne einen großen Zusatzaufwand bei KM durch Anpassung der Generierskripte zu erzeugen. So muss von KM in Build Forge ein neuer Server, ein neuer Kollektor, ein neuer Selektor und ein neuer Schritt angelegt werden. Dies ermöglicht eine sehr schnelle Generierung auf einer neuen Entwicklungsplattform.

Somit ermöglicht die Synchronisation eine problemlose Entwicklung auf bereits vorhandenen Plattformen und das relativ problemlose Einbinden neuer Entwicklungsplattformen in Build Forge. Da für die Tests der Synchronisation die Verkettung ausgeschaltet wurde, beendet sich Generierung nach diesem Projekt. Wäre die Verkettung eingeschaltet gewesen, so würde Build Forge automatisch mit der Generierung des nächsten Projekts fortfahren.

Das automatische Fortsetzen der Generierung durch die Verkettung führt auch zum nächsten zu untersuchenden Punkt. Welchen Einfluss hat Build Forge auf die Laufzeit der Erzeugung von Software.

6.3 Laufzeittest

In diesem Abschnitt wird der Einfluss von Build Forge auf die Generierzeit bei bestehenden Projekten behandelt. Für diesen Vergleich werden drei unterschiedliche Generierungen herangezogen. Das ist zum einen eine Betrachtung der Laufzeit des Testprojekts mit dem aktuell verwendeten Generierautomaten. Neben dieser Betrachtung erfolgt auch eine Laufzeitbetrachtung von Build Forge mit aktiviertem „Threading“ und

eine Betrachtung mit einer rein seriellen Schrittausführung durch Build Forge.

6.3.1 Laufzeit im aktuellen Generierautomaten

Um überhaupt beurteilen zu können, inwieweit Build Forge einen Einfluss auf die Laufzeit der Generierung hat, ist es notwendig Kenntnis über die aktuellen Dauer der Generierung mit dem aktuell verwendeten Generierautomaten zu besitzen.

Da für die Tests eine offizielle Version eines kleineren Projekts konvertiert wurde und innerhalb des Generierautomaten eine Zeitmessung erfolgt, werden diese gemessenen Werte als Grundlage für die Betrachtung der Laufzeit herangezogen. So zeigt die Tabelle aus Anhang C die aktuellen Generierzeiten der einzelnen Level und in der letzten Zeile die aufsummierte Dauer der Generierung des Projekts SIMOREG. Da durch die Last im Netz und die Belastung der einzelnen Maschinen auch immer ein Einfluss auf die Generierzeit ausgeübt wird, ist diese auch bei mehreren Generierungen der gleichen Version einer gewissen Schwankung unterworfen. Die Erfahrung zeigt, dass diese Schwankungen bei kleineren Projekten in der Regel im Bereich einer halben bis dreiviertel Stunde liegen. Wie aus der Tabelle im Anhang C entnommen werden kann, hat die dargestellte Generierung 4 Stunden und 13 Minuten gedauert. Somit erhält man einen Zeitraum einer Generierung, der sich zwischen 3,5 und 5 Stunden bewegt. Das bedeutet, dass Build Forge für eine Generierung einer SIMOREG-Version nicht länger als 5 benötigen sollte.

Allerdings muss erwähnt werden, dass Build Forge in der Testumgebung durch die in Abschnitt 5.2.1 beschriebene Installation ausgebremst wird, da auf einer SUN der Build Forge Server, die MySQL-Datenbank und ein Agent installiert wurde. Dieses Vorgehen wird von IBM nur für Testzwecke empfohlen, während im regulären Betrieb diese Installationen auf unterschiedlichen Servern laufen sollten. Da nun als Grundlage für weitere Tests die Generierzeit in der produktiven Umgebung feststeht, wird als nächstes der Einfluss von Threading auf die Generierzeit untersucht.

6.3.2 Laufzeit mit Threading

Da Build Forge die parallele Generierung von Schritten unterstützt, wurde auch die Laufzeit bei der parallelen Generierung mehrerer Schritte untersucht. Aufgrund der Beschränkung des Testsystems mit nur einem SUN-Agenten kann die Parallelgenerierung jedoch nicht die Wirkung entfalten, die bei einem Einsatz mehrerer Generierserver erreicht werden kann. Um den Generierserver nicht zu sehr zu belasten, wurde die Anzahl der möglichen parallelen BF-Schritte auf drei beschränkt. Die Generierzeit des getesteten Projekts liegt zwischen 4 und 5 Stunden. Die ausführliche Darstellung der Generierzeit findet sich in Anhang C. Bei der Erzeugung der Software mit Hilfe von Build Forge und aktiviertem „Threading“ beträgt die Generierzeit in etwa 4,5 Stunden.

Anhang D zeigt eine Abbildung, die mit Hilfe der Analysefunktionen von Build Forge erstellt wurde. Dort wird neben der Laufzeit der 9 BF-Projekte in Sekunden auch der jeweilige Startzeitpunkt angezeigt. Für eine bessere Darstellung der Generierzeiten, wurden diese aus der Grafik aus Anhang D entnommen und in Tab. 6.2 übertragen. Für eine bessere Vergleichbarkeit mit den anderen Laufzeitmessungen wurden diese in Sekunden angegeben Werte in Stunden umgerechnet. Wie bereits in Abschnitt 6.3 erwähnt wurde,

Projekt	**Generierzeit in Sekunden**	**Generierzeit in hh:mm:ss**
DCM_LEVEL_1	641	00:10:41
DCM_LEVEL_2	451	00:07:31
DCM_LEVEL_3	3883	01:04:43
DCM_LEVEL_4	686	00:11:26
DCM_LEVEL_5	242	00:04:02
DCM_LEVEL_6	1152	00:19:12
DCM_LEVEL_7	8392	02:19:52
DCM_LEVEL_8	426	00:07:06
DCM_LEVEL_9	71	00:01:11
SUMME:	15944	04:25:44

Tabelle 6.2: Generierzeiten parallel

unterliegt die Generierzeit auch Schwankungen, weshalb die längere Generierzeit von ca. 10 Minuten bei der parallelen Generierung nicht ins Gewicht fällt. Auch muss noch einmal darauf hingewiesen werden, dass bei der Installation von Build Forge in einem realen Softwareprojekt mehrere Server mit Agenten zur Verfügung stehen. Somit kann zum einen die Anzahl der parallelen Schritte erhöht werden und zum anderen wird die Auslastung auch auf mehreren Maschinen verteilt. Durch diese Verteilung auf mehrere Maschinen würde man in einem realen Projekt eine Beschleunigung der Generierung erhalten, da anstelle der quasi-parallelen Ausführung der Schritte auf einer Maschine eine wirkliche parallele Durchführung der Schritte erfolgt.
Neben der Betrachtung der parallelen Generierung ist für den Vergleich der Generierzeiten auch die serielle Erzeugung der Software, also analog zum bereits existierenden Vorgehen, von Interesse.

6.3.3 Laufzeit bei serieller Erzeugung

Da neben der parallelen Generierung mit der Threading-Funktionalität von Build Forge auch eine rein serielle Erzeugung der Software notwendig sein kann, wird in diesem Abschnitt ein Blick auf die Dauer der Generierzeit bei ausgeschaltetem Threading von Build Forge betrachtet. Um auch hier die Vergleichbarkeit einmal zu den offiziellen Generierungen als auch einen Vergleich zu der parallelen Generierung mit Build Forge zu gewährleisten, kommt bei diesen Tests ebenfalls die geschaffene Testumgebung zum Einsatz. Als einziger Unterschied zu der parallelen Generierung wurde die Anzahl der möglichen Threads auf Null gesetzt, während bei der parallelen Erzeugung mit Build

Forge, die Anzahl der parallelen Schritte durch Build Forge festgelegt wurde. Mit dieser Art der Generierung durch Build Forge erhält man die in Anhang. E dargestellten Generierzeiten. Auch diese Auswertung im Balkendiagramm wurde mit der Fähigkeit von Build Forge zum Erstellen von Statistiken erzeugt.

Um wie in Abschnitt 6.3.2 eine bessere Vergleichbarkeit zu erreichen, werden diese Werte in Stunden umgerechnet und in Tab. 6.3 dargestellt. Wie man erkennt, beträgt

Projekt	**Generierzeit in Sekunden**	**Generierzeit in hh:mm:ss**
DCM_LEVEL_1	678	00:11:18
DCM_LEVEL_2	464	00:07:44
DCM_LEVEL_3	4058	01:07:38
DCM_LEVEL_4	745	00:12:25
DCM_LEVEL_5	247	00:04:07
DCM_LEVEL_6	1140	00:19:00
DCM_LEVEL_7	6732	01:52:12
DCM_LEVEL_8	383	00:06:23
DCM_LEVEL_9	67	00:01:07
SUMME:	14514	04:01:54

Tabelle 6.3: Generierzeiten seriell

die Generierzeit bei der seriellen Generierung ungefähr 4 Stunden und 2 Minuten. Die serielle Generierung ist schneller, da der Prozessor nicht quasi-parallel mehrere Schritte abarbeiten muss, sondern der Prozessor von Build Forge jeweils nur für einen einzigen Schritt benötigt wird.
Nun müssen noch die Ergebnisse der einzelnen Zeitmessungen in Verbindung gebracht werden und die richtigen Schlüsse aus diesen gezogen werden.

6.3.4 Bedeutung der Zeitmessungen

In den vorangegangenen Abschnitten wurden die einzelnen Generierzeiten des Testprojekts bei der letzten offiziellen Generierung dieser Version, bei der parallelen Generierung mit der Build Forge Threading Technologie und der seriellen Generierung mit Build Forge ermittelt und kurz vorgestellt. In Tabelle 6.4 werden noch einmal die Gesamtgenerierzeiten aus den vorigen Abschnitten nebeneinander dargestellt.

	akt. GA (Kap. 6.3.1)	**BF parallel (Kap. 6.3.2)**	**BF seriell (Kap. 6.3.3)**
Dauer	04:14:33	04:25:44	04:01:54

Tabelle 6.4: Gegenüberstellung der Generierzeiten

In Tabelle 6.4 auf der vorherigen Seite zeigt sich, dass die Laufzeiten der drei untersuchten Methoden zwischen 4 und 4,5 Stunden liegen. In Abschnitt 6.3 wurde bereits erwähnt, dass die Laufzeit einer Generierung immer einer gewissen Schwankung unterliegt. Werden die Schwankungen der Dauer, von bis zu einer dreiviertel Stunde, berücksichtigt, kann aus den Tests mit Build Forge gefolgert werden, dass der Einsatz von Build Forge keine negativen Auswirkungen auf die Dauer der Erzeugung von Software hat.
Demgegenüber kann sogar davon ausgegangen werden, dass durch den Einsatz der „Threading“-Technologie von Build Forge die Gesamtlaufzeit noch reduziert werden kann, wenn mehrere Generierserver zum Einsatz kommen. Weiterhin wird Build Forge im Rahmen der Diplomarbeit natürlich auch durch die vorgegebene Installation des Build Forge Servers, des Datenbankservers und des SUN-Agenten auf einer Maschine verlangsamt.

6.4 Wiederverwendbarkeit generierter Komponenten

Da im untersuchten Projekt auch bereits erzeugte Komponenten in die Generierung höherer Level einfließen, muss Build Forge, wie der aktuelle Generierautomat auch, dieses Vorgehen unterstützen. Da bereits in Abschnitt 6.3 gezeigt wurde, dass eine komplette Generierung eines bereits bestehenden Projekts mit Build Forge erfolgen kann, ist auch schon gezeigt, dass Build Forge auch die Wiederverwendung von Komponenten beherrscht.
Wie bereits in Bild 5.7 auf Seite 58 dargestellt wurde, wurden die einzelnen Generierschritte des Generierautomaten in einzelne Build Forge Schritte umgesetzt. Durch den Aufbau der Projekte aus einzelnen Schritten und die Synchronisation innerhalb eines Projekts (Kap. 6.2.2) ist es möglich, die notwendigen Komponenten vor der Ausführung der Generierbefehle einzuziehen und in den nachfolgenden Generierschritten wieder zu verwenden. In Bild 6.8 auf der nächsten Seite wird der Aufruf für das Einziehen bereits generierter Komponenten dargestellt.

6.5 Parallele Generierung mehrerer Branches

Die parallele Generierung unterschiedlicher Branches ist ganz allgemein für die Softwareentwicklung in großen Projekten von entscheidender Bedeutung, da die Weiterentwicklung und die Wartung freigegebener Versionen auf unterschiedlichen Zweigen erfolgt. Das lässt den Schluss zu, dass KM die Fähigkeit besitzen muss mehrere Versionen parallel zu generieren, da ansonsten die Entwicklung unnötig aufgehalten wird. So wird eine Verzögerung bei einer wichtigen Fehlerbehebung auf einem Branch wegen einer Generierung auf der Hauptentwicklungslinie mit Sicherheit vom Management nicht toleriert. Somit ist die parallele Generierung mehrerer Branches durch Build Forge auch ein Hauptkriterium bei der Frage nach dem erfolgreichen Einsatz von Build Forge

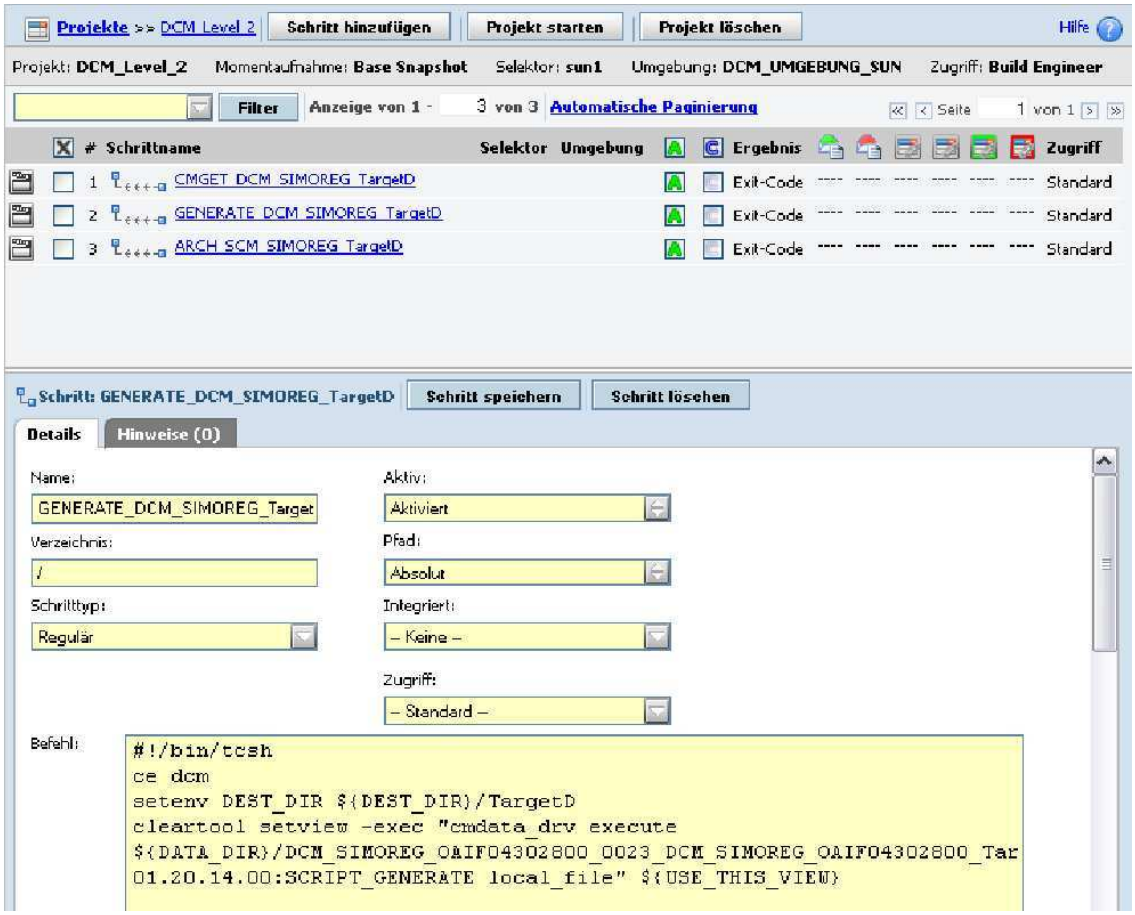


Abbildung 6.8: GETSTAGE-Aufruf in Build Forge

in Softwareprojekten.

Für die Tests der parallelen Generierung ist es erforderlich die in Abschnitt 5.1.2 beschriebene Testumgebung zu erweitern. So wurde für diese Tests ein zusätzlicher Branch namens lel_test_dcm und eine dazugehörige View angelegt, um das Generieren auf zwei unterschiedlichen Branches zu testen. Abb. 6.9 auf der nächsten Seite stellt dar, wie anhand von Build Forge der Branch über das Setzen der zu verwendenden View als BF-Umgebungsvariable generiert werden kann. Dazu muss die Umgebungsvariable „USE_THIS_VIEW“ auf die View gesetzt werden, mit der der zu generierende Branch sichtbar wird. Verwendet man diese Umgebungsvariable konsequent in allen BF-Schritten, werden diese BF-Schritte auch in der durch diese Variable ausgewählten View ausgeführt. Um zu zeigen, dass auch gleichzeitig auf zwei unterschiedlichen Branches generiert werden kann, wurde die Erzeugung der Komponenten aus einem Level kurz hintereinander gestartet, damit diese sich für einen bestimmten Zeitraum überlappen. Dadurch wird sichergestellt, dass Build Forge parallel auf beiden Branches generieren kann, ohne Einfluss auf die Generierung des jeweils anderen zu nehmen.

So wird in Bild 6.10 auf der nächsten Seite die parallele Bearbeitung eines Build Forge Projekts auf zwei unterschiedlichen Branches dargestellt. Wie man dieser Abb. entnehmen kann, wurden beide Projekte wirklich kurz hintereinander gestartet und haben beide den Status „Aktiv“. Dadurch konnte gezeigt werden, dass Build Forge

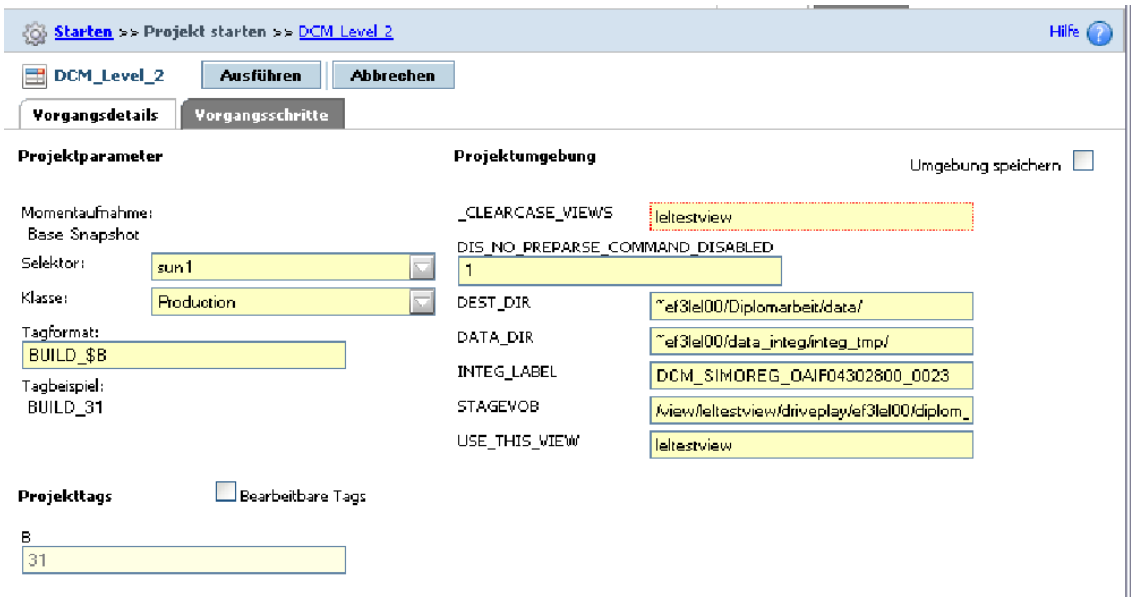


Abbildung 6.9: View-Auswahl durch Umgebungsvariable

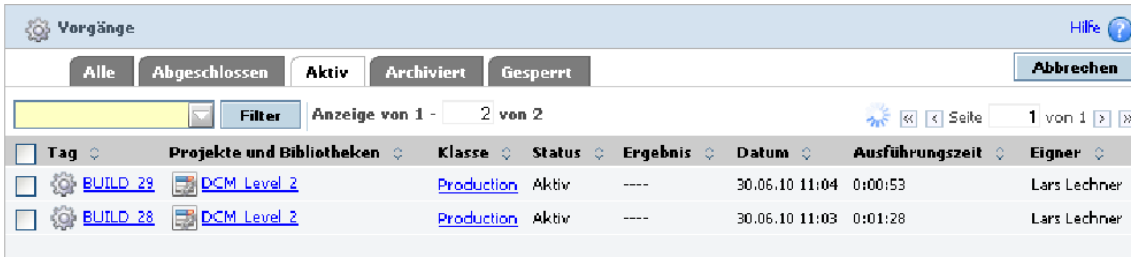


Abbildung 6.10: gleichzeitige Projektausführung

auch eine parallele Generierung mehrerer Branches ermöglicht, die jeweils auch ihren eigenen Status, wie in Abb. 6.11 dargestellt, zurückmelden. Zudem erkennt man bei der Betrachtung von Abb. 6.11 bereits, dass die Generierung mehrerer Branches nicht nur möglich ist, sondern auch beide erfolgreich beendet wurden. Somit ist eine der

Abbildung 6.11: Erfolgreicher Abschluss beider Branches

wichtigsten Anforderung großer Softwareprojekte an CM, nämlich die Unterstützung der Parallelentwicklung, erfüllt, womit Build Forge prinzipiell als Generierwerkzeug in Softwareprojekten eingesetzt werden kann.

Wie in diesem Kapitel durch Tests aufgezeigt wurde, erfüllt Build Forge die in Kapitel 4 geforderten Kriterien.

Jedoch ist neben der prinzipiellen Eignung als Generierwerkzeug, sei es als Ersatz für bereits bestehende Generierautomaten oder als ein von Beginn an eingesetztes Generierwerkzeug auch eine wirtschaftliche Betrachtung durchzuführen.

7 Betrachtung der Wirtschaftlichkeit

Wie im vorigen Kapitel gezeigt wurde, erfüllt Build Forge die technischen Anforderungen, die innerhalb bestehender Projekte oder auch neue Softwareprojekte an ein Werkzeug für die Automatisierung des Generierablaufs gestellt werden. Nun ist neben der reinen Betrachtung der technischen Seite auch ein Blick auf die Kostenseite notwendig, da dies bei der Entscheidung für den Zukauf oder die Eigenentwicklung auch eine entscheidende Rolle spielt. Um hier eine Entscheidungshilfe an die Hand zu bekommen, sind erst einmal Kenntnisse über die reinen Produktkosten von Build Forge notwendig.

7.1 Produktkosten

Da neben weiteren anfallenden Kosten, als erstes einmal der Anschaffungspreis ins Gewicht fällt, werden in Tabelle 7.1 die Kosten der einzelnen Komponenten von Build Forge dargestellt.

Komponente	Preis
IBM RATIONAL BUILD FORGE ENTERPRISE EDITION SERVER LICENSE + SW SUBSCRIPTION & SUPPORT 12 MONTHS	166.938,00 €
IBM RATIONAL BUILD FORGE ACCESS FLOATING USER LICENSE + SW SUBSCRIPTION & SUPPORT 12 MONTHS	4.183,00 €
IBM RATIONAL BUILD FORGE ACCESS AUTHORIZED USER LICENSE + SW SUBSCRIPTION & SUPPORT 12 MONTHS	1.454,00 €
IBM RATIONAL BUILD FORGE QUICK REPORT ENTERPRISE EDITION SERVER LICENSE + SW SUBSCRIPTION & SUPPORT 12 MONTHS	52.239,00 €
IBM RATIONAL BUILD FORGE ENTERPRISE EDITION ADAPTOR TOOLKIT SERVER LICENSE + SW SUBSCRIPTION & SUPPORT 12 MONTHS	37.796,00 €

Tabelle 7.1: Build Forge Preise Juni 2010

Wie man in der Tabelle 7.1 auch erkennt, wird Build Forge von IBM mit zwei unterschiedlichen Lizenzmodellen angeboten. Hierbei handelt es sich einmal um eine Floating Lizenz. Dies bedeutet, dass eine Lizenz Gültigkeit für alle Mitarbeiter besitzt, jedoch immer nur einer mit dieser Lizenz arbeiten kann. Daneben existiert noch das „ACCESS AUTHORIZED“-Modell. Im Gegensatz zum Floating-Lizenzmodell, wird hier die Lizenz dauerhaft einem Mitarbeiter zugeordnet und nur dieser kann mit Build Forge arbeiten. Die Entscheidung, ob mit einer Floating-Lizenz oder einem fest zugeordneten Lizenz gearbeitet wird, hängt von der Projektgröße bzw. falls Build Forge nur bei KM eingesetzt wird, von der Anzahl der KM-Mitarbeiter ab. Daneben bietet IBM auch noch erweiterte Analysemöglichkeiten an, die mit der Komponente „QUICK REPORT„ zugekauft werden

können. So wurden z. B. mit Hilfe von „QUICK REPORT“die Abbildungen aus Anhang D und E erstellt.
Des Weiteren wird als zusätzliche Komponente ein Werkzeugkasten für die Entwicklung eigener Adaptoren angeboten. Mit Hilfe der Adaptoren besteht die Möglichkeit weitere Versionsverwaltungssoftware, wie z. B. Subversion, in Build Forge zu integrieren. Da Build Forge jedoch von sich aus die ClearCase-Unterstützung bietet, wurden die Adaptoren im Rahmen der Diplomarbeit nicht untersucht. Für den sinnvollen Einsatz von Build Forge in einem Projekt wird mindestens der Build Forge Server und eine Zugriffslizenz benötigt, wodurch die Anschaffungskosten von Build Forge mindestens 168392 € betragen.

Da nun die Anschaffungskosten für den Einsatz von Build Forge bekannt sind, wird im nächsten Abschnitt ein Blick auf die Beantwortung der Frage einer Eigenentwicklung oder dem Zukauf von Build Forge geworfen.

7.2 Aufbau neuer Projekte: Build Forge oder Eigenentwicklung

Da im vorigen Abschnitt der Mindestanschaffungspreis von Build Forge bestimmt wurde, wird nun die Frage untersucht, ob eine Eigenentwicklung oder der Zukauf bereits existierender Werkzeuge für ein Softwareprojekt mehr Ersparnis bringt. Wie bereits in Abschnitt 7.1 gezeigt wurde, belaufen sich die Mindestausgaben für Build Forge mit einer Lizenz und Support auf 168392 €.

Bei der wirtschaftlichen Abschätzung werden als Stundensatz für einen Entwickler 100 € zu Grunde gelegt und die Abschätzung der Dauer für die Entwicklung eines ausgereiften Generierautomaten wird in Mannmonaten[21] (MM) vorgenommen. Die Rechnungseinheit Mannmonat besteht aus 20 Manntagen (MT), von denen jeder einzelne wiederum aus 8 Mannstunden (MS) besteht. Durch das Vorgehen die Menge der Arbeit, die pro Zeiteinheit geleistet werden kann, auszudrücken, lassen sich gut Abschätzungen für die anfallenden Kosten vornehmen.

Betrachtet man als Grundlage für die Abschätzung der Entwicklungszeit den Generierautomaten des SINAMICS-Projekts, lag die reine Entwicklungszeit bei ungefähr 9 Mannmonaten. So wurde eine neue Datenhaltung, die in Abschnitt 3.3.1.1 vorgestellte cmdata-Struktur, entwickelt. Neben der Neuentwicklung der cmdata-Struktur war es für die automatisierte Bearbeitung dieser Dateien auch nötig Perlmodule zu entwickeln, die einen Zugriff auf diese Strukturen ermöglichen. Nachdem die Datenhaltung und die Zugriffsfunktionen in den Perlmodulen geschaffen waren, mussten noch die Skripts aus Abschnitt 3.3.1.2 entwickelt werden, durch die eine automatische Abarbeitung der

[21] Die Menge an Arbeit die eine Person in einem Monat leistet. Analog ist der Manntag und die Mannstunde zu sehen

einzelnen Felder aus den cmdata-Dateien erfolgt.
Aufgrund dieser Erfahrungswerte aus dem SINAMICS-Projekt werden im Rahmen der Wirtschaftlichkeitsabschätzung 9-10 Mannmonate Entwicklungszeit für einen Generierautomaten in großen Softwareprojekten angenommen. Da die Kosten für eine Mannstunde mit 100 € bekannt sind, wird die Entwicklungszeit nach folgender Formel ebenfalls in Mannstunden umgerechnet: Anzahl der MS $*$ Anzahl der MT $*$ Anzahl der MM . Werden nun die bekannten Werte eingesetzt, erhält man folgende Gleichung: 8 MS $*\,20$ MT $*\,10$ MM . Somit dauert die Entwicklung eines Generierautomaten 1600 Mannstunden. Mit dem Stundensatz von 100 € und den berechneten 1.600 Mannstunden Entwicklungszeit, betragen die Kosten der Eigenentwicklung eines Generierautomaten 1600 MS $*\,100$ € $= 160000$ € .
Vergleicht man nun den Mindestkaufpreis von Build Forge mit den soeben berechneten Kosten einer Eigenentwicklung wie in Tabelle 7.2 so erkennt man, dass die Eigenentwicklung um 8392 € günstiger wäre.

	Variante	**Kosten**
	Build Forge	168392 €
	Eigenentwicklung	160000 €
Ersparnis		8392 €

Tabelle 7.2: Vergleich der Kosten

Neben dieser geringen Kosteneinsparung bei einer Eigenentwicklung muss jedoch auch berücksichtigt werden, dass bei dem Kauf von Build Forge noch 12 Monate Support durch IBM enthalten sind. Durch die Unterstützung von IBM wird ein schnellerer produktiver Einsatz gefördert, da bei Problemen kompetente Ansprechpartner vorhanden sind. Weiterhin steht bei der Eigenentwicklung immer auch eine größere zeitliche Verzögerung, bis diese letztendlich wirklich produktiv genutzt werden kann, da erst einmal eine gewisse Basis gelegt werden muss. Das bedeutet, dass anfangs Generierung noch händisch ausgeführt werden müssen, wodurch sich die Entwicklung des selbst geschriebenen Generierautomaten verzögert.
Daher kann gesagt werden, dass es sich durchaus lohnt die Möglichkeit eines Einsatzes von Build Forge in neuen Großprojekten zu überdenken, obwohl Build Forge auf den ersten Blick mehr Kosten verursachen mag als eine Eigenentwicklung.

Neben dem Einsatz neuer Projekte ist auch eine Portierung bereits bestehender Projekte möglich.

7.3 Portierungsaufwand bei bestehenden Projekte

Dieser Abschnitt wirft einen kurzen Blick auf den Einsatz von Build Forge in bereits bestehenden Softwareprojekten als Ersatz zu einem selbstentwickelten Generierautomaten. Berücksichtigt man hier, dass bereits Geld in die Eigenentwicklung eines Generierautomaten und auch in die Anpassung der Entwicklungsumgebungen geflossen ist, kommt der Einsatz von Build Forge bei der Umstellung eines Projekts teurer als das ein Projektstart mit Build Forge. Denn neben den bisher bezahlten Entwicklungskosten kommt auch in diesem Fall der Mindesteinkaufspreis von Build Forge von 168392 € aus Abschnitt 7.1 auf Seite 77 dazu. Wie bereits erwähnt, hat auch eine Entwicklung der Entwicklungsumgebung stattgefunden, wodurch die Arbeit mit Build Forge ebenfalls erschwert wird und auch hier zuerst Anpassungen vorgenommen werden müssen. Ein Beispiel dafür ist die in Abschnitt 5.2.3 beschriebene Arbeitsweise von Build Forge über den absoluten View-Pfad, während die Entwicklungsumgebung in vielen Projekten auf eine gesetzte View ausgelegt ist. Das bedeutet, dass neben den reinen Kosten für Build Forge auch zusätzliche Kosten und zeitlicher Verzug durch ein Anpassen der Entwicklungsumgebung entstehen.
Hier muss auch noch erwähnt werden, dass bei einer Ablösung bestehender Skripts durch Build Forge die Mitarbeiter auch auf Schulungen geschickt werden sollten. Zwar ist eine Einarbeitung in Build Forge auch ohne diese möglich, jedoch fällt einem der Einstieg nach der Wissensvermittlung durch den Kursleiter mit Sicherheit leichter. Diese Schulungen erhöhen die Kosten für eine Umstellung um 1487,50 € pro Person.

Als Fazit kann somit gesagt werden, dass im Fall eines Umstiegs ein viel größerer Nutzen erreicht werden muss als dies bei der Weiterentwicklung bereits bestehender Skripts der Fall wäre.

8 Zusammenfassung

Da nun die Testergebnisse aus Kapitel 6 und eine Wirtschaftlichkeitsabschätzung aus Kapitel 7 vorliegen, muss noch entschieden werden, ob die Verwendung von Build Forge nur bei neuen Projekten ratsam ist, oder ob auch ein Nutzen aus der Umstellung bereits bestehender Projekte gezogen werden kann.

8.1 Einsatztauglichkeit

Das Ziel der Diplomarbeit war es die Leistungsfähigkeit von Build Forge in großen Softwareprojekten zu untersuchen und ob es sinnvolle Einsatzmöglichkeiten innerhalb dieser gibt.

Betrachtet man nun die Testergebnisse aus Kapitel 6 so gelangt man aus technischer Sicht zu dem Schluss, dass sich Build Forge für den Einsatz in großen Softwareprojekten auf jeden Fall eignet. So erfüllt es die an einen Generierautomaten gestellte Forderung nach

- Plattformunabhängigkeit
- Generierung mehrerer Branches
- Unterstützung von Versionsverwaltungssoftware
- Unterstützung von Komponenten und Leveln

wie in den Tests bewiesen wurde.

Auch aus wirtschaftlicher Sicht (Kap. 7) gelangt man im Rahmen dieser Diplomarbeit zu dem Schluss, dass sich Build Forge für den Einsatz in großen Softwareprojekten anbietet, da die Kosten im Vergleich mit einer Eigenentwicklung nur unwesentlich höher ausfallen.

Allerdings muss zwischen zwei Anwendungsfällen, dem Einsatz in neuen Projekten und der Verwendung in Altprojekten, unterschieden werden.

8.1.1 Einsatz in neuen Projekten

Wie bereits in Abschnitt 8.1 ausgeführt wurde, besitzt Build Forge auf jeden Fall die Eignung für den Einsatz in großen Softwareprojekten. Durch das, nach kurzer Einarbeitung, problemlose Anlegen der BF-Projekte und den Einsatz der Agenten ist es KM möglich sehr schnell Softwarestände zu erzeugen. Des Weiteren besitzt Build Forge, um der Verlängerung der Generierzeit wachsender Projekte entgegenzuwirken, mit der

Unterstützung der parallelen Generierung durch „Threading“ ebenfalls ein sehr gute Lösung, die ansonsten zeitaufwendig entwickelt werden müsste. Neben einer wachsenden Anzahl der Komponenten spielt auch die Lauffähigkeit der Build Forge Agenten auf unterschiedlichen Plattformen eine sehr große Rolle, da im Laufe der Entwicklung auch andere Entwicklungsumgebungen in das Projekt mit einfließen können. Durch die Agenten können neue Entwicklungsplattformen problemlos in eine bereits bestehende Build Forge Umgebung integriert werden.

Bei der Entscheidungsfindung ist es auf jeden sinnvoll über den Zukauf von Build Forge nachzudenken. Als Nebeneffekt der Verwendung von Build Forge in einem Großprojekt ergibt sich auch eine höhere Wahrscheinlichkeit für den Einsatz von Build Forge in kleineren Softwareprojekten, da durch das große Projekt bereits eine Infrastruktur vorhanden ist. Dies hat automatisch homogenere Projekt zur Folge..

8.1.2 Einsatz in bestehenden Projekten

Im vorigen Abschnitt wurde der Einsatz von Build Forge in neuen Softwareprojekten betrachtet und es wurde aufgezeigt, dass die Anwendung dort vielversprechend ist. Demgegenüber steht der Einsatz von Build Forge in Altprojekten. Zwar ist Build Forge hier prinzipiell auch geeignet, allerdings müssen hier Einschränkungen vorgenommen werden. Bei bereits bestehenden Projekten sind im Laufe der Zeit eine Reihe von Lösungen entstanden, die nicht ohne Probleme und erheblichen Mehraufwand in Build Forge integriert werden können. Ein Beispiel hierfür ist die Arbeitsweise von Build Forge mit dem absoluten View-Pfad, während z. B. im Projekt SINAMICS mit gesetzten Views gearbeitet wird. Um bei der Umstellung die Entwicklung nicht zu blockieren, müsste Build Forge parallel zu einem laufenden Betrieb eingerichtet werden, was jedoch einen längeren Zeitraum in Anspruch nimmt, da auch weiterhin Alltagsaufgaben anfallen.

So sollte Build Forge in Altprojekten nur dann eingesetzt werden, wenn der zu erwartende Gewinn durch die Harmonisierung die Kosten der Einführung übertrifft.

8.2 Ausblick

Im Rahmen der Diplomarbeit wurde zuerst einmal die Tauglichkeit von Build Forge für KM untersucht. Neben dieser reinen technischen und wirtschaftlichen Untersuchung müssten vor einem Einsatz auch einmal die weiteren Werkzeuge von Build Forge für die Unterstützung von KM untersucht werden. Hierzu gehören die Analysefunktionen,die Adaptoren und der automatische Generierstart zu festgelegten Startzeiten.

Auch ist es für die Zukunft sicherlich sinnvoll Build Forge bzgl. der Verwendbarkeit durch den Softwareentwickler zu untersuchen, da dies ein einheitliches Vorgehen bei der Erzeugung von Software durch KM und den Softwareentwickler ermöglicht. In diesem

Zusammenhang würde sich auch eine Untersuchung der Integration von Build Forge in eine IDE, wie z. B. Eclipse, durch Plug-Ins anbieten.

Weiterhin sollten neben Build Forge noch andere Werkzeuge für die Automatisierung von Generierungen, wie z. B. den „Team Foundation Server“ von Microsoft oder „Electric Cloud“, untersucht werden, um eine breite Auswahl von Werkzeugen vergleichen zu können. So bietet der „Team Foundation Server“ zusätzlich zu der Automatisierung von Generierungen auch eine eigene Versionsverwaltung, wodurch zusätzliche Kosten für eine Versionsverwaltungssoftware, wie z. B. ClearCase, entfallen. Bei diesen Untersuchungen müssten die Unterschiede und die Verwendbarkeit dieser Produkte bzgl. gemischter Entwicklungsplattformen und der Unterstützung von parallelen Generierungen herausgearbeitet werden.

So bleibt, dass eine optimale Entscheidungsfindung über das zu verwendende Produkt erst nach einem Vergleich von Build Forge mit seinen Konkurrenzprodukten stattfinden kann

Anhang A: cmdata-Dateien

A.1 cmdata_paths

```
cmdata_paths: cmdata_file
{
  READ_PATHS:   Pfad der cmdata-Datei ausserhalb des VOBs
  WRITE_PATHS:  Pfad der cmdata-Datei  im VOB
  GROUP:        Zu welcher Gruppe, gehört diese Datei
                  -Komponente
                  -andere
}
```

A.2 cmdata_versions

```
cmdata_versions_KomponenteA: Vww.xx.yy.zz
{
  INTEG_LABEL:      enhält den Stempel der aktuellen Generierung
  GENERATE:         bei SUN-Komponenten steht hier der
     Generieraufruf
  GENERATE_PC:      bei PC-Komponeten steht hier der
     Generieraufruf
  ARCHIVE:          Kopieren der zip-Datei in den Stage-VOB und
     stempeln des Pfades
  COPY:             Kopieren der Generierergebnisse in ein temp.
      Verzeichnis
  PACKAGE:          Erzeugen einer zip-Datei mit dem Inhalt des
     temp. Verzeichnisses
  BRANCH:           Branchname auf dem generiert wird
  STAGEFILE_LIST:   Pfadname der Datei im Stage-VOB
  CMGET_COMPLIST:   Liste von zu verwendenden Komponenten
  LATEST:           Branchname, wenn es sich um den aktuellen
     Datensatz handelt
  DOGEN_LEVEL:      Generierlevel in dem diese Komponente
     generiert wird
}
```

A.3 cmdata_components

```
cmdata_components: Komponente
{
  ACTIVE:
  {
    Status der Komponente auf den Branches
    main     : yes  -> Die Komponente ist für main relevant
    branch1  : no   -> Kompontene ist für branch1 nicht mehr
       relevant
  }
  FLAGS:       zusätzliche Merkmale, z.~B. ob diese Komponente
     freigegeben werden kann
  GROUP_IF:    Welche Schnittstellen haben Interesse an dieser
     Komponente
  GROUP_GEN:   Zu welcher Generiergruppe gehört die Komponente.
}
```

A.4 cmdata_integrations

```
cmdata_integrations: STEMPEL
{
  STATUS:           Status der Integration
                     -> offen
                     -> geplant
  START:            Startdatum der Integration
  START_PLANNED:    geplantes Startdatum
  FINISH:           Enddatum der Integration
  MASTER_VERSION:   Hauptversion der Integration
  PRODUCT_STATE:    wurde diese Version freigegeben
}
```

A.5 cmdata_branches

```
cmdata_branches: BRANCH
{
  LABEL_BRANCH_STARTS: Von welchem Stempel wurde der Branch
      gezogen
  VIEWSTORE_INTEG:     auf welchem Laufwerk liegt das Viewstore
  ACTIVE:              Status des Branches
  DELIV_PREL_TO:       An wen wird dieser Branch vorab abgegeben
  SAFETY_KEY:          Welche Safetykeys gibt es auf diesem
      Branch
}
```

A.6 cmdata_interfaces

```
cmdata_interfaces: SCHNITTSTELLE
{
  SELECT_COMPONENTS: Wenn dieses Feld gesetzt ist, wird eine
      Komponentenliste zur Auswahl angeboten
  MAIL_DIST_*:       Mailverteiler
  ADD_FILES:         Zusätzliche Dateien für diese Schnittstelle
  SYSTEM_MODE:       Müssen neben dem Versenden der Mail noch
      zusätzliche Aktionen ausgeführt werden.
}
```

A.7 cmdata_project

cmdata_project: PROJECT
{
  PROJECT_CMDATA_PATHS: an welcher Stelle liegt das projektspezifische **cmdata_paths**
  SUPER_CMDATA_PATHS: Pfad zum projektübergeordneten **cmdata_paths**
  PROJECT_SITENAMES: Liste der Standorte, an denen entwickelt wird
  INTEG_FILELIST: Enhält die Pfade aller Dateien, die für eine Integration benötigt werden.
  NETWORK_DRIVE_HOME_GENUSER: Auf welches PC–Laufwerk wird das Homeverzeichnis des Generiernutzers verbunden.
}

Anhang B: Ausschnitt der Generierzeiten SINAMICS

INTEGRATION : ARM_I4403_0593		
SCRIPT : DOCMGEN		
STEP START	STEP STOP	DIFF
GLOBAL ACTIONS	GLOBAL ACTIONS FINISHED	00:08:11
*INFO : Executing cm_safety-check	FINISHED CMSAFETY	00:20:52
*INFO : checking for software changes	FINISHED CMLABEL	01:10:03
*INFO : CREATING LOG-FILES FOR SELECTED COMPS	FINISHED CREATING LOG-FILES	00:07:44
STEP 11-PREPARATION FOR GENERATIONON PC	FINISHED PREPARATION	00:00:08
STEP 12.1-GETTING FILES FROM STAGEVOB	FINISHED GETSTAGE1	00:09:26
STEP 13.1-EXECUTE GENERA-TION	FINISHED GENERATION1	03:18:13
STEP 14.1-CHECK LOGFILE,ZIP AND COPY RESULT	FINISHED CHECK & COPY1	01:10:26
*INFO : creating SUPPLY_base-files	FINISHED SUPPLY FILES1	00:46:28
...	...	...
STEP 14.5-CHECK LOGFILE, ZIP AND COPY RESULT	FINISHED CHECK & COPY5	01:51:10
*INFO : creating SUPPLY_base-files	FINISHED SUPPLY FILES5	00:09:03
*INFO : preliminary delivery	FINISHED PRELIMINARY DELI-VERY 5	00:00:11
*INFO : Send mail after	FINISHED SENDING MAIL5	00:00:05
...	...	...
STEP 15-PRINT SUMMARY AND UPDATE METRICS	FINISHED METRICS MAIL	00:00:08
		=34:53:16

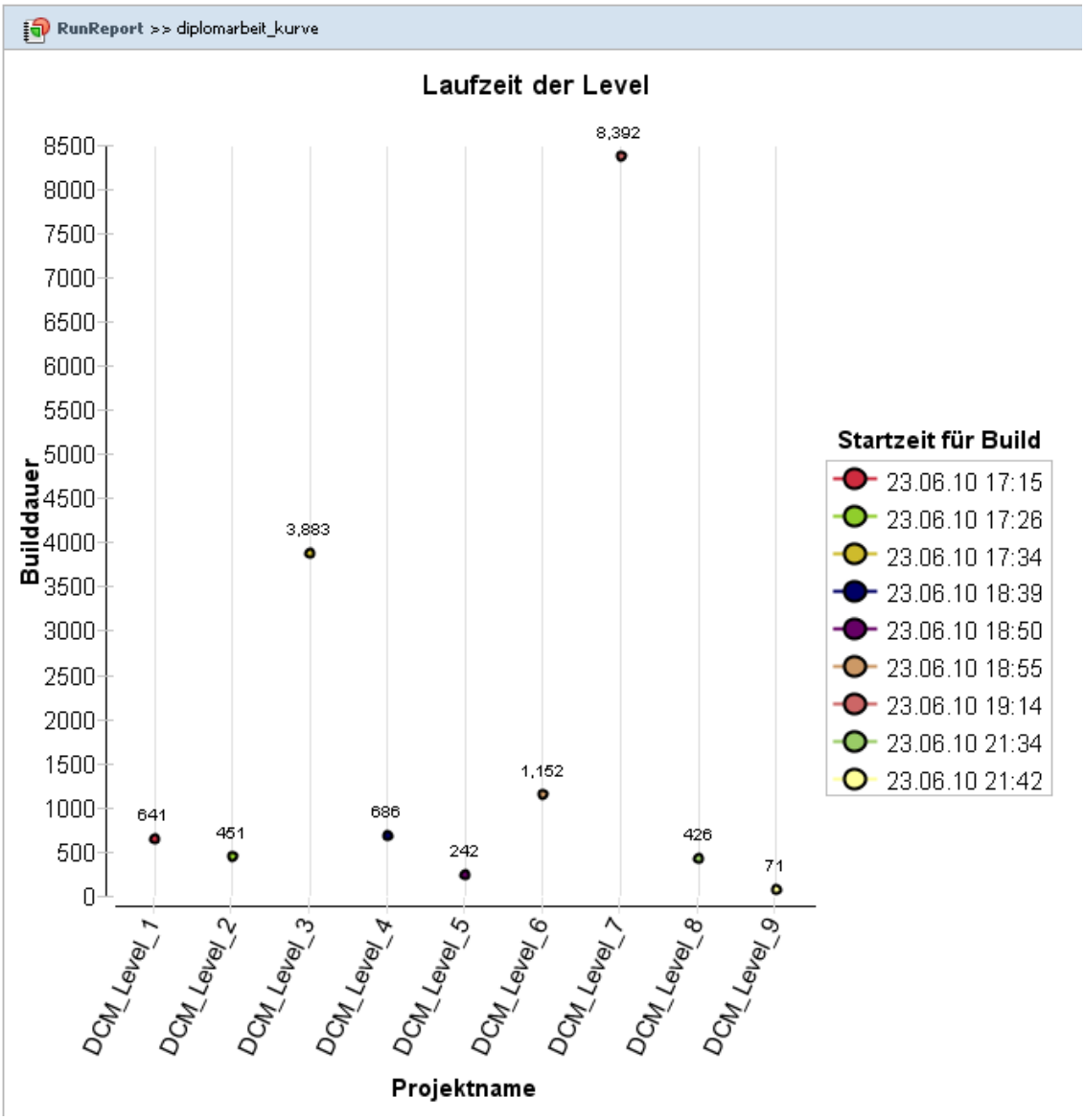
Anhang C: Originallaufzeit des Testprojekts

INTEG: DCM_SIMOREG_$LABEL		
SCRIPT: DOCMGEN		
STEP START	STEP STOP	DIFF
GLOBAL ACTIONS	GLOBAL ACTIONS FINISHED	00:01:29
* INFO : checking for software changes	FINISHED CMLABEL	00:15:25
* INFO : CREATING LOG-FILES FOR SELECTED COM-PONENTS	FINISHED CREATING LOG-FILES	00:00:39
STEP 11 - PREPARATION FOR GENERATION ON PC	FINISHED PREPARATION	00:01:11
STEP 12.1 - GETTING FILES FROM STAGE VOB	FINISHED GETSTAGE 1	00:01:21
STEP 13.1 - EXECUTE GENE-RATION	FINISHED GENERATION 1	00:04:29
STEP 14.1 - CHECK LOGFILE & ZIP & COPY RESULT	FINISHED CHECK & COPY 1	00:04:46
* INFO : creating SUPPLY_base files	FINISHED SUPPLY FILES 1	00:03:42
* INFO : preliminary delivery	FINISHED PRELIMINARY DELI-VERY 1	00:00:01
STEP 12.2 - GETTING FILES FROM STAGE VOB	FINISHED GETSTAGE 2	00:00:06
STEP 13.2 - EXECUTE GENE-RATION	FINISHED GENERATION 2	00:03:30
STEP 14.2 - CHECK LOGFILE & ZIP & COPY RESULT	FINISHED CHECK & COPY 2	00:00:17
* INFO : creating SUPPLY_base files	FINISHED SUPPLY FILES 2	00:00:11
* INFO : preliminary delivery	FINISHED PRELIMINARY DELI-VERY 2	00:00:02
STEP 12.3 - GETTING FILES FROM STAGE VOB	FINISHED GETSTAGE 3	00:00:16
STEP 13.3 - EXECUTE GENE-RATION	FINISHED GENERATION 3	01:02:37

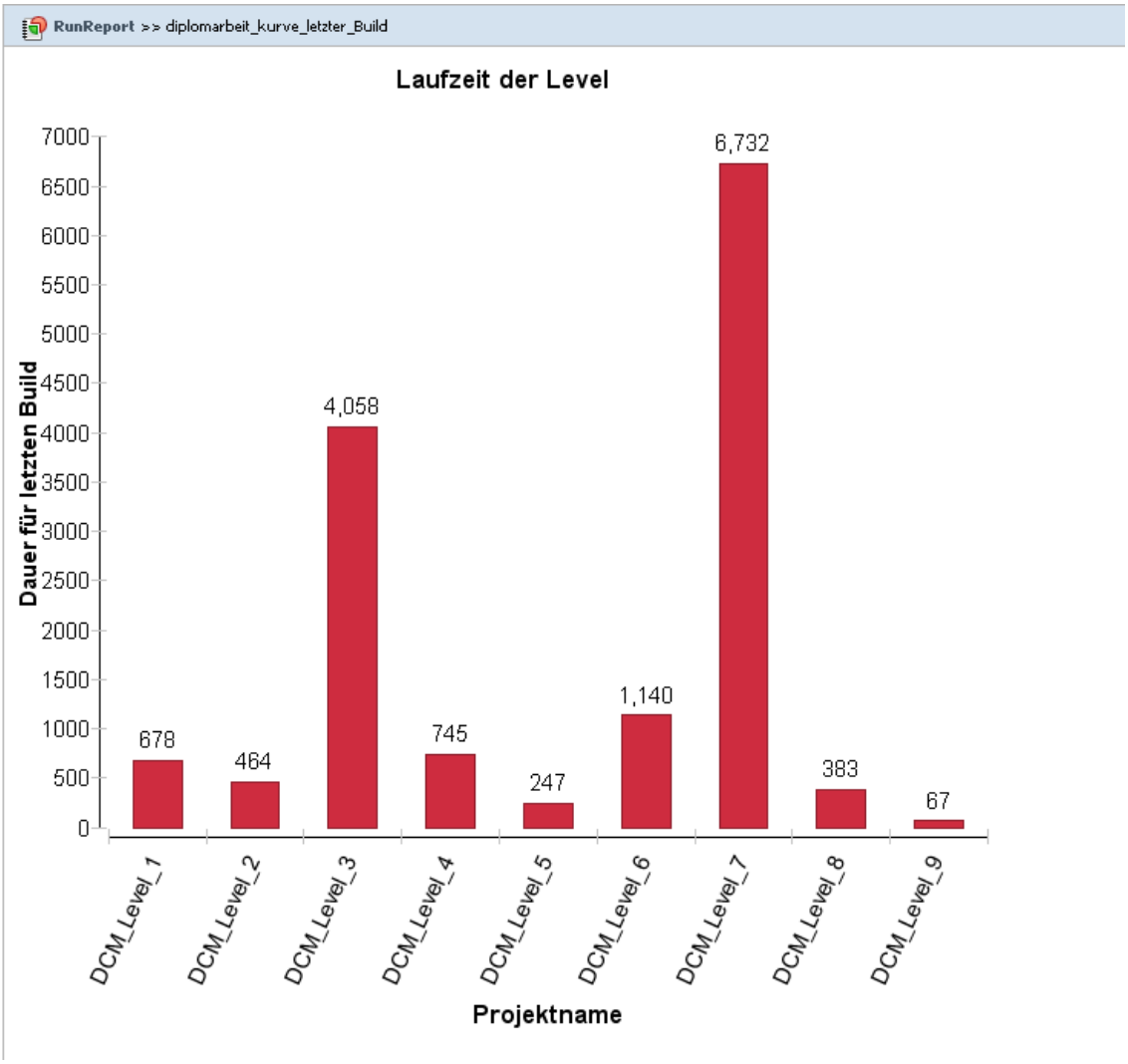
STEP 14.3 - CHECK LOGFILE & ZIP & COPY RESULT	FINISHED CHECK & COPY 3	00:02:15
* INFO : creating SUPPLY_base files	FINISHED SUPPLY FILES 3	00:01:21
* INFO : preliminary delivery	FINISHED PRELIMINARY DELIVERY 3	00:00:01
STEP 12.4 - GETTING FILES FROM STAGE VOB	FINISHED GETSTAGE 4	00:01:02
STEP 13.4 - EXECUTE GENERATION	FINISHED GENERATION 4	00:06:50
STEP 14.4 - CHECK LOGFILE & ZIP & COPY RESULT	FINISHED CHECK & COPY 4	00:02:52
* INFO : creating SUPPLY_base files	FINISHED SUPPLY FILES 4	00:01:44
* INFO : preliminary delivery	FINISHED PRELIMINARY DELIVERY 4	00:00:01
STEP 12.5 - GETTING FILES FROM STAGE VOB	FINISHED GETSTAGE 5	00:00:33
STEP 13.5 - EXECUTE GENERATION	FINISHED GENERATION 5	00:01:37
STEP 14.5 - CHECK LOGFILE & ZIP & COPY RESULT	FINISHED CHECK & COPY 5	00:02:07
* INFO : creating SUPPLY_base files	FINISHED SUPPLY FILES 5	00:01:31
* INFO : preliminary delivery	FINISHED PRELIMINARY DELIVERY 5	00:00:01
STEP 12.6 - GETTING FILES FROM STAGE VOB	FINISHED GETSTAGE 6	00:00:16
STEP 13.6 - EXECUTE GENERATION	FINISHED GENERATION 6	00:16:45
STEP 14.6 - CHECK LOGFILE & ZIP & COPY RESULT	FINISHED CHECK & COPY 6	00:00:33
* INFO : creating SUPPLY_base files	FINISHED SUPPLY FILES 6	00:00:12
* INFO : preliminary delivery	FINISHED PRELIMINARY DELIVERY 6	00:00:03
STEP 12.7 - GETTING FILES FROM STAGE VOB	FINISHED GETSTAGE 7	00:01:30
STEP 13.7 - EXECUTE GENERATION	FINISHED GENERATION 7	01:40:10

STEP 14.7 - CHECK LOGFILE & ZIP & COPY RESULT	FINISHED CHECK & COPY 7	00:03:16
* INFO : creating SUPPLY_base files	FINISHED SUPPLY FILES 7	00:01:02
* INFO : preliminary delivery	FINISHED PRELIMINARY DELIVERY 7	00:00:02
STEP 12.8 - GETTING FILES FROM STAGE VOB	FINISHED GETSTAGE 8	00:00:07
STEP 13.8 - EXECUTE GENERATION	FINISHED GENERATION 8	00:00:09
STEP 14.8 - CHECK LOGFILE & ZIP & COPY RESULT	FINISHED CHECK & COPY 8	00:06:37
* INFO : creating SUPPLY_base files	FINISHED SUPPLY FILES 8	00:00:34
* INFO : preliminary delivery	FINISHED PRELIMINARY DELIVERY 8	00:00:02
STEP 12.9 - GETTING FILES FROM STAGE VOB	FINISHED GETSTAGE 9	00:00:20
STEP 13.9 - EXECUTE GENERATION	FINISHED GENERATION 9	00:00:18
STEP 14.9 - CHECK LOGFILE & ZIP & COPY RESULT	FINISHED CHECK & COPY 9	00:00:20
* INFO : creating SUPPLY_base files	FINISHED SUPPLY FILES 9	00:00:12
* INFO : preliminary delivery	FINISHED PRELIMINARY DELIVERY 9	00:00:01
STEP 15 - PRINT SUMMARY & UPDATE METRICS	FINISHED METRICS MAIL	00:00:07
		= 04:14:33

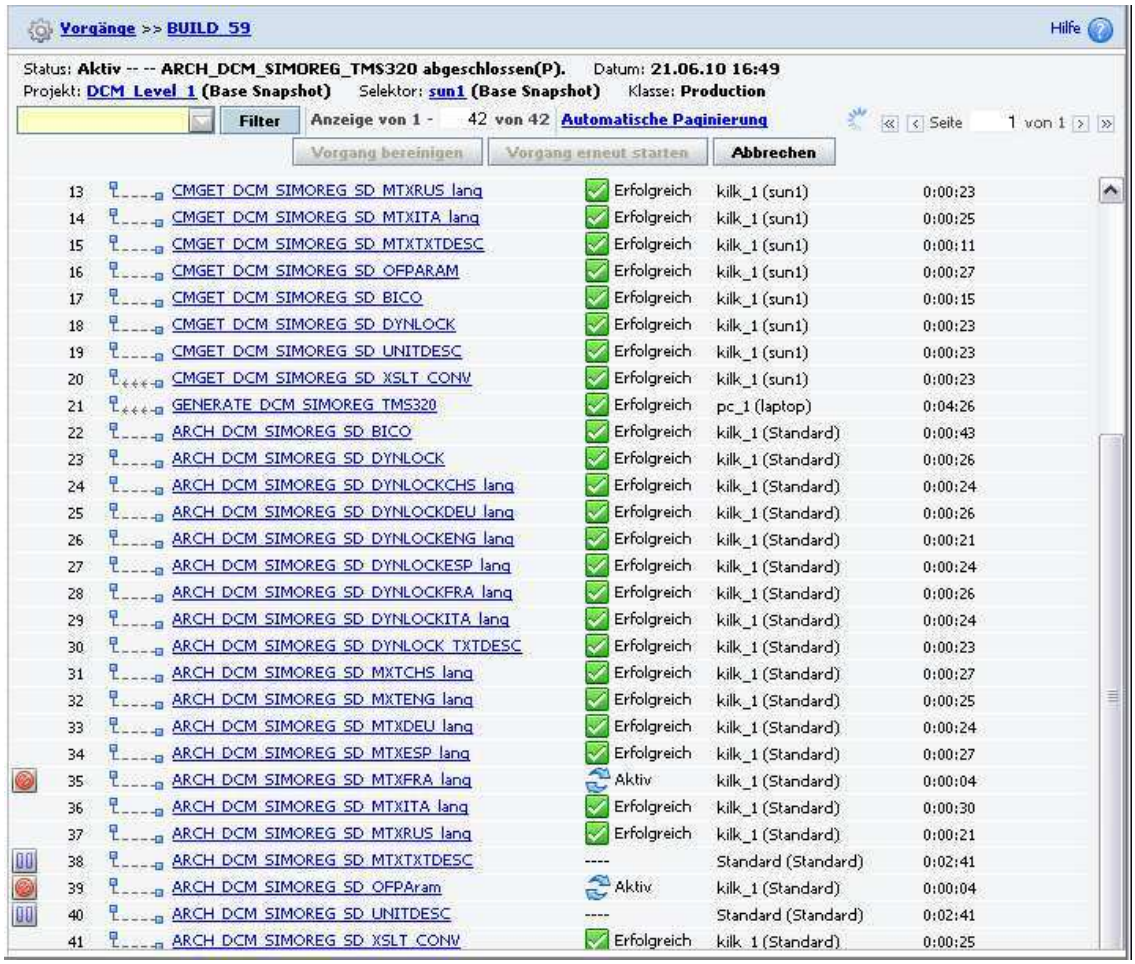
Anhang D: Laufzeit mit Threading



Anhang E: Laufzeit seriell



Anhang F: Synchronisation ARCHIVE_*



Vorgänge >> BUILD_59 Hilfe

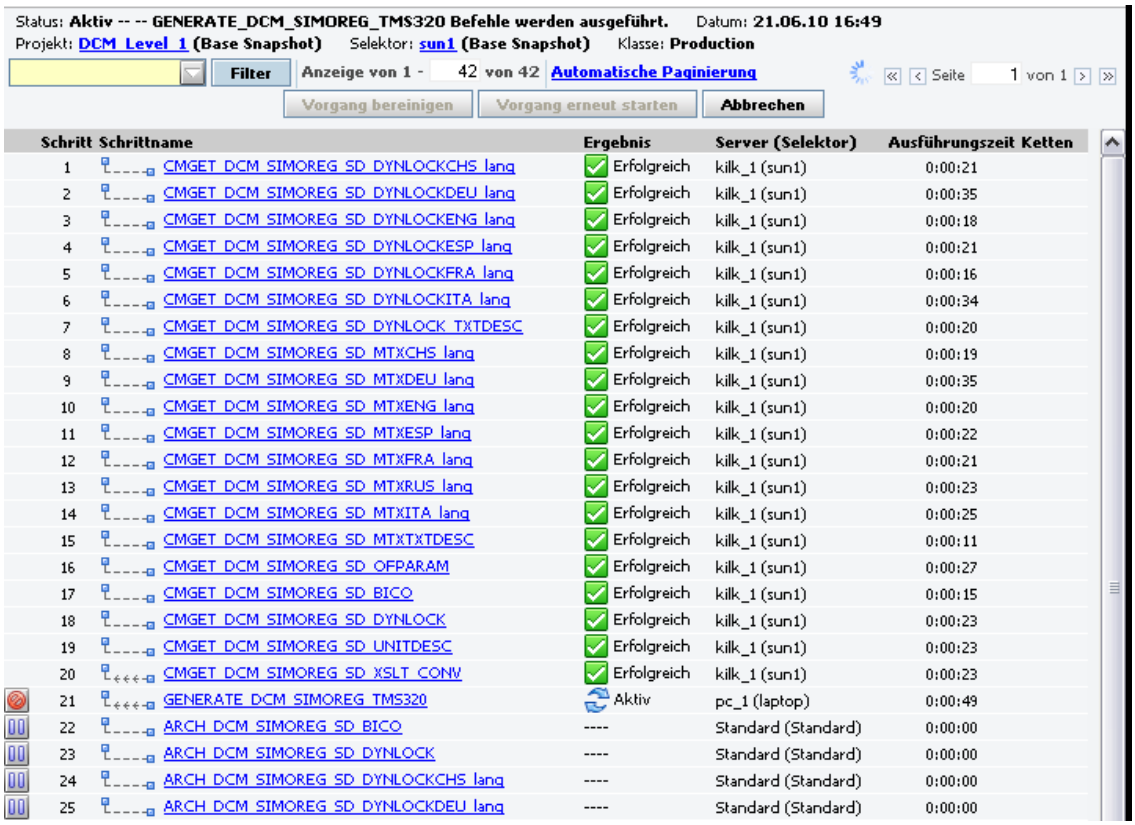
Status: **Aktiv** -- -- **ARCH_DCM_SIMOREG_TMS320 abgeschlossen(P).** Datum: **21.06.10 16:49**
Projekt: **DCM_Level_1 (Base Snapshot)** Selektor: **sun1 (Base Snapshot)** Klasse: **Production**

Filter | Anzeige von 1 - 42 von 42 | Automatische Paginierung | « ‹ Seite 1 von 1 › »

Vorgang bereinigen | Vorgang erneut starten | Abbrechen

13	CMGET_DCM_SIMOREG_SD_MTXRUS_lang	Erfolgreich	kilk_1 (sun1)	0:00:23
14	CMGET_DCM_SIMOREG_SD_MTXITA_lang	Erfolgreich	kilk_1 (sun1)	0:00:25
15	CMGET_DCM_SIMOREG_SD_MTXTXTDESC	Erfolgreich	kilk_1 (sun1)	0:00:11
16	CMGET_DCM_SIMOREG_SD_OFPARAM	Erfolgreich	kilk_1 (sun1)	0:00:27
17	CMGET_DCM_SIMOREG_SD_BICO	Erfolgreich	kilk_1 (sun1)	0:00:15
18	CMGET_DCM_SIMOREG_SD_DYNLOCK	Erfolgreich	kilk_1 (sun1)	0:00:23
19	CMGET_DCM_SIMOREG_SD_UNITDESC	Erfolgreich	kilk_1 (sun1)	0:00:23
20	CMGET_DCM_SIMOREG_SD_XSLT_CONV	Erfolgreich	kilk_1 (sun1)	0:00:23
21	GENERATE_DCM_SIMOREG_TMS320	Erfolgreich	pc_1 (laptop)	0:04:26
22	ARCH_DCM_SIMOREG_SD_BICO	Erfolgreich	kilk_1 (Standard)	0:00:43
23	ARCH_DCM_SIMOREG_SD_DYNLOCK	Erfolgreich	kilk_1 (Standard)	0:00:26
24	ARCH_DCM_SIMOREG_SD_DYNLOCKCHS_lang	Erfolgreich	kilk_1 (Standard)	0:00:24
25	ARCH_DCM_SIMOREG_SD_DYNLOCKDEU_lang	Erfolgreich	kilk_1 (Standard)	0:00:26
26	ARCH_DCM_SIMOREG_SD_DYNLOCKENG_lang	Erfolgreich	kilk_1 (Standard)	0:00:21
27	ARCH_DCM_SIMOREG_SD_DYNLOCKESP_lang	Erfolgreich	kilk_1 (Standard)	0:00:24
28	ARCH_DCM_SIMOREG_SD_DYNLOCKFRA_lang	Erfolgreich	kilk_1 (Standard)	0:00:26
29	ARCH_DCM_SIMOREG_SD_DYNLOCKITA_lang	Erfolgreich	kilk_1 (Standard)	0:00:24
30	ARCH_DCM_SIMOREG_SD_DYNLOCK_TXTDESC	Erfolgreich	kilk_1 (Standard)	0:00:23
31	ARCH_DCM_SIMOREG_SD_MXTCHS_lang	Erfolgreich	kilk_1 (Standard)	0:00:27
32	ARCH_DCM_SIMOREG_SD_MXTENG_lang	Erfolgreich	kilk_1 (Standard)	0:00:25
33	ARCH_DCM_SIMOREG_SD_MTXDEU_lang	Erfolgreich	kilk_1 (Standard)	0:00:24
34	ARCH_DCM_SIMOREG_SD_MTXESP_lang	Erfolgreich	kilk_1 (Standard)	0:00:27
35	ARCH_DCM_SIMOREG_SD_MTXFRA_lang	Aktiv	kilk_1 (Standard)	0:00:04
36	ARCH_DCM_SIMOREG_SD_MTXITA_lang	Erfolgreich	kilk_1 (Standard)	0:00:30
37	ARCH_DCM_SIMOREG_SD_MTXRUS_lang	Erfolgreich	kilk_1 (Standard)	0:00:21
38	ARCH_DCM_SIMOREG_SD_MTXTXTDESC	----	Standard (Standard)	0:02:41
39	ARCH_DCM_SIMOREG_SD_OFPAram	Aktiv	kilk_1 (Standard)	0:00:04
40	ARCH_DCM_SIMOREG_SD_UNITDESC	----	Standard (Standard)	0:02:41
41	ARCH_DCM_SIMOREG_SD_XSLT_CONV	Erfolgreich	kilk_1 (Standard)	0:00:25

Anhang G: Synchronisation GENERATE_*



Status: **Aktiv -- -- GENERATE_DCM_SIMOREG_TMS320 Befehle werden ausgeführt.** Datum: **21.06.10 16:49**
Projekt: **DCM_Level_1 (Base Snapshot)** Selektor: **sun1 (Base Snapshot)** Klasse: **Production**
Filter | Anzeige von 1 - 42 von 42 | Automatische Paginierung | Seite 1 von 1
Vorgang bereinigen | Vorgang erneut starten | Abbrechen

Schritt	Schrittname	Ergebnis	Server (Selektor)	Ausführungszeit Ketten
1	CMGET_DCM_SIMOREG_SD_DYNLOCKCHS_lang	Erfolgreich	kilk_1 (sun1)	0:00:21
2	CMGET_DCM_SIMOREG_SD_DYNLOCKDEU_lang	Erfolgreich	kilk_1 (sun1)	0:00:35
3	CMGET_DCM_SIMOREG_SD_DYNLOCKENG_lang	Erfolgreich	kilk_1 (sun1)	0:00:18
4	CMGET_DCM_SIMOREG_SD_DYNLOCKESP_lang	Erfolgreich	kilk_1 (sun1)	0:00:21
5	CMGET_DCM_SIMOREG_SD_DYNLOCKFRA_lang	Erfolgreich	kilk_1 (sun1)	0:00:16
6	CMGET_DCM_SIMOREG_SD_DYNLOCKITA_lang	Erfolgreich	kilk_1 (sun1)	0:00:34
7	CMGET_DCM_SIMOREG_SD_DYNLOCK_TXTDESC	Erfolgreich	kilk_1 (sun1)	0:00:20
8	CMGET_DCM_SIMOREG_SD_MTXCHS_lang	Erfolgreich	kilk_1 (sun1)	0:00:19
9	CMGET_DCM_SIMOREG_SD_MTXDEU_lang	Erfolgreich	kilk_1 (sun1)	0:00:35
10	CMGET_DCM_SIMOREG_SD_MTXENG_lang	Erfolgreich	kilk_1 (sun1)	0:00:20
11	CMGET_DCM_SIMOREG_SD_MTXESP_lang	Erfolgreich	kilk_1 (sun1)	0:00:22
12	CMGET_DCM_SIMOREG_SD_MTXFRA_lang	Erfolgreich	kilk_1 (sun1)	0:00:21
13	CMGET_DCM_SIMOREG_SD_MTXRUS_lang	Erfolgreich	kilk_1 (sun1)	0:00:23
14	CMGET_DCM_SIMOREG_SD_MTXITA_lang	Erfolgreich	kilk_1 (sun1)	0:00:25
15	CMGET_DCM_SIMOREG_SD_MTXTXTDESC	Erfolgreich	kilk_1 (sun1)	0:00:11
16	CMGET_DCM_SIMOREG_SD_OFPARAM	Erfolgreich	kilk_1 (sun1)	0:00:27
17	CMGET_DCM_SIMOREG_SD_BICO	Erfolgreich	kilk_1 (sun1)	0:00:15
18	CMGET_DCM_SIMOREG_SD_DYNLOCK	Erfolgreich	kilk_1 (sun1)	0:00:23
19	CMGET_DCM_SIMOREG_SD_UNITDESC	Erfolgreich	kilk_1 (sun1)	0:00:23
20	CMGET_DCM_SIMOREG_SD_XSLT_CONV	Erfolgreich	kilk_1 (sun1)	0:00:23
21	GENERATE_DCM_SIMOREG_TMS320	Aktiv	pc_1 (laptop)	0:00:49
22	ARCH_DCM_SIMOREG_SD_BICO	----	Standard (Standard)	0:00:00
23	ARCH_DCM_SIMOREG_SD_DYNLOCK	----	Standard (Standard)	0:00:00
24	ARCH_DCM_SIMOREG_SD_DYNLOCKCHS_lang	----	Standard (Standard)	0:00:00
25	ARCH_DCM_SIMOREG_SD_DYNLOCKDEU_lang	----	Standard (Standard)	0:00:00

Literaturverzeichnis

[Bel05] David Bellagio. *Software configuration management strategies and IBM Rational ClearCase : a practical introduction*. IBM Press, Upper Saddle River NJ, 2nd ed. edition, 2005.

[BH09] Bruno Bartl and Stefan Huhn. IBM Rational Software Conference 2009 Build Management & Automatisierung rund um den Globus. ftp://ftp.software.ibm.com/software/emea/de/rsc/tag1/WED_PM_TCJ_1_3_Bartl_IBM.pdf, 2009. [Online; Stand 13. Juli 2010].

[Cor03] IBM Corporation. Essentials of configuration management with rational clearcase,student manual,version 2003.06.00. Kursmaterial für ClearCase-Kurse bei IBM, 2003.

[Cor09] IBM Corporation. *IBM Rational Build Forge Onlinehilfe, Version 7.1.1.2*, 2009.

[Cor10a] IBM Corporation. Kursbeschreibung: Essentials of IBM Rational Build Forge. http://www.www-304.ibm.com/jct03001c/services/learning/ites.wss/de/de?pageType=course_description&courseCode=RS5440DE, 2010. [Online; Stand 04. August 2010].

[Cor10b] IBM Corporation. Rational Build Forge Enterprise Edition: Features and benefits. http://www-01.ibm.com/software/awdtools/buildforge/enterprise/features/index.html?S_CMP=rnav, 2010. [Online; Stand 28. Februar 2010].

[Cor10c] IBM Corporation. Rational Build Forge Enterprise Edition: Overview. http://www-01.ibm.com/software/awdtools/buildforge/enterprise/index.html, 2010. [Online; Stand 28. Februar 2010].

[Cor10d] IBM Corporation. Selecting the correct IBM Rational software product license. http://www-01.ibm.com/software/rational/howtobuy/licensing/, 2010. [Online; Stand 13. Juli 2010].

[Her99] Helmut Herold. *Linux-Unix-Shells : Bourne-Shell, Korn-Shell, C-Shell, bash, tcsh*. Linux/Unix und seine Werkzeuge. Addison-Wesley, Bonn [u.a.], 3. aktualisierte aufl. edition, 1999. Bis 2. Aufl. u.d.T.: Herold, Helmut: Unix-Shells.

[iee05] IEEE Standard for Software Configuration Management Plans. *IEEE Std 828-2005 (Revision of IEEE Std 828-1998)*, pages 0_1–19, 2005.

[Koc07a] Helga Koch. ClearCase Schulung. Bereitgestellt für Entwickler im VOB armdoc unter /armdoc/..../cc_schulung.ppt, 2007.

[Koc07b] Helga Koch. Project-Configuration-Management-Plan. Bereitgestellt für Entwickler im VOB armdoc unter /armdoc/..../projekt_cmplan.doc, 2007.

[Pop08] Gunther Popp. *Konfigurationsmanagement mit Subversion, Ant und Maven : Grundlagen für Softwarearchitekten und Entwickler*. dpunkt.-Verl., Heidelberg, 2., aktualisierte edition, 2008.

[Rei94] Christoph Reichenberger. *Konzepte und Verfahren für die Software-Versionsverwaltung*. Schriften der Johannes-Kepler-Universität Linz : Reihe C, Technik und Naturwissenschaften ; 5. Trauner, Linz, 1994. Zugl.: Linz, Univ., Diss.

Glossar

Branch Der Branch ist ein Zweig, auf dem parallel entwickelt werden kann.

ClearCase ClearCase ist eine Versionsverwaltungssoftware von IBM.

cmdata cmdata ist ein von KM entwickeltes Datenformat für die KM-Datenhaltung.

ConfigSpec Die ConfigSpec beschreibt die Sicht auf Objekte in VOBs.

Framework Ein Framework ist ein Rahmen, innerhalb dessen eine Entwicklung erfolgen kann.

IDE IDE ist eine Integrierte Entwicklungsumgebung, wie z. B. das Visual Studio.

Integration Verknüpfung der einzelnen Softwaremodule zum Produkt und dessen Generierung.

Release Das Release ist ein Softwarestand der mit einer externen Version freigegeben wurde und an Kunden verkauft wird..

SIMOTION SIMOTION ist ein Projekt bei Siemens MC RD das Maschinensteuerungen entwickelt.

SINAMICS SINAMICS ist ein Projekt bei Siemens MC RD das Motorsteuerungen entwickelt.

SINUMERIK SINUMERIK ist ein Projekt bei Siemens MC RD das Werkzeugmaschinensteuerungen entwickelt.

Threading Threading bezeichnet in Build Forge die Möglichkeit der parallelen Generierung.

Token Das Token ist ein Hilfsmittel zur Synchronisation zwischen SUN und PC.

Toolsmith der Toolsmith pflegt und Entwickelt die Generierumgebung und Skripts.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Erlangen, 05. August 2010