Bob

Build-Automatisierung für komplexe Embedded-Systeme

TechniSat AUTOMOTIVE

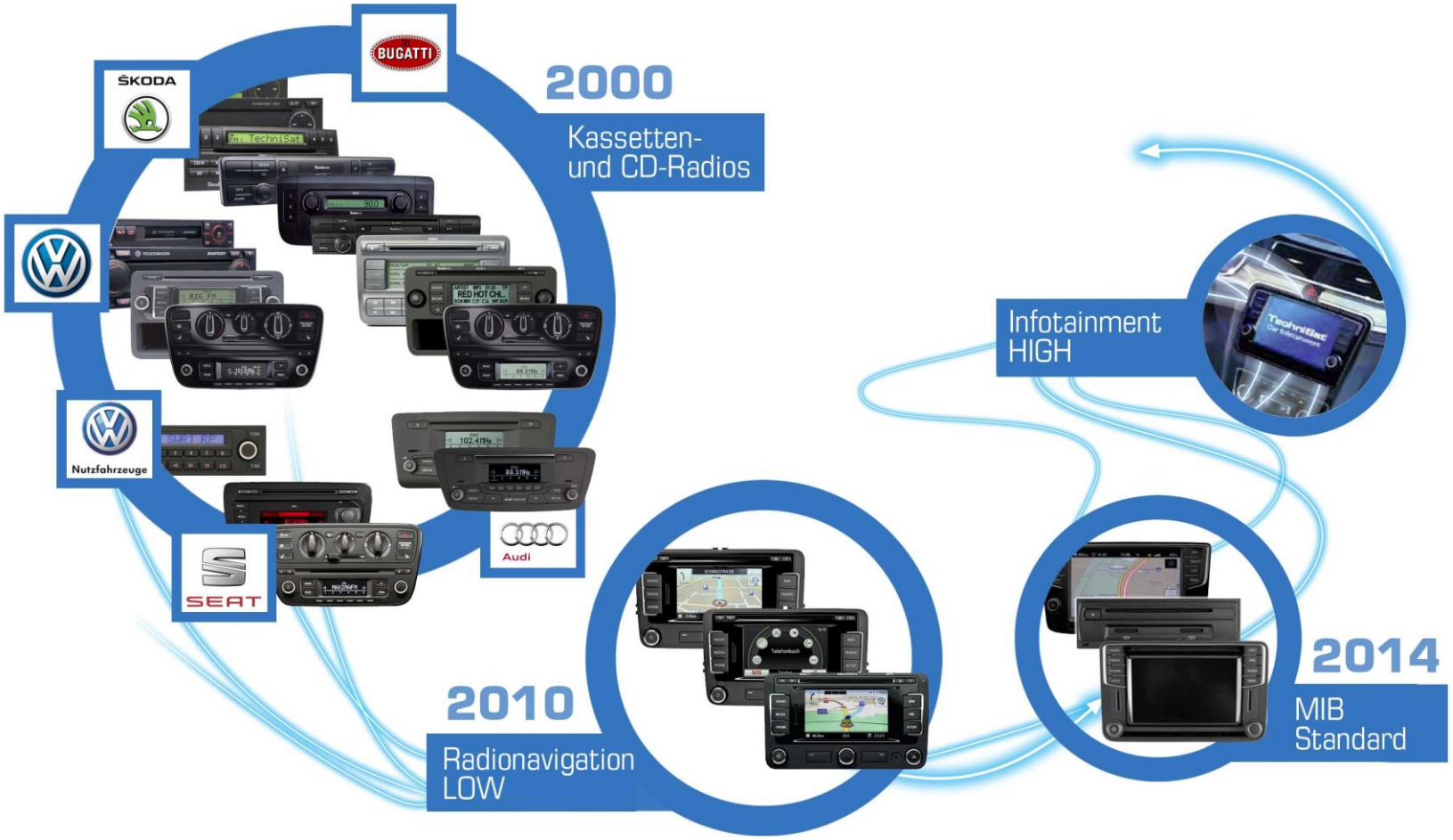
Inhalt

- Einleitung
- Entwicklung von Infotainmentsystemen
- Bob
  - Motivation
  - Theorie
    - Problemabgrenzung
    - Schritte zum Paketbau
    - Variantenhandling
  - Praxis
    - Arbeitsweise
    - Sandboxing
    - Binärartefakte
    - Jenkins
  - Einsatz bei TechniSat
- Zusammenfassung

Einleitung

- TechniSat Automotive
  - 1997 als Business-Unit gegründet
  - Kompetenz von Entwicklung über Test bis zur Produktion
  - Entwicklung von Software, Hardware und Mechanik -> Komplettsystem
  - R&D-Center in Dresden
- Zur Person
  - Jan Klötzke
  - Technischer Leiter der Systemgruppe
  - Verantwortung für das Basissystem (Betriebssystem, Treiber, OSAL)
  - Plus komplexe Querschnittsthemen (Stabilität, Performance)
  - Vorrangig integrationslastige Tätigkeit

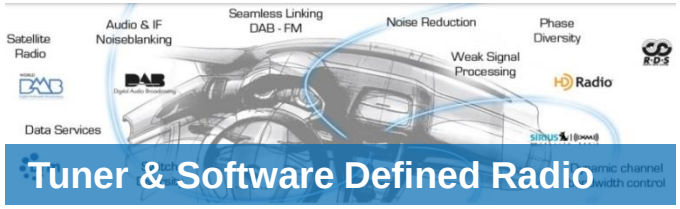
Mehr als 6 Millionen produzierte Einheiten für die VW-Gruppe



TechniSat AUTOMOTIVE

Entwicklung von Infotainmentsystemen - Komplexität

- Hardware
  - SMP-Applikationsprozessoren
  - DSPs, Coprozessoren
  - µC
- Umfang eines aktuellen Projekts
  - Ca. 50 Varianten (HW und SW)
  - Code-Größe
    - Mehrere Millionen Zeilen TechniSat-Code
    - Zulieferungen in Source- und Binär-Form
  - Komplexität
    - Mehrere Betriebssysteme im Gerät
    - Effektiv 3,x Prozessorarchitekturen
- Externe Schnittstellen
  - CAN, MOST, Ethernet, USB, CD, SD/MMC
  - AM/FM/DAB/Sirius, GPS, WLAN, Bluetooth, UMTS



TechniSat AUTOMOTIVE

Entwicklung von Infotainmentsystemen - Entwicklungsprozess

- In Spitzenzeiten mehr als 500 festangestellte und externe Ingenieure
- Am V-Modell orientiertes Entwicklungsmodell (Automotive SPICE)
- Abnahme des Gerätes durch den OEM in mehreren Musterstufen
- Deadline SOP – Start Of Production
  - Hardware ist final für den Produktionszeitraum
  - Freigegebene Software muss vorliegen
  - In der Serie erfolgen nur noch einzelne, genau abgesprochene Fixes
- Softwareerstellung
  - Nightly
  - Regelmäßiger Release-Zyklus für Entwicklungsversionen (wöchentlich)
  - Stable-Branches für SOPs
- Arbeitsweise
  - Entwickler erstellt SW am Arbeitsplatz auf dev-branch
  - Vorintegration übernimmt SW-Änderungen auf Zielbranch(es) und erstellt Tags
  - Integration übernimmt Tags in die Release-SW und Konfiguriert ein Jenkinssystem zum Bau der Release-SW

Entwicklung von Infotainmentsystemen - Integration

- Vom Code zum Image
  - Input
    - Inhouse-Code von über 400 Softwareentwicklern
    - Integration von externen Zulieferern durch unterschiedliche Arbeitsgruppen
  - Steuerung durch Projektleitung der einzelnen Projekte
  - Determinismus
    - Reproduzierbarkeit über viele Jahre gefordert
    - Abgegrenzte Fixes müssen sich im Softwareimage abbilden
      - Im Ergebnis dürfen sich nur betroffene Teile ändern
      - Der Rest muss binär identisch bleiben
- … und zurück
  - Entwicklung: Test von Entwicklungsständen auf aktueller Gerätesoftware
  - Bugfixing
    - Fehler wird in Version X.Y beobachtet
    - Entwickler benötigt genau diesen Stand zu Analyse, Entwicklung und Test

Bob

Bob – Intro

> Are you quite sure that all those bells and whistles, all those wonderful facilities of your so called powerful programming languages, belong to the solution set rather than the problem set?
>
> — Edsger W. Dijkstra

- Build-Tools sind spezielle Skriptsprachen für den Softwarebau
- Hauptproblem zuverlässiger Build-Systeme: Komplexität

Bob – Motivation

- Korrektheit
  - Fallstricke vermeiden
  - Einflüsse sichtbar zu machen
- Reproduzierbarkeit und Nachvollziehbarkeit
  - KISS, DRY
  - Explizite vs. implizite Konfiguration
  - Eine Quelle für alle Build-Backends (z.B. Jenkins)
- Produktivität
  - Geschwindigkeit, insbesondere beim Applikationsentwickler
  - Automatisiertes Variantenhandling
  - Continuous Integration
    - Inkrementeller Bau wo möglich, Clean-Build wo nötig
    - Robuster Bau von Branches, nicht nur von Tags/TGZs
    - Jenkins-Cluster-Builds
- Unterstützung von Linux und Windows als Entwicklungsumgebung

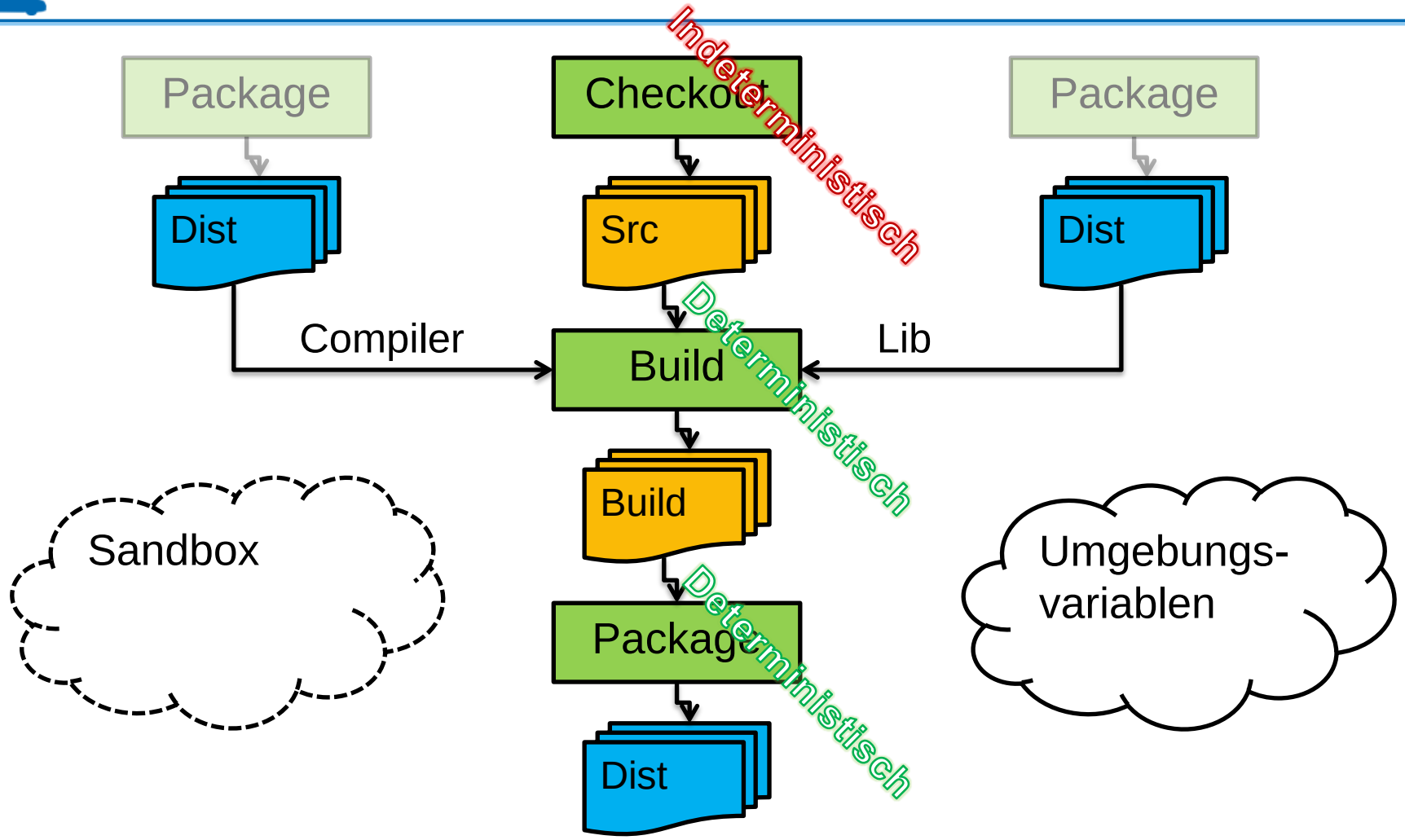
Theorie – Problemabgrenzung

- Reproduzierbares Cross-Compiling ist schwer
  - Versehentliche Verwendung von Host-Dateien
  - Unentdeckte Abhängigkeiten zu anderen Artefakten (z.B. #include – Ketten)
  - Hostkonfiguration von entscheidender Bedeutung
  - Unterschiedliche und exotische Buildsysteme in Zulieferungen
- Eigenschaften existierender Cross-Compiling-Buildtools (Bitbake/Yocto, Buildroot, …)
  - Größtenteils monolithisch (kein Bau auf verteilten Buildservern)
  - Sind nicht für den Einsatz beim Entwickler gedacht
- Eigenschaften traditionelle Paketsysteme (Portage, deb-build, …)
  - Grundlage: gemeinsames Dateisystem
  - Abhängigkeiten stellen nur sicher dass die benötigten Artefakte im Dateisystem vorhanden sind
  - Ort der jeweiligen Artefakte per Konvention bekannt, Akkumulation im Dateisystem → Abhängigkeiten beschreiben nur was mindestens vorhanden sein muss
  - Komplexe Tools mit sehr vielen Freiheitsgraden

Theorie – Funktionales Buildsystem

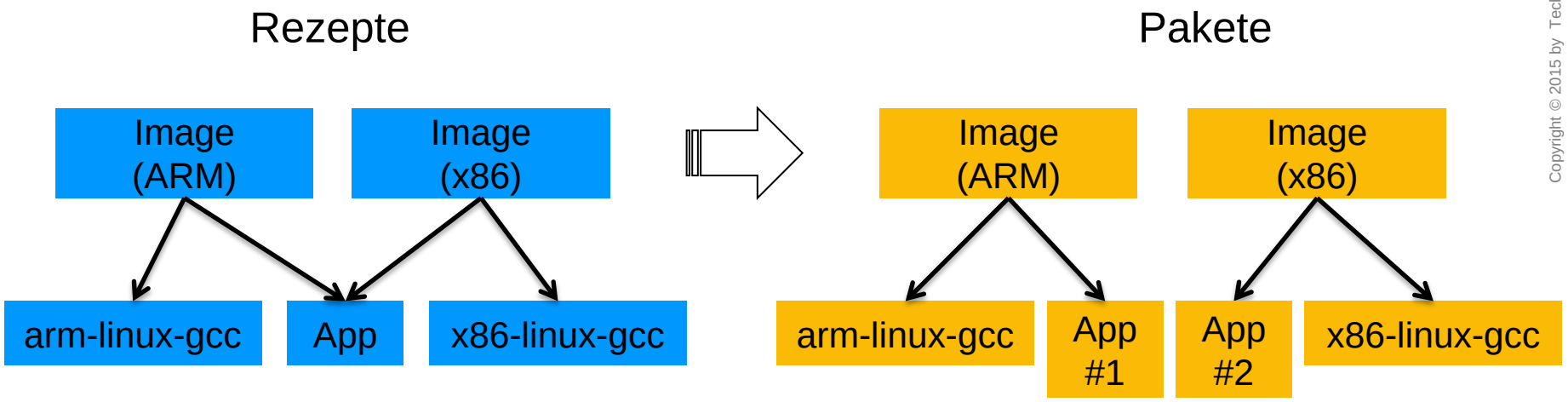
- Deklarative, funktionale „Programmierspache“
  - Lazy evaluation
  - Bob behandelt Pakete als Werte einer rein funktionalen Programmiersprache
  - Wenn man ein Paket baut fragt man Bob effektiv nach dem Wert eines Gleichungssystems
  - Deshalb: in Bob werden keine Pakete installiert sondern „berechnet“
- Beispiel:
  - Programmübersetzung: app = compiler(src)
  - Imagegenerierung: initramfs = cpio(init, busybox, …)
- Ziel: alles was zum Ergebnis beiträgt wird in dem Gleichungssystem abgebildet
  - Quellcode, Compiler, Hilfsprogramme, …
  - Über Sandbox bis hin zu den Coreutils
  - Bob: Gleichung == Rezept
- Grundlage: Kontrollierte Ausführung von Bash-Skripten in separaten Verzeichnissen für jeden Schritt

Theorie - Schritte zum Paketbau



Theorie - Variantenhandling

- Rezepte bilden einen gerichteten, azyklischen Graph
- Das selbe Rezept als Abhängigkeit unter verschiedenen anderen Rezepten
- Unterschiedliche Eingangsgrößen → verschiedene Varianten
  - Umgebungsvariablen, Abhängigkeiten, Tools, Sandbox
  - Automatische Bestimmung wie viele Varianten notwendig sind



Theorie - Rezepte

- Funktionale Sprache zur Beschreibung der Pakete
- Als YAML-Dateien abgelegt
- Ein Rezept beschreibt alles was in den Paketbau einfließt
  - Sourcecode und Abhängigkeiten zu anderen Paketen
  - Checkout-, Build- und Package-Skript
  - Umgebungsvariablen
- Varianten ergeben sich aus den Eingabeparametern

```
checkoutSCM:
   scm: git
   url: git://git.kernel.org/pub/scm/network/ethtool/ethtool.git
   branch: v3.10
buildVars: [CROSS_COMPILE]
buildTools: [make, toolchain]
buildScript: |
   $1/configure --host=$CROSS_COMPILE --prefix=/usr
   make
packageScript: |
   make -C $1 install DESTDIR=$PWD
```

Theorie - Gesamtsoftware

- Alle Rezepte zusammen beschreiben die Gesamtsoftware
  - Diese bilden immer nur den aktuellen Zustand ab (analog zu Sourcecode eines Moduls)
  - Fortentwicklung der Gesamtsoftware über Versionierung der Rezepte abgebildet
- Einstiegspunkt: Root-Rezepte
- Gemeinsame Funktionalität mehrerer Rezepte kann in Klassen ausgelagert werden

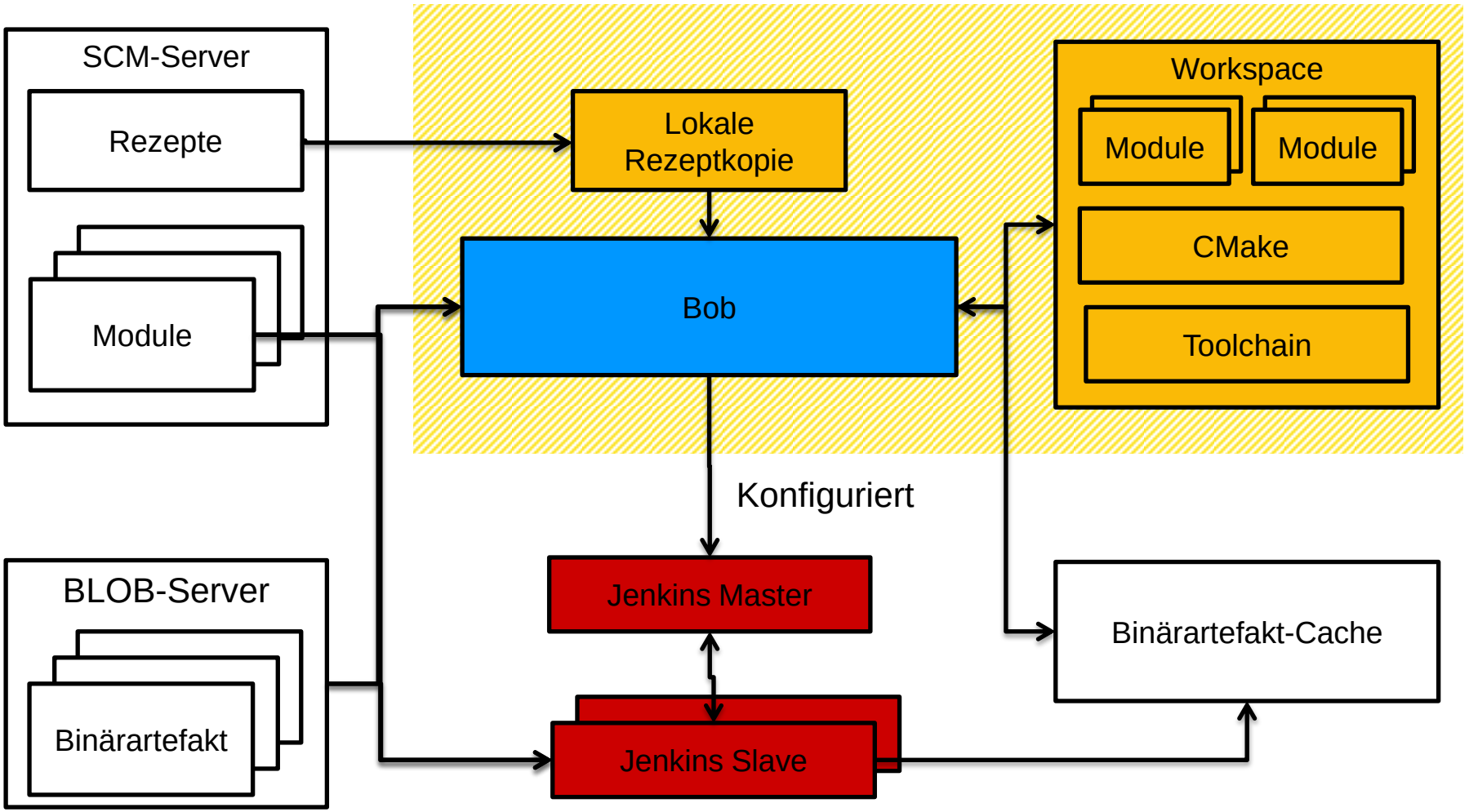
```
bob-tutorials/sandbox/
        classes/
                make.yaml
        recipes/
                toolchain/
                        arm-linux-gnueabihf.yaml
                busybox.yaml
                linux.yaml
                vexpress.yaml
```

SCM

Praxis – Arbeitsweise

- Verschiedene Modi
  - „bob build“ → Release Mode
    - Ziel: Korrektheit
    - Reproduzierbar Releases bauen (Sandbox)
  - „bob dev“ → Development Mode
    - Ziel: Entwicklerperformance
    - Kontinuierliche Arbeitsweise, auch über Rezeptupdates hinweg
  - „bob jenkins“ → Jenkins Mode
    - Ziel: Automatische Jenkins-Konfiguration
    - Continuous Build → Continuous Integration
- md5(Script+Environment+Abhängigkeiten) → Hash-Tree → Variant-Id
  - gleich: erneute Ausführung an vorhergehender Stelle
  - ungleich: neues Verzeichnis für betroffenen Schritt UND alle davon abhängigen Schritte
  - Konservativ, dafür meist korrekt (keine Rücksicht auf alte Skripte nötig)
  - Grundlage für Variantenhandling

Praxis – The Big Picture



TechniSat AUTOMOTIVE

Praxis – Unterschied Release- bzw. Development-Mode

	Release-Mode	Development-Mode
Arbeitsverzeichnisse	Pro Variant-Id ein Verzeichnis	Pro *vorkommende* Variant-Id ein Verzeichnis
Änderungen am Checkout-Step	Durch veränderte Variant-Id Checkout in ein neues Verzeichnis	Inkrementelles Update des vorhandenen Verzeichnisses
Sandbox (Voreinstellung)	Enabled	Disabled
Binärartefakte	Paket direkt herunterladen	Abhängigkeiten herunterladen

Praxis – Sandboxing (Linux only)

- Unbezahlbar um Cross-Compiling-Fehler aufzudecken
- Grundlage für binär identisches bauen der Pakete
- Voreinstellung für Release- und Jenkins-Mode
  - Rootfs wird über ein Sandbox-Rezept bereitgestellt
  - Jeglicher Input für den auszuführenden Schritt ist read-only
  - Nur Ausgabeverzeichnis ist schreibbar
  - Stabile Pfadnamen basierend auf der Variant-Id
- Arbeitsweise
  - Setzt auf Linux User-Namespaces auf → keine Root-Rechte notwendig
  - Ausführung der Schritte in einem eigenen Namespace
  - Bind-Mount der benötigten Verzeichnisse in die Sandbox
  - Verzeichnisse in der Sandbox sind unabhängig vom Host immer gleich
  - Vernachlässigbarer Overhead
- Sandbox ist selbst wieder als Rezept beschrieben
- Build-Host wird bis auf Kernel komplett austauschbar

Praxis – Binärartefakte

- Herausforderung reproduzierbarer Builds
  - Setzt komplett deterministische Pakete voraus (inklusive *aller* Abhängigkeiten)
  - Anderenfalls ist nicht nachvollziehbar wie das Artefakt entstanden ist
  - Erinnerung: Checkout ist nicht immer deterministisch (Bau von Branches, Grundlage für Continuous Build/Integration)
- Bob: Caching von Artefakten, Build-Id als Zugriffsschlüssel
  - Berechnung analog zu Variant-Id
  - Checkout von Tags/Commit-Ids: a-priori berechenbar da deterministisch
  - Indeterministischer Checkout: Hash des Sourcecodebaumes
- Typische Anwendung: Jenkins befüllt Cache, Entwickler lesen davon

```
$ bob build --upload …
$ bob build --download=yes …
$ bob jenkins add --upload …
```

Praxis - Jenkins

- Bob konfiguriert Jenkins über REST-API
- Zirkelschluss möglich
  - Job auf Jenkins welcher Rezepte auscheckt und Bob ausführt
  - Jenkins konfiguriert sich selbst
- Jenkins führt Schritte direkt aus (als Shell-Step)
- Nur wenige Plugins notwendig
  - SVN/Git
  - Multiple SCMs Plugin
  - CopyArtifact Plugin
  - Conditional Buildstep Plugin
- Durch Sandboxing werden Jenkins-Slaves komplett austauschbar

```
$ bob jenkins add --root vexpress --prefix nightly- nightly http://...
$ bob jenkins push nightly
```

Einsatz bei TechniSat

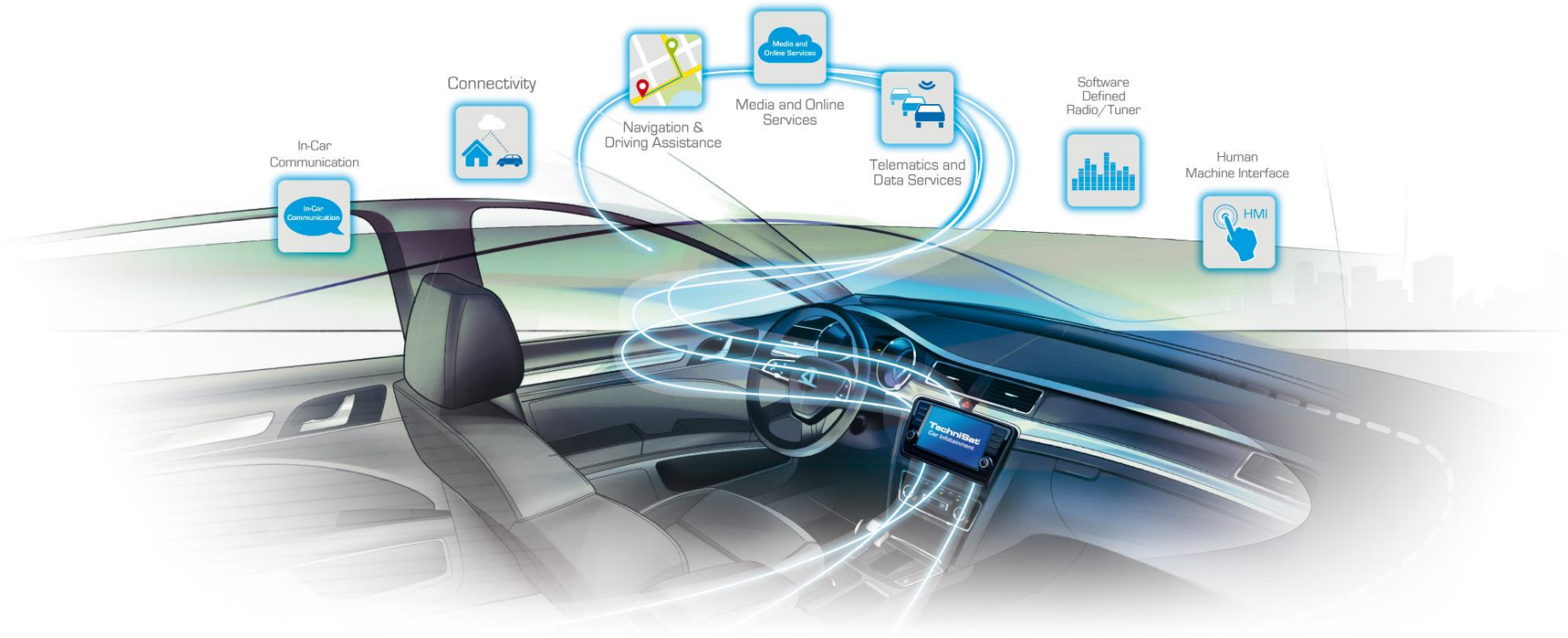
- Ausschließlich in der Vorentwicklung eingesetzt
- Kein Ersatz für Yocto
  - TechniSat-Software und Zulieferungen werden mit Bob gebaut
  - Standard-Komponenten aus Yocto-SDK
- Continous Build mit Jenkins
  - Gut skalierbar über generische Jenkins-Slaves
    - Slaves als Docker-Images verfügbar
    - Durch Sandboxing nur noch ein generisches Image notwendig
  - Rezepte von Arbeitsgruppen betreut
    - Integration durch alle beteiligten Arbeitsgruppen
    - Verteilung der Aufgabe auf viele Schultern
  - Jenkins bekommt Trigger bei Checkin
    - Betroffene Pakete werden inkrementell gebaut
    - Ca. 5 Minuten nach Checkin ist ein fertiges Image verfügbar

Zusammenfassung

- Zielgruppe
  - Komplexe Cross-Compiling-Projekte
  - Zuverlässiger Bau von Software mit vielen Abhängigkeiten
  - Bau von vielen Varianten einer Software
- Binär reproduzierbarer Softwarebau sollte der Standard sein
  - Problem: ist das heruntergeladene Paket wirklich aus den angegebenen Quellen erstellt? → Vertrauensproblem
  - Integrität von Images kann unabhängig nachvollzogen werden (z.B. IoT-Device)
  - Bob ist da eine weiter Lösung

- Ressourcen
  - GitHub: https://github.com/BobBuildTool
  - Manual: http://bob-build-tool.readthedocs.org/en/latest/
  - Mailing-Liste: http://www.freelists.org/list/bob-build-tool

Vielen Dank für Ihre Aufmerksamkeit



Alles aus einer Hand

Umfassende Kompetenz in Entwicklung und Produktion

TechniSat AUTOMOTIVE