



Bernhard Dorninger | Software Competence Center Hagenberg

Experiences with OSGi in industrial applications

Content

- Who we are
- Requirements, Implementation and Experiences
 - Runtime platform for computation algorithms (HA)
 - Middleware for machine control
 - Vehicle sensor data gateway / simulator
- More lessons learnt
- Conclusion

Who we are

- Institution for applied sciences
 - outsourced R&D
 - deliver and prepare product ideas
 - Partner for innovative projects with new technologies
- Founded July 1999 by departments of the Johannes Kepler University Linz
- Integral part of the Softwarepark Hagenberg
- Form of Organisation: Non-Profit Ltd
- ~50 staff members
- Partially founded by the Austrian Government



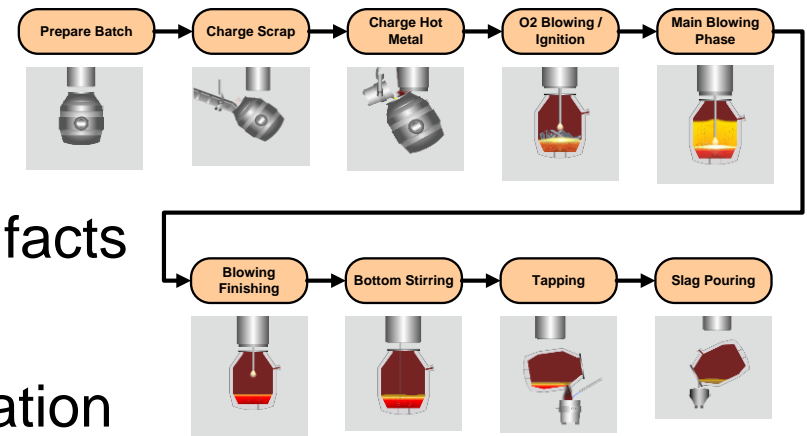
Competence Centers for
Excellent Technologies

Computation Runtime: Context



- Domain Steelmaking
- Complex computational process models (PM)
- PM guiding and controlling various tasks during phases of steelmaking process

- PM date back to the 70s → Typical problems of ageing software
- PMs scattered over IT infrastructure
- Various types of deployment artifacts
- Inconsistent interfaces
- Inaccurate or lacking documentation

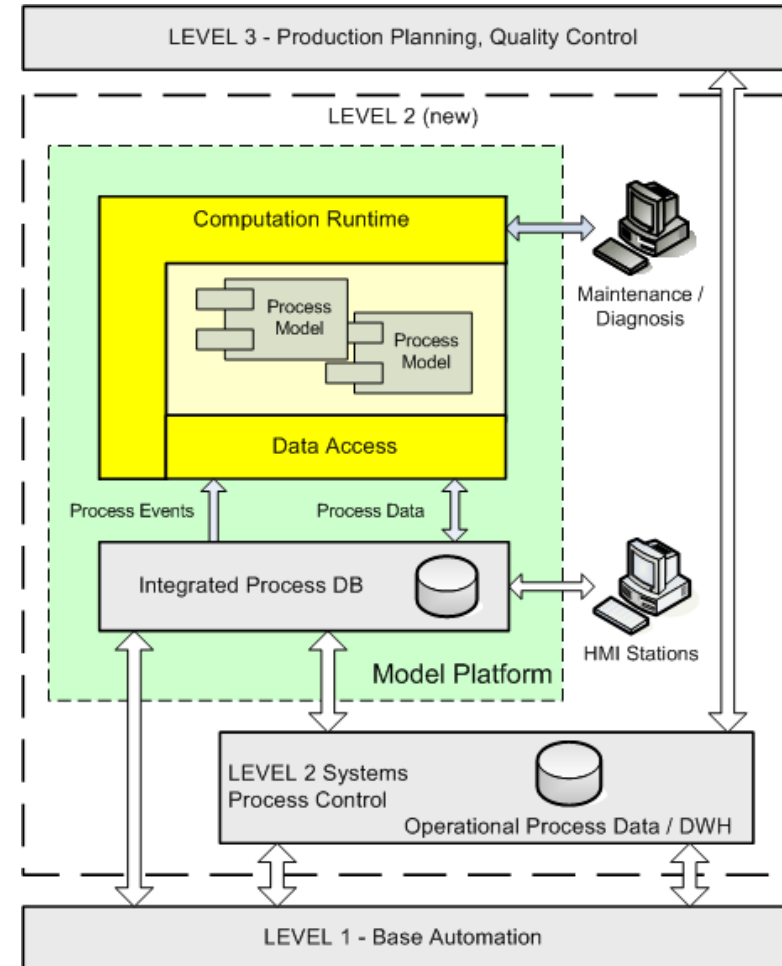


Computation Runtime: Requirements

- Overall project goals
 - Reducing efforts for PM maintenance and testing
 - Shortening PM rollout times
 - Reducing troubleshooting times
- Functional Requirements
 - Pluggable PM components
 - Versioning support
 - Unified data access
 - Legacy code integration
 - Rule based & parallel execution
 - Integration with modern IDE
- NF Requirements
 - High availability (24x7)
 - Fault tolerant
 - Portability (Win, OpenVMS)
 - Preferably based OSS
 - Simplicity for PM maintainers (metallurgists)

Computation Runtime: Implementation

- Located at Level 2 (ISA-95)
- CR acts as container for PM, manages PM instances and their metadata
- executes computations based on simple rules
- Generic data access framework: passive or active data acquisition
- Uses third party process DB



Computation Runtime: Implementation / Experiences

- Implementation based on Eclipse Equinox
 - from V3.2 on sufficiently stable
 - Some issues with OpenVMS → changes in OSGi core and runtime
 - Some infrastructure bundles (CM) from Knopflerfish
- Pluggable PM modules with versions: out of the box
- Legacy code support: there, but doesn't solve JNI stability issues
- Parallel execution with help of Eclipse runtime jobs
- Execution rules: compiled Boolean expressions

Computation Runtime: Stability & HA

- Framework itself proved very stable
- PM bundles might be unstable
 - Unstable native code crashes whole CR
 - launch PMs containing native code in own process
 - Automatic version fallback: “last known good PM version”
 - Computations (or callbacks) may block
 - CR and PM communication must be interruptible any time
- PM bundles might be greedy / evil
 - Restricting PM code via OSGI Permission Management
 - Works for e.g. thread or socket operations
 - Not for excessive memory usage/turnover and CR's SPI usage

Computation Runtime: Stability & HA

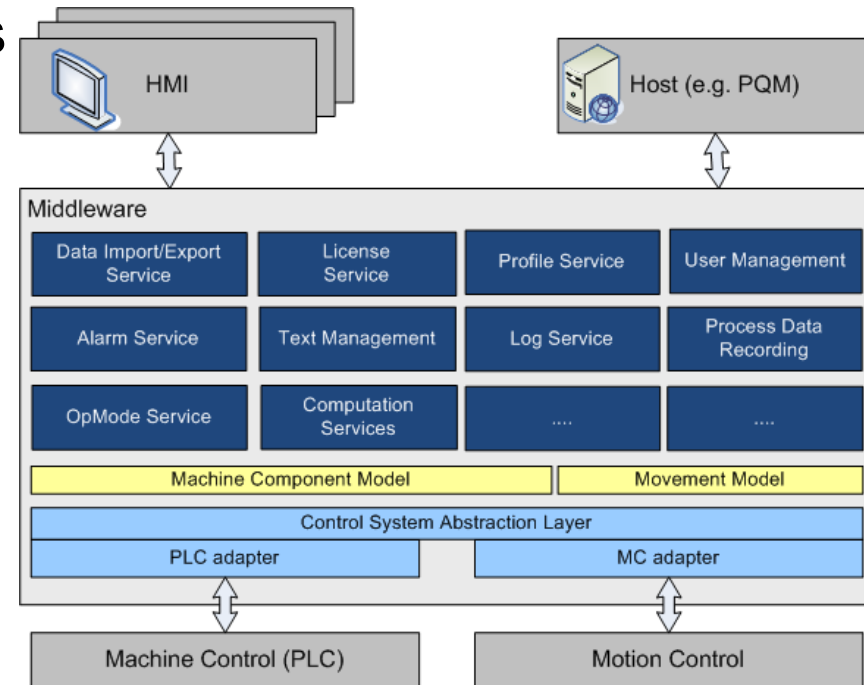
- More measures to achieve High Availability:
 - Full standby system – may be switched within seconds
 - Isolation of experimental PMs in their own runtime.
 - Delayed PM bundle updates: only in certain process phases
 - Blocking PM configuration (via OSGi CM) on operative PMs
- Recovery
 - OSGi framework state
 - additional state information saved via OSGi preferences service

Machine Control MW: Context / Req.

- Domain machine automation
- Middleware for decoupling HMI & other clients from PLCs
- Must support product lines
 - Large number of different (but related) machines
 - Vast variety of feature options for each machine
 - Different vendors of PLCs
- Restricted HW resources (CPU, RAM)
- Dynamic updates of SW during machine operation
- Broad range of functions: read and influence machine state, variables, operations

Machine Control MW: Implementation

- CS adapters as set of bundles
- Machine and Movement model based on Eclipse EMF
- Services realized as set of bundles
- Varying functionality as full bundles or as fragments
- Binding between modules via declarative services
- HMI integrated into MW - directly using OSGi services
 - Coupling remote HMIs via ECF/remote OSGi
 - Other clients (host) coupled via WebServices (Jetty as httpd)

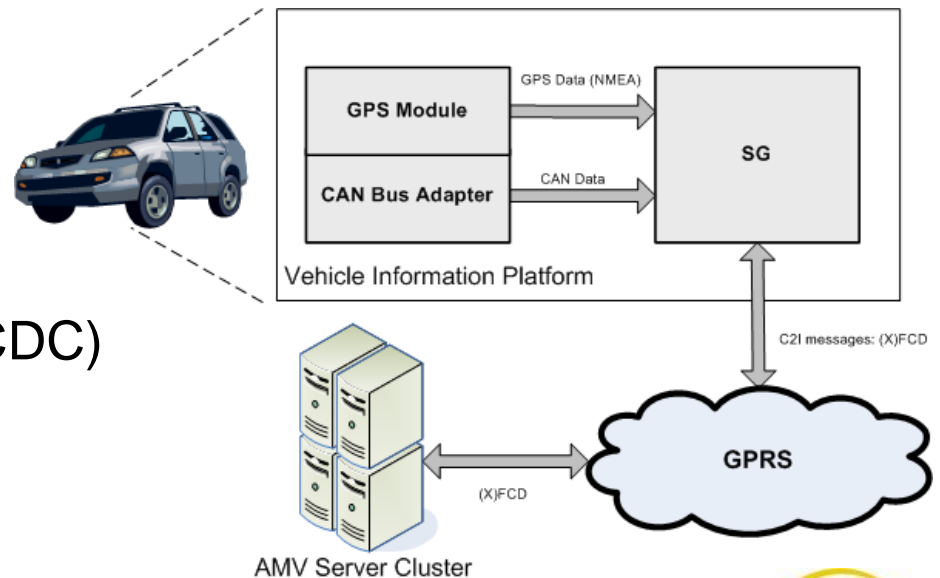


Machine Control MW: Experiences

- Declarative services: Easy to use, dependencies more comprehensible
- ECF/remote OSGi
 - Alternatives examined: dOSGi, RMI based proprietary solution
 - ECF because of support in Eclipse + performance
 - Runs well, only some minor issues: e.g. rOSGi Provider silently fails upon CNFE
 - ECF/rOSGi – EMF: (Eclipse Issue 245014): problems with serializing of EMF objects: workaround via manual externalization
- Machine Data Sync initially with Eclipse CDO
 - Promising functionality, but too much heap dynamics
 - too low performance on target system
 - dependencies bloated code base

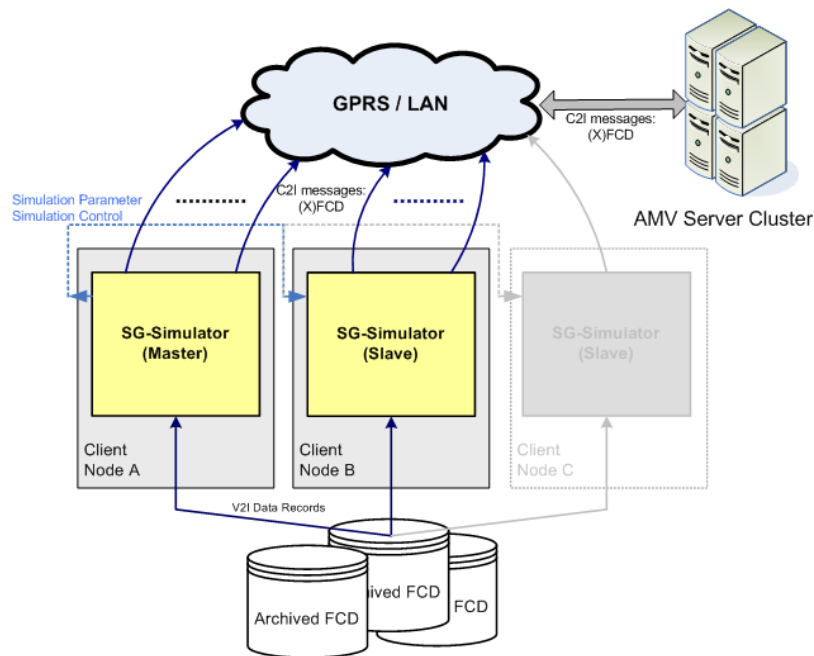
Sensor Data Gateway (SG)

- SW gateway for collection, preprocessing and transmission of vehicle data
- Deployed into car - part of infotainment system
 - Easy update of components during operation
 - Remote diagnosis
 - Configurable data filters
 - Event detection (e.g. emergencies)
 - Limited resources (Java CDC)
 - “self healing” capabilities



SG Simulator

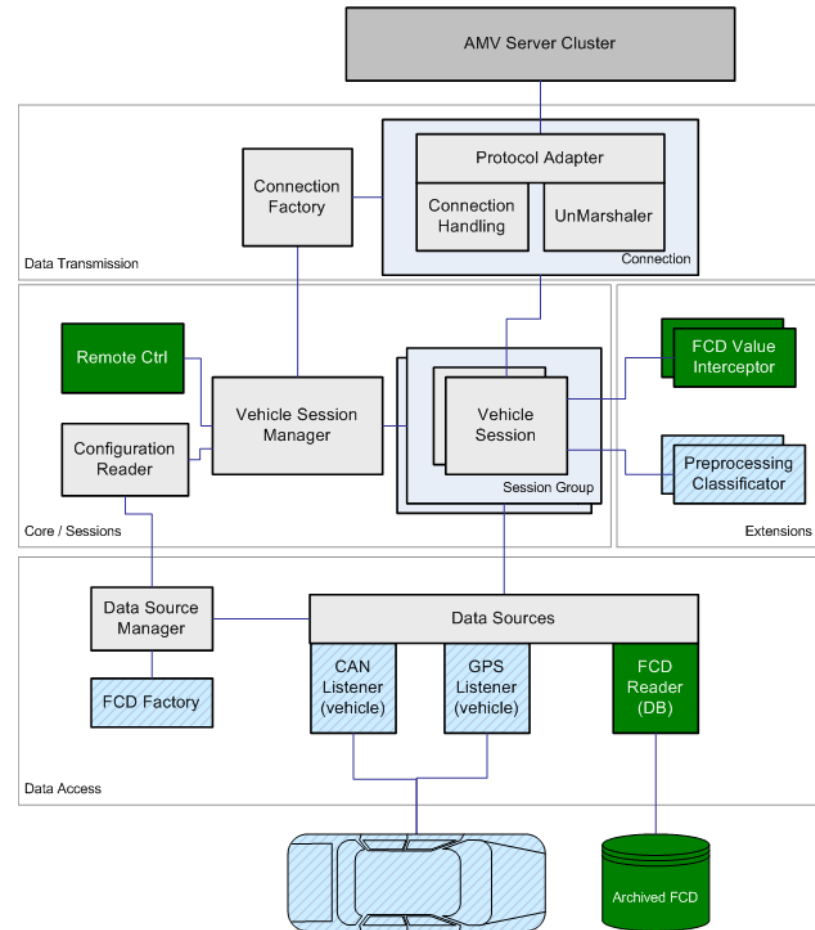
- Distributed simulation of vast amount of vehicles
- Flexible amount of simulator slave nodes
- Playback of prerecorded vehicle data



- Flexible data variance per vehicle session
- Multiple data sources per simulation scenario possible
- Reuses most of SG code base

SG: Architecture / Implementation

- Layered architecture
- Core bundles are common to SG and simulator
- Special functionality for SG and Simulator in optional bundles
 - SG: CAN + GPS Listener, FCDFactory
 - Sim: Remote Control (M/S), FCDReader
- Dynamic extensions
 - SG: Preprocessing Classifiers
 - Sim: Value interceptors → Data variance



SG: Experiences

- Straightforward application customization
 - Different applications with shared core
 - Module structure eases maintenance
- SW updates on running vehicle out of the box
- Sim M/S communication:
 - First prototype used proprietary protocol (DRPC)
 - switching to ECF/remoteServices in the future
- OSGi Execution environments proved helpful
 - SG: Java Mobile CDC
 - Simulator: Java 6

More lessons learnt

- Following best practices pays off
- Granularity of bundles: Encapsulate code in own bundle if
 - can be reused in other context
 - has to be maintained independently
- Separation of API interfaces and implementation bundles:
 - Increases # bundles
 - decouples dependent bundles: allows easy change of implementation
- Bundle dependencies may become very complex
 - Increases with package level dependencies (and version matching)
 - Impossible to handle without tool support

More lessons learnt

- Dynamic nature of OSGi must be taken care of
 - Can't rely on permanent bundle/service availability
 - Possible awkward behavior upon bundle update: e.g. Dependency graph rerouting restarts bundles.
 - Classloading may raise confusing issues esp. combined w. reflection
- Declarative Services help reducing complexity
 - DS for optional or lazily needed services
 - exposed and needed services obvious at development time
- Testing
 - Testing can easily be automated
 - Full test bundles for (partial) integration testing → Continuous Integration
 - Bundle fragments for non public package testing

Conclusion

- We use OSGi mainly for headless, server-like applications
- OSGi applicable for industrial applications
 - Sufficient stability (of implementation of choice)
 - Offers everything for dynamic modifications
 - Rather small footprint
 - Runs in reduced environments
- Some constraints though
 - Not for time critical (Hard RT) applications
 - Cannot eliminate Java's stability risks