Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2018,8
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

An Industry 4.0 Case Study: The Integration of CoCoME and xPPU

Rudolf Biczok, Kiana Busch, Robert Heinrich,
Ralf Reussner

2018

An Industry 4.0 Case Study: The Integration of CoCoME and xPPU

Technical Report

Rudolf Biczok, Kiana Busch, Robert Heinrich, Ralf Reussner

July 5, 2018

Contents

List of Figures

List of Tables

Acknowledgement

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593 and the MWK (Ministry of Science, Research and the Arts Baden-Württemberg) in the funding line Research Seed Capital (RiSC).

1 Introduction

Researches from multiple engineering disciplines study the evolution process of long-term systems through empirical research. Recent contributions are so called community case studies, which aim to provide reference systems for comparable and replicable case studies. The Common Component Modelling Example (CoCoME) [6] is an example for a community case study for information systems. It is specifically designed for collaborative empirical research. CoCoME supports the aspects of evolution that have the highest relevance in the software engineering community. Time horizon (i.e., targeting the run-time or design-time of a system) or supported activities (i.e., targeting requirements engineering, design process, etc.) are examples of evolutional aspects in software engineering. The eXtended Pick and Place Unit (xPPU) [11] is another example of a community case study that is used to research the different evolution cycles in automated production systems.

Modern trends in industry blur the boundaries between pure information systems and automated production systems. As stated by Vogel-Heuser et al. [19], the proportion of software in automated production systems is increasing and the demand for highly customizable production systems will require higher involvement of multiple engineering disciplines. The emerging appearance of Industry 4.0 systems will increase the demand for empirical methods to study evolution cycles in these heterogeneous environments.

There exist case studies about demonstrators that illustrate concepts of Industry 4.0 environments (e.g., the demonstrator myJoghurt [18] or a coffee machine as prototypical Cyber-Physical Systems (CPS) [9]). However, these prototype systems target specific problems where the automation aspect is dominant. The software systems, which utilize the physical parts of these prototypes, do not comprehend the complexity of information system in real world scenarios. Additionally, a community case study is supposed to be a standardized or at least widely used reference for projects with the same research topics. This requires a demonstrator that is not only easily accessible and expendable but also comprehends the most significant aspects of evolution in Industry 4.0 scenarios.

We therefore propose a new community case study for Industry 4.0, which extends the existing community case studies CoCoME and xPPU. This new variant of CoCoME implements common use cases in Industry 4.0 environments, i.e., ordering a customizable product, creating a production plan for a customizable product on multiple abstraction layers, and observing the progress of batch size one productions. We enabled event-based communications between information system and automated production units through a webservice-orientated communication model. It is possible to define and emulate the automated production systems in a web-based frontend, or to integrate a real automated production system into CoCoME variant that is accessible through a REpresentational State Transfer (REST) webservice Application

Programming Interface (API). Researchers, who are interested in Industry 4.0 as research target can use this CoCoME variant as starting point for their empirical work.

The next chapters of this technical report summarize the required background and published prototype systems (see chapter 2) from which we derived repetitive properties of Industry 4.0 systems (as described in chapter 3). Additionally, we give an overview over the implementation details (chapter 4), and guides for typical development tasks in CoCoME itself (chapter 5). In the last chapter (chapter 6) we discuss the overall outlook of the Industry 4.0 variant of CoCoME.

2 Related work

In this project we investigate the evolution of systems from two different domains: software engineering and automation engineering (see Figure 2.1). We define evolution in engineering domains as a continuous process where systems have to adapt over a long period of time to preserve their functionality and maintainability. This corresponds with Lehman's laws [13] of software evolution - the term used to describe the evolution process for software-driven environments (e.g., information systems).

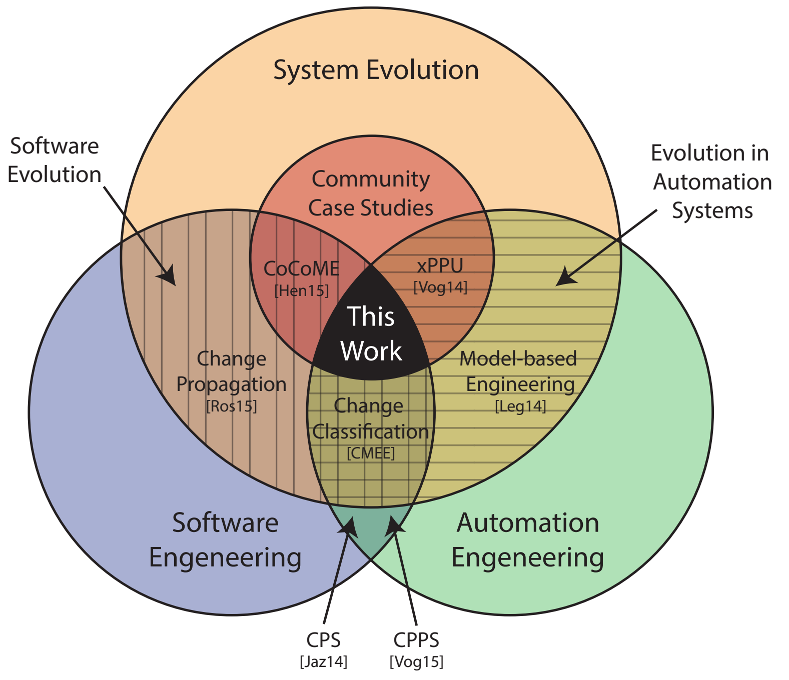


Figure 2.1: Overview of related work

2.1 Empirical approaches and community case studies

There exist several ways to conduct a case study on either information systems or automated production systems. Researchers from the field of software engineering, for instance, analyze source code repositories [4] or conduct maintenance operation on open-source projects [14] to infer relationships between software artifacts and project activities. Case studies in automation technology are conducted in a similar way. For example, by applying model-based engineering methods to laboratory plants [12], designing a joined demonstrator for Cyber-Physical Production Systems (CPPS) [18], or by collecting data from commercial systems [1].

Vogel-Heuser et al. [20] and Heinrich et al. [6] state that comparing or replicating empirical data from case studies can be difficult. Major shortcomings in case studies from both domains are:

Lack of Standardization Different case studies rely on different reference systems (e.g., (open-source) software applications or laboratory plants). Comparing case studies that do not use the same reference system is hard to impossible because of diverse frame condition and project structure [6].

Week Comprehensiveness Especially in the area of software evolution, most case studies cover only few aspects needed for comprehensive empirical work (e.g., documentation of the evaluation process [6]). This makes it difficult to replicate the findings under different conditions.

High Setup & Reproduction Costs Researchers have to replicate results if they want to investigate alternative solutions for already covered problems. If the previously used reference system is not easily accessible, s/he must invest time to setup his own test subject. This is especially an expensive problem in case studies for automation systems, where physical machines must be constructed or altered [20].

These issues are the reason for the development of community case studies[1]. They serves as standardized and comprehensive test subjects to ensure replicability and comparability between other approaches [6].

2.1.1 CoCoME

CoCoME originated from a joint GI-Dagstuhl research seminar [17]. Multiple research groups used CoCoME to apply and evaluate methods for formal component modeling. Since then, CoCoME is part of a collaboration platform with the goal to standardize the procedure of conducting case studies [6].

The CoCoME platform provides multiple **evolution subjects**, which represent different variants of CoCoME (e.g., service-oriented variant [3] or hybrid-cloud variant [7]). Researchers who want to study evolution of specific software architectures can select the CoCoME variant that corresponds with their interests. They then use the appropriate evolution subject for their case study by analyzing the already available project data or by doing their own software engineering tasks. New Findings can then be used for verifying predictions, validation new architectural patterns, or estimating maintenance costs.

For this project we choose the hybrid-cloud variant of CoCoME[2], because of its distributed and webservice-oriented nature [7]. This variant is designed for trading enterprises with cloud-based infrastructures. As it can be seen in Figure 2.2, the hybrid-cloud variant of CoCoME provides software for four subsystems: Enterprises, stores, cash desk lines, and service providers [7]. A **store** ❶ represents the the local place where customers purchase products. The main part of each store is a server holding informations about stocks and available products.

[1] or open case studies, as called by Vogel-Heuser et al. [20]

[2] `https://github.com/cocome-community-case-study/cocome-cloud-jee-platform-migration`

Each store has multiple **cashe desk lines** ❷ which represent a cashier's work place PC including peripheral devices (barcode scanner, cash box, printer). The **enterprise** ❸ subsystem is responsible for managing all connected stores. The **service provider database** ❹ encapsualtes all data relevant for the whole trading enterprise in a cloud-based environment [7, 8].

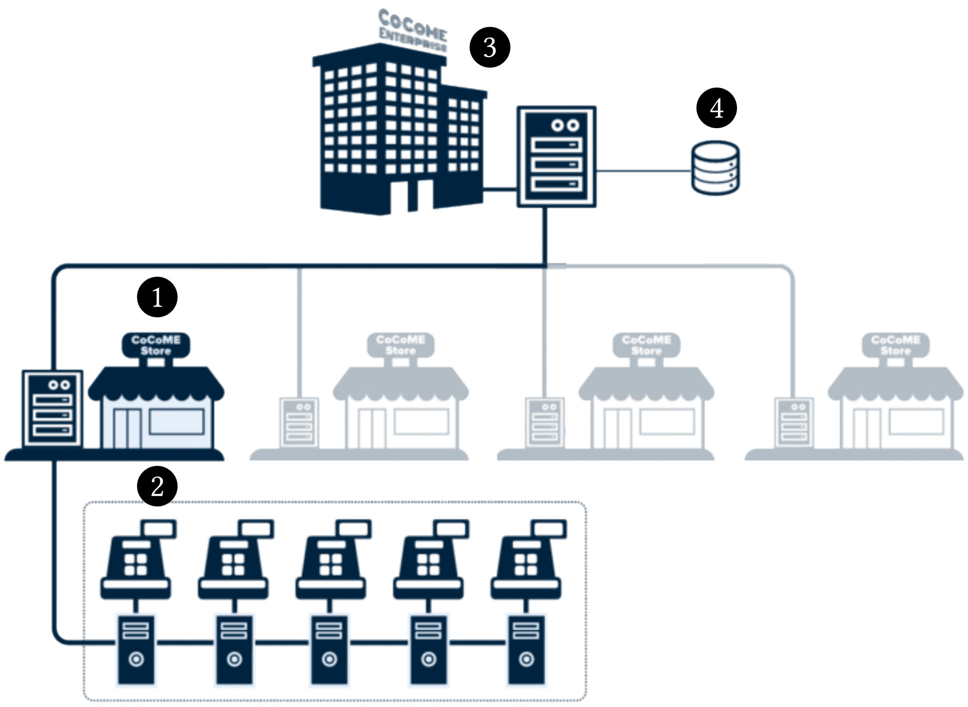


Figure 2.2: Overview of the hybrid-cloud variant of CoCoME [7]

The architecture of CoCoME follows the principles of Component-Based Software Engineering (CBSE). CoCoME is therefore subdivided into readily reusable software components with hierarchical structures. The interface between components is described by general concepts like Plain Old Java Objects (POJO) or higher level language specifications like Enterprise Java Beans (EJB). Figure 2.3 shows a coarse-grained component model of CoCoME:

❶ **TradingSystem::CashDeskLine** contains the business logic for managing cash desk lines (e.g., barcode scanner, display and printer management) [7].

❷ **TradingSystem::Inventory** contains the data structures and business logic used in stores and enterprises (e.g., product definition and provisioning, business reporting) [7].

❸ **WebService::CashDesk** provides public API access from other components through Simple Object Access Protocol (SOAP) webservices [7].

❹ **WebService::Inventory** provides public API access from other components through SOAP webservices[7].

❺ **WebFrontend::UseCases** holds web-based front ends for all subsystems [7].

❻ **ServiceAdapter** database abstraction through RESTful webservices. It allows an entrepreneur to outsource the inventory data to cloud providers [7].

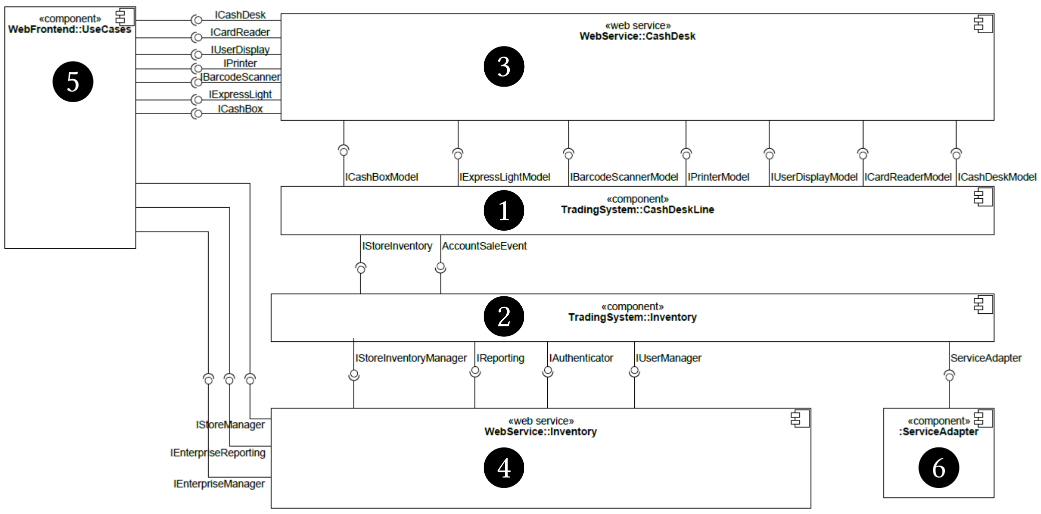


Figure 2.3: Simplified component model of the hybrid-cloud variant of CoCoME [7]

2.1.2 xPPU

In order to study evolutionary aspects of an automated production system, the Institute of Automation and Information Systems (AIS) at TU München developed the demonstrator xPPU [11]. It reassembles a manufacturing unit that is able to pick and categorize materials with different properties (e.g., color, weight). Hence, it incorporates most commonly used physical components available in commercial automation systems without becoming too complex [11].

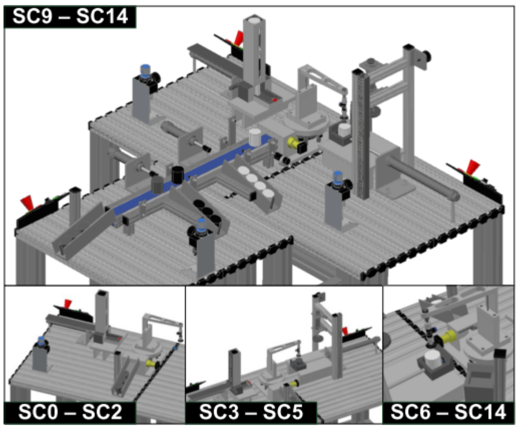


Figure 2.4: Image of the xPPU [11]

Similar to a reference system in software evolution, xPPU can be used to research the impacts of certain evolution scenarios[3]. At first, researchers select a variant of xPPU as research subject. Figure 2.4 illustrates the variant from scenario SC0 that consists only of a crane, a ramp, and a stack. They then apply the changes needed to realize the evolution scenario. Scenario SC1 [20], for instance, is the result of increasing capacity of the output storage in SC0 [20]. Table 2.1 shows an excerpt of several evolution scenarios conducted on xPPU with their corresponding impacts. These impacts are classified by the affected part (e.g., ramp, crane, etc.) and the involved engineering discipline (i.g. Mechanic parts, electronics, or embedded software). Vogel-Heuser et al. analyzed more than 13 scenarios [20].

[3]also called evolution steps by Vogel-Heuser et al. [20]

Cause of evolution	Scenario	Scope of change											
		Stack			Crane			Stamp			Ramp		
		M	E	S	M	E	S	M	E	S	M	E	S
increasing transportation throughput	SC0	A	A	A	A	A	A	-	-	-	A	A	A
increasing capacity of output storage	SC1	o	o	o	o	o	o	-	-	-	M	o	o
two different products are processed	SC2	A	A	A	o	o	o	-	-	-	o	o	o
one product has to be labeled	SC3	o	o	o	M	M	M	A	A	A	o	o	o
...			...			...			...			...	

Legend: A: Added; M: Modified; -: not included in scenario; o: no Changes; M: Mechanics; E: Electronics; S: Software

Table 2.1: Evolution scenarios and their impact on xPPU (table content from Legat et al. [11])

2.2 Contributions in Industry 4.0

There exists no common definition of an Industry 4.0 system. Instead, we found papers presenting structural and functional characteristics of such systems. Lee et al. , for instance, propose a 5-level architecture for CPS in Industry 4.0 environments [10]. In that architecture, a CPS can be classified by its structural depth and responsibility (e.g., level 1 for systems that simply interconnects machines and computing devices for data exchange or level 5 for self-adaptive, intelligent systems). Other contributions such as the work of N. Jazdi [9] illustrate certain aspects of Industry 4.0 systems on real prototype systems with low complexity. Such aspects are ❶ smart networking, ❷ mobility, ❸ flexibility, ❹ integration of customers, and ❺ new innovative business models [9].

2.2.1 MyJoghurt

The demonstrator MyJoghurt is another case study reassembling an agent-oriented, CPPS for Industry 4.0 scenarios [18]. Its architecture focuses on the aspects mobility, flexibility, and integration of customers. Customers order their individual yoghurt through a smart device app (mobility aspect) where they enters the desired ingredients and packaging options (integration of customers aspect, as seen in Figure 2.5) [15].

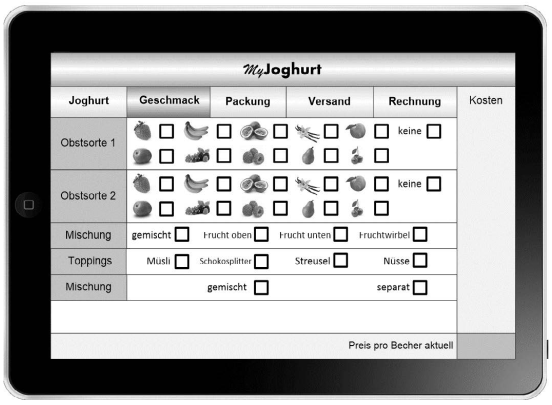


Figure 2.5: Tablet app in MyJoghurt [15]

For each submitted order, MyJoghurt then uses an automatic discovery mechanism to create a prediction plan based on a fixed production template.

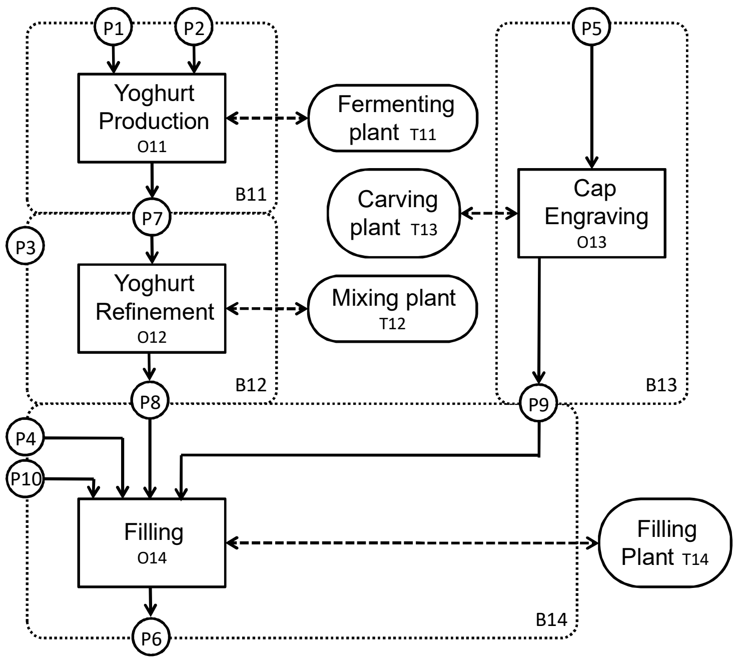


Figure 2.6: Production plan in MyJoghurt. Shapes with sharp corners represent necessary production steps. Shapes with round corners represent operations resolved by the discovery system of MyJoghurt [18] .

However, MyJoghurt does not include other subsystems except production plants. In a modern Industry 4.0 environment, we expect to see multiple involved system types (e.g., information and automated production systems).

3 Evolution Scenario

In the previous section, we presented functional and structural aspects of Industry 4.0 systems and how a subset of them are implemented in demonstrator systems like MyJoghurt. We now use those findings for the following chapter to formulate a new evolution scenario for the hybrid cloud variant of CoCoME. In other words, we describe the structural modifications on CoCoME and by extension any traditional enterprise needed to transform the hybrid cloud system to a level 5 Industry 4.0 system (according to the 5C architecture by Lee et al. [10]). Additionally, we present details about the communication model for processing batch size one productions and the domain-specific language needed to execute low-level production tasks. These three points are required to realize the use cases and the functional properties of Industry 4.0 systems (as described by Jazdi [9]) in CoCoME.

3.1 Structural changes

The main difference between the hybrid could variant and the Industry 4.0 variant is the introduction of plants as additional subsystems (as seen in Figure 3.1). Plants represent autonomous production sites, which offer low-level production operations as a service to an enterprise and schedule these operations across a heterogeneous set of production units. Each production unit is accessible from the local network of a plant and offers a RESTful interface for state information retrieval and command execution. Webservice endpoints for production units have the advantage that they can hide machine-specific complexity through a generic (RESTful) web API. For the integration we used one of the xPPU models, which also includes an implementation of a web API on a Raspberry Pi [2]. Researchers have also the option to define custom production units and simulate their execution within the Industry 4.0 variant. The separation between high-level production management (done by the enterprise subsystem) and low-level production management (done by the plant subsystem) also complies with the 5C architecture, as illustrated in Table 3.2.

3.2 Communication model

As we saw in existing prototype systems like MyJoghurt, customer involvement through highly personalize products plays a major role in Industry 4.0. However, translating orders for personalized products into a production cycle is a challenging task. On the one hand, product designers wish to define production templates that consist of easily combinable and reusable production steps. On the other hand, each type of production unit offers different set

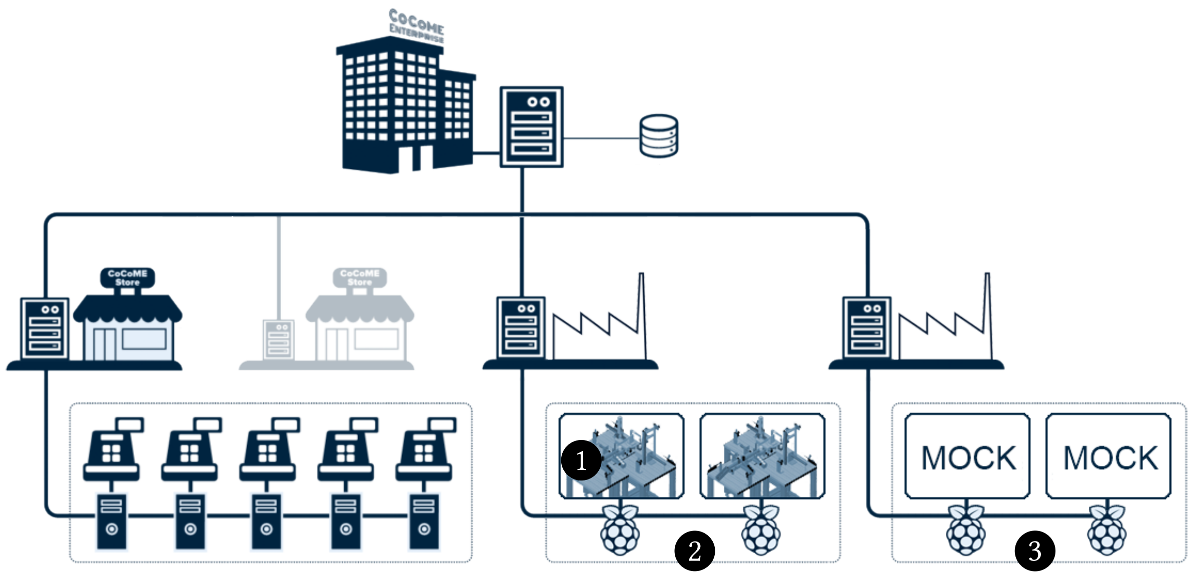
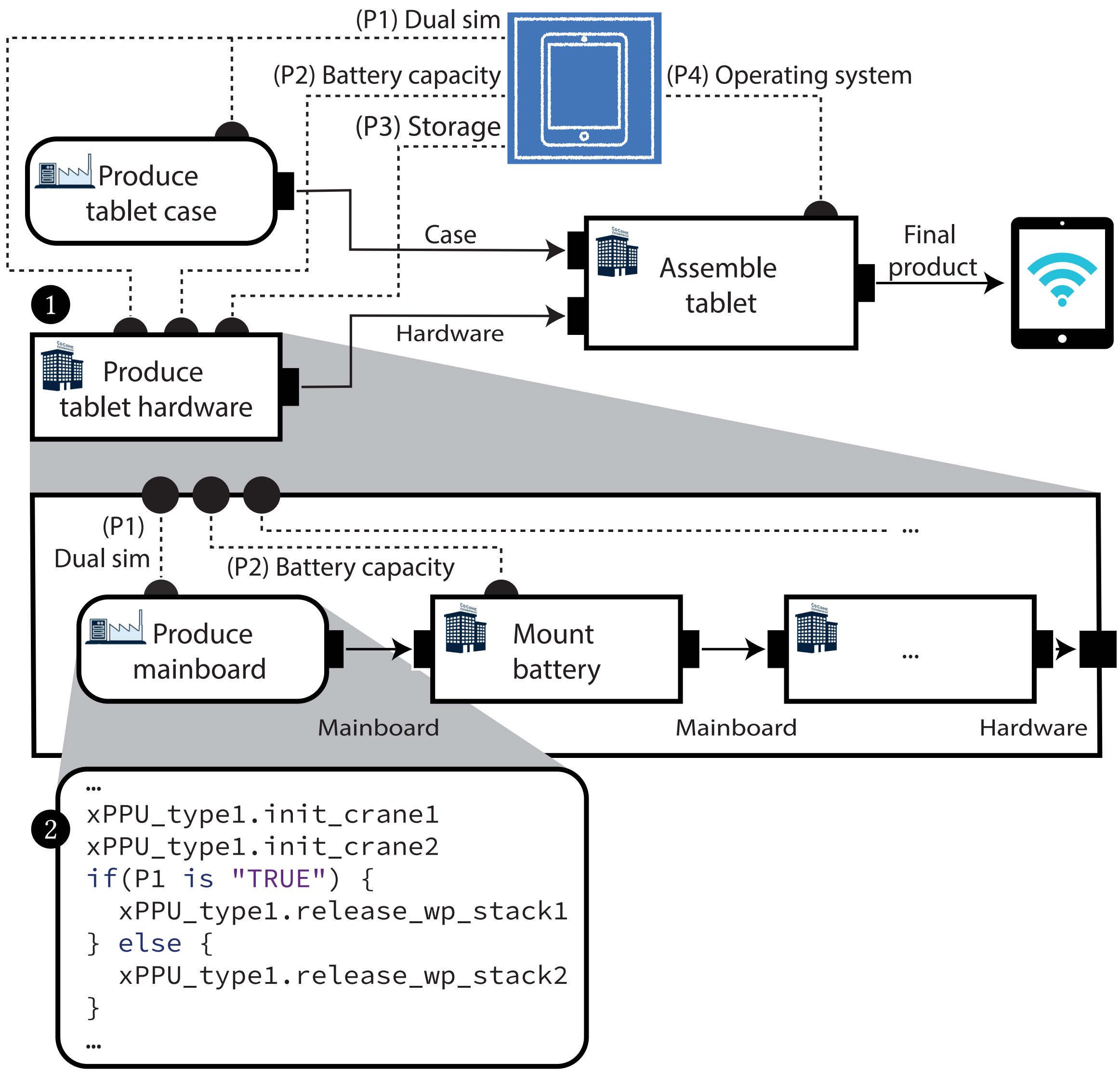


Figure 3.1: Overview of Industry 4.0 variant of CoCoME: Each plant 1 is responsible for low-level production tasks and multiple production units. These production units are either remotely connected through a RESTful web API 2 or run as virtualized unit 3.

of low-level instructions that require a profound knowledge in automation technology and the unit type itself. In case of xPPU, for instance, an automation engineer must know how to control each of the subparts of xPPU in order to utilize the production device (e.g., initializing crane upon first use, or moving crane to stack to pick up work pieces).

We therefore propose two types of template definition: Plant operations and production recipes. The plant operations are templates that utilize different sets of production unit instructions (cf. [2]). This allows an automation engineer to hide the complexity of the production units when providing atomic, general-purpose production steps as a service. The product designer of an enterprise can then use the plant operations to define production recipes. The purpose of production recipes is to encapsulate multiple plant operations into more coarse-grained and reusable services or to define the entire production procedure for a customizable product. Both plant operations and production recipes can be parameterized to make the production procedure more configurable by the customer. In addition, every plant operation and production recipe offer (typed) input and output ports for work pieces that are consumed or produced during the operation execution. Figure 3.2 illustrates how plant operations and (nested) production recipes can be combined by a product designer to describe the production of a tablet.



```
...
xPPU_type1.init_crane1
xPPU_type1.init_crane2
if(P1 is "TRUE") {
  xPPU_type1.release_wp_stack1
} else {
  xPPU_type1.release_wp_stack2
}
...
```

Figure 3.2: Example of a production recipe for a tablet as directed acyclic graph. The production recipe consists of sub-recipes (rectangles) and plant operations (rectangles with rounded corners). Solid lines represent the (typed) material flow between each production step. Dashed lines symbolize the parameter mappings between the product and the various recipe steps. ❶ shows the production recipe of the high-level recipe step "Produce tablet hardware". In ❷ we can see the implementation of plant operation [2] "Produce mainboard" in a DSL.

3.3 DSL for cyber-physical production systems

As hinted in the section below, coordinating production units to fulfill a specific task requires much domain knowledge and is hard to generalize. Therefore, it is important to provide an automation engineers with a Domain-Specific Language (DSL), where they can specify all production unit instructions and dynamic elements needed to realize a specific plant operation. The DSL we propose consists of two different type of expressions:

Expression	**Function**
`PUType.inst`	Execute instruction `inst` on a production unit of type `PUType`
`if(Param == "TEST_VALUE") {` `expressionList1` `} else {` `expressionList2` `}`	Test if parameter `Param` has the exact value `TEST_VALUE`. If true, execute `expressionList1`, else execute `expressionList2`

Table 3.1: Expression available in the DSL

These expressions are sufficient enough to perform different sets of instructions based on the parameter values passed by a customer (as seen in the example in Figure 3.2).

3.4 Use cases and roles

This evolution scenario covers one primary use case: Ordering personalized products as a customer. As we can see in Figure 3.3, the process of fulfilling a customer's order involves use cases of all other subsystems (i.e., store, enterprise, and plant). At first, when customers submit personalized orders to their local store, the store subsystem delegates the order to its associated enterprise. Then the enterprise resolves the production recipe associated with the ordered product and schedules the production. This includes requesting services from plants (i.e., plant operations) or other enterprises (i.e., production recipes), and proceeding with other production tasks while waiting for the corresponding plant / enterprise to deliver its service. When a plant receives a plant operation order from an enterprise, a batch system on each plant site must schedule necessary production unit instructions accordingly to the plant operation template (cf. [2]).

For the enterprise subsystem and the plant subsystem, we need special user roles for defining both the appropriate production recipes and plant operations. In this scenario, we assigned the existing enterprise manager role from the hybrid cloud variant with the task of defining product recipe. An enterprise manager in CoCoME can, for instance, be seen as product designer or entrepreneur. The plant manager is a new role in the Industry 4.0 variant of CoCoME. This role reassembles an automation engineer who is responsible for creating and maintaining plant operation templates.

In summary, we can use the work by Jazdi [9] and Lee et al. [10] to categorize the sub use cases in a structural way (as seen in Table 3.2).

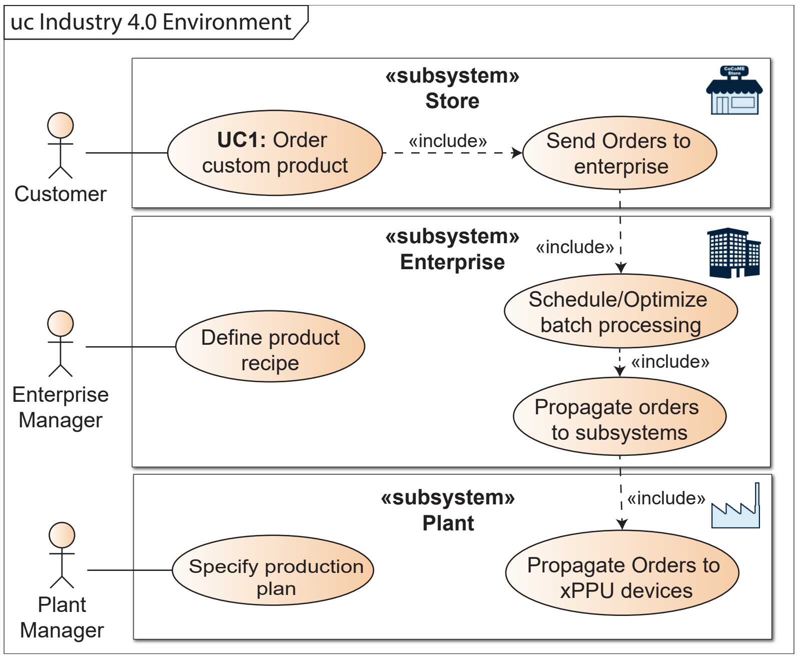


Figure 3.3: Use Cases in the Industry 4.0 variant of CoCoME

5C level	Subsystem	Function	Industry 4.0 Feature
Configure	Store	Product order customization	Integration of customers
	Store / Enterprise	Monitoring the production order	Flexibility
Cognition	Enterprise	Product template and product recipe definition	New business models
		Production scheduling and surveillance based on production recipe	Flexibility
Cyber	Plant	Plant operation scheduling and surveillance based on DSL	Smart networking
Conversion	Production Unit (RESTful service agent)	Submission system for device operations, encapsulated state machine, self-observation	Direct system extension
Connection	Production Unit (mechanical device)	Interaction with production unit sensors through OPC UA [5]	Smart networking

Table 3.2: Overview of functional and structural properties of the Industry 4.0 variant based on the structural categorization by Jazdi [9] and the functional categorization by Lee et al. [10]

4 Implementation

This chapter showcases the implementation details about the batch size one production scheduling, which includes both the production order scheduling on the enterprise level and plant order scheduling on the plant level. Beyond that, we present an overview of altered and added components in the component model of CoCoME. In the last part of this chapter, we describe the RESTful interface provided by the AIS and how it is integrated in CoCoME.

4.1 Component model

The transformation of the hybrid-cloud variant to an industry 4.0 system is a complex task affecting all six major components of CoCoME [7], as seen in Figure 4.1:

1. **TradingSystem::CashDeskLine** The cash desk subsystem now provides the customer with an additional product configuration display, represented by the **:Cunfigurator** subcomponent. A customer activates the configurator display upon typing the barcode of a custom product into the **:BarcodeScanner**. Other subcomponents remain unchanged.
2. **TradingSystem::Inventory** This component houses the business logic for the batch size one production mechanisms and newly added data structures. The **:Application::Production** subcomponent is responsible for the high-level production order scheduling on an enterprise server. The **:Application::Plant** subcomponent encapsulates all functionalities of a plant server. These functionalities include the low-level plant order scheduling, creating or importing production unit types, and interfacing or virtualizing production unit instances. Corresponding data structures for describing production units, production unit types, plant properties, and the plant operation DSL reside in the **:Data::Plant** subcomponent. Data structures for custom product definitions, production operation ordering, plant operation ordering, and the production recipe DSL reside in the existing **:Data::Enterprise** subcomponent. In the **:Data::Store** component, we added data types for managing custom products as store items. CoCoME uses an internal Comma Seperated Values (CSV) format to exchange database records between **:Data::Persistence** and **ServiceAdapter**. This made it necessary to implement routines for converting between Java runtime objects and CSV records for each altered or additional data class. **:Application::Reporting** and **:Application::ProductDispatcher** are unaffected.
3. **WebService::CashDesk** We added a SOAP webservice in **:ConfiguratorService** alongside to the **TradingSystem::CashDeskLine::Configurator** component. Other subcomponents remain unchanged.

❹ **WebService::Inventory** We added the subcomponent **:Plant** to expose the plant server business logic in **TradingSystem::Inventory::Application::Plant** as SOAP webservice. Additionally, we added webservice calls in **:Store** and **:Enterprise** to support event-based messaging between subsystems, and to expose Create, Read, Update, Delete (CRUD) operations for the new data structures in **WebService::Inventory::Data**. The **:Reporting** webservice remains unchanged.

❺ **WebFrontend::Web** In **:PlantView**, we created web-based views for maintaining production unit classes, production units, and plant operation recipes for the plant manager role. For the enterprise manager role, we added views in **:EnterpriseView** for creating and editing production order recipes, plants, and custom products. Finally the **:StoreView** offers additional web controls for configuring custom products from a cash desk, a view for managing custom products inside a store stock, and a view for observing the status of an ordered custom product. The components **PlantConnector** and **PlantData** operate in the background and provide other frontend subcomponents with plant-related CRUD operations.

❻ **ServiceAdapter** Modifications or additions in **TradingSystem::Inventory::Data** are also reflected in this component. Every new data structure needs a corresponding Java Persistence API (JPA) entity and a set of RESTful webservices to provide appropriate CRUD operations. In addition, we extended the internal data exchange mechanism to support polymorphic queries. This allows other components to use the base type for querying certain records inside the cloud storage, instead of querying each subtype separately (e.g., SELECT * FROM PRODUCT also returns all custom products)

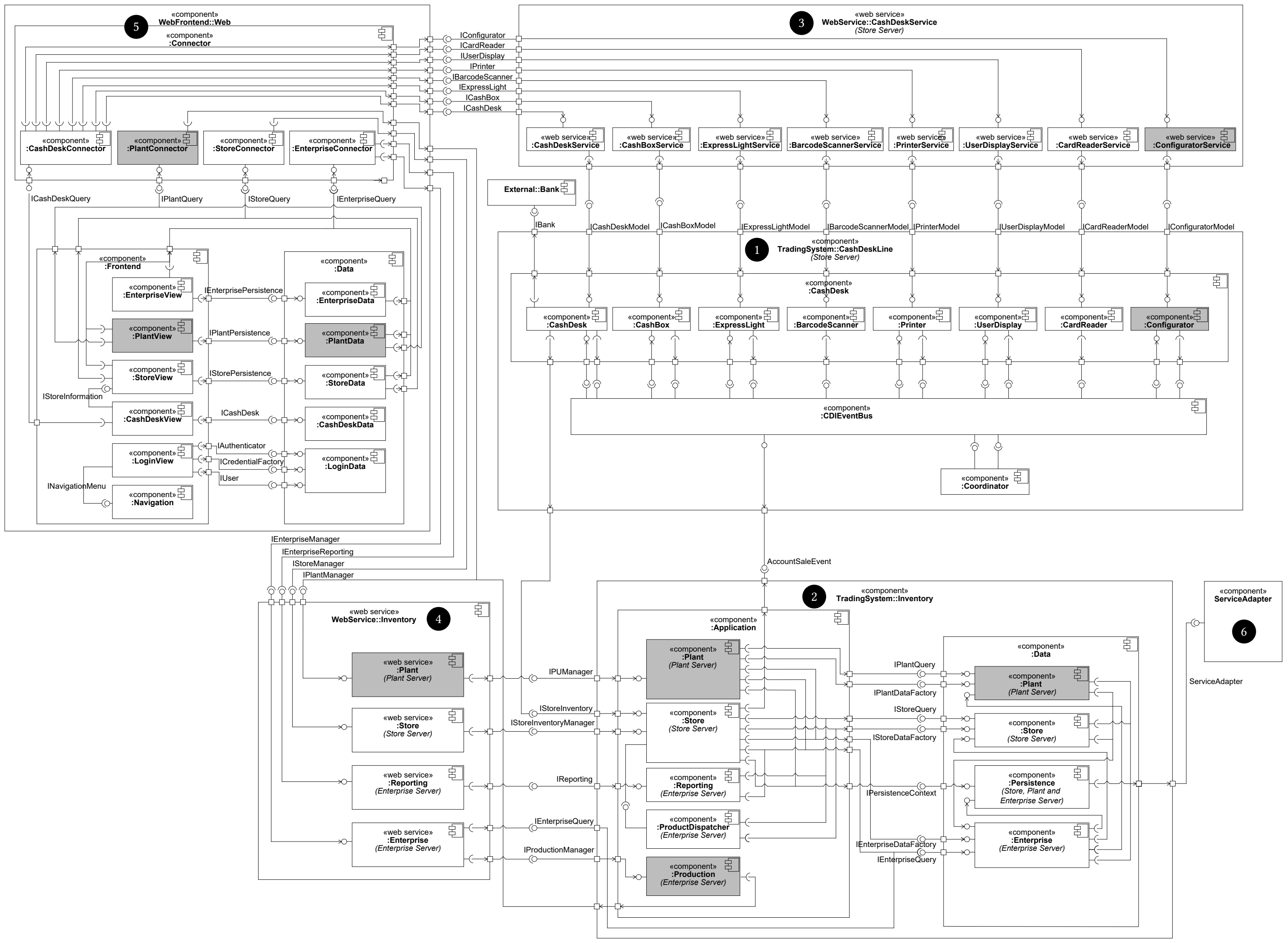
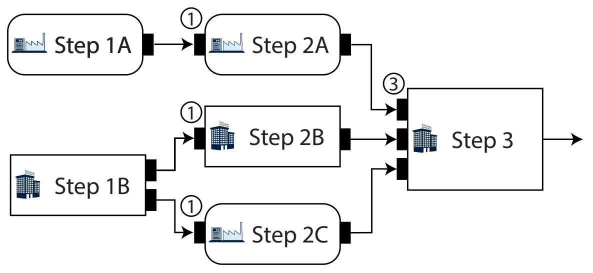

«component» WebFrontend::Web
«component» :Connector
«component» :CashDeskConnector
«component» :PlantConnector
«component» :StoreConnector
«component» :EnterpriseConnector
ICashDeskQuery
IPlantQuery
IStoreQuery
IEnterpriseQuery
«component» :Frontend
«component» :EnterpriseView
«component» :PlantView
«component» :StoreView
«component» :CashDeskView
«component» :LoginView
«component» :Navigation
IStoreInformation
INavigationMenu
«component» :Data
«component» :EnterpriseData
«component» :PlantData
«component» :StoreData
«component» :CashDeskData
«component» :LoginData
IEnterprisePersistence
IPlantPersistence
IStorePersistence
ICashDesk
IAuthenticator
ICredentialFactory
IUser
«web service» WebService::CashDeskService (Store Server)
IConfigurator
ICardReader
IUserDisplay
IPrinter
IBarcodeScanner
IExpressLight
ICashBox
ICashDesk
«web service» :CashDeskService
«web service» :CashBoxService
«web service» :ExpressLightService
«web service» :BarcodeScannerService
«web service» :PrinterService
«web service» :UserDisplayService
«web service» :CardReaderService
«web service» :ConfiguratorService
External::Bank
IBank
ICashDeskModel
ICashBoxModel
IExpressLightModel
IBarcodeScannerModel
IPrinterModel
IUserDisplayModel
ICardReaderModel
IConfiguratorModel
«component» TradingSystem::CashDeskLine (Store Server)
«component» :CashDesk
«component» :CashDesk
«component» :CashBox
«component» :ExpressLight
«component» :BarcodeScanner
«component» :Printer
«component» :UserDisplay
«component» :CardReader
«component» :Configurator
«component» :CDIEventBus
«component» :Coordinator
AccountSaleEvent
IEnterpriseManager
IEnterpriseReporting
IStoreManager
IPlantManager
«web service» WebService::Inventory
«web service» :Plant (Plant Server)
«web service» :Store (Store Server)
«web service» :Reporting (Enterprise Server)
«web service» :Enterprise (Enterprise Server)
IPUManager
IStoreInventory
IStoreInventoryManager
IReporting
IEnterpriseQuery
IProductionManager
«component» TradingSystem::Inventory
«component» :Application
«component» :Plant (Plant Server)
«component» :Store (Store Server)
«component» :Reporting (Enterprise Server)
«component» :ProductDispatcher (Enterprise Server)
«component» :Production (Enterprise Server)
IPlantQuery
IPlantDataFactory
IStoreQuery
IStoreDataFactory
IPersistenceContext
IEnterpriseDataFactory
IEnterpriseQuery
«component» :Data
«component» :Plant (Plant Server)
«component» :Store (Store Server)
«component» :Persistence (Store, Plant and Enterprise Server)
«component» :Enterprise (Enterprise Server)
«component» ServiceAdapter
ServiceAdapter
1
2
3
4
5
6

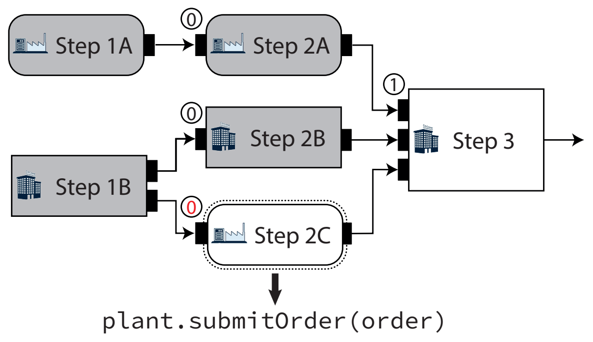
Figure 4.1: Component model of the Industry 4.0 variant of CoCoME

4.2 Production recipe processing

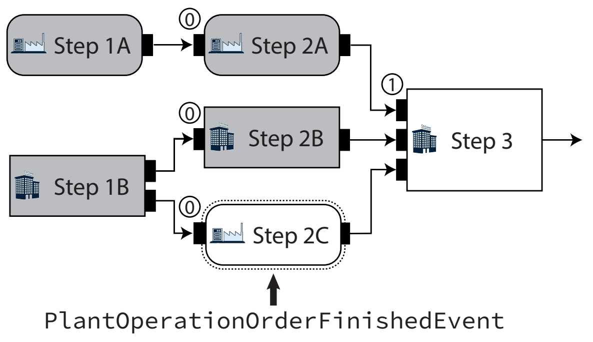
The most crucial part of the recipe scheduling is handled by the `ProductionManager` class. Every time the `EnterpriseManager`[1] class receives a production order from a plant or another enterprise, it makes an asynchronous call to `ProductionManager.submitOrder(order)`[2]. From there on, the `ProductionManager` loads the corresponding production order recipe, translates it into an execution graph and stores the data with other order-specific data into a hash table structure.



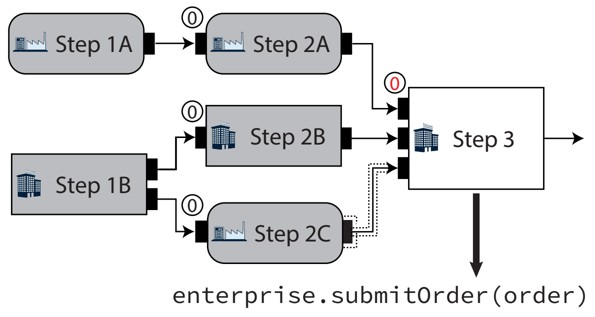
(a) Initial execution graph



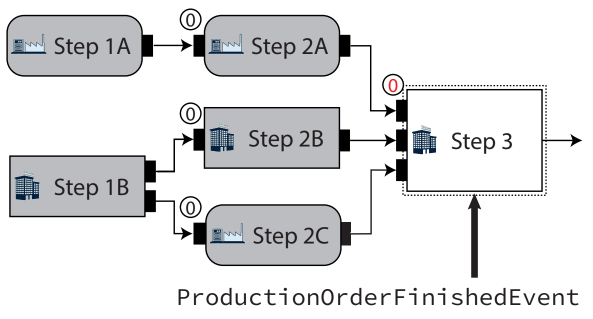
(b) Issuing plant operation order "Step 2C"



(c) Receiving event after order completion



(d) Issuing production order "Step 3"



(e) Receivng event after order completion

Figure 4.2: Production order scheduling on the enterprise server

The execution graph is an optimized runtime data structure, whose nodes represent either

[1]Module **enterprise-logic-webservice**, package **org.cocome.cloud.webservice.enterpriseservice**
[2]Module **enterprise-logic-ejb**, package **org.cocome.tradingsystem.inventory.application.production**

plant operations or other production operations. Each execution graph node has a counter initialized to the number of input ports (i.e., the number of incoming edges). After initializing the execution graph, the `ProductionManager` starts with the leaves of the execution graph and sends orders to subsystems who provide the particular operation. The `ProductionManager` then pauses his execution and the associated runtime thread can return to the application server thread pool. If the enterprise server receives a `PlantOperationOrderFinishedEvent`[3] or an `ProductionOrderFinishedEvent`[3], it delegates the event to the `ProductionManager`, which in turn resumes the event-causing execution graph. The `ProductionManager` then flags the operation as finished, traverses to the neighboring nodes and reduces each counter by one. Nodes whose counters are reduced to zero mark the next operations to be processed by the `ProductionManager`. This means the `ProductionManager` issues new orders for these nodes and again pauses the execution or reaches the end of the execution graph.

Figure 4.2a shows an example of an execution graph in its initial state. Figure 4.2b shows the execution graph after several cycles where the gray nodes are already processed. The `ProductionManager` detects a zero counter on node "Step 2C" and issues an order to the corresponding plant. After some time, the enterprise server receives an `PlantOperationOrderFinishedEvent` and decrements the counter of operation "Step 3" (as seen in Figure 4.2c and Figure 4.2d). The enterprise manager then detects the zero counter of "Step 3" and sends an appropriate order to the 3nd party enterprise (Figure 4.2e).

4.3 Plant operation processing

In contrast to the `ProductionManager` of an enterprise server, the `PUManager`[4] is the primary executing element for plant operations. Processing a plant operation order includes the following steps (as seen in Figure 4.3):

❶ Fetch the corresponding plant operation recipe from the cloud storage and evaluate the recipe markup based on the given parameters in the order. Additionally, group the evaluated instruction list into work packages based on the target production unit classes (keep sequential order of the instructions)

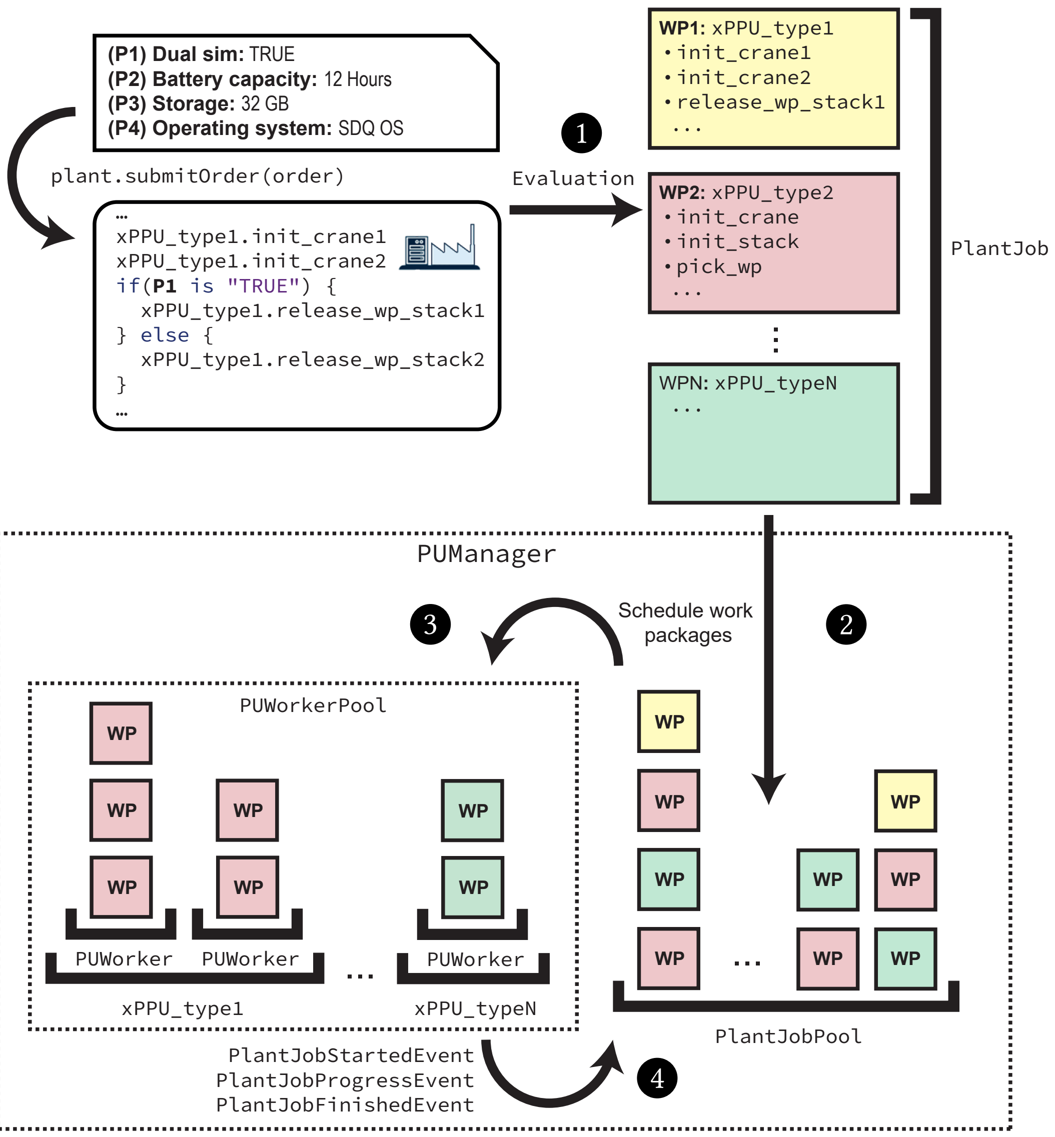
❷ Add the list of work packages as job to the `PlantJobPool`[4]. Generate a job id and use it as hash table key for efficient access.

❸ Pick the first available working package of the job and put it on the job queue of the next free `PUWorker`. If all instances of the desired production unit class are busy, associate the working package to the unit with the smallest queue.

❹ Suspend the plant operation execution until a `PUWorker` issued a new plant job event. If it is a `PlantJobFinishedEvent`[5], pick the affected job from the `PlantJobPool` and repeat step ❸ and ❹.

[3]Module **enterprise-logic-ejb**, package **org.cocome.tradingsystem.inventory.application.production**
[4]Module **plant-logic-ejb**, package **org.cocome.tradingsystem.inventory.application.plant.pu**
[5]Module **plant-logic-ejb**, package **org.cocome.tradingsystem.inventory.application.plant.pu.event**

The plant operation scheduling for a specific order is finished when no work package is left after step ❹. The plant server then sends a `PlantOperationOrderFinishedEvent` to the caller to signal the completion. Each `PUWorker` has its own CPU thread and encapsulates the REST-API calls required to control the production unit. This additional abstraction is required to provide an event-based production environment, even if the interfaced production units do not offer REST callback routines.



```
...
xPPU_type1.init_crane1
xPPU_type1.init_crane2
if(P1 is "TRUE") {
  xPPU_type1.release_wp_stack1
} else {
  xPPU_type1.release_wp_stack2
}
...
```

Figure 4.3: Plant order scheduling on the plant server

4.4 xPPU Interface

The AIS implemented an interface for providing a remote access to their xPPU instances (as seen in Figure 4.4) [2]. It hides device-specific communication protocols under a REST-based API (e.g., the machine-to-machine communication protocol Object linking and embedding for Process Control Unified Architecture (OPC UA)).

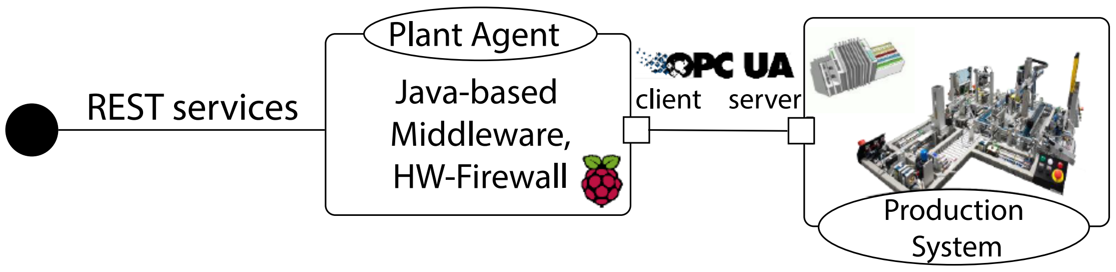


Figure 4.4: Concept of the PPU REST-Interface [2]

Table 4.1 shows an excerpt of the available REST operations on the xPPU interface [2].

Command	Meaning
/operation	Lists all available operations
/history	Shows all finished and running operations
/instance	Returns the ISA-88[16] instance model

Command	Meaning
/start	Starts a new operations
/hold	Pauses the execution of an issued operation
/startbatch	Buck execution of operations

Table 4.1: Excerpt of the available REST operations on the xPPU interface [2]

5 Developer Guide

Implementing new functionalities in CoCoME can be difficult for developers who didn't work with the community case study before. We therefore use the next pages to illustrate the necessary implementations steps based on a test evolution scenario. In this test scenario, we extent the production recipe by the data structure ResourceType. This allows an enterprise manager to define types of resources that are transported between the plants / enterprises. The current version of CoCoME uses a string property inside the EntryPoint data structure to distinguish between different material types. Implementing this feature will affect multiple subcomponents in CoCoME, which include the **ServiceAdapter**, the **TradingSystem::Inventory::Data**, **WebService::Inventory**, and **WebFrontend::Web**.

5.1 Prerequirements

It is required to have a working CoCoME setup before proceeding with the next sections. A typical CoCoME installation process includes the following steps:

1. Create a subdirectory `cocome`
2. Clone the main CoCoME project and the service adapter project into the `cocome` directory by executing this command line commands:

```
cd cocome
git clone
↪  git@github.com:cocome-community-case-study/cocome-cloud-jee-platform-migration.git
↪  core
git clone git@github.com:cocome-community-case-study/cocome-cloud-jee-service-adapter.git
↪  service-adapter
```

3. Switch to the GIT branch `xppu_integration`:

```
(cd core && git checkout xppu_integration)
(cd service-adapter && git checkout xppu_integration)
```

4. Follow the preparation guides inside `cocome/core/cocome-maven-platform/doc`. Especially the instructions in "Installation in a nutshell.md" will be helpful

5.2 Extending the ServiceAdapter component

For this section, we will modify the code of the following submodules of the service adapter:

```
cocome/service-adapter/service-adapter-ejb  # JPA Entities and converters
cocome/service-adapter/service-adapter-rest # RESTful Java servlet
```

1. **Modify the JPA classes in `service-adapter-ejb`.** The JPA classes exist in the package `org.cocome.tradingsystem.inventory.data`.
An example JPA class for our test evolution scenario could look like this:

```java
package org.cocome.tradingsystem.inventory.data.plant.recipe;

@Entity
public class ResourceType implements Serializable, NameableEntity {

  private static final long serialVersionUID = 1L;

  private long id;
  private String name;
  private TradingEnterprise enterprise;

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  public long getId() { return this.id; }

  public void setId(final long id) {this.id = id;}

  @Basic
  public String getName() {return this.name;}

  public void setName(final String name) {this.name = name;}

  @ManyToOne
  @NotNull
  public TradingEnterprise getEnterprise() {return this.enterprise;}

  public void setEnterprise(final TradingEnterprise enterprise) {this.enterprise =
  ↪   enterprise;}
}
```

Additionally, we need to add an $n : 1$ relationship between the JPA classes EntryPoint and ResourceType:

```
@Entity
public class EntryPoint implements Serializable, NameableEntity {
  ...
  private ResourceType resourceType;

  @NotNull
  @ManyToOne
  public ResourceType getResourceType() {return resourceType;}

  public void setResourceType(ResourceType resourceType) {this.resourceType =
  ↪   resourceType;}
  ...
}
```

2 **Modify the Data Access Object (DAO) in service-adapter-ejb.** They provide methods for serializing and deserializing JPA entities during network exchange and are located in package
`org.cocome.tradingsystem.remote.access.dao`
In the test evolution scenario, we need to add a new DAO class ResourceTypeDAO and to modify the existing DAO class EntryPointDAO for the corresponding type EntryPoint.

3 In module `service-adapter-rest`, **add references to all newly created DAO classes in class `cocome.cloud.sa.serviceprovider.impl.ServiceProviderDatabase`.**
For example, we would add the following entries for the ResourceTypeDAO class:

```
@WebServlet("/Database/ServiceProviderDatabase")
public class ServiceProviderDatabase extends HttpServlet {
  ...
  @Inject
  private ResourceTypeDAO resourceTypeDAO;
  ...
  @PostConstruct
  protected void initDAOMap() {
    ...
    daoMap.put(resourceTypeDAO.getEntityTypeName(), resourceTypeDAO);
    ...
  }
  ...
 }
```

If you want to add CRUD operations for supertypes to the service adapter, you can add a DAO class that extends the AbstractInheritanceTreeDAO class (e.g., see ParameterDAO for details).

4. **Redeploy the service adapter module to make the changes visible to other CoCoME subsystems.** If you don't use an IDE for redeployment, you may need to rebuild a new EAR archive through the `mvn package` command.

5.2.1 Unit testing

We use simple Java SE unit tests for newly created DAO classes. Unit tests reside in a corresponding package within the `src/test` subfolder of the `service-adapter-ejb` module. Adding new unit tests requires the following steps:

1. **Add newly created JPA classes to `src/test/resources/META-INF/persistence.xml`.** For example, we have to add the following line to the test persistence configuration to make the ResourceType class usable **during testing**:

```
...
<class>org.cocome.tradingsystem.inventory.data.plant.recipe.ResourceType</class>
...
```

2. **Add DAO unit test to the corresponding Java package.** For the test scenario, we have to add a new unit test ResourceTypeDAOTest.
3. **Make sure that existing tests do not break.** Modifications on the JPA classes usually effect the persistence behavior of other entities. During the implementation of the test scenario, we have to adjust the unit tests EntryPointDAOTest, RecipeDAOTest, EntryPointInteractionDAOTest, and PlantOperationDAOTest.

5.3 Extending the TradingSystem::Inventory::Data component

In this section we will perform all modifications inside the following maven submodules:

```
cocome/core/cocome-maven-project/cloud-logic-service/cloud-logic-core-api
cocome/core/cocome-maven-project/cloud-logic-service/cloud-logic-core-impl
cocome/core/cocome-maven-project/cloud-logic-service/cloud-enterprise-logic/enterprise-logic-ejb
```

5.3.1 Creating data structures and transfer objects

First, we have to add equivalent data classes and transfer object types to the main maven project. The transfer objects are solely used for data exchange between webservices and do not necessarily have the same structures as the data classes in the service adapter project. The regular data classes on the other hand should reflect the same properties and aggregations as their service adapter counterpart. The reason why we have to define data classes twice is because of technical limitations in the CoCoME service adapter, which will be addressed in future iterations.

1. **Create an interface for all new JPA classes created in the service adapter.** These interfaces should be located in the `cloud-logic-core-api` module. An example for the test scenario could look like this:

```java
package org.cocome.tradingsystem.inventory.data.plant.recipe;

import org.cocome.tradingsystem.inventory.data.INameable;

public interface IResourceType extends INameable {

  long getId();
  void setId(final long id);

  String getName();
  void setName(final String name);

  ITradingEnterprise getEnterprise() throws NotInDatabaseException;
  void setEnterprise(final ITradingEnterprise name);
}
```

2. **Add a corresponding implementation** for each added interface to the module `cloud-logic-core-impl`:

```java
package org.cocome.tradingsystem.inventory.data.plant.recipe;

@Dependent
public class ResourceType implements Serializable, IResourceType {

  private static final long serialVersionUID = 1L;

  private long id;
  private String name;
  private ITradingEnterprise enterprise;
  private enterpriseId;

  @Inject
14    private Instance<IEnterpriseQuery> enterpriseQueryInstance;
15    private IEnterpriseQuery enterpriseQuery;
16
17    @PostConstruct
18    public void initPlant() {
19      enterpriseQuery = enterpriseQueryInstance.get();
20      operation = null;
21    }
22
23    public long getId() {return this.id;}
24    public void setId(final long id) {this.id = id;}
25
26    public String getName() {return this.name;}
27    public void setName(final String name) {this.name = name;}
28
29    public int getEnterpriseId() {return this.enterpriseId;}
30    public void setEnterprseId(final int enterpriseId) {this.enterpriseId = enterpriseId;}
31
32    public IRecipeOperation getEnterprise() throws NotInDatabaseException {
33      if (enterprise == null) {
34        enterprise = enterpriseQuery.queryEnterpriseById(operationId);
35      }
36      return enterprise;
37    }
38
39    public void setEnterprise(ITradingEnterprise enterprise) {this.enterprise =
      ↪   enterprise;}
40  }
```

The IEnterpriseQuery is used in line 34 to load the corresponding enterprise instance lazily from the service adapter.

3 **Add transfer object classes** to module `cloud-logic-core-impl`:

```
1  package org.cocome.tradingsystem.inventory.application.plant.recipe;
2
3  import org.cocome.tradingsystem.inventory.application.INameableTO;
4
5  @XmlType(name = "ResourceTypeTO",
6    namespace = "http://recipe.plant.application.inventory.tradingsystem.cocome.org")
7  @XmlRootElement(name = "ResourceTypeTO")
8  @XmlAccessorType(XmlAccessType.FIELD)
9  public class ResourceTypeTO implements Serializable, INameableTO {
```

```java

  private static final long serialVersionUID = 1L;

  @XmlElement(name = "id", required = true)
  private long id;
  @XmlElement(name = "name", required = true)
  private String name;
  @XmlElement(name = "enterprise", required = true)
  private EnterpriseTO enterprise;

  public long getId() {return this.id;}
  public void setId(final long id) {this.id = id;}
  public String getName() {return this.name;}
  public void setName(final String name) {this.name = name;}
  public EnterpriseTO getEnterprise() {return this.enterprise;}
  public void setEnterprise(final EnterpriseTO enterprise) {this.enterprise =
  ↪   enterprise;}
}
```

5.3.2 Extending data structure converters and factories

After creating data classes and transfer object classes, we have to extend data factory and data conversion classes. These classes are utility classes used to convert transfer objects to primary data classes back and forward. There exist three different variants:

Classes	Package
`IEnterpriseDataFactory` `EnterpriseDatatypesFactory`	`org.cocome.tradingsystem.inventory.data.enterprise`
`IPlantDataFactory` `PlantDatatypesFactory`	`org.cocome.tradingsystem.inventory.data.plant`
`IStoreDataFactory` `StoreDatatypesFactory`	`org.cocome.tradingsystem.inventory.data.store`

IEnterpriseDataFactory, IPlantDataFactory, and IStoreDataFactory are interfaces inside the `cloud-logic-core-api` module. The module `cloud-logic-core-impl` holds the implementations EnterpriseDatatypesFactory, PlantDatatypesFactory, and StoreDatatypesFactory.

For the test evolution scenario, we added the following methods to IPlantDataFactory / PlantDatatypesFactory:

Classes	Package
`IResourceType getNewResourceType();`	CDI-based factory method
`ResourceTypeTO fillResourceTypeTO( IResourceType rt) throws NotInDatabaseException;`	Converts data objects to transfer objects, which may involves database lookups
`IResourceType convertToResourceType( ResourceTypeTO rtTO);`	Converts transfer objects to data objects

Additionally, we have to modify the existing methods fillEntryPointTO convertToEntryPoint to reflect the $n : 1$ relationship between EntryPoint end ResourceType.

5.3.3 Extending persistence methods

In this step we extend the **Persistence** subcomponent, which provides low-level operations for creating, updating, and deleting data records. All modifications will affect the package

```
org.cocome.tradingsystem.inventory.data.persistence
```

in either the `cloud-logic-core-api` or `cloud-logic-core-impl` module.

1. **Add persistence method signatures to the interface IPersistence.** The interface can be found in module `cloud-logic-core-api`. For example, we add the following methods for ResourceType:

Operation	Function
`void createEntity(IResourceType rt) throws CreateException;`	Persist a given instance to the service provider storage
`void updateEntity(IResourceType rt) throws UpdateException;`	Updates an instance in the service provider storage denoted by its ID
`void deleteEntity(IResourceType rt) throws UpdateException;`	Deletes a given instance from the service provider storage denoted by its ID

2. **Add CSV headers to the class ServiceAdapterHeaders.** The class can be found in module `cloud-logic-core-impl`. In our test scenario, we also have to add the Resource-TypeId to the CSV header of type EntryType.
3. **Add converter methods to ServiceAdapterEntityConverter.** The class can be found in module `cloud-logic-core-impl`. In the test scenario, we also have to modify existing converter methods of type `EntryType`.
4. **Implement persistence methods in CloudPersistenceContext.** The class can be found in module `cloud-logic-core-impl` and consists of delegation methods to the ServiceAdapterEntityConverter class.

5.3.4 Extending parsing utilities

The following step involves the **Parsing** subcomponent, which is used in CoCoME to parse CSV records obtained from the service adapter storage. All modifications will affect the package

```
org.cocome.tradingsystem.remote.access.parsing
```

in either the `cloud-logic-core-api` or `cloud-logic-core-impl` module.

1. **Add parsing method signatures to the interface IBackendConversionHelper**. The interface can be found in module `cloud-logic-core-api`. For our test scenario, we would add the following signature to the class:

```
Collection<IPlantOperationOrder> getResourceType(String resourceType);
```

2. **Implement parsing methods in `CSVHelper`**.
The class can be found in module `cloud-logic-core-impl`. For the test evolution scenario, we implemented `getResourceType` and change `getEntryPoint`.

5.3.5 Extending query providers

Next, we have to extend the query provider used to fetch existing data from the service adapter. The subsystems store, plant, and enterprise have their own query provider. The interfaces are all part of the module `cloud-logic-core-api`:

Class	Package
`IEnterpriseQuery`	`org.cocome.tradingsystem.inventory.data.enterprise`
`IPlantQuery`	`org.cocome.tradingsystem.inventory.data.plant`
`IStoreQuery`	`org.cocome.tradingsystem.inventory.data.store`

The implementations of these interfaces can be found in different modules:

Implementation	Interface	Module
`EnterpriseQueryProvider`	`IEnterpriseQuery`	`enterprise-logic-ejb`
`ProxyEnterpriseQueryProvider`	`IEnterpriseQuery`	`cloud-logic-core-impl`
`PlantEnterpriseQueryProvider`	`IEnterpriseQuery`	`plant-logic-ejb`
`StoreEnterpriseQueryProvider`	`IEnterpriseQuery`	`store-logic-ejb`
`EnterprisePlantQueryProvider`	`IPlantQuery`	`cloud-logic-core-impl`
`EnterpriseStoreQueryProvider`	`IStoreQuery`	`cloud-logic-core-impl`

IPlantQuery and IStoreQuery with their implementation classes are only used in their dedicated subsystems. The IEnterpriseQuery is accessible from all major subsystems including enterprises, plants and stores. This makes it necessary to provide different proxy implementation for the plant subsystem (PlantEnterpriseQueryProvider) and the store subsystem (StoreEnterpriseQueryProvider). The superclass ProxyEnterpriseQueryProvider provides the complete implementation for the concrete proxy classes.

ResourceType class is part of the production recipe DSL, it is required to modify IEnterpriseQuery, EnterpriseQueryProvider, and ProxyEnterpriseQueryProvider. However, it is not possible to extend EnterpriseQueryProvider and ProxyEnterpriseQueryProvider in one step, because the ProxyEnterpriseQueryProvider requires the existence of enterprise webservice stubs. However these webservice stubs cannot be generated if a compilation error exist. For our test evolution scenario, we have to delay the modification ProxyEnterpriseQueryProvider until the end of the up-following section.

5.3.6 Integration Testing

It is possible to use CDI-based integration testing to verify if the correctness of changed persistence- and query classes. An example can be found within the `enterprise-logic-ejb` module:

```
org.cocome.tradingsystem.inventory.data.enterprise.QueryProviderAndPersistenceIT
```

These tests require a running application server with a running service adapter instance.

5.4 Extending the WebService::Inventory component

We so far implemented the back end of the CRUD operations. In the next stage of the test scenario, we add webservice calls for these operations to make them visible to other subsystems. The SOAP-based webservices in CoCoME consist of four components:

- The general service definition inside the `pom.xml` file of the `cloud-logic-core-services` module
- An annotated webservice interface inside the `cloud-logic-core-services` module
- An implementation of the webservice interface (the location can vary.)
- An Apache CXF mapping file for the transfer objects

The mapping file reside in `cloud-logic-core-services/src/main/resources` and are necessary during the webservice stub generation. This generation process is managed by the webservice framework Apache CXF. During the stub generation, Apache CXF processes the Java interfaces to generates Webservice Definition Language (WSDL) documents and client source code. The mapping files are used to exclude the transfer objects to be replaced with stub classes during the source code generation.

The files / packages of the three subsystem webservices (i.e., enterprise, plant, and store) are:

Class	**Maven Module**
`IEnterpriseManager`	`cloud-logic-core-webservice`
`EnterpriseManager`	`cloud-enterprise-logic/enterprise-logic-webservice`
`IPlantManager`	`cloud-logic-core-webservice`
`PlantManager`	`cloud-plant-logic/plant-logic-webservice`
`IStoreManager`	`cloud-logic-core-webservice`
`StoreManager`	`cloud-store-logic/store-logic-webservice`
webservice	**Package**
Enterprise manager	`org.cocome.logic.webservice.enterpriseservice`
Plant manager	`org.cocome.logic.webservice.plantservice`
Store manager	`org.cocome.logic.webservice.storeservice`
webservice	**Mapping file**
Enterprise manager	`enterpriseManagerBindings.xml`
Plant manager	`plantManagerBindings.xml`
Store manager	`storeManagerBindings.xml`

For the test evolution scenario, we modified the enterprise manager webservice by performing the following steps:

1 Add CRUD operations to the interface `IEnterpriseManager`:

```java
...
@WebMethod ResourceTypeTO queryResourceTypeById(
  @XmlElement(required = true)
  @WebParam(name = "resourceTypeID") long resourceTypeId)
  throws NotInDatabaseException;

@WebMethod Collection<ResourceTypeTO> queryResourceTypeByEnterpriseId(
  @XmlElement(required = true)
  @WebParam(name = "enterpriseID") long enterpriseId)
  throws NotInDatabaseException;

@WebMethod long createResourceType(
  @XmlElement(required = true)
  @WebParam(name = "resourceTypeTO") ResourceTypeTO resourceTypeTO)
  throws CreateException, NotInDatabaseException;

@WebMethod void updateResourceType(
  @XmlElement(required = true)
  @WebParam(name = "resourceTypeTO") ResourceTypeTO resourceTypeTO)
  throws UpdateException, NotInDatabaseException;

@WebMethod void deleteResourceType(
  @XmlElement(required = true)
  @WebParam(name = "resourceTypeTO") ResourceTypeTO resourceTypeTO)
  throws UpdateException, NotInDatabaseException;
...
```

2 **Implement webservice methods in `EnterpriseManager`.**
3 **Repackage the core package of CoCoME with the `mvn package` command.**
4 **Add newly created transfer objects to `enterpriseManagerBindings.xml`:**

```
...
<bindings if-exists="true" schemaLocation="IEnterpriseManager_schema2.xsd">
  ...
  <bindings node="//xsd:complexType[@name='ResourceTypeTO']">
    <class
      ref="org.cocome.tradingsystem.inventory.application.plant.recipe.ResourceTypeTO"/>
  </bindings>
  ...
</bindings>
...
```

One way to determine which `IEnterpriseManager_schema*.xsd` to use for the mapping below, is to perform step 4 and look inside the XML Schema Definitioon (XSD) files in the `src/main/resources` directory of the `cloud-logic-core-webservice` module.

5 **Repackage the core package of CoCoME with the `mvn package` command.**
6 **Add new available webservice calls to the ProxyEnterpriseQueryProvider**, if necessary.
7 Redeploy the modified CoCoME maven modules to the application server.

5.4.1 Integration Testing

We use integration tests that use the Apache CFX clients to call running webservice directly. There exist one test for each manager webservice, e.g., EnterpriseManagerIT, PlantManagerIT, and StoreManagerIT.

5.5 Extending the web frontend

Extending the web-based user interfaces requires modifications in this maven submodule:

```
cocome-cloud-jee-platform-migration/cocome-maven-project/cloud-logic-web
```

For the test evolution scenario, we want to add an additional web interface for retrieving and adding resource types. To make the enterprise manager responsible for controlling the resource types, we have to perform the following steps:

1. **Add view data classes for each new data structure**. For the class `ResourceType`, it would look like this:

```java
package org.cocome.cloud.web.data.enterprisedata;

import org.cocome.cloud.web.data.ViewData;
import org.cocome.tradingsystem.inventory.application.plant.recipe.RecourceTypeTO;

public class RecourceTypeTOViewData extends ViewData<RecourceTypeTO> {

        public RecourceTypeViewData(RecourceTypeTO data) {
                super(data);
        }

        @Override
        public long getParentId() {
                return this.data.getEnterprise().getId();
        }
}
```

2. **Add a view DAO class for each new data structure**. These DAO classes encapsulate webservice calls and provide an automatic caching mechanism. Data retrial through webservice calls can lead to performance degeneration if CoCoME is executed on a single computer. Examples of view DAO classes exist inside package `org.cocome.cloud.web.data.enterprisedata`.
3. **Add a view class for each web view**. These classes build the bridge between Java Server Faces (JSF) documents and the view DAO classes. Examples of view classes exist inside package `org.cocome.cloud.web.frontend.enterprise`.
4. **Add JSF pages for each web view**. These EXtensible HyperText Markup Language (XHTML) documents can be found inside the directory `src/main/webapp`. For our test scenario, we added a new JSF page inside the subfolder `enterprise`. Additionally, we modified the pages `show_enterprises.xhml`, `create_input_entry_point.xhml`, and `create_output_entry_point.xhml`.
5. **Add localized messages** by adding message keys and values to the properties file in `src/main/resources/cocome/frontend/Strings.properties`.

6 Conclusion

In this technical report we gave an overview over existing case studies and concept papers about Industry 4.0. Based on our findings, we designed an evolution scenario that combines the hybrid-cloud variant of CoCoME with the REST-enabled xPPU into an Industry 4.0 community case study. We identified reoccurring structural [10] and functional [9] properties of Industry 4.0 systems in the implemented evolution scenario. Examples of these properties are the user involvement through customizable products, the flexibility and modularity in defining production templates, as well as the hierarchy between production units, plants, and enterprises.

With the new variant of CoCoME, we can provide a platform for automation engineers and software engineers who want to conduct empirical Industry 4.0 studies. They can observe the impacts of their applied methods on a more heterogenous environment (e.g., predictability of performance impacts on the high-level CoCoME use cases with changing production unit hardware) [7]. Case studies on the Industry 4.0 variant of CoCoME are more replicable and comparable, since they fulfill the requirements of a research platform (e.g., available tools, documentation, and design-type / run-tine artifacts).

For the future, we plan to implement other functional requirements on Industry 4.0 systems that are not already part of our evolution scenario. Known examples are real-time monitoring of production unit utilization, auto-adjustment of the production process depending on changing pricing policies or order quantities, and a meta-model for describing resources and products with their different production stages [9]. Furthermore, the research communities in automation engineering and software engineering lack a fine-grained categorization of Industry 4.0 system characteristics, like quality of service requirements, design- and run-time activities, or used middleware technologies other than SOAP-based web services. A systematic literature research could be an appropriate method to obtain this information.

Acronyms

API Application Programming Interface. 7, 8, 25

CBSE Component-Based Software Engineering. 11
CDI Contexts and Dependency Injection. 34, 36
CoCoME Common Component Modelling Example. 7
CPPS Cyber-Physical Production Systems. 9, 13
CPS Cyber-Physical Systems. 7, 13
CRUD Create, Read, Update, Delete. 21, 30
CSV Comma Seperated Values. 20, 34, 35

DAO Data Access Object. 29, 30, 39
DSL Domain-Specific Language. 17–20, 36

EJB Enterprise Java Beans. 11

IDE Integrated Development Environment. 30

JPA Java Persistence API. 21, 28–31
JSF Java Server Faces. 39

OPC UA Object linking and embedding for Process Control Unified Architecture. 19, 26

POJO Plain Old Java Objects. 11

REST REpresentational State Transfer. 7, 25

SOAP Simple Object Access Protocol. 11

WSDL Webservice Definition Language. 36

XHTML EXtensible HyperText Markup Language. 39
xPPU eXtended Pick and Place Unit. 7
XSD XML Schema Definitioon. 38

Bibliography

[1] M. Bonfè, C. Fantuzzi, and C. Secchi. Design patterns for model-based automation software design and implementation. *Control Engineering Practice*, 21(11):1608 – 1619, 2013. Advanced Software Engineering in Industrial Automation (INCOM'09).

[2] S. Bougouffa, K. Meßzmer, S. Cha, E. Trunzer, and B. Vogel-Heuser. Industry 4.0 interface for dynamic reconfiguration of an open lab size automated production system to allow remote community experiments. In *2017 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pages 2058–2062, 2017.

[3] B. Demchak, V. Ermagan, E. Farcas, T.-j. Huang, I. H. Krüger, and M. Menarini. A rich services approach to cocome, 01 2007.

[4] M. Goeminne, A. Decan, and T. Mens. Co-evolving code-related and database-related changes in a data-intensive software system. *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 353–357, 2014.

[5] S. Grüner, J. Pfrommer, and F. Palm. Restful industrial communication with opc ua. *IEEE Transactions on Industrial Informatics*, 12(5):1832–1841, Oct 2016.

[6] R. Heinrich, S. Gärtner, T.-M. Hesse, T. Ruhroth, R. Reussner, K. Schneider, B. Paech, and J. Jürjens. A Platform for Empirical Research on Information System Evolution. *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE'15)*, pages 415–420, 2015.

[7] R. Heinrich, K. Rostami, and R. Reussner. The CoCoME Platform for Collaborative Empirical Research on Information System Evolution. 2016.

[8] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller. CoCoME - The common component modeling example. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5153 LNCS:16–53, 2008.

[9] N. Jazdi. Cyber physical systems in the context of industry 4.0. In *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*, pages 1–4, May 2014.

[10] J. Lee, B. Bagheri, and H.-A. Kao. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, 3:18 – 23, 2015.

[11] C. Legat, J. Folmer, and B. Vogel-Heuser. Evolution in industrial plant automation: A case study. In *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, pages 4386–4391, 2013.

[12] C. Legat, D. Schütz, and B. Vogel-Heuser. Automatic generation of field control strategies for supporting (re-)engineering of manufacturing systems. *Journal of Intelligent Manufacturing*, 25(5):1101–1111, 2014.

[13] M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213 – 221, 1979.

[14] P. Mäder and A. Egyed. Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering*, 20(2):413–441, 2015.

[15] D. Pantförder, F. Mayer, C. Diedrich, P. Göhner, M. Weyrich, and B. Vogel-Heuser. *Agentenbasierte dynamische Rekonfiguration von vernetzten intelligenten Produktionsanlagen – Evolution statt Revolution*, pages 145–158. Springer Fachmedien Wiesbaden, 2014.

[16] J. Parshall and L. Lamb. *Applying S88: Batch Control from a User's Perspective.* ISA, 1999.

[17] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil. The common component modeling example. *Lecture notes in computer science*, 5153, 2008.

[18] B. Vogel-Heuser, C. Diedrich, D. Pantförder, and P. Göhner. Coupling heterogeneous production systems by a multi-agent based cyber-physical production system. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 713–719, 2014.

[19] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy. Evolution of software in automated production systems: Challenges and research directions. *Journal of Systems and Software*, 110:54 – 84, 2015.

[20] B. Vogel-Heuser, C. Legat, J. Folmer, and S. Feldmann. Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit. *Technical Report*, (TUM-AIS-TR-01-14-02), 2014.