Karlsruhe Reports in Informatics 2017,2
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

CoCoME with Security

Simon Greiner, Mihai Herda

2017

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

CoCoME with Security

Simon Greiner
Simon.Greiner@kit.edu

Mihai Herda
Mihai.Herda@kit.edu

Abstract

In this technical report we provide the documentation of the functional requirements of a component-based system representing the IT infrastructure supermarket along with the description of confidentiality properties in the form of information flow requirements for the system. From an architectural point of view, we describe for each interface all services on a functional level. We identify a number of possible attackers and assign for each attacker what inputs to the system she may gain knowledge about and which outputs she may be able to observe. The architecture and security properties of the system are modeled using an extension of the Palladio Component Model

1 Introduction

Computer systems are getting more complex and are increasingly deployed in security critical areas. The rising complexity of these systems increases the probability of developers introducing bugs and thus of an attacker finding an exploit. Increased deployment of systems in scenarios where security plays a very important role increases the potential damage that can be done by an attacker.

Component based systems engineering is a method used to deal with complexity and maintainability of systems. A component based system is made of components, each providing part of the functionality while gaining the full functionality by composing single components. The components are independent of each other in the sense that they do not share a state and comminucate via provided and required interfaces which specify the services that actually implement the functionality.

In this technical report we model the IT infrastructure of a supermarket using a specification language designed for model based systems engineering. We additionally model security requirements for the system in the form of information flow properties. In the context of information flow security the inputs and outputs of the system are assigned to security domains *high* (confidential) and *low* (public). A system is noninterferent if information provided as high input does not interfere with information provided as low output of the system. The formal definition of noninterference for component based systems that we use in this case study are taken from [GG16].

The system we model is based on the Common Component Modeling Example (CoCoME) [HKW+08], a system definition frequently used for

illustration of component-based specification techniques. We use the Palladio Component Model (PCM) approach to model components. While Palladio has been developed to primarily model quality aspects of components, an extension to include confidentiality properties is in development. This approach has the advantage that security properties are modeled early in the development process, thus allowing the early identification of security issues.

The reminder of the document is structured as follows: in Section 2 we present the definition of noninterference for components as we use it in this report. In Section 3 we describe the system to be modeled, the components that make it up, the interfaces, and services for each component. We show how the system is modeled using PCM in Section 3.5. In Section 4 we provide the specification of the information flow properties for the system and show how we specified them in the PCM. Finally, we conclude in Section 5.

2 Noninterference for Components

In this section we provide a short overview of the definition of noninterference and compositionality for components as used inb this report. For a full account of the noninterference property, the interested reader may refere to [GG16].

Noninterference, in general, requires that a system does not publish any secret information on public channels. The inputs and outputs of a system are partitioned in *high* (secret) and *low* (public) inputs and outputs and the noninterference property requires the low outputs to be independent of the high inputs. This means that if the low inputs to the system are equivalent for two runs, the low outputs must also be equivalent. We consider an equivalence relation $\sim$ that determines whether two inputs or two outputs are low equivalent. The equivalence relation $\sim$ is subject to specification. In the remainder of this section we will describe a noninterference property that is adequate for component based systems. We also show how the characteristics of component based system influenced the choice of the definition of noninterference for components and why the definition by [GG16] is an appropriate one.

Components are software parts that implement a certain functionality. Components provide and require interfaces. The interfaces contain service signatures that declare the parameters and return types of each service. The environment can call the services of the provided interfaces of a component and thus access the functionality implemented in the component. A component may need to call services provided by the environment in order fullfill its functionality. To specify the services a component may depend on, a component may have required interfaces. For convenience, and without loss of generality, in the rest of this section we will consider that services are provided and required directly by components, and instead of interfaces as is done in [GG16].

Two components, by definition, do not share a state, and the only way a component can communicate with other components is through the required services. Thus, components communicate only through message

passing and not through memory sharing. However, the components themselves do have a state and all services provided by the component have access to that state. This means, for a component, that two service calls with the same input parameters may return different results depending on previous calls of services of that component. In this context a service call event itself can be information that is published by the system. Service calls that may not influence the future outputs of the system are called *invisible*.

Thus, an appropriate noninterference property must be a property of sequences of service calls, and not of one service in isolation. The sequences of service calls are defined as traces of inputs and outputs to the system. Two traces t_1 and t_2 are low equivalent $t_1 \sim t_2$ if the inputs and outputs in t_1 and t_2 are element-wise low equivalent, after removing the inputs and outputs of invisible service calls from the two traces.

The noninterference property defined by [GG16] considers an environment that can send inputs to a system (by calling services and choosing their input parameters) and reading the outputs of the system (by reading the return value of a service call). The environment chooses its next input based on the observation of the previous inputs and outputs of the system. The environment is formally modeled using a strategy function, $\omega : \mathbb{T} \mapsto \mathcal{P}(\mathbb{I})$ which maps a trace (i.e. an observation) to a set of possible inputs. Given two low equivalent observations, a strategy is required to return two low equivalent sets of inputs. Two strategies are low equivalent if given the same observation they generate low equivalent inputs.

A component is non-interferent, if for all low equivalent strategies ω_1 and ω_2 and for every trace t_1 that can be generated by the component under ω_1, the component can produce a trace t_2 under strategy ω_2 such that $t_1 \sim t_2$.

Two components C_1 and C_2 where C_2 provides some of the services required by $C1$ can be combined in a *composition*. Using this mechanism, multiple components can be used to assemble a system. Since one of the main advantages of component based systems is the re-usability of components it is important for the noninterference property for components to allow the reuse of the analysis of a component. Thus we require that the noninterference property to be *compositional*, i.e. if the components of a composition are noninterferent, the composition itself is also noninterferent. The noninterference property shown in this section is compositional provided the environment fulfills the following three conditions:

1. Every service required by a composition is provided by the environment.
2. Every service called by a component terminates
3. The environment does not leak whether an invisible service was called or not.

The proof for the compositionality can be found in [GG16].

3 Description of the System

The system we consider for this case study is the IT infrastructure of a super market containing one cash desk and the necessary store management infrastructure comprised of a store server and a store client, as shown in Figure 1. The model of our system is based on the CoCoME model which represents a component based model of a super market network.

Figure 1: High level view of the system

The store server stores the data relevant to the store and also allows the store client and cash desk to make certain queries. It is comprised of three components: the application component and the data component and the persistence. The application component provides the necessary interfaces which are used by the clients (the cash desk and the store client) for accessing the server. The data component provides the interface that offers services for querying the database. The persistence provides the connection to the database. The store client is used by the store manager to manage the inventory, for example to view the currently available items on stock, to change the price of an item or to order new items.

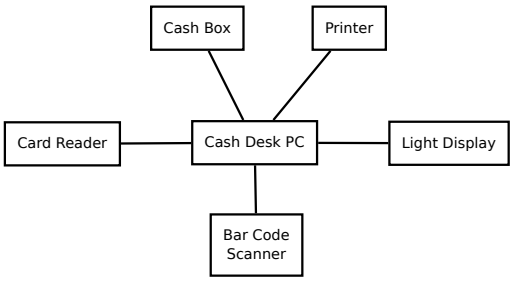

Figure 2: The cash desk

The cash desk is composed of a cash desk PC which is connected to one of each of the following devices(see Figure 2):

- LightDisplay - for displaying the scanned items and the amount to be paid
- BarCodeScanner - for reading the barcode of an item
- CardReader - for paying with a credit card
- CashBox - for managing cash payments
- Printer - for printing the receipt and the purchased items

The cash desk is also connected to a bank in order to support payment with credit card.

The system allows for a customer to purchase items at the cash desk and pay for them. The cashier uses the barcode scanner to scan the items of the customer and the scanned items along with other information relevant to the transaction are shown on the light display and printed on the receipt by the printer. The customer can pay either by cash or by credit card. The cash desk has a cash box for managing cash transactions and a credit card reader for credit card transactions. The store is managed by the store manager who uses the store client to view the inventory of the store, change the prices of the items and order new products with the store client.

Unlike the original CoCoME model our model represents a single supermarket with a single cash desk (as opposed to multiple super markets connected to an enterprise server).

3.1 Actors

We consider different classes of actors that model involved people in the physical world. It is sufficient to consider only one instance of each class symbolically. (A by-stander can be a customer in another transaction, and vice versa.)

We consider the actors from the use cases described in [HKW+08], namely the *Customer*, *Cashier*, *StoreManager* and *Bank* and the *OtherStore*. We introduce the additional actors *ByStander* and *OtherStore* which are not considered in the functional requirements but must be considered as potential attackers. The ByStander represents a person in the checkout queue next to the Customer. The OtherStore represents a store from which products are ordered for our store.

3.2 Interfaces for Interaction with the Environment

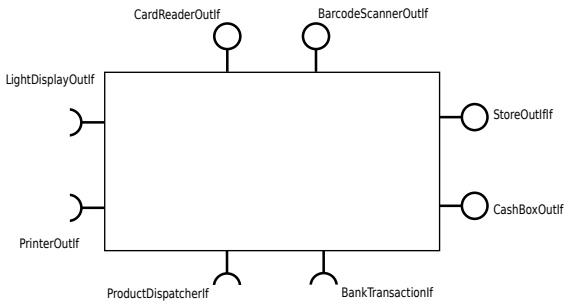

Figure 3: External interfaces of the system

This section describes the external interfaces of the system, also shown in Figure 3. The interfaces PrinterOutIf, LightDisplayOutIf, CardReaderOutIf, BarcodeScannerOutIf, and CashBoxOutIf are provided by the

devices present in a cash desk, namely the printer, the light display, the card reader, the barcode scanner, and the cash box respectively. The StoreOutIf is provided by the store client. The interfaces PrinterOutIf, LightDisplayOutIf, CardReaderOutIf, BarcodeScannerOutIf, and CashBoxOutIf and StoreOutIf are not part of the original CoCoME model. We use them to model the interaction of the system with the environment.

In order to support credit card transactions the required interface BankTransactionIf provides the necessary functionality for communication with the bank.

In the following, we describe the external interfaces of the system.

3.2.1 LightDisplayOutIf

The interface *LightDisplayOutIf* is required by the system and models the interaction of the light display with the environment. Its services output the displayed contents to the environment and can be read by anyone who is near the cash desk, including the customer, the cashier and any other by stander.

The interface offers the following services:

- `String displayShopItem(int id, int price)`
- `String displayTotal(int grossTotal)`
- `String displayPaymentCash(int amountPayed, int change)`

The service *displayShopItem* outputs the item id (*id*) and the price (*price*) of each scanned item. The service *displayTotal* outputs the total amount (*grossTotal*) that needs to be paid by the customer. The service *displayPaymentCash* outputs the amount paid (*amountPaid*) and the change (*change*) received by the customer in case of a cash payment.

3.2.2 CardReaderOutIf

The interface *CardReaderOutIf* is provided by the system and models the interaction of the customer with the card reader. Its services are reading the credit card number and pin both of which are provided by the customer.

The interface offers the following services:

- `void readCardNumber(int number)`
- `void readPIN(int pin)`

The service *readCardNumber* reads the credit card number (*number*) of the customer's credit card. The service *readPIN* reads the pin (*pin*) of the customer's credit card.

3.2.3 CashBoxOutIf

The interface *CashBoxOutIf* is provided by the system and models the interaction of the cash box with the cashier.

The interface offers the following services:

- `void startNewSale()`

- `void completeItemRegistration()`
- `boolean acknowladgeCashPayment(int amountPayed)`

The service *startNewSale* registers the start of a new sale in the system. The service *completeItemRegistration* marks the end of an item scanning process when called. The service *acknowladgeCashPayment* is called by the cashier in order to acknowledge that he received a certain amount (*amount*) of cash from the customer.

3.2.4 BarCodeScannerOutIf

The interface *BarCodeScannerOutIf* is provided by the system and models the interaction of the barcode scanner with the cashier.

The interface offers the following service:

- `void readBarcode(int barcode)`

The service *readBarcode* reads the barcode (*barcode*) from an item.

3.2.5 PrinterOutIf

The interface *PrinterOutIf* is required by the system and models the interaction of the printer with the environment. Its services output the printed contents to the enviornment and can be read by customer and the cashier.

The interface offers the following services:

- `String printShopItem(int id, String name, int price, int vat)`
- `String printTotal(int netTotal, int grossTotal)`
- `String printPaymentCard(int cardNumber)`
- `String printPaymentCash(int amountPayed)`

The service *printShopItem* outputs the *item id* and the *price* of each scanned item. The service *printTotal* outputs the total amount (*grossTotal*) that needs to be paid by the customer. The service *printPaymentCard* outputs the last four digits of the customer's credit card number (*cardNumber*) in case of a credit card payment. The service *printPaymentCash* outputs the amount paid (*amountPaid*) and the change (*change*) received by the customer in case of a cash payment.

3.2.6 BankTransactionIf

The interface BankTransactionIf is required by the system and provided by the bank. It models the interaction of the cash desk with the bank.

The interface offers the following services:

- `Acknowledgement requestTransaction(int cardnumber, Account account, int amount)`

The service *requestTransaction* requests the transfer of an amount (*amount*) from the customer's credit card (*cardnumber*) to the store's account (*account*).

3.2.7 StoreOutIf

The interface *StoreOutIf* is provided by the system and models the interaction of the store manager with the store client.

The interface offers the following services:

- `StoreWithEnterpriseTO getStore()`
- `List<ProductWithStockItemTO> getProductsWithLowStock()`
- `List<ProductWithSupplierTO> getAllProducts()`
- `List<ProductWithSupplierAndStockItemTO> getAllProductsWithOptionalStockItem()`
- `List<ComplexOrderTO> orderProducts(ComplexOrderTO complexOrder)`
- `ComplexOrderTO getOrder(int orderId)`
- `void rollInReceivedOrder(ComplexOrderTO complexOrderTO)`
- `ProductWithStockItemTO changePrice(StockItemTO stockItemTO)`
- `void markProductsUnavailableInStock(ProductMovementTO requiredProductsAndAmount)`

The service *StoreWithEnterpriseTO* returns the data regarding the store. The returned *StoreWithEnterpriseTO* contains the id, name and location of the store. The service *getProductsWithLowStock* returns a list of stock items which are currently in low stock. Each *ProductWithStockItemTO* contains a stock item and its appropriate product description. The service *getAllProducts* returns a list of all products in the data base with the suppliers from which the products can be ordered. The service *getAllProductsWithOptionalStockItem* returns a list of all products in the data base with the suppliers from which the products can be ordered and the items of that product which are on stock. The service *orderProducts* allows the store manager to issue an order(*complexOrder*) for certain products. The service *getOrder* returns the ComplexOrderTO object with the id (*orderid*) given by the store manager. The service *rollInReceivedOrder* allows the store manager to mark an order (*complexOrderTO*) as received and add the ordered items to the inventory. The service *changePrice* allows the store manager to change the price of a stock item (*stockItemTO*). The service *markProductsUnavailableInStock* allows the store manager to mark some given products(*requiredProductsAndAmount*) as unavailable in stock.

3.2.8 ProductDispatcherIf

The interface *ProductDispatcherIf* is requried by the system and provides a service for ordering products at another store.

The interface offers the following service:

- `ProductAmountTO[] orderProductsAvailableAtOtherStores( EnterpriseTO enterpriseTO, StoreTO callingStore, Collection<ProductAmountTO> productAmounts)`

The service *orderProductsAvailableAtOtherStores* allows is used to order a collection of products (*productAmounts*) from a trading enterprise (*enterpriseTO*) to the store (*callingStore*). The serivce returns an array of the product amounts as a confirmation.

3.3 Interfaces of the Internal Components

This section provides a description of the internal components of the system. The devices found at the cash desk (light display, card reader, cash box, barcode scanner, and printer) as well as the store client provide very similar interfaces to the ones described in Section 3.2. Additional internal interfaces are provided by the cash desk, application, data and persistence components of the store server. The interfaces described in this section are part of the original CoCoME case study [HKW+08]. The internal view of the system is also described in Figure 7.

3.3.1 LightDisplayIf

The *LightDisplayIf* interface is provided by the light display and required by the cash desk. Its services tell the light display what to display to the environment.

The interface offers the following services:

- `void displayShopItem(int id, int price)`
- `void displayTotal(int grossTotal)`
- `void displayPaymentCard(int cardNumber)`
- `void displayPaymentCash(int amountPayed, int change)`

The service *displayShopItem* outputs the item the *item id* and the *price* of each scanned item. The service *displayTotal* outputs the total amount (*grossTotal*) that needs to be paid by the customer. The service *displayPaymentCard* outputs the last four digits of the customer's credit card number (*cardNumber*) in case of a credit card payment. The service *displayPaymentCash* outputs the amount paid (*amountPaid*) and the change (*change*) received by the customer in case of a cash payment.

3.3.2 CardReaderIf

The interface *CardReaderIf* is required by the card reader and provided by the cash desk. Its services notify the system that the card reader has read the card number and pin of the customer's credit card.

The interface offers the following services:

- `void readCardNumber(int number)`
- `void readPIN(int pin)`

The service *readCardNumber* reads the credit card number (*number*) of the customer's credit card. The service *readPIN* reads the pin (*pin*) of the customer's credit card.

3.3.3 CashBoxIf

The interface *CashBoxIf* is required by the cash box and provided by the cash desk. Its services allow for the registration of a purchase by the customer and for payment by cash.

The interface offers the following services:

- `void startNewSale()`
- `void completeItemRegistration()`
- `boolean acknowladgeCashPayment(int amountPayed)`

The service *startNewSale* registers when called the start of a new sale in the system. The service *completeItemRegistration* marks the end of an item scanning process when called. The service *acknowladgeCashPayment* is called by the cashier in order to acknowledge that he received a certain amount (*amount*) of cash from the customer.

3.3.4 ScannerIf

The interface *ScannerIf* is required by the barcode scanner and provided by the cash desk. It offers a service for scanning the bar code of a product.

The interface offers the following service:

- `void readBarcode(int barcode)`

The service *readBarcode* reads the barcode (*barcode*) from an item.

3.3.5 PrinterIf

The *PrinterIf* interface is provided by the printer and required by the cash desk. Its services produce a printed receipt for the environment.

The interface offers the following services:

- `void printShopItem(int id, String name, int price, int vat)`
- `void printTotal(int netTotal, int grossTotal)`
- `void printPaymentCard(int cardNumber)`
- `void printPaymentCash(int amountPayed)`

The service *printShopItem* outputs the item the *item id* and the *price* of each scanned item. The service *printTotal* outputs the total amount (*grossTotal*) that needs to be paid by the customer. The service *printPaymentCard* outputs the last four digits of the customer's credit card number (*cardNumber*) in case of a credit card payment. The service *printPaymentCash* outputs the amount paid (*amountPaid*) and the change (*change*) received by the customer in case of a cash payment.

3.3.6 StoreIf

The *StoreIf* interface is provided by the application component of the store server and required by the store client.

The interface offers the following services:

- `StoreWithEnterpriseTO getStore()`
- `List<ProductWithStockItemTO> getProductsWithLowStock()`
- `List<ProductWithSupplierTO> getAllProducts()`
- `List<ProductWithSupplierAndStockItemTO> getAllProductsWithOptionalStockItem()`
- `List<ComplexOrderTO> orderProducts(ComplexOrderTO complexOrder)`
- `ComplexOrderTO getOrder(int orderId)`
- `void rollInReceivedOrder(ComplexOrderTO complexOrderTO)`
- `ProductWithStockItemTO changePrice(StockItemTO stockItemTO)`
- `void markProductsUnavailableInStock(ProductMovementTO requiredProductsAndAmount)`

The service *StoreWithEnterpriseTO* returns the data regarding the store. The returned *StoreWithEnterpriseTO* contains the id, name and location of the store. The service *getProductsWithLowStock* returns a list of stock items which are currently in low stock. Each *ProductWithStockItemTO* contains a stock item and its appropriate product description. The service *getAllProducts* returns a list of all products in the data base with the suppliers from which the products can be ordered. The service *getAllProductsWithOptionalStockItem* returns returns a list of all products in the data base with the suppliers from which the products can be ordered and the items of that product which are on stock. The service *orderProducts* allows the store manager to issue an order(*complexOrder*) for certain products. The service *getOrder* returns the ComplexOrderTO object with the id (*orderid*) given by the store manager. The service *rollInReceivedOrder* allows the store manager to mark an order (*complexOrderTO*) as received and add the ordered items to the inventory. The service *changePrice* allows the store manager to change the price of a stock item (*stockItemTO*). The service *markProductsUnavailableInStock* allows the store manager to mark some given products(*requiredProductsAndAmount*) as unavailable in stock.

3.3.7 CashDeskConnectorIf

The interface *CashDeskConnectorIf* is provided by the application component of the store server and required by the cash desk. Its services allow for the registration of a new sale and for getting relevant product data for a given barcode.

The interface offers the following services:

- `void bookSale(saleTO sale)`

- `ProductWithStockItemTO getProductWithStockItem(int productBarcode)`

The service *bookSale* hands the data of a completed sale (*saleTO*) to the application server. The service *getProductWithStockItem* returns the data of a product (*ProductWithStockItemTO*) from a barcode (*productBarcode*).

3.3.8 PersistenceIf

The interface PersistenceIf is provided by the persistence server and required by the application component of the store server.

The interface offers the following service:

- `PersistenceContext getPersistenceContext()`

The service *getPersistenceContext* returns the PersistenceContext that can then be used by the store server when accessing inventory data.

3.3.9 StoreQueryIf

The interface *StoreQueryIf* is provided by the data component of the store server and required by the application component of the store server. The services it offers can be used to query the persistence server for data relevant to the operation of the store. The PersistenceContext object used by the services of this interface represents the connection to the database.

The interface offers the following services:

- `Store queryStoreById (int storeId, PersistenceContext pctx)`
- `Collection<Product> queryProducts (int storeId, PersistenceContext pctx)`
- `Collection<StockItem> queryLowStockItems (int storeId, PersistenceContext pctx)`
- `Collection<StockItem> queryLowStockItemsWithRespectToIncomingProducts (int storeId, PersistenceContext pctx)`
- `Collection<StockItem> queryAllStockItems (int storeId, PersistenceContext pctx)`
- `StockItem queryStockItem (int stockId, PersistenceContext pctx)`
- `StockItem queryStockItemById (int stockId, PersistenceContext pctx)`
- `ProductOrder queryOrderById (int orderId, PersistenceContext pctx)`
- `Product queryProductById (int productId, PersistenceContext pctx)`
- `Collection<StockItem> getStockItems (int storeId, int productId, PersistenceContext pctx)`

The service *queryStoreById* returns a store (*Store*) for a given store id (*storeIf*) and PersistenceContext (*pctx*). The service *queryProducts* returns a collection of products (*Collection<Product>*) which are available for a given store id (*storeId*) and PersistenceContext (*pctx*). The service *queryLowStockItems* returns a collection of items with low stock (*Collection<StockItem>*)for a given store id (*storeId*) and PersistenceContext (*pctx*). The service *queryLowStockItemsWithRespectToIncomingProducts* returns a collection of items (*Collection<StockItem>*) with low stock even when accounting for the already ordered products for a given store id (*storeId*) and PersistenceContext (*pctx*). The service *queryAllStockItems* returns a collection containing all stock items (*Collection<StockItem>*) for a given store id (*storeId*) and PersistenceContext (*pctx*). The service *queryStockItem* returns a stock item (*StockItem*) for a given stock id (*stockId*) and PersistenceContext (*pctx*). The service *queryStockItemById* returns a stock item (*StockItem*) for a given stock id (*stockId*) and PersistenceContext (*pctx*). The service *queryOrderById* returns an order (*ProductOrder*) for a order id (*orderId*) and PersistenceContext (*pctx*). The service *queryProductById* returns a product (*Product*) for a given product id (*productId*) and PersistenceContext (*pctx*). The service *getStockItems* returns a collection of stock items (*Collection<StockItem>*) for a given store id (*storeId*), product id (*productId*) and PersistenceContext (*pctx*).

3.4 Data Types

Besides primitive data types like int, the system also uses the following transfer objects (TOs) for the purpose of transferring the required data between the application layer and the GUI, but not any reference to an object.

- EnterpriseTO represents a trading enterprise with an id and a name.
- StoreTO represents a store with an id, name and a location.
- StoreWithEnterpriseTO represents a store of an enterprise, and contains the combined data of a StoreTO and an EnterpriseTO.
- ProductWithStockItemTO represents a stock item and it's appropriate product. The product has an id, barcode, a purchase price and a name. The stock item has an id, a sales price, an amount, a minimum stock and a maximum stock.
- ProductWithSupplierTO represents a product and it's appropriate supplier. The product has an id, barcode, a purchase price and a name. The supplier has an id and a name.
- ProductWithSupplierAndStockItemTO represents a stock item and it's appropriate product and supplier. The product has an id, barcode, a purchase price and a name. The stock item has an id, a sales price, an amount, a minimum stock and a maximum stock. The supplier has an id and a name.
- ComplexOrderTO represents an order. It has an id, a delivery date and an order date. It can also contain any number of ComplexOrderEntryTO which contain an amount and a ProductWithSupplierTO.

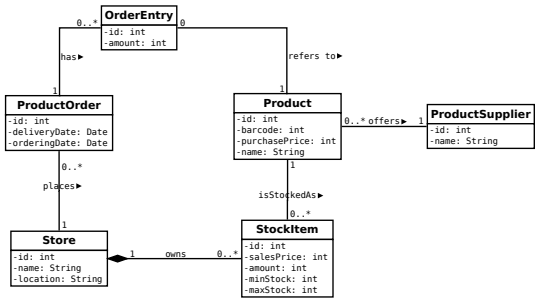

Figure 4: The data model of the system

- StockItemTO represents a stock item. It has an id, a sales price, an amount, a minimum stock and a maximum stock.
- ProductAmountTO and ProductMovementTO represent the quantity of a product that is to be ordered.

The system uses an object relationship mapping framework (Hibernate) to store objects in the database while hiding the communication with the database from the developer. (also see Figure 4):

- Store represents a store and has an id, name and a location.
- Product represents a type of items that can be stocked at the store. It has an id, a barcode number, a purchase price a name and a supplier.
- StockItem represents an item that can be sold at the store. It has an id, a price, an amount, and a maximum and minimum amount.
- ProductOrder represents an order of one or many products. It has an id, a delivery date and an order date. It can also contain one or more OrderEntry objects which contain an amount and a Product.

Other types that do not fit the two previous categories are the following:

- PersistenceContext represents a persistence context used by the Hibernate object-relational mapping framework. It is a cache that keeps a set of objects in the main memory and also manages a connection with the database.
- Acknowledgement represents the response given by the bank to a transaction request. It has a boolean flag that determines whether the transaction was successful.
- Account represents an account to/from which money can be transferred. It has an account number and an owner's name.

3.5 Modeling with the Palladio Component Model

The Palladio Component Model (PCM) [KBHR08] is a special language that allows the design and specification of component based software.

Functional and non-functional (e.g. performance or security) properties can be specified and simulations can be conducted at an early development stage of the software. The advantage of this approach is that shortcomings of the component based software will be identified early and thus save the costs of re-implementation that are usually incurred by defects found at a late development stage. The Palladio approach defines four roles that come into play during the development of the system and provides each of these roles with the possibility of specifying artifacts. The four roles supported by the PCM are

- *The component developer*, which can specify and implement components. Components are reusable parts of software systems that do not share a state. The functionality which they provide or require is specified using interfaces, which in turn offer services. Interfaces are associated with components using roles. The specification and implementation of the components, interfaces and roles are put inside a specification repository. Additionally these repositories can also contain additional data types.
- *The software architect* uses the contents of the repositories created by the component developers and compose an *assembly model* of a system. In the system composed this way, the components are instantiated in assemblies such that a component can be used multiple times in a system. The software architect also defines the interfaces that are provided and required by the system (the system roles). All artifacts created by the software architect are put inside an assembly model.
- *The system deployer* models the hardware architecture on which the system modeled by the software architect will run. This hardware model can then be used to simulate the performance of the system at an early design stage.
- *The domain expert* models the interaction of the user with the system. This model can be used in order to determine which parts of the system need to be tested in order to ensure a high reliability of the system.

In addition to these four roles an extension that is currently in development allows for the specification of information flow properties for services.

For this case study we take an existing repository of the CoCoME system and add interfaces that are to be used for the interaction of the system with the environment. We use the components, interfaces and types from this repository in order to create the assembly model of a system representing a supermarket. With the PCM confidentiality extension we specify the information flow constrains for all services.

3.6 Creating the Component Repository

In the first step of the PCM development approach, the components, interfaces, roles and data types are specified and deposited in a repository. We use an existing repository which already containes most of the components,

interfaces and data types presented in Section 3. As part of the current case study we added several new interfaces (see Section 3.2.) that model the interaction of the system with the environment. Figures 5 and 6 show the repository diagram on which the system is based. Note, that the repository contains components for enterprise functionality described in the CoCoME technical report that are not necessary for the current case study. The repository diagram is a visual representation of the component repository and shows what components and interfaces are available to a software architect in order to assemble a system.

3.7 Creating the Assembly Model

Following the second step of the PCM approach we use the components defined in the repository and composed them into an assembly model (which is shown in Figure 7). The assembly model is the system composed of the instantiated components, also called assemblies, from the repository.

The system roles associate the provided and required interfaces described in Section 3.2 with the system. The system provides the interfaces used to interact with the cash desk devices (printer, light display, barcode scanner, card reader and cash box) and with the store client. The only required interface by the system is the interface for communicating with the bank.

The components are taken from the component repository described in the previous section and instantiated as assemblies inside the assembly model. The system contains an assembly *CashDesk_CashDesk_1* representing the cash desk. The assemblies *Assembly_CashDesk.Cashbox*, *Assembly_CashDesk.BarCodeScanner*, *Assembly_CashDesk.Printer*, *Assembly_CashDesk.CardReader*, and *Assembly_CashDesk.LightDisplay* represent the cash desk devices and are part of the system. They interact with the cash desk through the interfaces described in Section 4.4.

The store server is composed of three assemblies: *Application.Store_Store_1_Server* representing the application component, *Data.Store_Store_1_Server* representing the data component, and *Data.Persistence_Store_1_Server* representing the persistence component. The application component is connected to the cash desk and communicates through the *CashDeskConnectorIf* interface. The data component of the store server provides the *StoreQueryIf* interface which offers functionality for querying the database of the store. The persistence component provides the connection to the database that can then be used by the other store server components. It is connected to the application component through the interface *PersistenceIf*.

The store client is represented by the *GUI.Store_Store_1_Client* assembly which is connected to the data component of the store server through the interface *StoreIf*.

4 Security Requirements

In this section we show the information flow security requirements for the system. We define the attackers based on the actors described in Section 3

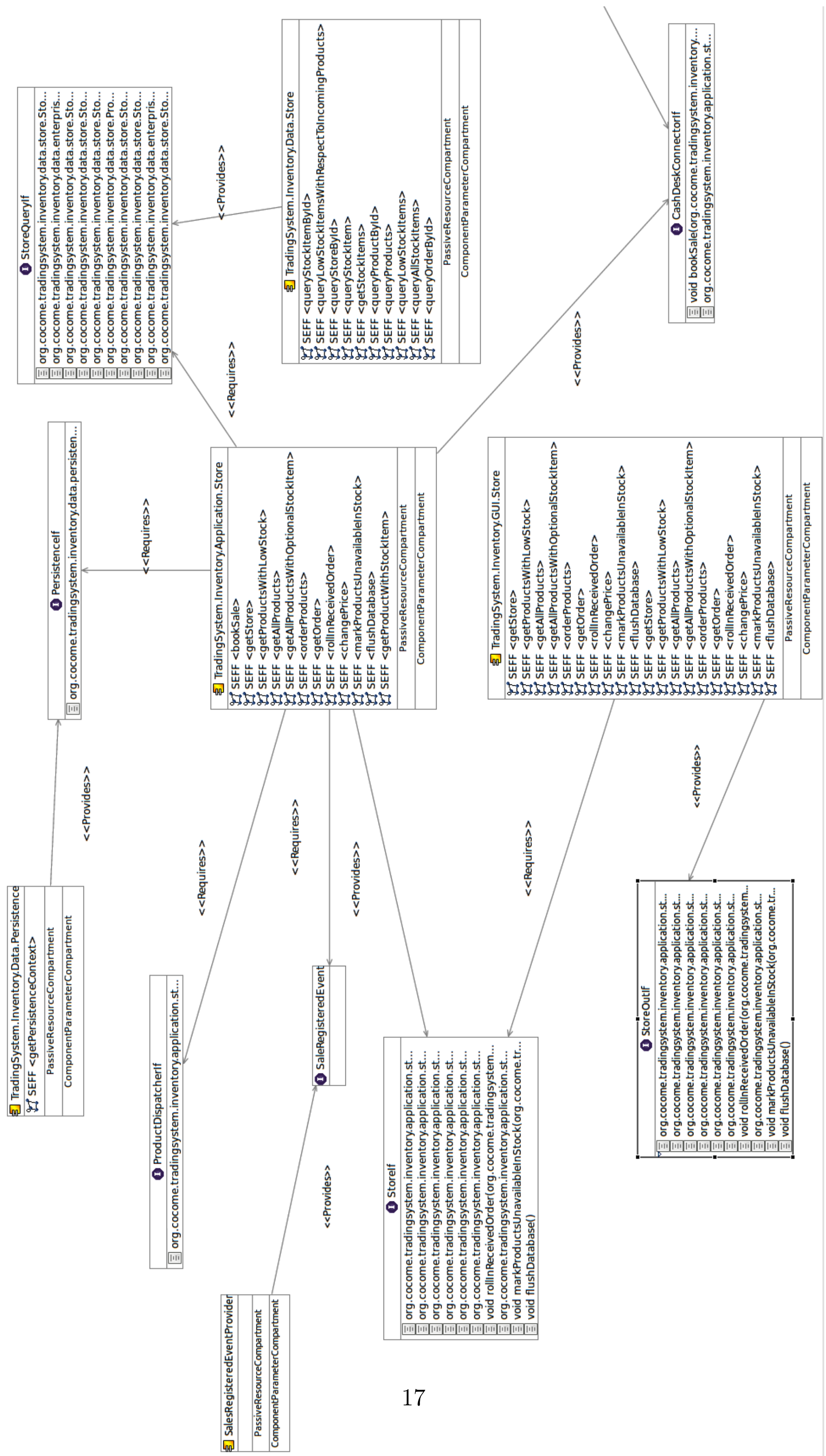

Figure 5: The repository diagram of cash desk components and interfaces

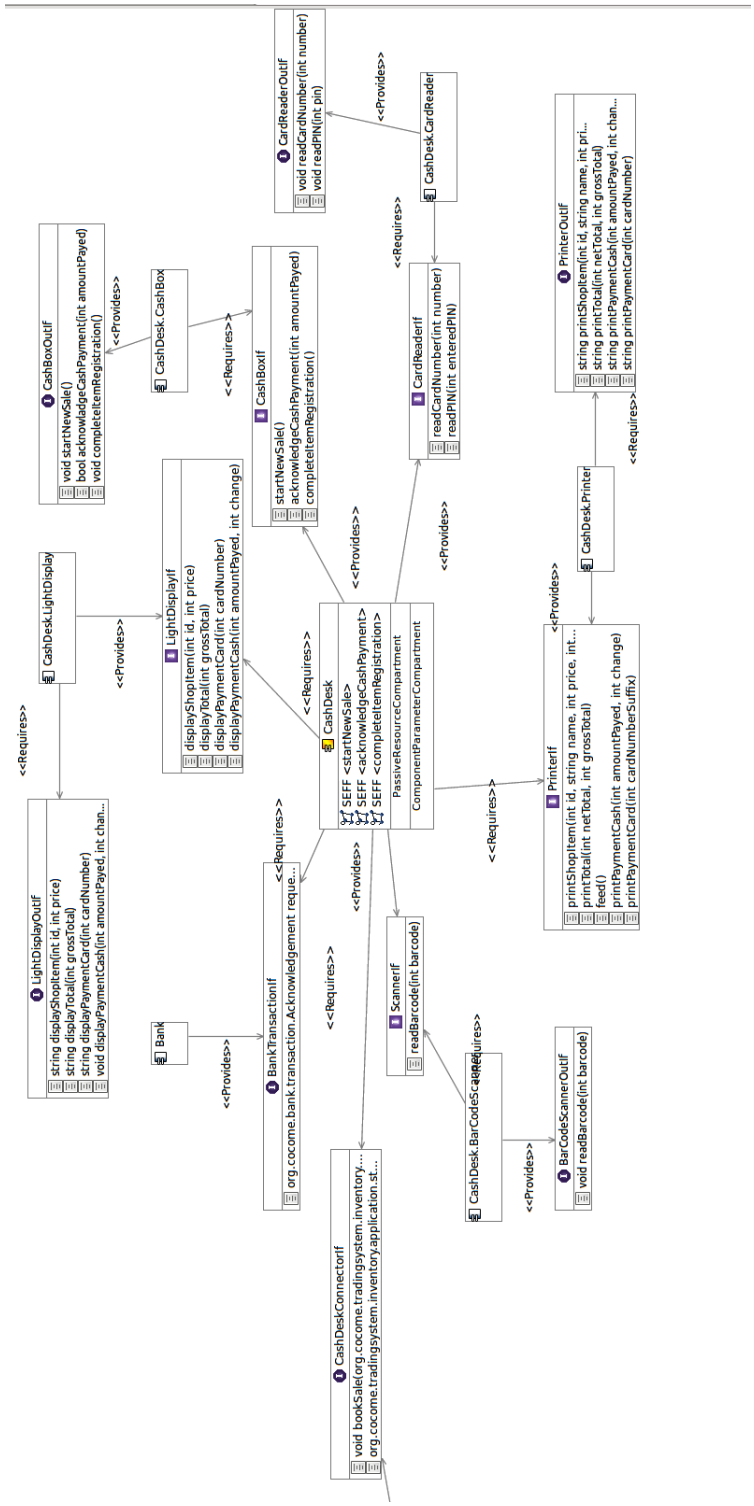

Figure 6: The repository diagram of store client and server components

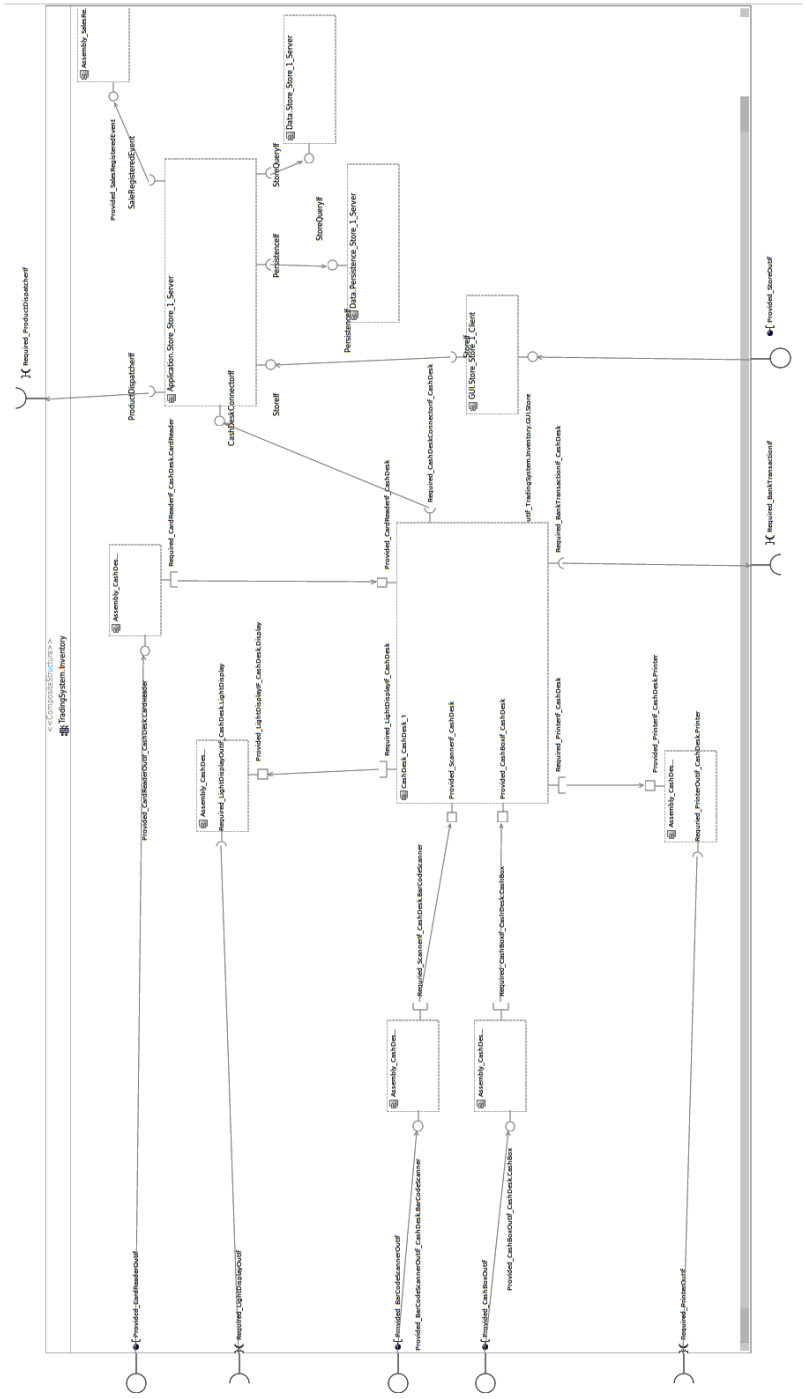

Figure 7: The system diagram of the model

and assign exactly one security domain for each attacker. For each service we assign security domains to its input parameters, return value, call event or expressions thereof. The meaning of these assignments is that only the attackers of the assigned security domains are allowed to have knowledge of the values of the contents of the security domains.

4.1 Attackers

We consider the actors described in Section 4.1 as possible attackers. For each attacker we define a security domain which holds the data to which the actor is allowed to know. Thus, we define the following security domains:

- Customer
- Cashier
- Bank
- ByStander
- StoreManager
- OtherStore

4.2 Provided Interfaces of the System

In this section we specify the security requirements for the provided interfaces of the system. The parameters of the services are inputs of the system while the return values of the services are outputs of the system. The attacker/domain from which an input originates is written in **bold**, while the other attackers that are also allowed to know that input. In addition to the service parameters found in the service signature, we use the $\backslash call$ and $\backslash result$ parameters representing the call event of a service and the return value of a service respectively.

4.2.1 Interface BarcodeScannerOutIf

This interface provides only one service for reading the bar code of an item. The input is made by the Cashier, this data may also be known by the Customer, ByStander and StoreManager since it is displayed on the LightDisplay, printed on the Printer and can also be found in the StoreServer from where it can be read by the StoreManager.

The following table provides the specification of the parameters of the services of the interface BarcodeScannerOutIf (see Section 3.2.4 for the functional description):

Service	Parameter	Domains
readBarcode	barcode	**Cashier**, Customer, ByStander, StoreManager

4.2.2 Interface CardReaderOutIf

The CardReaderOutIf interface provides two services for reading the credit card number and the pin of a credit card. The input is made in both cases by the Customer and other than him, only the Bank is allowed to know this data.

The following table provides the specification of the parameters of the services of the interface CardReaderOutIf (see Section 3.2.2 for the functional description):

Service	Parameter	Domains
readCardNumber	number	**Customer**, Bank
readCardNumber	number (last four digits)	**Customer**, Bank, Cashier, ByStander, StoreManager
readPin	pin	**Customer**, Bank

4.2.3 Interface CashBoxOutIf

This interface provides two services, startNewSale and completeItemRegistration which mark the start of a new sale, thus the beginning of the registration of the Customer's items and the completion of the item registration respectively. Since these two services don not have any parameters or return value the only information we consider is the call event of the services. The services are called by the cashier, but it can also be observed by ByStander and Customer who are physically at the cash desk. A third service provided by this interface is acknowladgeCashPayment which is called by the Cashier in order to input the the amount paid by the customer by cash. This amount may be also known by the ByStander and Customer.

The following table provides the specification of the parameters of the services of the interface CashBoxOutIf (see Section 3.2.3 for the functional description):

Service	Parameter	Domains
startNewSale	\call	**Cashier**, ByStander, Customer, StoreManager
completeItemRegistration	\call	**Cashier**, ByStander, Customer, StoreManager
acknowladgeCashPayment	amountPayed	**Cashier**, ByStander, Customer, StoreManager

4.2.4 Interface StoreOutIf

The interface StoreOutIf provides the services orderProducts, getOrder, rollInReceivedOrder, changePrice, and markProductsUnavailableInStock which allow the StoreManager to perform changes on the items in the inventory. All the inputs for these services are done by the StoreManager and he is the only one who may know this data. The services of StoreOutIf provide data regarding the current state of the store inventory. This data can only be read by the StoreManager on the StoreClient.

The following table provides the specification of the parameters of the services of the interface StoreOutIf (see Section 3.2.7 for the functional description):

Service	Parameter	Domains
orderProducts	complexOrder	**StoreManager**
getOrder	orderId	**StoreManager**
rollInReceivedOrder	complexOrderTO	**StoreManager**
changePrice	stockItemTO	**StoreManager**, Customer, Cashier, ByStander
markProducts-UnavailableInStock	requiredProducts-AndAmount	**StoreManager**
orderProducts	\result	StoreManager
getOrder	\result	StoreManager
rollInReceivedOrder	\result	StoreManager
changePrice	\result	StoreManager
markProducts-UnavailableInStock	\result	StoreManager
getStore	\result	StoreManager
getProductsWithLow-Stock	\result	StoreManager
getAllProducts	\result	StoreManager
getAllProductsWith-OptionalStockItem	\result	StoreManager

4.3 Required Interfaces of the System

4.3.1 Interface BankTransactionIf

The BankTransactionIf interface is required by the system and contains the service requestTransaction for requesting the transaction of the amount to be paid from the customer's account to the store account. The customer's account can be known only by the Customer and the Bank, the store's account can be known only by the Cashier, Bank, and StoreManager. The amount to be paid can be known by Cashier, ByStander, StoreManager, Customer, and Bank. Whether the transaction request was approved can be seen by the Customer, Cashier, ByStander and Bank.

The following table provides the specification of the parameters of the services of the interface BankTransactionIf (see Section 3.2.6 for the functional description):

Service	Parameter	Domains
requestTransaction	\result	**Bank**, Customer, Cashier, ByStander, StoreManager
requestTransaction	cardnumber	**Cashier**, Customer, StoreManager, Bank
requestTransaction	amount	**Cashier**, Customer, StoreManager, Bank, ByStander
requestTransaction	account	**Customer**, Bank

4.3.2 Interface PrinterOutIf

The PrinterOutIf interface provides four services for printing each shop item, the total amount to be paid, the amount paid by cash and the last four digits of the credit card number if a credit card was used. This data can be read by the Customer and the Cashier.

The following table provides the specification of the parameters of the services of the interface PrinterOutIf (see Section 3.2.5 for the functional description):

Service	Parameter	Domains
printShopItem	\result, id, name, price, vat	Customer, Cashier, ByStander
printTotal	\result, netTotal, grossTotal	Customer, Cashier, ByStander
printPaymentCash	\result, amountPayed, change	Customer, Cashier, ByStander
printPaymentCard	\result, cardnumber (last four digits)	Customer, Cashier, ByStander

4.3.3 Interface LightDisplayOutIf

The LightDisplayOutIf interface provides four services for printing each shop item, the total amount to be paid, the amount paid by cash and the last four digits of the credit card number if a credit card was used. This data can be read by the Customer, Cashier and ByStander.

The following table provides the specification of the parameters of the services of the interface LightDisplayOutIf (see Section 3.2.1 for the functional description):

Service	Parameter	Domains
displayShopItem	\result, id, price	Customer, Cashier, ByStander
displayTotal	\result, grossTotal	Customer, Cashier, ByStander

4.3.4 Interface ProductDispatcherIf

The ProductDispatcherIf offers a service for ordering products at another store.

The following table provides the specification of the parameters of the services of the interface LightDisplayOutIf (see Section 3.2.8 for the functional description):

Service	Parameter	Domains
orderProductsAvailable-AtOtherStores	enterpriseTO, callingStore, productAmounts, \result	**StoreManager**, OtherStore

4.4 Interfaces of the Internal Components

For the internal interfaces of the system we assign domains (with each domain belonging to exactly one attacker) to the service parameters and expressions of service parameters. An expression may be known by an attacker if it is in the domain of that attacker. The internal interfaces are required and provided by the components of the system, we will describe their security requriements from the provided perspective, i.e. the parameters of the services are considered inputs and the return values of the services are treated as outputs.

4.4.1 Interface LightDisplayIf

The following table provides the specification of the parameters of the services of the interface LightDisplayIf (see Section 3.3.1 for the functional description):

Service	Parameter	Domains
displayShopItem	id, price	Customer, Cashier, ByStander, StoreManager
displayTotal	grossTotal	Customer, Cashier, ByStander, StoreManager

4.4.2 Interface PrinterIf

The following table provides the specification of the parameters of the services of the interface PrinterIf (see Section 3.3.5 for the functional description):

Service	Parameter	Domains
printShopItem	id, price, name, vat	Customer, Cashier, ByStander, StoreManager
printTotal	netTotal, grossTotal	Customer, Cashier, ByStander, StoreManager
printPaymentCash	amountPaid, change	Customer, Cashier, ByStander, StoreManager
printPaymentCard	cardNumber (last four digits)	Customer, Cashier, ByStander, StoreManager

4.4.3 Interface ScannerIf

The following table provides the specification of the parameters of the services of the interface ScannerIf (see Section 3.3.4 for the functional description):

Service	Parameter	Domains
readBarcode	barcode	Cashier, Customer, ByStander, StoreManager

4.4.4 Interface CardReaderIf

The following table provides the specification of the parameters of the services of the interface CardReaderIf (see Section 3.3.2 for the functional description):

Service	Parameter	Domains
readCardNumber	number	Customer, Bank
readCardNumber	number (last four digits)	**Customer**, Bank, Cashier, ByStander, StoreManager
readPin	pin	Customer, Bank

4.4.5 Interface CashBoxIf

The following table provides the specification of the parameters of the services of the interface CashBoxIf (see Section 3.3.3 for the functional description):

Service	Parameter	Domains
startNewSale	\call	Cashier, ByStander, Customer, StoreManager
completeItemRegistration	\call	Cashier, ByStander, Customer, StoreManager
acknowladgeCashPayment	amountPayed	Cashier, ByStander, Customer, StoreManager

4.4.6 Interface CashDeskConnectorIf

The following table provides the specification of the parameters of the services of the interface CashDeskConnectorIf (see Section 3.3.7 for the functional description):

Service	Parameter	Domains
bookSale	saleTO	Cashier, ByStander, Customer, StoreManager
getProductWithStockItem	barcode	Cashier, ByStander, Customer, StoreManager

4.4.7 Interface StoreIf

The following table provides the specification of the parameters of the services of the interface StoreIf (see Section 3.3.6 for the functional description):

Service	Parameter	Domains
orderProducts	complexOrder, \result	StoreManager
getOrder	orderID, \result	StoreManager
rollInReceivedOrder	complexOrderTO, \result	StoreManager
changePrice	stockItemTO, \result	StoreManager, Customer, Cashier, ByStander
markProducts-UnavailableInStock	requiredProducts-AndAmount, \result	StoreManager
getStore	\result	StoreManager
getProductsWith-LowStock	\result	StoreManager
getAllProducts	\result	StoreManager
getAllProducts-WithOptionalStockItem	\result	StoreManager

4.4.8 Interface PersistenceIf

The following table provides the specification of the parameters of the services of the interface PersistenceIf (see Section 3.3.8 for the functional description):

Service	Parameter	Domains
getPersistenceContext	\result	StoreManager

4.4.9 Interface StoreQueryIf

The following table provides the specification of the parameters of the services of the interface StoreQueryIf (see Section 3.3.9 for the functional description):

Service	Parameter	Domains
queryStoreById	\result, storeId, pctx	StoreManager
queryProducts	\result, storeId, pctx	StoreManager, Cashier, Customer, ByStander
queryLowStockItems	\result, storeId, pctx	StoreManager
queryLowStock-ItemsWRIP	\result, storeId, pctx	StoreManager
queryAllStockItems	\result, storeId, pctx	StoreManager
queryStockItem	\result, stockId, pctx	StoreManager
queryStockItemById	\result, stockId, pctx	StoreManager
queryOrderById	\result, orderId, pctx	StoreManager
queryProductById	\result, productId, pctx	StoreManager, Cashier, Customer, ByStander
getStockItems	\result, storeId, productId, pctx	StoreManager

4.5 Specifying the Information Flow Properties in Palladio

The Palladio confidentiality extension allows the software architect to specify information flow properties. The process that needs to be followed in order to specify information flow properties is the following:

1. Definition of attackers
2. Definition of data sets
3. Assignment of parameters (or expressions thereof) of services to data sets
4. Application of the parameter - data set associations to the services

In the first step we use the extension of the PCM-Bench to define the possible attackers. Depending on the context each actor can be considered an attacker, thus the attackers correspond to the actors presented in Section 4.1.

The second step of the specification process requires the definition of data sets. These data sets abstractly represent data belonging to the same security domain. We defined exactly one data set for each attacker.

In the third step we assign service parameters, return and call events and expressions thereof to data sets using the *ParametersAndDataSets* construct. In the current case study we use the method parameter name to represent the parameters, \result for the return value of a service and \call for the call event of a service. The only instance in which we used an expression was *number % 10000* signifying the last four digits of the parameter *number*. The PCM bench extension for specifying information flow properties does not require any syntax for the expressions that are

assigned to domains. For this case study we use side-effect free Java expressions. For each method we created one or more ParametersAndDataSets. In order to keep the specification as simple as possible we minimized the number of ParametersAndDataSets, thus only services where two or more parameters must be in different data sets did have more than one ParametersAndDataSets.

In the final step we assign one or more ParametersAndDataSets constructs to each service.

As a result of the specification of information flow properties we assign one or more domains to each parameter and return values of the services of all interfaces. Additionally for some services we assign domains for call events and for expressions of parameters. By construction the defined domains have exactly one corresponding attacker (or actor). The assignments of domains to parameters, return values, call events or expressions of parameters correspond to the descriptions of the security requirements of services that can be found in Sections 4.2, 4.3 and 4.4.

In order for the system to fulfill the specified information flow properties, each attacker that knows only the data belonging to his own domain before the execution will not be able to infer anything about the data not belonging to the security domain of the attacker. Thus a tool that analyzes the specified system must perform an analysis for each service and for each attacker. The expressions that are contained in the security domain of the analyzed attacker must be considered low, while all other expressions will have to be considered high. Of course, the noninterference property analyzed by the tool will have to be compositional, i.e. if the property holds for all services of a component it must hold for any sequence of service calls of that component. The computational model and definitions of components, services and noninterference property described in [GG16] can, with a minor adjustment be used as a semantics for the specification described in this document. The minor adjustment that needs to be addressed is the fact that in [GG16] all services need to have at least one parameter, while in our model there are such services without parameters. The reason the semantics can be used anyway is that services without parameters from our model can be translated as services with a dummy parameter in the semantics.

The modeled system can be downloaded here: `http://formal.iti.kit.edu/~herda/pub/cocome.zip`

5 Conclusion

In this technical report we describe the modeling of a component-based system that represents the It infrastructure of a supermarket. The system is composed of a cash desk that has a printer, light display, barcode scanner, card reader and a cash box. The cash desk is connected to a store server which is also connected to a store client. The system model is based on CoCoME, a component based example system. We describe the functional requirements of the services of the components' interfaces.

The goal is to specify security requirements that can be then analyzed by various tools. We consider several existing actors from the use cases of the

system (as described in [HKW+08]) and define two new actors, ByStander and OtherStore, which can be considered as attackers in different scenarios. We specify security domains for these actors and then specify the data belonging to the security domain of each actor.

We used Palladio Bench, a software modeling tool that supports the modeling and simulation of component-based software in order to model the system. We use an extension of the Palladio Bench that supports the specification of information flow property to specify the security requirements.

Our future goal is to analyze the system described in this document and to check whether the specified information flow properties hold in a proof-of-concept implementation of the system.

References

[GG16] Simon Greiner and Daniel Grahl. Non-interference with what-declassification in component-based systems. In *Proceedings of the Computer Security Foundations Symposium (CSF 2016)*, June 2016.

[HKW+08] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. The Common Component Modeling Example. In Andreas Rausch, Ralf H. Reussner, Raffaela Mirandola, and Frantisek Plasil, editors, *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*, chapter CoCoME – The Common Component Modeling Example, pages 16–53. Springer-Verlag Berlin Heidelberg, 2008.

[KBHR08] Heiko Koziolek, Steffen Becker, Jens Happe, and Ralf Reussner. Evaluating performance of software architecture models with the palladio component model. In Jörg Rech and Christian Bunse, editors, *Model-Driven Software Development: Integrating Quality Assurance*, pages 95–118. IDEA Group Inc., December 2008.