9 Modelling the CoCoME with the JAVA/A Component Model*

Alexander Knapp[1], Stephan Janisch[1], Rolf Hennicker[1], Allan Clark[2], Stephen Gilmore[2], Florian Hacklinger[1], Hubert Baumeister[3], and Martin Wirsing[1]

[1] Institut für Informatik, Ludwig-Maximilians-Universität München
{hacklinger, hennicker, janisch, knapp, wirsing}@ifi.lmu.de

[2] Laboratory for Foundations of Computer Science, University of Edinburgh
{a.d.clark, Stephen.Gilmore}@ed.ac.uk

[3] Informatik og Matematisk Modellering, Danmarks Tekniske Universitet, Lyngby
hub@imm.dtu.dk

9.1 Introduction

The JAVA/A approach aims at semantically well-founded and coherent modelling and programming concepts for components: based on sound theoretical foundations it enhances the widely used UML 2.0 component model by modular analysis and verification techniques and a Java-based architectural programming language.

9.1.1 Goals and Scope of the Component Model

The JAVA/A component model, formally characterised within the algebraic semantic framework of [1], is inspired by ideas from "Real-Time Object Oriented Modeling" (ROOM [2]): components are strongly encapsulated, instantiable behaviours and any interaction of components with their environment is regulated by ports. We took up the ROOM model in its version integrated into the "Unified Modeling Language 2.0" (UML 2.0 [3]), though in a simplified form which, however, keeps the main structuring mechanisms and focuses on strong encapsulation as well as hierarchical composition.

In contrast to interface-based component approaches (like COM, CORBA, Koala, KobrA, SOFA; see [4] for an overview), the primary distinguishing feature of ROOM, and hence of the JAVA/A component model, is the consistent use of ports as explicit architectural modelling elements. Ports allow designers to segment the communication interface of components and thus the representation of different "faces" to other components. Moreover, ports are equipped with behavioural protocols regulating message exchanges according to a particular viewpoint. Explicit modelling of port behaviours supports a divide-and-conquer approach to the design of component-based systems on the level of components and also on the level of particular views to a component.

* This research has been partially supported by the EC 6th Framework project SENSORIA "Software Engineering for Service-Oriented Overlay Computers" (IST 016004) and the GLOWA-Danube project (01LW0303A) sponsored by the German Federal Ministry of Education and Research.

Furthermore, taking components to be strongly encapsulated behaviours communicating through ports fosters modular verification, which is one of the aims of the JAVA/A approach. Using our semantic foundations, we employ the model checking tools HUGO/RT and LTSA to verify that components comply to their ports and that connected ports can communicate successfully; by a compositionality theorem we can lift these properties to hierarchical, composite components. Starting with deadlock-free component and port behaviours, we verify the absence of deadlocks in composed behaviours. Since HUGO/RT and LTSA are general model checking tools we may also analyse arbitrary temporal logical formulae. For quantitative properties, we represent the component semantics, though rather abstractly, in the PEPA process algebra [5] and use continuous-time Markov chains for performance analysis with the IPC tool [6].

The second aim of the JAVA/A approach is the representation of software architecture entities in a programming language. For this purpose we introduced an "architectural programming" language, JAVA/A [7], which allows programmers to represent software architectures directly in the implementation language and thus helps to prevent "architectural erosion" [8]. We used the code generation engine HUGO/RT to translate the behaviour of components into Java code; the code generation is not formally verified but strictly adheres to the UML-semantics of the state machines. The usage of the same tool for verification and code generation helps in transferring design properties into the code. The generated Java code is then embedded into JAVA/A source code, which in turn is compiled and deployed to yield the executable system. The deployment is not described separately from the logical behaviour of the components.

9.1.2 Modelled Cutout of the CoCoME

We present selected parts of our model of the CoCoME trading system in Sect. 9.3 including the embedded system part with the CashDeskLine and the information system part with the Inventory. In particular we treated the structural aspects of CoCoME completely. The functional (behavioural) aspects were modelled also completely for the embedded system part of the CashDeskLine and partly for the information system part with the Inventory at its core. We touched only upon non-functional aspects by provision of an example in the context of express checkouts. We did not specify pure data-based specifications such as invariants and pre-/post-conditions but focused on the control flow behaviours of ports and components instead.

All deviations from the original architecture of the CoCoME are discussed in detail in Sect. 9.3.1. Our approach does not directly allow for n-ary connectors and broadcast messages between components. In both cases adaptor components need to be applied; as far as the modelling of the CoCoME is concerned we did not feel this to be a severe restriction. The full description of our model of the CoCoME as well as more details on the theoretical background for the functional analysis can be found on our web page [9].

9.1.3 Benefits of the Modelling

In Sect. 9.4 we show how the formal algebraic basis of our model allows one to thoroughly analyse and verify important qualitative and quantitative properties. In particular, we show that the composite component CashDeskLine is correct and deadlock free; as an example for performance analysis we study the use of express checkouts and show that their advantage is surprisingly small. In Sect. 9.5 we briefly present the model checking and architectural programming tools we used for the CoCoME.

9.1.4 Effort and Lessons Learned

We spent about two and a half person months on modelling, analysing and implementing the CoCoME with the JAVA/A component model and the JAVA/A architectural programming language. A completion of the modelling endeavour, including, in particular, data-based specifications would take about two more person months. However, an inclusion of all non-functional requirements would require some more work on the foundational aspects of the JAVA/A component model.

The CoCoME helped us on the one hand to consolidate the JAVA/A metamodel. On the other hand, we gained important insights into glass-box views for composite components and on the formulation of criteria for modular verification on the basis of a port based component model. We plan to extend our model with data-based specifications (invariants, pre-/post-conditions) to develop and study an approach to state/event-integration.

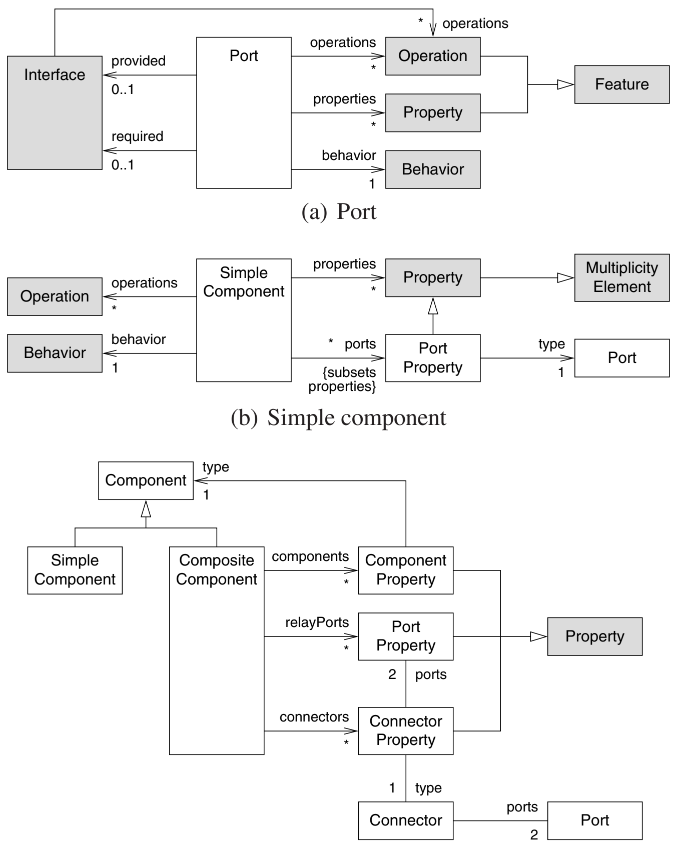
9.2 Component Model

In the JAVA/A component model, components are strongly encapsulated behaviours. Only the exchange of messages with their environment according to provided and required operation interfaces can be observed. The component interfaces are bound to ports which regulate the message exchange by port behaviour and ports can be linked by connectors establishing a communication channel between their owning components. Components can be hierarchical containing again components and connections.

In the following we briefly review JAVA/A's component metamodel, see Fig. 1. Although being based on corresponding concepts in the UML 2.0 and, in fact, easily mappable, we at least strive to give a more independent definition which only relies on well-known UML 2.0 concepts. In Fig. 1, UML concepts modified by the JAVA/A component model are shown with a white background while unmodified UML concepts are shown with grey background. We assume a working UML 2.0 knowledge [3] when explaining some of the specialities of our modelling notation and semantics. An algebraic semantics framework for the JAVA/A component model can be found in [1].

Port. A *port* (see Fig. 1(a)) describes a view on a component (like particular functionality or communication), the operations offered and needed in the context of this view and the mandatory sequencing of operation calls from the outside and from the inside. The operations offered by a port are summarised in its *provided interface*; the operations needed in its *required interface*. The sequencing of operations being called from and on a port is described in a *port behaviour*. Any auxiliary attributes and operations in order to properly define the port behaviour are added as internal features.

As an example, consider the port C-CD (showing stereotype «port») in Fig. 12 describing the coordinator's view on a cash desk: Its provided and required interfaces (attached to the port by using the ball and socket notation, respectively) declare only



(a) Port

(b) Simple component

(c) Composite component

Fig. 1. JAVA/A component metamodel

asynchronous operations (i.e., on calling these operations the caller does not wait for the callee to handle the call; for succinctness we use a stereotype ≪async≫ to express that all operations of an interface are meant to be asynchronous); it also shows an internal attribute for storing the identity of a cash desk. The behaviour of port C-CD is described by the UML state machine in Fig. 12 (top right). Besides the UML state machine features we use, on the one hand, a special completion trigger ≪tau≫ for modelling internal choice (of the port's owning component) which is enabled on state completion but, in contrast to completion triggers (used for those transitions not showing a trigger), is not prioritised over other events. On the other hand, we assume all state machines to behave like UML 2.0's protocol state machines in the sense that it is an error if an event occurs in a state where it cannot be handled by one of the outgoing transitions.

Simple component. A *simple component* (see Fig. 1(b)) consists of port properties (sometimes referred to as port declarations), internal attributes and operations, and a behaviour linking and using these ingredients. Port properties (like all properties) are equipped with a *multiplicity*, setting lower and upper bounds on how many port instances of a particular type exist during runtime, permitting dynamic reconfiguration.

As an example, consider the simple component Coordinator (showing stereotype ≪component≫) in Fig. 12. Besides an internal attribute and two internal operations it declares a port property cds of type C-CD with unlimited multiplicity. The behaviour of Coordinator is laid down in the state machine of Fig. 12 (bottom right). In fact, as indicated by the stereotype ≪orthogonal≫ with tag { param = cd : cds }, this behaviour description abbreviates a composite orthogonal state with as many orthogonal regions (containing the given behaviour) as there are currently port instances in cds. Note also that the internal operation updateSaleHistory is declared to be { sequential }, that is, all calls to this operation are sequentialised.

Composite component. A *composite component* (see Fig. 1(c)) groups components, simple as well as composite, by declaring component properties, and connectors between ports of contained components, by declaring connector properties. A *connector* (which is always binary) describes the type of a connector property which links two port properties such that the ports of the connector match the ports of the port properties. Composite components do not show a behaviour of their own, their behaviour is determined by the interplay of the behaviours of their contained component instances and the connections, i.e., connector instances. The ports offered by a composite component are exclusively *relay ports*, i.e., the mirroring of ports from the contained components which must not be declared to be connected by some connector property.

As an example, consider the composite component CashDeskLine (showing stereotype ≪component≫) in Fig. 6. It declares, via the component properties cashDesks and coordinator, components CashDesk and Coordinator as sub-components where each instance of CashDeskLine must show at least one instance of CashDesk. Port property co of CashDesk is connected to port property cds of Coordinator meaning that at runtime each port instance in co of a cash desk in cashDesks is connected to a port instance in cds of coordinator. The port declarations i and b of CashDesk are relayed. However, as in fact there may be several cash desks but there is to be only a single port instance of CDA-Bank we would have to introduce an adapter component which declares a port with multiplicity 1 to be relayed to the outside of CashDeskLine and a port with multiplicity 1..* (matching the multiplicity of cashDesks) to be connected to the different b instances of the different cashDesks. We abbreviate this by using the

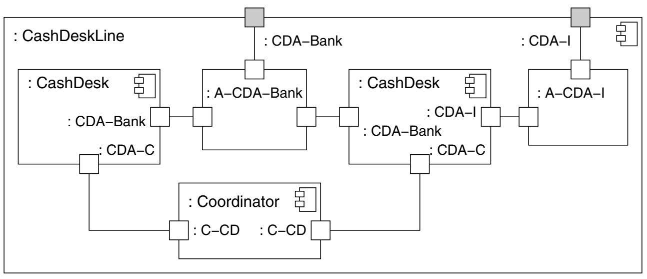


Fig. 2. Sample configuration of component CashDeskLine of Fig. 6

stereotype ≪adapter≫ on connector declarations; in particular, as indicated by the tagged value { kind = ”seq” }, the adapter component sequentialises the different calls.

A sample runtime configuration of the composite component CashDeskLine is given in Fig. 2. This configuration shows two instances of component CashDesk and a single instance of component Coordinator. The CDA-C port instances of the cash desks are connected to two different instances of the C-CD coordinator port. The CDA-Bank port instances of the cash desks are adapted to one relay port instance of the cash desk line by the auxiliary adapter component A-CDA-Bank. Similarly, the CDA-I port instance of the cash desk to the right (the other cash desk does not show a CDA-I port instance, in accordance with the multiplicity of the port feature declaration i) is relayed.

9.3 Modelling the CoCoME

We started the design of the CoCoME from the use case descriptions and sequence diagrams as given in [10]. After and during static structure modelling for simple (non-composite) components we designed component and port behaviours hand in hand, in case of the embedded system part accompanied by formal analysis. Finally, the simple components were applied for the design of the composite components, yielding a first draft of the complete architecture. Within the next iterations, the alternative and exceptional processes of the use case descriptions were taken into account to extend and correct the initial design. In case of ambiguous or unclear requirements our design followed the prototype implementation of the CoCoME. Since we fully agree with the data model provided in [10], we omit specifications of data types, transfer objects for data exchange and enumerations.

Our specifications comprise UML 2.0 class and composite structure diagrams to specify the static structure of components, ports and interfaces; and UML 2.0 state machine diagrams to specify the behavioural view for ports and components. Familiarity with terms, notions and functional requirements of the trading system [10] is assumed. For lack of space we do not discuss the specifications of components and ports shown in grey in Figs. 3–7. Instead we restrict our attention on a detailed presentation of the embedded system part which is also the focus of our functional analysis. Additionally we provide a representative extract of our model for the information system part of the CoCoME. Specifications not discussed here can be found on our web page [9].

9.3.1 Architectural Deviations

Two of the essential features of our component model are, on the one hand, its strict use of ports as first-class citizen to encapsulate (parts of) component behaviour and, on the other hand, the distinction between component, port types respectively and their instantiation. These features enabled us to model some aspects of the trading system in a more convenient way. In the following we describe and justify structural deviations from the original modelling along Fig. 3. On the left-hand side the component hierarchy as described in [10] is shown. The right-hand side shows the corresponding modelling within our approach. From deviations with our static model almost naturally some deviations in behavioural aspects follow. We omit a detailed discussion here and refer to [9] for component-wise details on this issue.

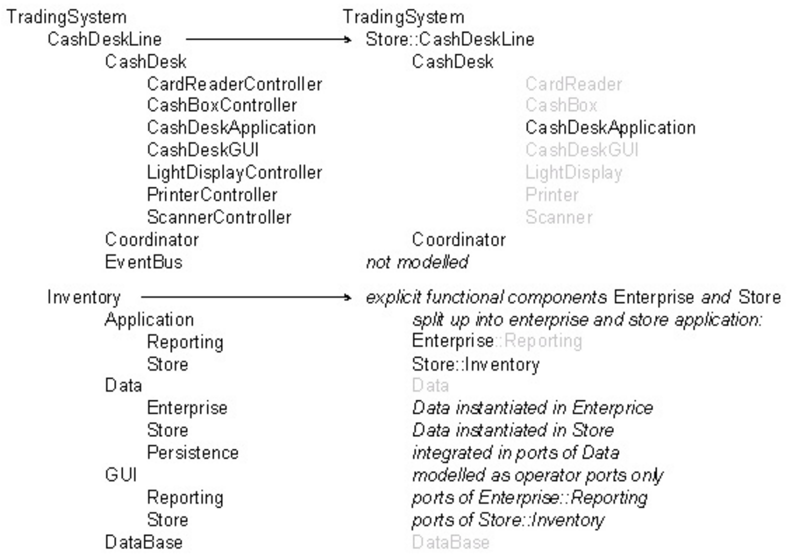


Fig. 3. Component hierarchy of given (left) and modelled (right) architecture

As the most important deviation in the embedded system part we did not model the event bus component, which is used in the CoCoME for the communication between the subcomponents of the cash desk line on the one hand, and between those, the coordinator and external components on the other hand. Instead of a functional component, our approach provides explicit models of component communications and interactions using port and component behaviour specifications. We consider the integration of an event bus to be an implementation decision where the implementation should guarantee that the specified communication structure of the design model is respected.

Explicit modelling of the cash desk's internal interaction structure constitutes the internal topology of our composite component CashDesk (see Fig. 7) deviating from the cash desk's inner structure as shown in [10, Fig. 6]. During the modelling of the subcomponent's communication it soon became apparent that, in our approach, the most appropriate topology for the CashDesk is the one specified in Fig. 7. The central functionality of handling sales almost always requires the cash desk application to receive some signal or message respectively from an "input" component such as the cash box and to send a corresponding notification to an "output" component such as the cash desk GUI or the printer.

Furthermore we dropped the distinction of functional and controller components within the cash desk component of the CoCoME. In our approach, controller components such as the CashBoxController [10, e.g. Fig. 6, 6], linking the middleware and the hardware devices, could be modelled with ports.

The main structural deviation from the CoCoME requirements of the information system part concerns the layered approach to the modelling of the component Inventory on the left-hand side of Fig. 3. The layers Application, Data and GUI (represented by components) distinguish between an "enterprise part" and a "store part": within the component Data this distinction is manifested directly in the components Enterprise and Store. Within the components Application and GUI the distinction is between Reporting and Store. The former is, according to the deployment of Inventory::Application::Reporting on the EnterpriseServer (see [10, Fig. 6]), not only part of the enterprise context but also located at the store server (as part of Inventory::Application). However, according to the use case descriptions and sequence diagrams of [10], Reporting seems actually not to be used in the store context. In fact, the functionality of this component is exclusively to generate reports for the enterprise manager. Therefore we decided to model Store and Enterprise as functional components on their own. An enterprise may contain a number of stores comprising an inventory and a cash desk line. Reporting then is part of Enterprise but not of Store as this seems to model the application domain of the required trading system more naturally. The Data layer of the CoCoME is represented by the component Data with ports modelling the enterprise and the store related data aspects. Notice that, as required in the CoCoME, different instances of the Data component may share the same instance of a DataBase component. This issue depends on a concrete system configuration only. Last, the GUI layer is represented by the operator ports of the components Enterprise and Store.

Further structural deviations concern the original component Data::Persistence which is in our approach not modelled explicitly but integrated with the port modelling of the Data component instead. Also, the sequence diagram [10, Fig. 6] concerning the product exchange among stores of the same enterprise (Use Case 8) shows a component ProductDispatcher which is not mentioned in the structural view of the CoCoME. We modelled this component as part of the enterprise component.

9.3.2 Trading System — Stores and Enterprises

This section describes the specifications for the root component TradingSystem as well as the two fundamental components Stores and Enterprises, all of them being composed from further components.

TradingSystem. The composite component TradingSystem in Fig. 4 provides flexible instantiation possibilities for different system configurations. As will be evident from the specifications of the composite components Enterprise and Store described hereafter, the former contains, among others, a number of stores and a Store in turn contains further components such as the CashDeskLine. In fact a system configuration following the hierarchy further down from one instance of Enterprise already suffices to meet the original CoCoME requirements for a trading system with an enterprise and a number of stores which belong to this enterprise. In this case the sets of simpleStores and simpleStoresDB would be empty as these are used only in case of extended system configurations with stores independent from any enterprise.

The bank component, required for card payment at the cash desks of a store, is considered external to the trading system. Therefore the component TradingSystem declares a relay port of type CDA-Bank to delegate incoming and outgoing communications between a bank and internal components, respectively. The port multiplicity with a lower bound of 1 of indicates that for a proper instantiation of TradingSystem it is strictly required to provide appropriate bank connections to the system.

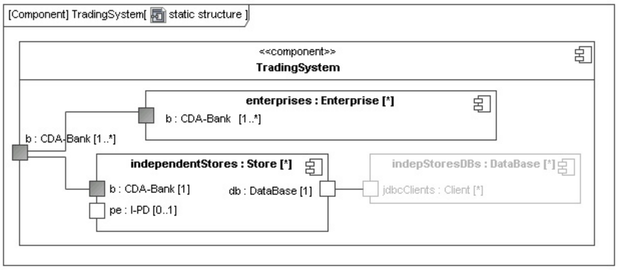


Fig. 4. Static structure of the component TradingSystem

Beyond, the component TradingSystem allows for several further system configurations. For example the system might be used with store components which are independent of any enterprise. On this account the TradingSystem contains a set of stores (simpleStores), each of them connected to its own instance of a data base (simpleStoresDB). Since the port feature pe:I-PD is required only for stores belonging to an enterprise, the port must not be connected in this case.

Enterprise. An enterprise is modelled by the composite component Enterprise in Fig. 5. It consists of a number of stores, of a component Reporting which provides means to create several reports from the enterprise manager's point of view, of a component Data as an intermediate layer between Reporting and the concrete DataBase which

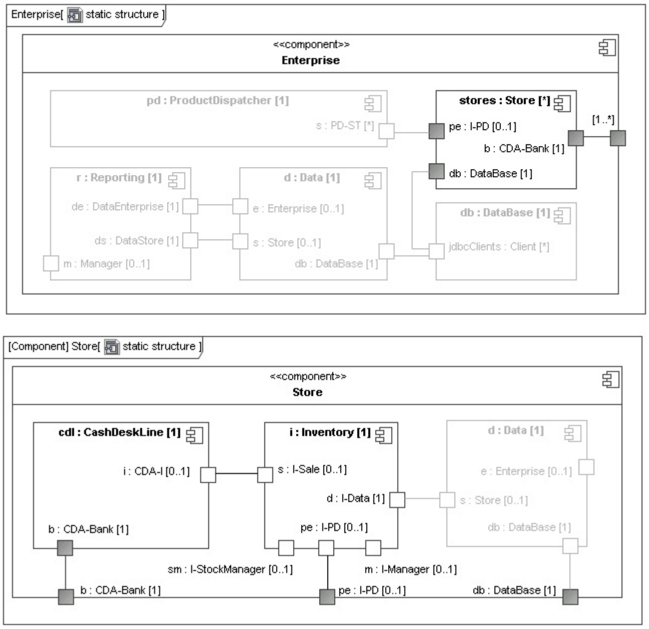


Fig. 5. Static structure of the components Enterprise and Store

is also instantiated as part of an Enterprise and shared between Data and the stores, and finally of a component ProductDispatcher to coordinate the product exchange among the stores of the enterprise.

In order to provide connections for the bank ports b of the contained stores, the component uses a relay port with multiplicity 1..* and instantiates one port for each store. Hence, in contrast to the cash desks as part of the CashDeskLine (Fig. 6) the different stores of an enterprise do not share the same bank connection.

Store. As depicted in Fig. 5 the component Store is a composition of a CashDeskLine (see Sect. 9.3.3), an Inventory (see Sect. 9.3.4) and an instance of the component Data. The inventory is connected to the Data instance hiding the concrete data base from the application as required in the CoCoME. In contrast to the enterprise context, the port e : Enterprise of Data is not used, when instantiating the component as part of a Store. The optional operator ports of Inventory remain unconnected, as we did not model explicit GUI components. These would be connected to the particular ports for testing purposes; in the deployed system we may also connect actual operator interfaces.

The component Store uses mandatory relay ports to connect to a bank component and a data base. Relaying the port I-PD of component Inventory is optional, in order to take into account the requirements of the exceptional processes in Use Case 8 (enterprise server may be temporally not available). Optionality is also required for system configurations with stores that are independent of any enterprise. In this case, there is definitely no other store to exchange products with.

9.3.3 Cash Desks — The Embedded System Part

Any store instantiated as part of the trading system comprises a cash desk line which in turn represents a set of cash desks, monitored by a coordinator. Each cash desk consists of several hardware devices managed by a cash desk PC. The specification of the cash desk line models the embedded system part of the CoCoME with characteristic features of reactive systems such as asynchronous message exchange or topologies with a distinguished controller component. The former is illustrated by the subsequent behaviour specifications for ports and components, the latter is exemplified directly in the static structure of the composite component CashDesk with the cash desk application playing the role of the controlling component at the centre (Fig. 7). Due to this topology, most of this section is devoted to the specification of the component CashDeskApplication.

CashDeskLine. A CashDeskLine (Fig. 6) consists of at least one cash desk connected to a coordinator which decides on the express mode status of the cash desks. The composite component CashDeskLine declares two relay ports delegating the communication between the cash desks, the inventory (i : CDA-I) and the bank (b : CDA-Bank).

The connector declarations in Fig. 6 are annotated with the stereotype adapter of kind seq, meaning that the communication between the ports of the cash desks and the relay ports i and b respectively, is implemented by a sequential adapter. In contrast, the communication between the cash desks and the coordinator does not need to be adapted, because each CashDesk instance is linked via its CDA-C port to its own instance of the coordinator port C-CD. To share the bank connection among the desks of a cash desk

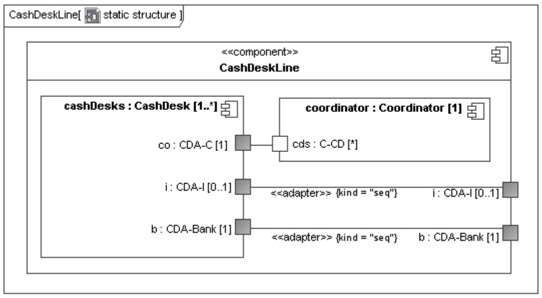

CashDeskLine[static structure]
<<component>>
CashDeskLine
cashDesks : CashDesk [1..*]
coordinator : Coordinator [1]
co : CDA-C [1]
cds : C-CD [*]
i : CDA-I [0..1]
<<adapter>> {kind = "seq"}
i : CDA-I [0..1]
b : CDA-Bank [1]
<<adapter>> {kind = "seq"}
b : CDA-Bank [1]

Fig. 6. Static structure of the component CashDeskLine

line follows the CoCoME requirement in [10, Fig. 6] which shows a multiplicity of 1 for the particular required Bank interface, port respectively.

CashDesk (CD). The CashDesk component specified in Fig. 7 is the most complex composite of the trading system. The component consists of six components modelling the hardware devices as described in the CoCoME and one component modelling the cash desk application. A cash desk has three relay ports to allow for the communication with a bank, inventory and coordinator component. The component and port multiplicities of the static structure in Fig. 7 reflect the requirements of the CoCoME. Since an exceptional process for Use Case 1 (Process Sale [10]) explicitly mentions that the inventory might not be available, the relay port i may sometimes not be connected. The optional ports of CashBox, Scanner and CardReader model the communication of an operator with the particular hardware device. In case of a cash desk actually deployed, these ports might be connected with some low-level interrupt handler.

CashDeskApplication (CDA). The cash desk application links all internal components of a cash desk and communicates with components external to the cash desk such as a

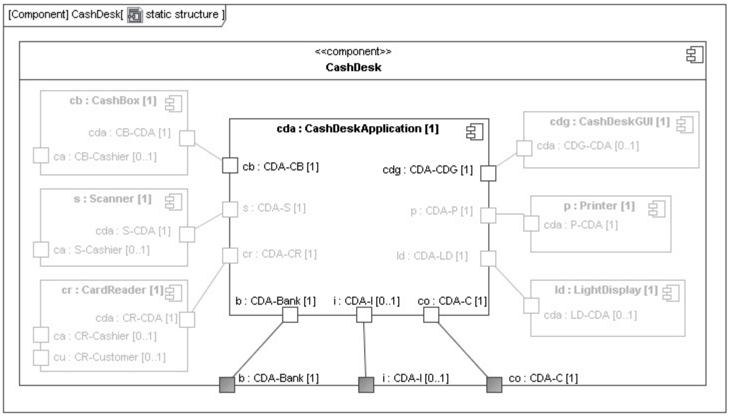


Fig. 7. Static structure of the component CashDesk

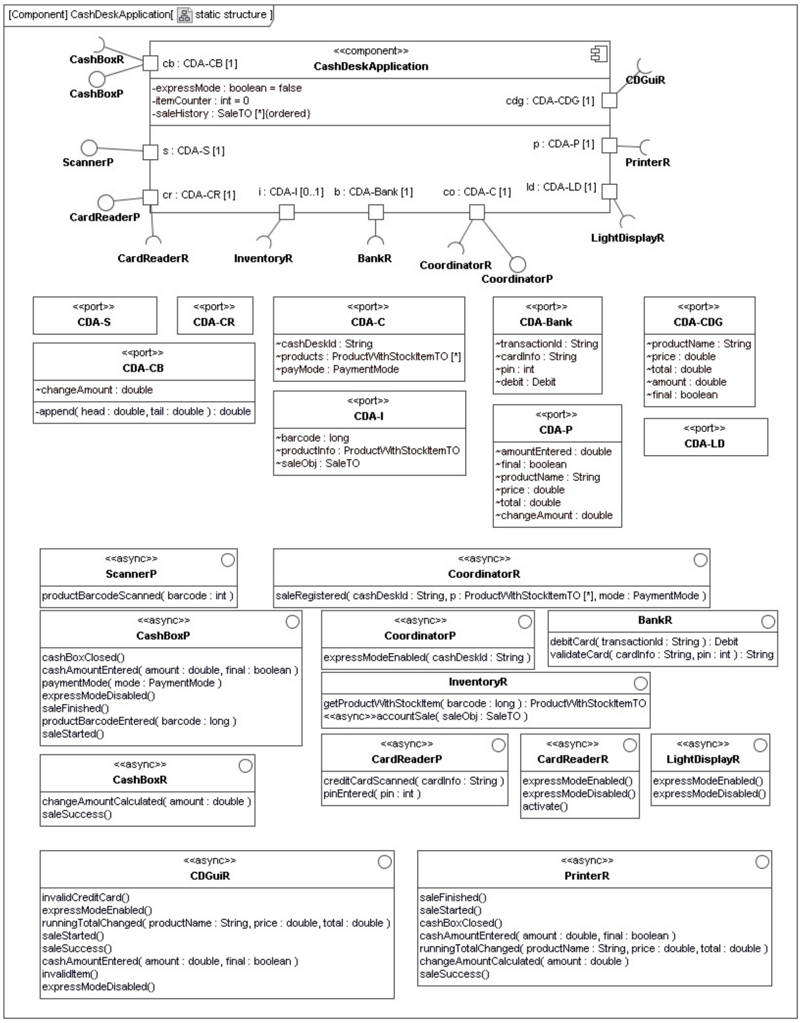


Fig. 8. Static structure of the CDA component

bank, the inventory or the coordinator. In order to facilitate the particular communication, CashDeskApplication declares one port for each. Figure 8 (top) shows an overview of the component with its private state as well as its ports and interfaces; the ports' state attributes and the interface details are given in the middle and lower region respectively. As a naming convention, we have used component abbreviations such as CDA-CB for the port types and the suffixes R (P) for interfaces required (provided) by a port.

In the following we briefly describe representative port behaviours of the CDA. Thereafter, we discuss the specification of the component behaviour which interrelates the constituent port behaviours within a single state machine.

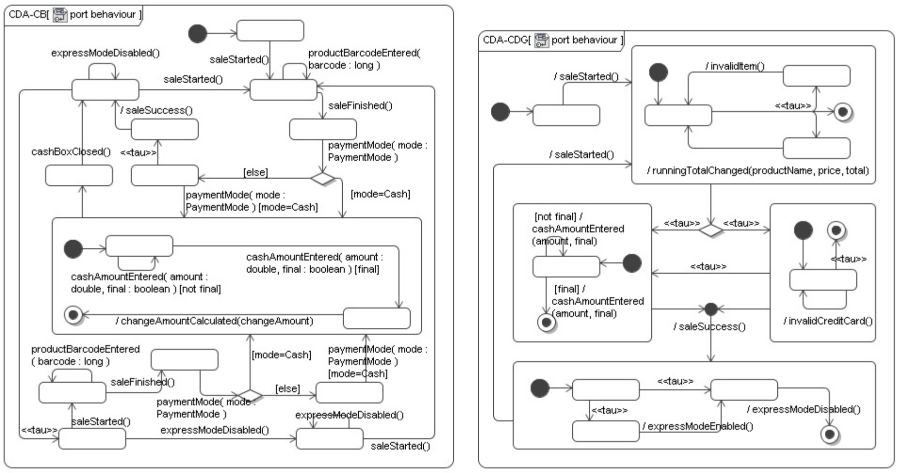

CDA-CB[port behaviour]
expressModeDisabled()
saleStarted()
productBarcodeEntered(barcode : long)
saleStarted()
/ saleSuccess()
saleFinished()
paymentMode(mode : PaymentMode)
cashBoxClosed()
<<tau>>
[else]
paymentMode(mode : PaymentMode) [mode=Cash]
[mode=Cash]
cashAmountEntered(amount : double, final : boolean) [final]
cashAmountEntered(amount : double, final : boolean) [not final]
/ changeAmountCalculated(changeAmount)
productBarcodeEntered (barcode : long)
[mode=Cash]
paymentMode(mode : PaymentMode) [mode=Cash]
saleFinished()
paymentMode(mode : PaymentMode)
[else]
expressModeDisabled()
saleStarted()
<<tau>>
expressModeDisabled()
saleStarted()
CDA-CDG[port behaviour]
/ saleStarted()
/ invalidItem()
<<tau>>
/ saleStarted()
/ runningTotalChanged(productName, price, total)
[not final] / cashAmountEntered (amount, final)
<<tau>>
<<tau>>
<<tau>>
<<tau>>
[final] / cashAmountEntered (amount, final)
/ saleSuccess()
/ invalidCreditCard()
<<tau>>
<<tau>>
/ expressModeDisabled()
/ expressModeEnabled()

Fig. 9. Behaviour of the CDA ports CB and CDG

CDA Port Behaviour. The state machine CDA-CB in Fig. 9 specifies the communication between the cash desk application and the cash box (CB). In general, the state machine for a port receives events and signals named in the provided interface of that port and sends out signals and events named in the required interface of that port. After initially having received the signal saleStarted, defined in the port's provided interface CashBoxP in Fig. 8, the port may receive arbitrary many manually entered product bar codes before moving to the next state due to the reception of saleFinished; manually entered product bar codes are part of an exceptional process in Use Case 1 of the CoCoME. The next step within the sale process is the selection of card or cash payment as the payment procedure. If payment mode Cash was selected, the port waits for messages concerning the cash amount received from the customer. It sends back information concerning the change amount calculated (by sending changeAmountCalculated defined in CDA-CB's required interface CashBoxR in Fig. 8), assumes that the cash box is subsequently opened and finally waits for the cash box to be closed again. If payment mode CreditCard was chosen, the port changes to a state where the chosen mode may be cancelled by switching to cash payment and, additionally, a τ-transition may be triggered internally, whereupon the cash box should be prepared to receive a saleSuccess as a signal for the successful clearing of the sale process via card payment. In both cases the port waits afterwards for the next saleStarted and until then allows to disable the express mode the cash desk may have been switched into in the meantime.

In contrast to the behaviour at an "input" port, Fig. 9 also shows an example for an "output" port behaviour: the specification of CDA-CDG, which is intended to connect to the cash desk's GUI. In fact the behaviour is very similar to the specification of CDA-P, connecting to the printer (not shown here). Both ports signal the status of the sale process such as saleStarted or saleFinished and both show mostly internal choice transitions. The main difference is that only the GUI port signals problems with the credit card (invalidCreditCard). Also, besides the light display, it is the GUI which is notified in case of mode switches from or to express mode.

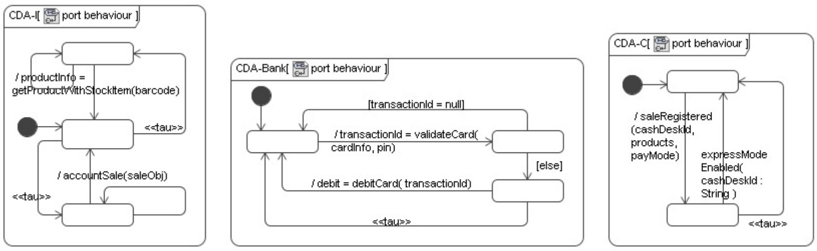

CDA-I[port behaviour]
/ productInfo = getProductWithStockItem(barcode)
<<tau>>
/ accountSale(saleObj)
<<tau>>
CDA-Bank[port behaviour]
[transactionId = null]
/ transactionId = validateCard(cardInfo, pin)
[else]
/ debit = debitCard(transactionId)
<<tau>>
CDA-C[port behaviour]
/ saleRegistered (cashDeskId, products, payMode)
expressMode Enabled(cashDeskId : String)
<<tau>>

Fig. 10. Behaviour of the CDA ports I, C and Bank

The communication with components external to the cash desk is prescribed by the port behaviour specifications for CDA-I, CDA-Bank and CDA-C shown in Fig. 10. The port behaviours of CDA-I and CDA-Bank demonstrate our modelling of synchronous communication. The behaviour of CDA-C is used in Sect. 9.4.1 to illustrate our formal analysis of functional requirements.

For a discussion on the behaviour specifications of the remaining ports CDA-S, CDA-CR, CDA-P and CDA-LD not relayed to the environment, we refer to [9].

CDA Component Behaviour. Figure 11 specifies the component behaviour of the cash desk application. Using the port declarations of the static structure in Fig. 8 it shows the dependencies and inter-linkages between the different ports of the CDA. For example messages sent via ports p or cdg such as p.saleStarted and cdg.saleStarted are sequentially arranged after the message cb.saleStarted was received at port cb. Furthermore port attributes as well as component attributes such as itemCounter are assigned as an effect, and afterwards used as actual parameters in messages sent.

Since the specification of the cash desk application's behaviour is rather involved we used submachines to factor out the major steps of the entire sale process. For lack of space we refer to [9] for their detailed specification. However, a brief description may

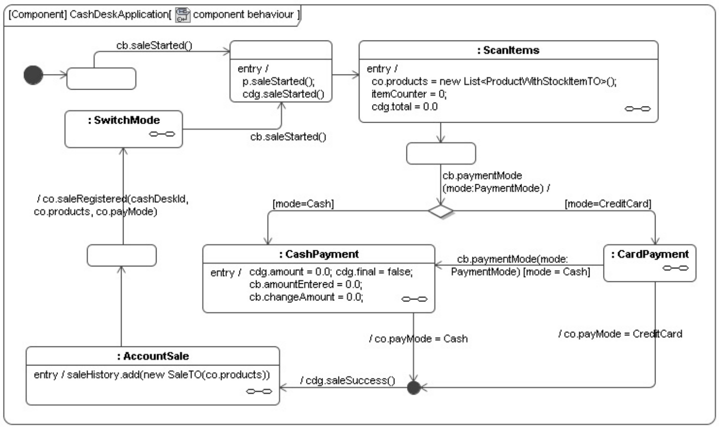

[Component] CashDeskApplication[component behaviour]
cb.saleStarted()
entry / p.saleStarted(); cdg.saleStarted()
: ScanItems
entry / co.products = new List<ProductWithStockItemTO>(); itemCounter = 0; cdg.total = 0.0
: SwitchMode
cb.saleStarted()
cb.paymentMode (mode:PaymentMode) /
/ co.saleRegistered(cashDeskId, co.products, co.payMode)
[mode=Cash]
[mode=CreditCard]
: CashPayment
entry / cdg.amount = 0.0; cdg.final = false; cb.amountEntered = 0.0; cb.changeAmount = 0.0;
cb.paymentMode(mode: PaymentMode) [mode = Cash]
: CardPayment
/ co.payMode = Cash
/ co.payMode = CreditCard
: AccountSale
entry / saleHistory.add(new SaleTO(co.products))
/ cdg.saleSuccess()

Fig. 11. Component behaviour of CashDeskApplication

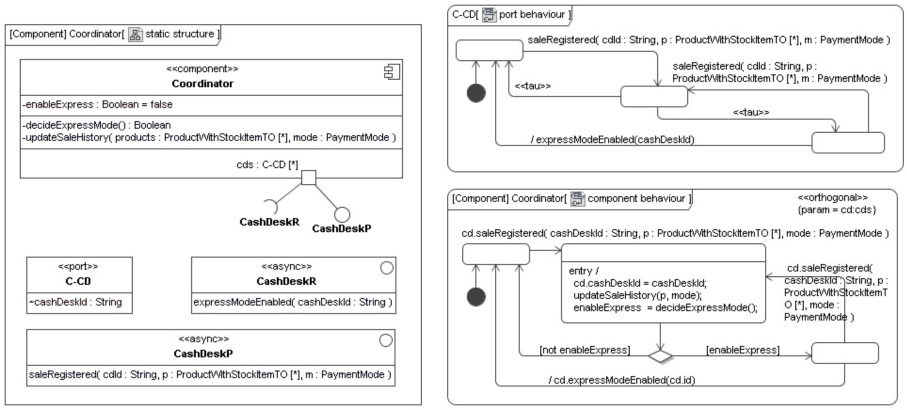

[Component] Coordinator[static structure]
<<component>>
Coordinator
-enableExpress : Boolean = false
-decideExpressMode() : Boolean
-updateSaleHistory(products : ProductWithStockItemTO [*], mode : PaymentMode)
cds : C-CD [*]
CashDeskR
CashDeskP
<<port>>
C-CD
~cashDeskId : String
<<async>>
CashDeskR
expressModeEnabled(cashDeskId : String)
<<async>>
CashDeskP
saleRegistered(cdId : String, p : ProductWithStockItemTO [*], m : PaymentMode)
C-CD[port behaviour]
saleRegistered(cdId : String, p : ProductWithStockItemTO [*], m : PaymentMode)
saleRegistered(cdId : String, p : ProductWithStockItemTO [*], m : PaymentMode)
<<tau>>
<<tau>>
/ expressModeEnabled(cashDeskId)
[Component] Coordinator[component behaviour]
<<orthogonal>> {param = cd:cds}
cd.saleRegistered(cashDeskId : String, p : ProductWithStockItemTO [*], mode : PaymentMode)
entry / cd.cashDeskId = cashDeskId; updateSaleHistory(p, mode); enableExpress = decideExpressMode();
cd.saleRegistered(cashDeskId : String, p : ProductWithStockItemTO [*], mode : PaymentMode)
[not enableExpress]
[enableExpress]
/ cd.expressModeEnabled(cd.id)

Fig. 12. Static structure and behaviour specifications of Coordinator

provide an intuitive understanding of the component's behaviour: after saleStarted was received at the cash box port cb, the submachine ScanItems repeatedly receives product bar codes and notifies the printer and the GUI about product details like name and price. Thereafter the payment mode must be chosen, resulting in a transition to the corresponding submachine. CardPayment may be left at any state by the reception of cb.paymentMode(Cash) modelling the cashier's ability to switch from card to cash payment, e.g., in case of problems with the credit card. Before the sale process is completed the component tries to account the sale at the inventory using port i within AccountSale. If the inventory is not available the sale information is stored locally and delivered during the next sale processes. Finally, in SwitchMode the component waits for a signal to switch into express mode, to disable a previous express mode, or to start a new sale.

Coordinator (C). The cash desk line (see Fig. 6) of a store consists of a number of cash desks and an instance of Coordinator, see Fig. 12, which decides on the mode of the cash desks (express or normal). The component declares its port of type C-CD with multiplicity * to allow to connect an arbitrary number of cash desks, which should be monitored by this coordinator. Note that even if the coordinator decided for express mode, the port may receive yet another sale registration from the same cash desk because the communication partners are executing concurrently. In this case the sale registration has precedence over the coordinator's decision: the port recomputes its internal decision.

The component behaviour shown alongside the port behaviour in Fig. 12 illustrates the modelling of internal actions of a component which are hidden by τ-transitions in the port behaviour specification. For each cash desk the component keeps track of the particular sale history and decides upon this history to eventually switch a cash desk into express mode. The update of the sale history is synchronised (annotation sequential) due to the concurrent execution of the port instances cd in cds.

9.3.4 Inventory — The Information System Part

The information system part is modelled with an inventory at the core. The inventory plays a crucial role in the Use Cases 3, 4, 7 and 8 (see [10, Figs. 6,6,6,6]), which

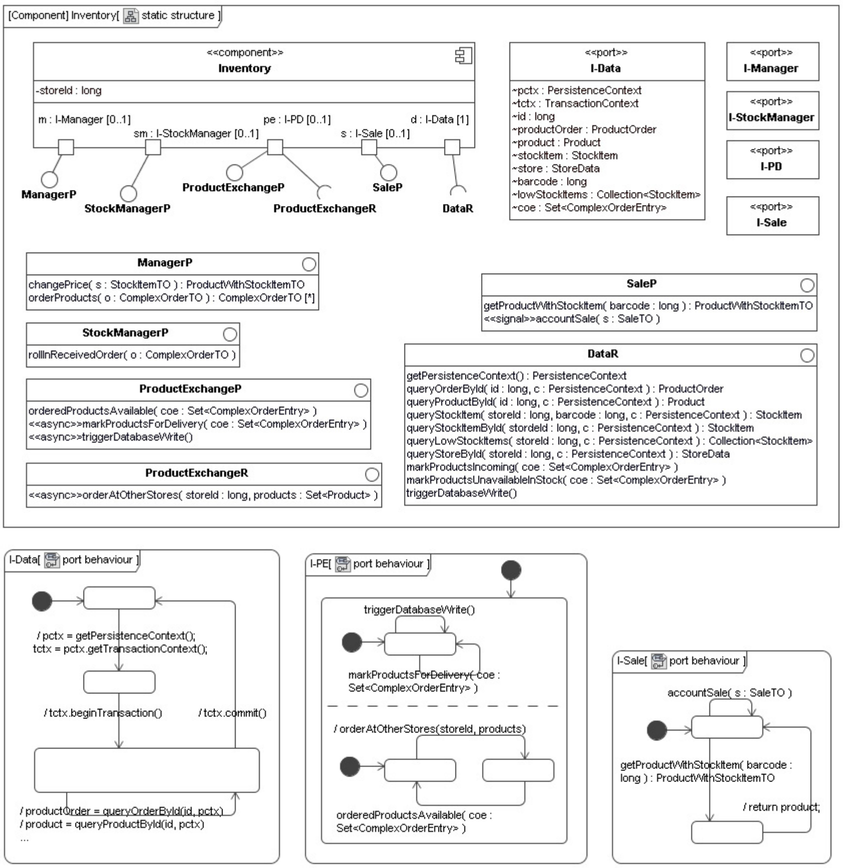

[Component] Inventory[static structure]
<<component>>
Inventory
-storeId : long
m : I-Manager [0..1]
sm : I-StockManager [0..1]
pe : I-PD [0..1]
s : I-Sale [0..1]
d : I-Data [1]
ManagerP
StockManagerP
ProductExchangeP
ProductExchangeR
SaleP
DataR
<<port>>
I-Data
~pctx : PersistenceContext
~tctx : TransactionContext
~id : long
~productOrder : ProductOrder
~product : Product
~stockItem : StockItem
~store : StoreData
~barcode : long
~lowStockItems : Collection<StockItem>
~coe : Set<ComplexOrderEntry>
<<port>>
I-Manager
<<port>>
I-StockManager
<<port>>
I-PD
<<port>>
I-Sale
ManagerP
changePrice(s : StockItemTO) : ProductWithStockItemTO
orderProducts(o : ComplexOrderTO) : ComplexOrderTO [*]
StockManagerP
rollInReceivedOrder(o : ComplexOrderTO)
ProductExchangeP
orderedProductsAvailable(coe : Set<ComplexOrderEntry>)
<<async>>markProductsForDelivery(coe : Set<ComplexOrderEntry>)
<<async>>triggerDatabaseWrite()
ProductExchangeR
<<async>>orderAtOtherStores(storeId : long, products : Set<Product>)
SaleP
getProductWithStockItem(barcode : long) : ProductWithStockItemTO
<<signal>>accountSale(s : SaleTO)
DataR
getPersistenceContext() : PersistenceContext
queryOrderById(id : long, c : PersistenceContext) : ProductOrder
queryProductById(id : long, c : PersistenceContext) : Product
queryStockItem(storeId : long, barcode : long, c : PersistenceContext) : StockItem
queryStockItemById(storeId : long, c : PersistenceContext) : StockItem
queryLowStockItems(storeId : long, c : PersistenceContext) : Collection<StockItem>
queryStoreById(storeId : long, c : PersistenceContext) : StoreData
markProductsIncoming(coe : Set<ComplexOrderEntry>)
markProductsUnavailableInStock(coe : Set<ComplexOrderEntry>)
triggerDatabaseWrite()
I-Data[port behaviour]
/ pctx = getPersistenceContext();
tctx = pctx.getTransactionContext();
/ tctx.beginTransaction()
/ tctx.commit()
/ productOrder = queryOrderById(id, pctx)
/ product = queryProductById(id, pctx)
...
I-PE[port behaviour]
triggerDatabaseWrite()
markProductsForDelivery(coe : Set<ComplexOrderEntry>)
/ orderAtOtherStores(storeId, products)
orderedProductsAvailable(coe : Set<ComplexOrderEntry>)
I-Sale[port behaviour]
accountSale(s : SaleTO)
getProductWithStockItem(barcode : long) : ProductWithStockItemTO
/ return product;

Fig. 13. Static structure and port behaviour of the component Inventory

describe how to order products, receive ordered products, change the price of a product and how products might be exchanged among the stores of a enterprise. The most prominent new modelling aspect with respect to the embedded system part in Sect. 9.3.3 is the specification of synchronous message call receptions. The specifications of Data, Reporting, ProductDispatcher and DataBase can be found on our web page [9].

Inventory (I). The component Inventory[1] is a model of the store's portion of the application layer of the CoCoME. As depicted in the static structure of Fig. 13, Inventory provides two optional ports m : I-Manager and sm : I-StockManager to allow for manager and stock manager requests. The behaviour specifications of these ports are trivial and omitted here. The ports may be used for instance to connect simulation components in order to test the developed system or, of course, to connect actual control interfaces in the deployed system.

[1] Note that our component Inventory models Inventory::Application::Store of [10], see Sect. 9.3.1.

The ports of type I-Data and I-Sale are used to connect to the data layer of a store component and to the cash desks of the store, respectively. As the port behaviour of I-Data in Fig. 13 exemplifies for two operations of the interface DataR, any operation call on the data layer is transactional, i.e., is framed by an explicit transaction start (tctx.beginTransaction) and end (tctx.commit); the remaining operations of DataR are applied analogously. Connections via I-Sale support the reception of messages required during the sale process at a cash desk.

The component declares a port pe : I-PE in order to cope with product exchange among stores as described in Use Case 8 of the CoCoME. The port behaviour specification in Fig. 13 uses an orthogonal state with two regions to model the two distinct roles of an inventory: the port may request to order some products at the other stores of the enterprise, i.e., play the active role of initiating product exchange; on the other hand, it provides a trigger for a data base update with possibly cached and not yet submitted data, as well as to mark products for delivery to other stores, i.e. playing the passive role of being asked for product exchange. Both messages are eventually received during a connection with the component ProductDispatcher (see Fig. 5) responsible to coordinate the product exchange among the stores (see [10, Fig. 6]).

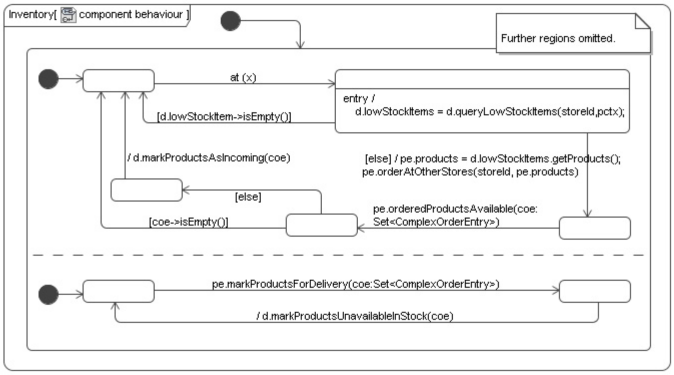

Inventory[component behaviour]
Further regions omitted.
at (x)
entry /
d.lowStockItems = d.queryLowStockItems(storeId,pctx);
[d.lowStockItem->isEmpty()]
/ d.markProductsAsIncoming(coe)
[else] / pe.products = d.lowStockItems.getProducts();
pe.orderAtOtherStores(storeId, pe.products)
[else]
pe.orderedProductsAvailable(coe:
Set<ComplexOrderEntry>)
[coe->isEmpty()]
pe.markProductsForDelivery(coe:Set<ComplexOrderEntry>)
/ d.markProductsUnavailableInStock(coe)

Fig. 14. Component behaviour of the Inventory (regions omitted)

The component behaviour specification of Inventory comprises of six orthogonal regions, essentially each of them modelling the interaction with one possible communication partner along the ports of the component. For a lack of space, Fig. 14 shows only two of them. Again, for more details we refer to [9]. The regions shown illustrate part of the interplay between the ports d and pe in the course of executing a product exchange among stores. The upper region specifies the component's behaviour with respect to the inventory's check if the stock is getting low for some products. The check occurs cyclical after some not further specified time period x. The lower region shows the direct forwarding of a product order actually to be executed. Note, that we do not show the message exchange related to the transactional data communication with port d. For this purpose the operation calls on d would simply be framed by begin and commit transitions analogously to the port behaviour of I-Data.

9.4 Analysis

9.4.1 Analysis of Functional Requirements

For the analysis of the functional requirements we focus on the semantical properties of our CoCoME model specified by the behaviour specifications of ports and components in Sect. 9.3. We consider the asynchronous part of our model for the CoCoME and we will check deadlock-freeness and component correctness. For the synchronous, information-oriented part we believe that the behavioural aspects of message exchange and parallel execution, which are in the centre of our interest here, are not so relevant.

The basic idea of our approach is to proceed hierarchically, starting from the analysis of (local) properties of simple components and their ports from which we can then derive, by a compositionality theorem, properties of composite components. Thus, following the hierarchical construction of components, we obtain results for the behaviour of the global system. Our analysis process consists of the following steps:

1. For each simple component we analyse
   - the behaviour specification of each of its ports,
   - the behaviour specification of the component itself, and
   - the relationships between the component behaviour and the behaviour specified for each of its ports.
2. For each composite component we analyse
   - the interaction behaviour of connected ports,
   - the behaviour of the composite component which can be inferred from the behaviours of its constituent parts, and
   - the relationships between the behaviour of the composite component and the behaviour of each of its relay ports.

The semantic basis of our study are the given UML state machines for the behaviour specifications of ports and components. In order to treat them in a formal way, we represent them by labelled I/O-transition systems which are automata with an initial state and with distinguished *input* (or *provided*), *output* (or *required*), and *internal* labels. Additionally, we assume that there is a special *invisible* (or *silent*) action τ.[2] Two I/O-transition systems A and B (over the same I/O-labelling) are *observationally equivalent*, denoted by $A \approx B$, if there exists a weak bisimulation relation between A and B, e.g. in the sense of [11], where all actions apart from τ are considered to be visible. We also use standard operators on I/O-transition systems like *relabelling*, *hiding* and the formation of *products*.[3] In some cases we will need a simple form of relabelling of an I/O-transition system A, denoted by $n.A$, where the labels of A are just prefixed by a given name n thus obtaining a copy of A. An I/O-transition system A is *deadlock-free* if for any reachable state there is an outgoing sequence of transitions $\tau^*\ a\ \tau^*$ with some label $a \neq \tau$ preceeded and followed by arbitrary many τ actions. The observational equivalence relation is compatible with deadlock-freeness and, moreover, it

[2] Internal actions are not invisible; to construct particular component views, they can, however, be hidden.

[3] The product of two I/O-transition systems is used to model their parallel composition with synchronisation on corresponding input/output labels.

is preserved by the above mentioned operators on transition systems. For the precise technical definitions we refer to [9].

Let us first consider how behaviour specifications of ports are represented by I/O-transition systems. As pointed out in Sect. 9.2 a port has a provided and a required interface. For calls of operations of the provided interface we use input labels; for sending an operation request according to the required interface of a port we use output labels. In most cases the label is just the name of an interface operation where we have abstracted from operation parameters and results which is possible if the transitions in the original state machine do not depend on the arguments and results of the operation. In the other cases we must assume, for the purpose of model checking later on, that the impact of arguments and/or results and/or guards occurring on UML transitions can be resolved by a finitary case distinction which is encoded by appropriate labels. Note, that transitions with the invisible action τ can occur in the behaviour specification of a port in order to model a possible internal choice (of the port's owner component) which is not visible at the port but may have an impact on the future behaviour of the port.

As a concrete example we consider the component Coordinator and the behaviour specification of its port C-CD; see Fig. 12. The corresponding I/O-transition system, shown in Fig. 15, is directly inferred from the behaviour specification.[4] According to the given behaviour specification of the port, the silent action τ represents a non-visible choice whether an express mode should be enabled or not.

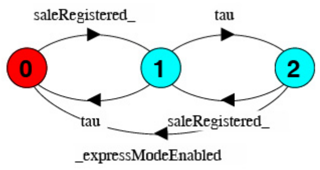


Fig. 15. I/O-transition system for port C-CD

Let us now look to the behaviour specifications of simple components and their representation by I/O-transition systems. A simple component contains a set of port declarations of the form $p : P[mult]$ where p is a port name, P its port type and $mult$ specifies the port multiplicity indicating how many instances of that port a component (instance) can have. Since a component can only communicate with its environment via its ports, any input label of a component has the form $p.i$ where p is a port name and i is an input label of the port. Similarly, the output labels of a component have the form $p.o$. For the definition of input and output labels of components we do not take into account here the multiplicities of ports. This is possible if we assume that actions of different port instances of the same port declaration are independent from each other which is indeed the case in our example. In the following we will always omit multiplicities in port declarations. In contrast to ports, components can have internal labels which are just given by some name representing an internal action of the component.

[4] For the representation of the transition systems we have used the LTSA tool (cf. Sect. 9.5) which does not support the symbol "/" used in UML state machines. In order to indicate that i is an input label we use $i\_$ and, symmetrically, to indicate that o is an output label we use $\_o$.

Again, for the purpose of model checking, we assume that arguments, results and/or guards of internal operations are encoded into appropriate labels.

As an example we consider the behaviour specification of the Coordinator component (see Fig. 12). The behaviour specification uses an entry action and a pseudostate for a guarded alternative which both have to be resolved in the corresponding transition system. For representing the entry action we introduce the (internal) label entry and for representing the two guarded alternatives we introduce two internal labels enableExpress, describing the decision that the express mode should be enabled, and notEnableExpress, expressing the converse case. Operation calls have now the prefix cds of the port on which the operation is received or sent. The whole transition system representing the behaviour of the Coordinator component is shown in Fig. 16.

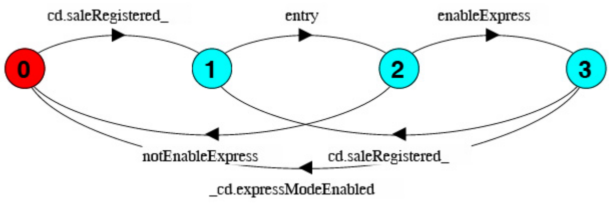


Fig. 16. I/O-transition system for component Coordinator

Analysis of Simple Components. In the first step of our model analysis we consider simple components which are the basic building blocks of our system model. For each simple component we check the deadlock-freeness of the behaviour specifications of each of its ports and of the component itself. Obviously, this condition is satisfied for all simple components and ports of our behavioural model for the CoCoME.

A more subtle point concerns the relationships between the behaviour of a component and the behaviour specified for each of its ports which must in some sense fit together. To consider this issue more closely, let C be a component with associated behaviour represented by the I/O-transition system A_C and let $p : P$ be a port declaration of C such that the behaviour specification associated to P is represented by the I/O-transition system A_P. Intuitively, the component C is correct w.r.t. its port declaration $p : P$ if the component behaviour supports the behaviour specified for that port. Apparently this is the case if the component's behaviour observable at port p is observationally equivalent to the behaviour specification of P (up to an appropriate relabelling).

Formally, the *observable behaviour of C at port p*, denoted by $obs_p(C)$, can be constructed by hiding all labels of A_C which do not refer to p. Using the hiding operator, $obs_p(C)$ is just $A_C \setminus H$ where the set H of hidden labels consists of the internal labels of A_C together with all input or output labels $q.op$ of A_C such that $q \neq p$. Since the transition system $obs_p(C)$ has no internal labels and, up to the prefix p, the same input and output labels as A_P we can now require that it is observationally equivalent to $p.A_P$, i.e., $obs_p(C) \approx p.A_P$ (where $p.A_P$ is the copy of A_P explained above). In this case we say that the component C is *correct w.r.t. its port declaration* $p : P$.

Let us illustrate how we can check the correctness of the component Coordinator w.r.t. to a port instance cd of type C-CD. First we consider the observable behaviour of the Coordinator at the port instance cd which is just the transition system shown in Fig. 16 where all labels which are not prefixed by cd are replaced by τ. If we minimise this transition system w.r.t. observational equivalence then we obtain (up to the prefix cd) the transition system in Fig. 15 which represents the behaviour of the port type C-CD. This shows the correctness of the Coordinator component. Indeed we have checked with the LTSA tool (cf. Sect. 9.5) that all simple components occurring in the CashDesk and CashDeskLine composite components (cf. Sect. 9.2) are correct.

The definition of the observable behaviour of a component at a particular port can be generalised in a straightforward way to arbitrary subsets of the port declarations of a component and, in particular, to the case where all ports of a component are simultaneously considered to be observable. For a component C, the latter is called the *(fully) observable behaviour* of C and denoted by $obs(C)$.

Obviously, the above definitions of correctness and observable behaviour apply not only to simple but also to composite components considered in the next step.

Analysis of composite components. The analysis of composite components is related to the task of a system architect who puts components together to build larger ones. Before we can analyse the behaviour of a composite component it is crucial to consider the connections that have been established between the ports of their subcomponents.

Analysis of connectors. For the analysis of connectors one has first to check whether the connections between the ports of components are syntactically well-defined. After that we can analyse the interaction behaviour of two connected ports.

In the following let us consider a connection between two port declarations $p_\mathrm{l} : P_\mathrm{l}$ and $p_\mathrm{r} : P_\mathrm{r}$ occurring in components C_l and C_r respectively. The connection is syntactically well-defined, if the operations of the required interface of P_l coincide with the operations of the provided interface of P_r and conversely.[5] To study the interaction behaviour of the two ports, let A_{P_l} and A_{P_r} be the I/O-transition systems representing the behaviour of P_l and P_r, respectively. Any communication between the connected ports is expressed by synchronising output labels of one port with the corresponding input labels of the other port. Hence, the interaction behaviour of A_{P_l} and A_{P_r} can be formally represented by the *port product* $A_{P_\mathrm{l}} \otimes A_{P_\mathrm{r}}$ of A_{P_l} and A_{P_r}; see [9] for the definition of products.

A first semantic condition which should be required for a port connection is that any two port instances can communicate with each other without the possibility to run into a deadlock which means that the port product is deadlock-free. In this case we say that the ports are *behaviourally compatible*.

Let us, for instance, consider the composite component CashDeskLine (cf. Fig. 6) which has one connector between the port CDA-C of the CashDesk component and the port C-CD of the Coordinator. The transition system representing the behaviour of the port CDA-C (cf. Fig. 10) is shown in Fig. 17 (top) and the transition system representing the behaviour of the port C-CD was shown in Fig. 15. Hence, the interaction behaviour of the two ports is represented by the transition system of their port product which is

[5] Our approach would also allow a more general condition where the required operations of one port are included in the provided operations of the other one which, however, is not needed for the current case study.

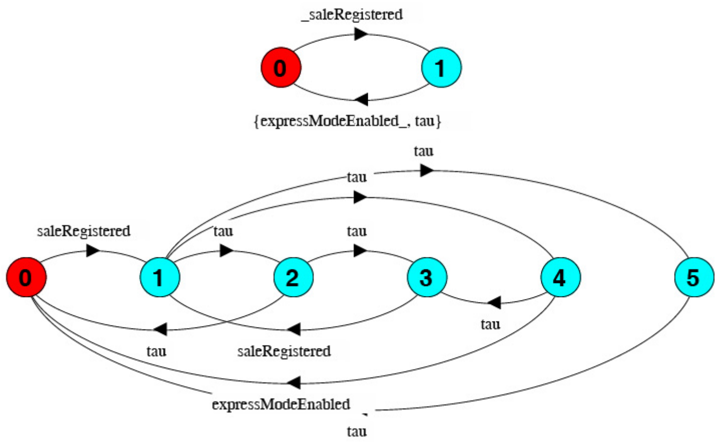


Fig. 17. Port product of CDA-C and C-CD

shown in Fig. 17 (bottom). Obviously, the port product has no deadlock and therefore the two ports are behaviourally compatible.

In general, the potential capabilities for interaction of a port will not be used when the port is connected to another port. In this case the behaviour specified for that port is restricted by the interaction with another port. It is, however, often the case that this restriction applies only to one side of a connection while the behaviour of the other port is not restricted and hence fully reflected by the interaction. Given two ports P_l and P_r with behaviours represented by I/O-transition systems $A_{P_\mathrm{l}}, A_{P_\mathrm{r}}$ respectively, the interaction behaviour of P_l and P_r *reflects* the behaviour of P_l, if the port product is observationally equivalent to the behaviour of P_l, i.e. $A_{P_\mathrm{l}} \approx (A_{P_\mathrm{l}} \otimes A_{P_\mathrm{r}})$. This property plays an essential role for the compositionality of component behaviours considered below. An obvious consequence of this definition is that if the interaction behaviour of P_l and P_r reflects the behaviour of P_l (P_r resp.) and if the behaviour of the port P_l (P_r resp.) is deadlock-free, then P_l and P_r are behaviourally compatible.

For instance, let us consider again the port product of CDA-C and C-CD in Fig. 17 (bottom). After minimisation of the transition system w.r.t. observational equivalence with the LTSA tool we obtain just the transition system of the port CDA-C; cf. Fig. 17 (top). Hence, the interaction behaviour of CDA-C and C-CD even reflects the behaviour of the port CDA-C.

Analysis of the behaviour of composite components. In contrast to simple components the behaviour of a composite component is not explicitly specified by the developer but can be derived from the behaviours of the single parts of the composite component. For that purpose we construct the product of the transition systems representing the observable behaviours of all subcomponents declared in a composite component whereby the single behaviours of the subcomponents observed at their ports are synchronised according to the shared labels determined by the connectors. Hence, we focus on the interactions between the subcomponents (via their connected ports) and on the actions on the relay ports of the composite component while the internal behaviour of the subcomponents is not relevant; see [9] for the detailed definitions.

Of course, one may again construct the observable behaviour of a composite component which then could be used for further analysis but also for the construction of the behaviour of another composite component on the next hierarchy level. When climbing up the hierarchy of composite components one can always first perform a minimisation of the observable behaviour of the subcomponents before the behaviour of the composite component on the next level is constructed. This technique can indeed be very efficient to reduce the state space of nested components because, depending on the application, many (or even all) τ-transitions may be removed.[6] In fact, our experience shows that in this way there is often not even an increment of the size of the state space [12].

In the following we focus on checking the deadlock-freeness of the behaviour of a composite component. It is well-known that, in general, the deadlock-freeness of subcomponents does not guarantee the deadlock-freeness of a global system (as nicely illustrated by Dijkstra's philosophers example). Indeed this is unfortunately still the case if all components are correct w.r.t. their ports (in the sense from above) and if all ports are connected in a behaviourally compatible way, as soon as more than two subcomponents are involved. Hence, we are looking for particular topologies of component structures where deadlock-freeness is preserved. An appropriate candidate are (acyclic) star topologies as shown in Fig. 18 containing one central component C with n ports such that each port is connected to the port of one of the components C_i for $i = 1, \ldots, n$.

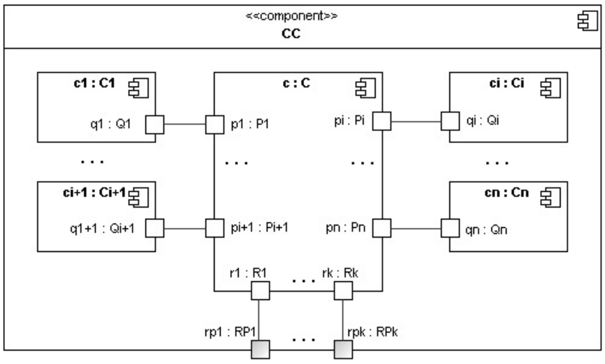


Fig. 18. Star topology of composite component

We assume that all single subcomponents, C and C_i, are correct w.r.t. their ports and that their local behaviours are deadlock-free. Then, if all connected ports are behaviourally compatible, the composite component CC can only deadlock if at least two ports $p_\alpha : P_\alpha, p_\beta : P_\beta$ of the central component C are connected to ports $q_\alpha : Q_\alpha$, $q_\beta : Q_\beta$ of components C_α and C_β resp. such that the behaviours specified for both port

[6] Only τ-transitions occurring in an alternative may not be removable according to the well-known fact that alternatives are i.g. not compatible with the observational equivalence.

types Q_α and Q_β properly restrict the behaviour of P_α *and* of P_β in an incompatible way.[7] This may happen, if C introduces a dependency between P_α and P_β that is incompatible with the simultaneous restrictions imposed by the connections with Q_α and Q_β on both sides of C. An example for such a situation is provided in [12]. If, however, at most the behaviour of one port of C is restricted by the connection and the interaction behaviours of all other connections reflect the behaviour of all other ports of C then deadlock-freeness is preserved by the component composition. This fact is expressed by the following theorem (for the proof see [12]) which shows that indeed for the global deadlock check it is enough if the subcomponents are locally checked for deadlock-freeness and correctness and if the architect of the composite component checks each single port connection on the basis of the interaction behaviour of the connected ports.

Theorem 1 (Deadlock-freeness of composite components). *Let CC be a composite component with component structure as shown in Fig. 18. Let the following hold:*

1. *The components $C_1, \ldots, C_n$ are correct w.r.t. their ports $q_1 : Q_1, \ldots, q_n : Q_n$ resp., and C is correct w.r.t. to each of its ports $p_1 : P_1, \ldots, p_n : P_n$.*
2. *All I/O-transition systems representing the behaviours of $C_1, \ldots, C_n$ and C are deadlock-free.*
3. *For all $i \in \{1, \ldots, n-1\}$ the interaction behaviour of P_i and Q_i reflects the behaviour of P_i.*
4. *The ports P_n and Q_n are behaviourally compatible.*

Then the I/O-transition system representing the behaviour of CC is deadlock-free.

This theorem is related to a result of Bernardo et al. [13] which is also motivated by the derivation of global properties from local ones, not using an explicit port concept, however. In [13] a significantly stronger condition is used requiring what we call "behaviour reflection" for all connections between components with no exception where behavioural compatibility is sufficient as in the above theorem. A further generalisation of the theorem to arbitrary many non behaviour reflecting but behavioural compatible connections is given in [12] which, however, needs further assumptions.

We can directly apply Thm. 1 to analyse the behaviour of the composite component CashDesk (cf. Fig. 7) where CashDeskApplication plays the role of the central component C. As pointed out above all subcomponents of CashDesk are correct w.r.t. their respective ports and their behaviour is deadlock-free. We also have analysed (with HUGO/RT and the LTSA tool, see Sect. 9.5.1) the given connectors between the ports of CashDeskApplication and the ports of the other subcomponents of CashDesk. It turns out that the port CDA-CB of CashDeskApplication is behaviourally compatible with the port CB-CDA of the CashBox component and that for all other connected ports the interaction behaviour even reflects the behaviour of the corresponding port of CashDeskApplication. Hence, Thm. 1 shows that the component CashDesk does not deadlock. We can also show (see [9]) that the CashDesk component is correct w.r.t. its relay ports.

[7] Note that it is sufficient to consider the behaviours of the *ports* of C_α and C_β instead of considering the observable behaviour of the components C_α and C_β since both components are assumed to be correct w.r.t. their respective ports.

Following our analysis method we now go one step up in the component hierarchy and consider the composite component CashDeskLine (cf. Fig. 6) which has connected subcomponents of type CashDesk and Coordinator. Obviously, the structure of CashDeskLine fits again to the component structure assumed in Thm. 1. Hence, we can directly apply Thm. 1 since we know that CashDesk is correct and deadlock-free, Coordinator is correct and deadlock-free (see paragraph on the analysis of simple components), and that the connection between the ports (of type) CDA-C and C-CD reflects the behaviour of CDA-C (see paragraph on the analysis of connectors). Thus component CashDeskLine does not deadlock and, according to the reflection of the appropriate port behaviour, it is also correct w.r.t. its relay ports.

Note again that we did not take into account here multiplicities of component declarations which means in this example, that we have disregarded the number of CashDesk instances that are connected to one Coordinator instance. This abstraction works because, first, the Coordinator instance has as many port instances of type C-CD as there are cash desks connected, and, more importantly, the interactions of the coordinator with the single cash desks are independent. More formally, this means that if there are n cash desks connected to the coordinator then arbitrary interleaving is allowed and thus deadlock-freeness of the cash desk line does not depend on n.

Let us now come back to the original proposal of the CashDeskLine structure which has used an event bus for communication [10]. We have refrained from using the event bus in the design model, as we believe that the introduction of an event bus is an implementation decision to be taken after the design model has been established and analysed. Indeed we could introduce right now an implementation model which implements the communication between the components of the CashDesk and CashDeskLine in terms of an event bus, provided that the bus follows the first-in first-out principle. Then, obviously, the order of communications between the single components specified in our design model would be preserved by the implementation model and hence the deadlock-freeness of the design model would also hold for the event bus based implementation.

This concludes the behavioural analysis of the asynchronous part of our model for the CoCoME which was in the centre of our interest. For the synchronous, information-oriented part we suggest to apply pre-/post-condition techniques which have been lifted to the level of components in our previous [14] and recent [15] work.

9.4.2 Non-functional Requirements

We perform quantitative analysis of the Process Sale Use Case 1 by modelling the example in the process algebra PEPA [5] and mapping it onto a Continuous-Time Markov Chain (CTMC) for performance analysis. The analysis shows that the advantage of express checkout is not as great as might be expected. Currently, however, this analysis has to be performed manually; a closer integration could be achieved by decorating UML state machines and sequence diagrams with rate information using the UML performance profile as in the Choreographer design platform [16], allowing a PEPA model to be extracted from the UML diagrams.

Model. Markovian models associate an exponentially distributed random variable with each transition from state to state. The random variable expresses quantitative information about the rate at which the transition can be performed. Formally, a random variable is said to have an exponential distribution with parameter λ (where $\lambda > 0$) if it has the probability distribution function F with $F(x) = 1 - e^{-\lambda x}$ if $x > 0$, and $F(x) = 0$ otherwise. The mean, μ, of this exponential distribution is $\mu = \int_0^\infty x\lambda e^{-\lambda x} dx = 1/\lambda$. Thus if we know the mean value of the duration associated with an activity then we can easily calculate from this the rate parameter of the exponential distribution: $\lambda = 1/\mu$.

In the case where we only know the mean value then the exponential distribution is the correct distribution to use because any other distribution would need to make additional assumptions about the shape of the expected distribution. However, for example in the case of waiting for credit card validation, the extra-functional properties state that we have a histogram representing the distribution over the expected durations stating that with probability 0.9 validation will take between 4 and 5 seconds and with probability 0.1 it will take between 5 and 20 seconds.

We encode distributions such as these in the process algebra PEPA as an immediate probabilistic choice followed by a validation occurring at expected rate (4.5 is mid-way between 4 and 5 and 12.5 is mid-way between 5 and 20 so we use these as our means).

$$\begin{aligned} &(\tau, 0.9 : \mathbf{immediate}).(\mathit{validate}, 1/4.5)\ldots \\ + &(\tau, 0.1 : \mathbf{immediate}).(\mathit{validate}, 1/12.5)\ldots \end{aligned}$$

Whichever branch is taken, the next activity is validation; the only difference is the rate at which the validation happens. In Fig. 19 we show how 700000 values from a uniformly-distributed interpretation of the histogram for credit card validation would differ from the exponentially-distributed interpretation.

In our experience, a distribution such as that shown in Fig. 19 (left) is unlikely to occur in practice. For example, it has the surprising property that delays of four seconds are very likely but delays of three seconds are impossible. Also, there is a very marked difference between the number of delays of five seconds and delays of six. In contrast, distributions as seen from our sample in Fig. 19 (right) occur frequently because they are a convolution of two heavy-tailed distributions. Other histogram-specified continuous distributions are treated similarly. We first make a weighted probabilistic choice and then delay for a exponentially-distributed time.

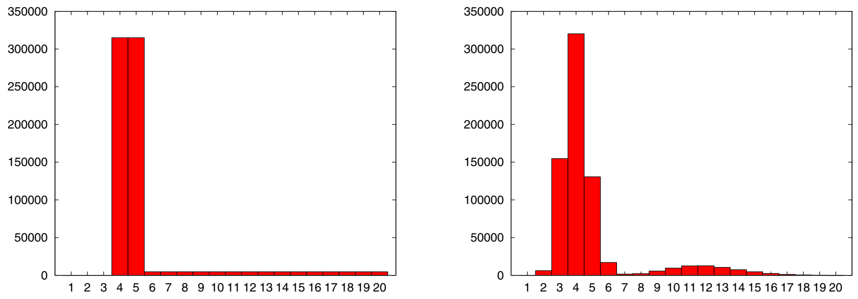


Fig. 19. Specified (left) and sampled (right) distributions for credit card validation

Analysis. From our process algebra model we obtain a finite-state continuous-time Markov chain represented as a matrix, Q, to be analysed to find the probability of being in each of the states of the model. At equilibrium the probability flow into every state is exactly balanced by the probability flow out so the equilibrium probability distribution can be found by solving the *global balance equation* $\pi Q = 0$ subject to the normalisation condition $\sum_i \pi(x_i) = 1$. From this probability distribution can be calculated performance measures of the system such as throughput and utilisation.

From this information we can assess a service-level agreement for the system. A service-level agreement typically incorporates a time bound and a probability bound on paths through the system behaviour. We considered the advantage to be gained by using the express checkout where customers in the queue have no more than 8 items to purchase. As would be expected, the sale is always likely to be completed more quickly at the express checkout but the advantage is not as great as might be expected. At the express checkout 50% of sales are completed within 40 seconds as opposed to the 44 seconds spent at a normal checkout (see Fig. 20 (right)). In our model we included the possibility of customers with 8 items or fewer choosing to go to a normal checkout instead of the express checkout, because we have seen this happening in practice. This goes some way to explaining why the difference in the results is not larger.

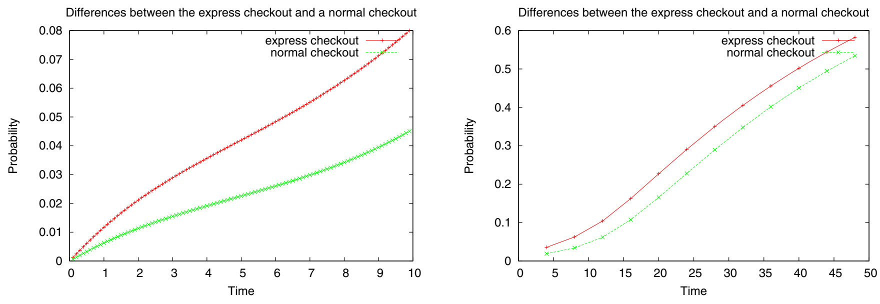


Fig. 20. Graphs showing how the advantage of using the express checkout (8 items or fewer) over using a normal checkout (100 items or fewer) varies over 10 (left) and 50 seconds (right)

9.5 Tools

9.5.1 Qualitative and Quantitative Analysis

HUGO/RT. HUGO/RT [17] is a UML model translator for model checking, theorem proving, and code generation: A UML model containing active classes with state machines, collaborations, interactions, and OCL constraints can be translated into the system languages of the real-time model checker UPPAAL, the on-the-fly model checker SPIN, the system language of the theorem prover KIV, and into Java and SystemC code. The input can either be directly be given as an XMI (1.0, 1.1) file or in a textual format called UTE (for an example see Fig. 22).

In the CoCoME, we use HUGO/RT for two purposes: On the one hand, we check the deadlock-freedom of connectors by translation into a model checker; however, as currently HUGO/RT's model checking support is limited to finite-state systems, abstraction has to be applied (manually) to infinite parameter domains. On the other hand, we use code generation into Java for component behaviours (see Sect. 9.5.2).

LTSA. For producing the graphs of the I/O-transition systems used in the behavioural analysis of our model and for the analysis of component correctness and behaviour reflection of ports we have used the Labelled Transition System Analyser (LTSA [11]). The LTSA tool supports the process algebra FSP [11] and, indeed, we have defined appropriate FSP processes for most of the transitions systems used in our model for the CoCoME. In this way we have expressed port products by parallel composition with synchronisation on shared labels and we have proved observational equivalence of transition systems by exploiting the minimisation procedure of LTSA. The concrete form of the used FSP processes is not shown here but can be examined in [9].

PEPA. PEPA (Performance Evaluation Process Algebra [5]) is a process algebra that allows for quantitative analysis of the CoCoME using Continuous-Time Markov Chains (CTMC). For the quantitative analysis of Sect. 9.4.2 we used the IPC tool [6].

9.5.2 Architectural Programming

Java/A [1,7] is a Java-based architectural programming language which features syntactical constructs to express Java/A component model elements (see Sect. 9.2) directly in source code. Thus, the implementation of a component-based system using Java/A is straightforward. Additionally, Java/A programs gain robustness with respect to architectural erosion: it is obvious to software maintainers which parts of the code belong to the architecture and therefore need special attention during maintenance.

We implemented a subset of the CoCoME, consisting of one component CashDesk including all of its sub-components and a simplified component Inventory to highlight Java/A as implementation language for component-based systems. Fig. 21 shows the source code of the top level composite component SimplifiedStore which contains the CashDesk and the Inventory. The assembly (l. 2–3) declares the sets of component and connector types which may be used in the configuration of the composite component. The initial configuration, consisting of one cash desk connected to one inventory, is established in the constructor of the composite component (l. 4–11). Components which are declared with the additional keyword `active` will be started after the

```
   composite component SimplifiedStore {
2    assembly { components { Inventory, CashDesk }
                connectors { Inventory.Sale, CashDesk.CDAI; } }
4    constructor Store() {
       initial configuration {
6        active component Inventory inv = new Inventory();
         active component CashDesk cd = new CashDesk();
8        connector Connector con = new Connector();
         con.connect(inv.Sale, cd.CDAI);
10     }
     }
12 }
```

Fig. 21. Java/A composite component SimplifiedStore

```
   simple component CashDeskApplication {
 2   int itemCounter = 0; ...
     port CDACB {
 4     provided { async saleStarted();
                  async productBarCodeEntered(int barcode);
 6                async saleFinished();
                  async paymentModeCash(); ... }
 8     required { void changeAmountCalculated(double amount);
                  void saleSuccess(); }
10     protocol <! behaviour {
         states { initial init;
12                simple a; simple b; simple e; ... simple h; }
         transitions { init -> a;
14                     a -> b { trigger saleStarted; }
                       b -> b { trigger productBarCodeEntered; }
16                     ...
                       e -> h { effect out.saleSuccess(); }
18                     h -> b { trigger saleStarted; }
                     } } !>
20   } ...
     void saleStarted() implements CDACB.saleStarted() {
22     Event event = Event.signal("send saleStarted", new Object[]{});
       this.eventQueue.insert(event);
24   } ...
     void processSaleStarted() {
26     try {
         CDAP.saleStarted();
28       CDACDG.saleStarted();
       }
30     catch (ConnectionException e) { e.printStackTrace(); }
     } ...
32 }
```

Fig. 22. The JAVA/A simple component CashDeskApplication

initialisation process (which basically consists of initialising and connecting the components).

In Fig. 22 we give a very brief overview of the JAVA/A implementation of the component CashDeskApplication.[8] In lines 3–20 the port `CDACB` is declared (see Fig. 9). Provided operations annotated with the keyword `async` instead of a return type are asynchronous. The port protocol (lines 10–19) is specified using the language UTE. In order to verify the absence of deadlocks of the connection of two ports, the JAVA/A compiler is closely integrated with HUGO/RT (see Sect. 9.5.1).[9]

The operations declared in a port's provided interface must have a realisation in the respective port's component. The implementation of the provided operation `saleStarted` of the port `CDACB` is shown in lines 21–24. In the body of the private helper method `processSaleStarted` (lines 25–31) required port operations are invoked. These invocations leave the component's boundaries and therefore the checked exception `ConnectionException` has to be handled.

We have used HUGO/RT to generate Java-based implementations of the state machines to realise the components' behaviour. Thus the components' behaviour adheres

[8] Of course, most of the component's body is omitted here. However, the complete implementation is available online [9].

[9] In contrast to Sect. 9.3, in the JAVA/A implementation port names are written without hyphens due to Java naming conventions.

strictly to the specifications given in the previous sections. However, to use the specified state machines for code generation, a few minor adoptions have been necessary: i.e., calls to required port operations are delegated to internal helper operations (e.g. `processSaleStarted` in Fig. 22, lines 25sqq.) and parameter values of incoming operation calls are stored in helper variables. The complete JAVA/A implementation of the simplified CoCoME is available online [9].

9.6 Summary

Our approach to modelling the CoCoME has been based on a practical component model with a strong focus on implementability and modular (component-wise) verification. In fact our UML based component model was originally introduced for the architectural programming language JAVA/A supporting encapsulation of components by ports. Based on the semantic foundations, we have that our strategy for modular verification of properties of hierarchically constructed components works for the architectural patterns used in the CoCoME. The semantic basis of our functional analysis was given in terms of I/O-transition systems to which we could apply standard operators, for instance for information hiding, thus focusing only on the observable behaviour of components on particular ports. Although the port-based approach alone does not suffice for guaranteeing full component substitutability, the analysis method can be used to ensure special properties, like deadlock-freedom, compositionally. These properties are transferred to the architectural programming language JAVA/A by code generation where deadlock-freedom corresponds to the ability of the components to communicate indefinitely. For non-functional properties, we used continuous-time Markov chains to quantify performance.

Currently we are developing support for modelling runtime reconfigurations of component networks. This will be necessary if the CoCoME requirements would be extended, e.g., to use cases for opening and closing cash desks. Also, the current component model does not directly integrate means for specifying non-functional properties. Our component model assumes that all connectors are binary which, due to the possibility to define ports with an arbitrary multiplicity, is no proper restriction. However, our analysis method actually supports multiplicities greater than one only if the actions of parallel executing instances of the same port or component declaration can be arbitrarily interleaved, which was indeed the case in the example. In the centre of our behavioural analysis was the interaction behaviour of components with asynchronous message exchange via their ports. For synchronous, data-oriented behaviours we still should add assertion-based techniques (e.g., in terms of pre- and post-conditions) whose integration in a concurrent environment, however, needs further investigation.

Acknowledgement. We would like to thank the organisers for the detailed preparation of the common component modelling example. We gratefully acknowledge many very useful and detailed comments made by the referees of a previous version of this study. We also would like to thank Mila Majster-Cederbaum for many fruitful discussions on the topic of interacting systems and their verification.

References

1. Baumeister, H., Hacklinger, F., Hennicker, R., Knapp, A., Wirsing, M.: A Component Model for Architectural Programming. In: Barbosa, L., Liu, Z. (eds.) Proc. 2nd Int. Wsh. Formal Aspects of Component Software (FACS 2005). Elect. Notes Theo. Comp. Sci, vol. 160, pp. 75–96 (2006)
2. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. John Wiley & Sons, New York (1994)
3. Object Management Group: Unified Modeling Language: Superstructure, Version 2.0. Technical report, OMG (2005)
4. Lau, K.K., Wang, Z.: A Survey of Software Component Models (Second Edition). Technical Report CSPP-38, School of Computer Science, The University of Manchester (2006)
5. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press, Cambridge (1996)
6. Bradley, J., Clark, A., Gilmore, S.: User manual for ipc: The Imperial PEPA Compiler (05/02/07), `http://www.doc.ic.ac.uk/ipc`
7. Hacklinger, F.: JAVA/A – Taking Components into Java. In: Proc. 13th ISCA Int. Conf. Intelligent and Adaptive Systems and Software Engineering (IASSE 2004), ISCA, Cary, NC, pp. 163–169 (2004)
8. Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architecture. ACM SIGSOFT Softw. Eng. Notes 17(4), 40–52 (1992)
9. Baumeister, H., Clark, A., Gilmore, S., Hacklinger, F., Hennicker, R., Janisch, S., Knapp, A., Wirsing, M.: Modelling the CoCoME with the JAVA/A Component Model (05/02/07), `http://www.pst.ifi.lmu.de/Research/current-projects/cocome/`
10. Reussner, R., Krogmann, K., Koziolek, H., Rausch, A., Herold, S., Klus, H., Welsch, Y., Hummel, B., Meisinger, M., Pfaller, C., Mirandola, R.: CoCoME — The Common Component Modelling Example. In: The Common Component Modeling Example: Comparing Software Component Models, ch. 3 (2007)
11. Magee, J., Kramer, J.: Concurrency — State Models and Java Programs. John Wiley & Sons, Chichester (1999)
12. Hennicker, R., Janisch, S., Knapp, A.: On the Compositional Analysis of Hierarchical Components with Explicit Ports (submitted, 2007) (06/22/07), `http://www.pst.ifi.lmu.de/Research/current-projects/cocome/`
13. Bernardo, M., Ciancarini, P., Donatiello, L.: Architecting Families of Software Systems with Process Algebras. ACM Trans. Softw. Eng. Methodol. 11(4), 386–426 (2002)
14. Hennicker, R., Baumeister, H., Knapp, A., Wirsing, M.: Specifying Component Invariants with OCL. In: Bauknecht, K., Brauer, W., Mück, T. (eds.) Proc. GI/OCG-Jahrestagung, vol. 157/I of books@ocg.at., pp. 600–607. Austrian Computer Society (2001)
15. Bidoit, M., Hennicker, R.: A Model-theoretic Foundation for Contract-based Software Components (submitted, 2007), `http://www.pst.ifi.lmu.de/people/staff/hennicker/`
16. Buchholtz, M., Gilmore, S., Haenel, V., Montangero, C.: End-to-End Integrated Security and Performance Analysis on the DEGAS Choreographer Platform. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 286–301. Springer, Berlin (2005)
17. Knapp, A.: Hugo/RT Web page (05/02/07), `http://www.pst.ifi.lmu.de/projekte/hugo`