1 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

1.1 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. Our JAVA/A component model is inspired by ideas from "Real-Time Object Oriented Modeling" (ROOM [1]): components are strongly encapsulated 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 [2]), though in a simplified form which, however, keeps the main structuring mechanisms and focuses on strong encapsulation as well as hierarchical composition. In [3], we devised an algebraic semantic framework for this model; furthermore, we introduced an "architectural programming" language, JAVA/A, which allows programmers to represent software architectures directly in the implementation language and thus helps to prevent "architectural erosion" [4].

In contrast to interface-based component approaches (like COM, CORBA, Koala, KobrA, SOFA; see [5] 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, in particular, the representation of different "faces" to other components. Moreover, ports are equipped with behavioural protocols regulating message exchanges according to a particular viewpoint. Similar to JAVA/A, ArchJava [6] and ComponentJ [7] are architectural programming languages which integrate

* 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.

architectural concepts into Java. However, neither ArchJava nor ComponentJ provide means for port protocol or component behaviour specification (aside from source code).

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 then lift these properties to hierarchical, composite components. For quantitative properties, we represent the component semantics, though rather abstractly, in the PEPA process algebra [8] and use continuous-time Markov chains for performance analysis with the IPC tool [9]. The second aim of the JAVA/A approach is the representation of software architecture entities in a programming language. We use the code generation engine HUGO/RT to translate the behaviour of components into Java code; the usage of the same tool for verification and code generation helps in transferring design properties into the code.

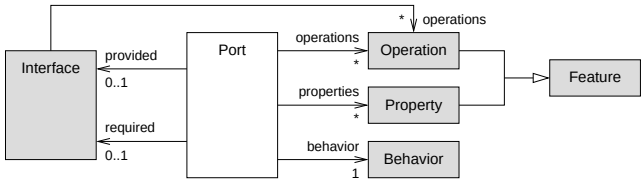
The remainder of this chapter is structured as follows: After a short introduction to the JAVA/A component model in Sect. 1.2 we present (selected parts of) our model of the CoCoME trading system in Sect. 1.3 including the CashDeskLine and the Inventory. In Sect. 1.4 we show how the formal algebraic 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. 1.5 we briefly present the model checking and architectural programming tools we used for the CoCoME and, finally, in Sect. 1.6, we summarise our results and discuss further research issues. The complete model of the CoCoME can be found at [10].

1.2 Component Model

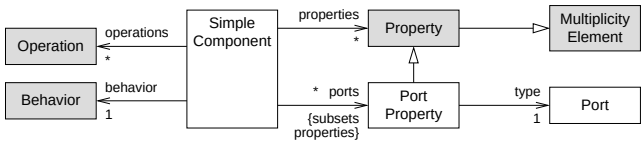
In the JAVA/A component model, components are strongly encapsulated behaviours. Only the exchange of messages with their environment according to provided and required operation interfaces can be observed. The component interfaces are bound to ports which regulate the message exchange by port behaviour 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 [2] 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 [3].

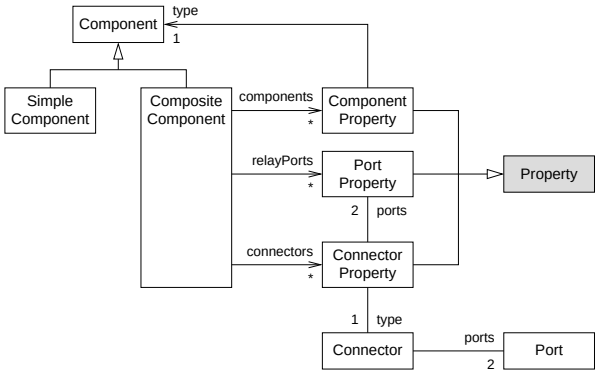
Port. A *port* (see Fig. 1(a)) describes a view on a component (like particular functionality or communication), the operations offered and needed in the context of this view



(a) Port



(b) Simple component



(c) Composite component

Figure 1. JAVA/A component metamodel

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. 21 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 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. 21 (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. 21. 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. 21 (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 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

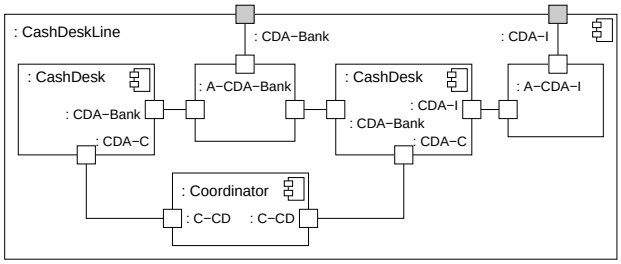


Figure 2. Sample configuration of component CashDeskLine of Fig. 6

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.

1.3 Modelling the CoCoME

We started the design of the CoCoME from the use case descriptions and sequence diagrams as given in [11]. 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 [11], 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 [11] is assumed.

1.3.1 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 showing, on the left-hand side, the component hierarchy as described in [11] and, on the right-hand side, the corresponding modelling within our approach. Behavioural deviations are discussed within dedicated paragraphs of the particular component specifications described in sections 1.3.2 up to 1.3.4.

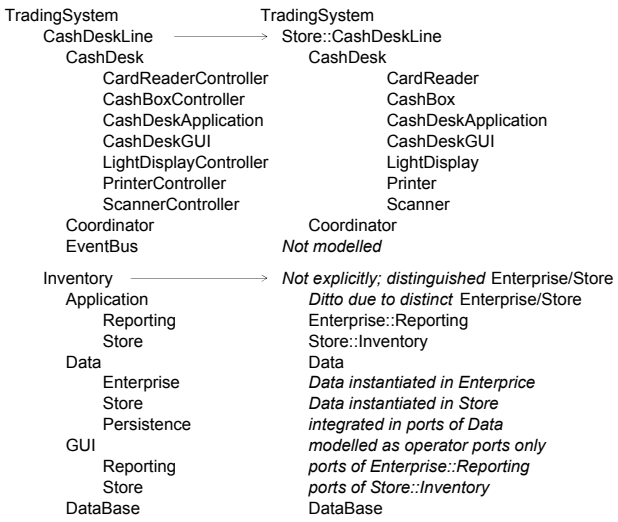


Figure 3. Component hierarchy of given (left) and modelled (right) architecture

As the most apparent structural difference in the embedded system part we did not model the event bus, 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. For this reason, we consider the modelling of an event bus to be an implementation decision whose manifestation should be shown to be correct w.r.t. the behaviour specified in the design model.

The corresponding explicit modelling of the cash desk's internal interaction structure directly constitutes the internal topology of our composite component CashDesk (see Fig. 7) deviating from the cash desk's inner structure as shown in [11, Fig. 1.6]. 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 [11, e.g. Fig. 1.6, 1.6], 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 dis-

tinction is between Reporting and Store. The former is, according to the deployment of Inventory::Application::Reporting on the EnterpriseServer (see [11, Fig. 1.6]), 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 [11], 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 [11, Fig. 1.6] 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.

1.3.2 Trading System — Stores and Enterprises

This section describes the specifications for the root component TradingSystem and the two fundamental components Store and Enterprise. All of these are composed from further components which are described and specified in succeeding sections.

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 indicates that for a proper instantiation of TradingSystem it is strictly required to provide appropriate bank connections to the system.

Beyond, the component TradingSystem allows for several further system configurations. For example the system might be used with store components which are

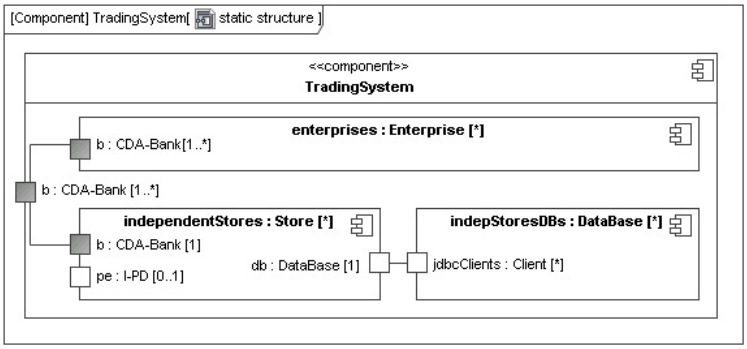


Figure 4. Static structure of the component TradingSystem

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 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, an Inventory 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. Also, 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.

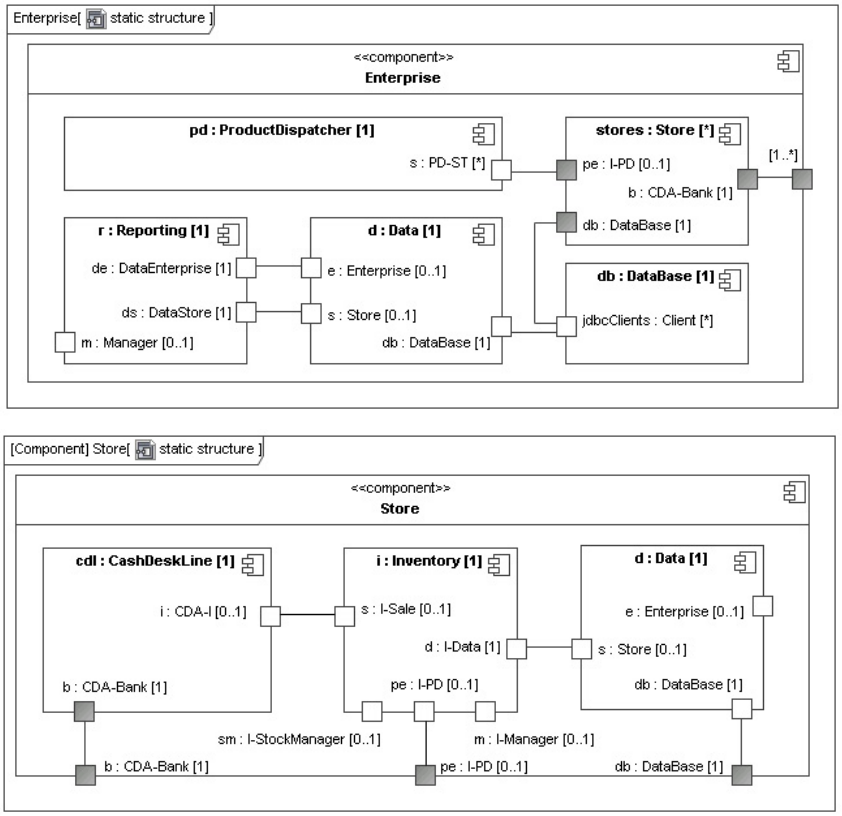


Figure 5. Static structure of the components Enterprise and Store

1.3.3 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).

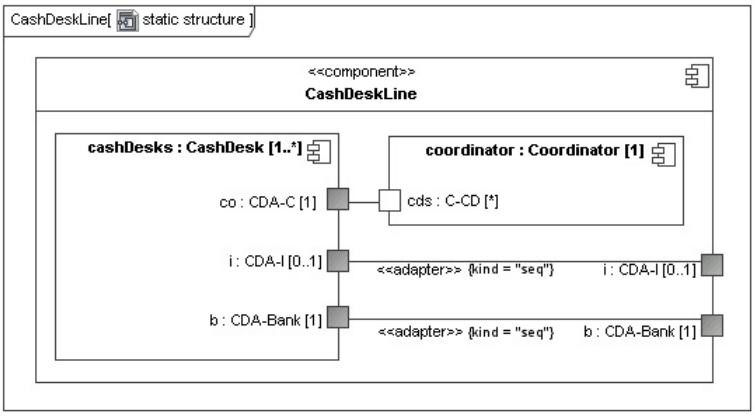


Figure 6. Static structure of the component CashDeskLine

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 line follows the CoCoME requirement in [11, Fig. 1.6] 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 [11]) 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 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.

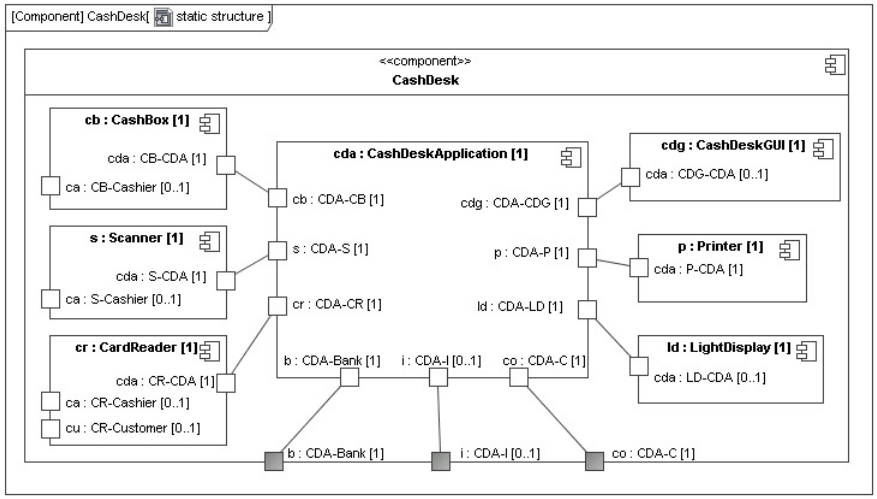


Figure 7. Static structure of the component CashDesk

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 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 "input" port behaviours of Fig. 9, Fig. 10 shows the "output" port behaviours of the CDA. The behaviour of CDA-CDG, intended to connect to the cash

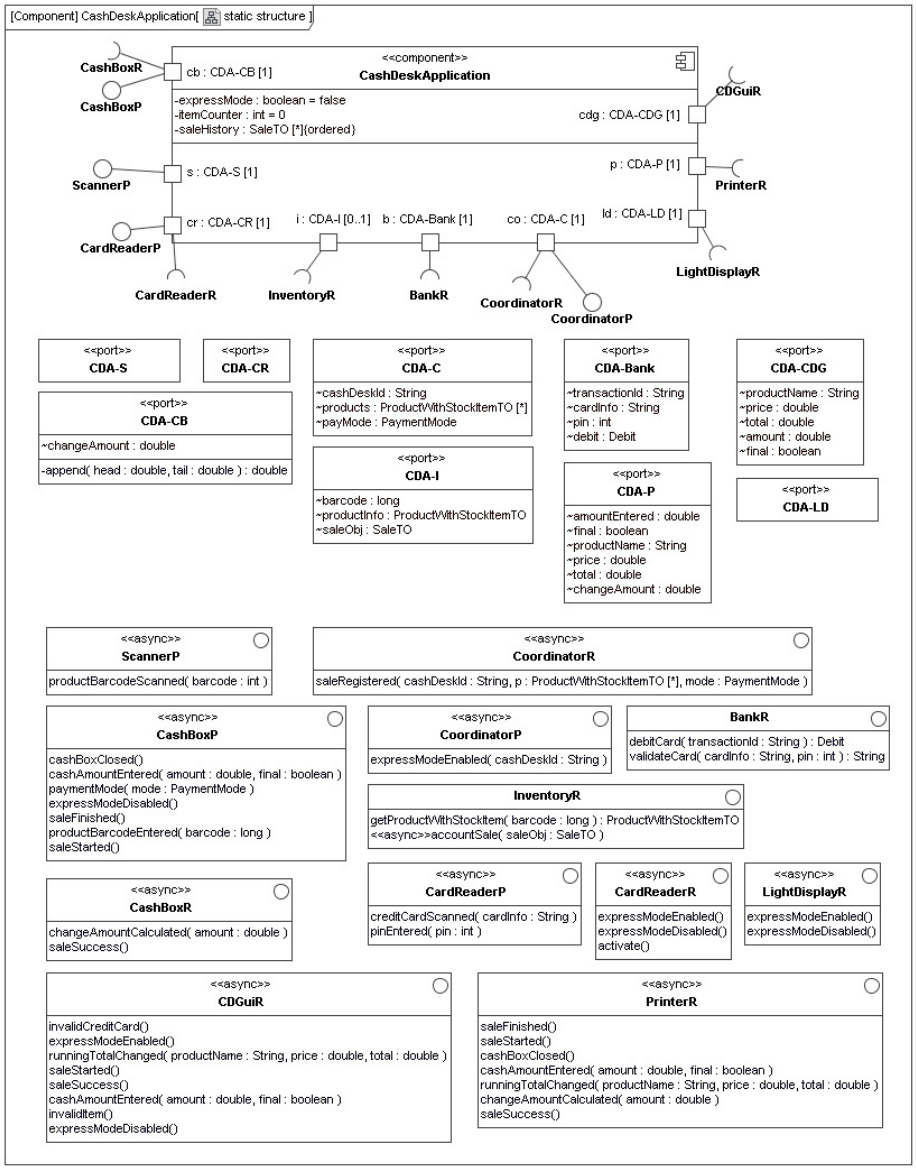


Figure 8. Static structure of the CDA component

desk's GUI is very similar to the specification of CDA-P, connecting to the printer. 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 (CDA-LD in Fig. 10), only the GUI is notified of mode switches from or to express mode.

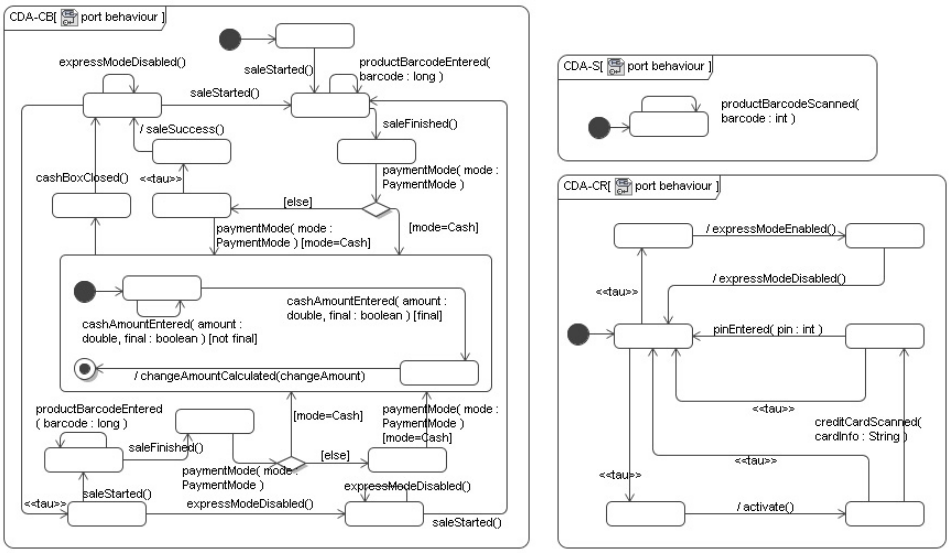


Figure 9. Behaviour of the CDA ports CB, S and CR

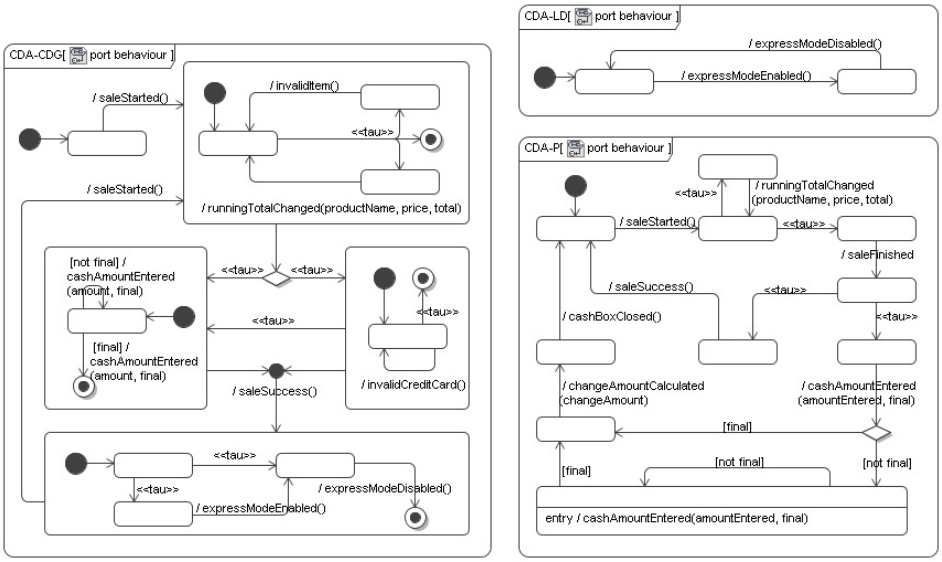


Figure 10. Behaviour of the CDA ports CDG, LD and P

The communication with components external to the cash desk is prescribed by the port behaviour specifications for the communiciation with the inventory (CDA-I), the bank (CDA-Bank) and the cash desk's coordinator (CDA-C) shown in Fig. 11. The communication with the inventory and the bank demonstrates our modelling of synchronous

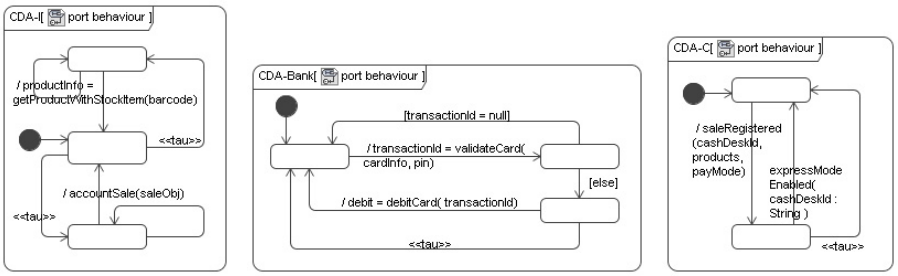


Figure 11. Behaviour of the CDA ports I, C and Bank

communication, which usually assigns returned values to port attributes. In general, the particular values are either used to evaluate succeeding guards in the port behaviour specification or might be read by an implementation of the component which owns the respective port. The port behaviours shown in Fig. 11 illustrate the former case. Finally, the behaviour of CDA-C, specifically the application of τ, should be noticed. It is used in Sect. 1.4.1, in combination with the behaviour specifications of the coordinator (see Fig. 21), to illustrate our approach to the analysis of functional requirements.

CDA Component Behaviour. Figure 12 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, for instance, as actual parameters in messages sent.

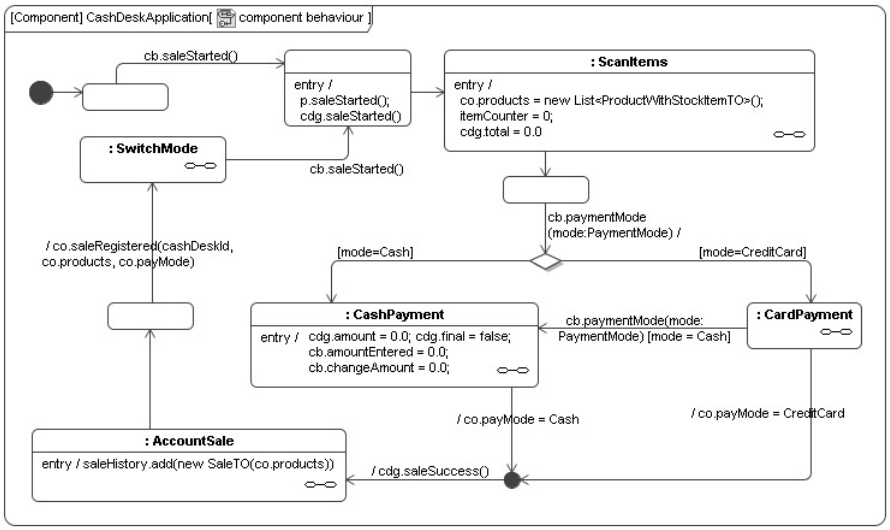


Figure 12. Component behaviour of CashDeskApplication (for details see Fig. 13)

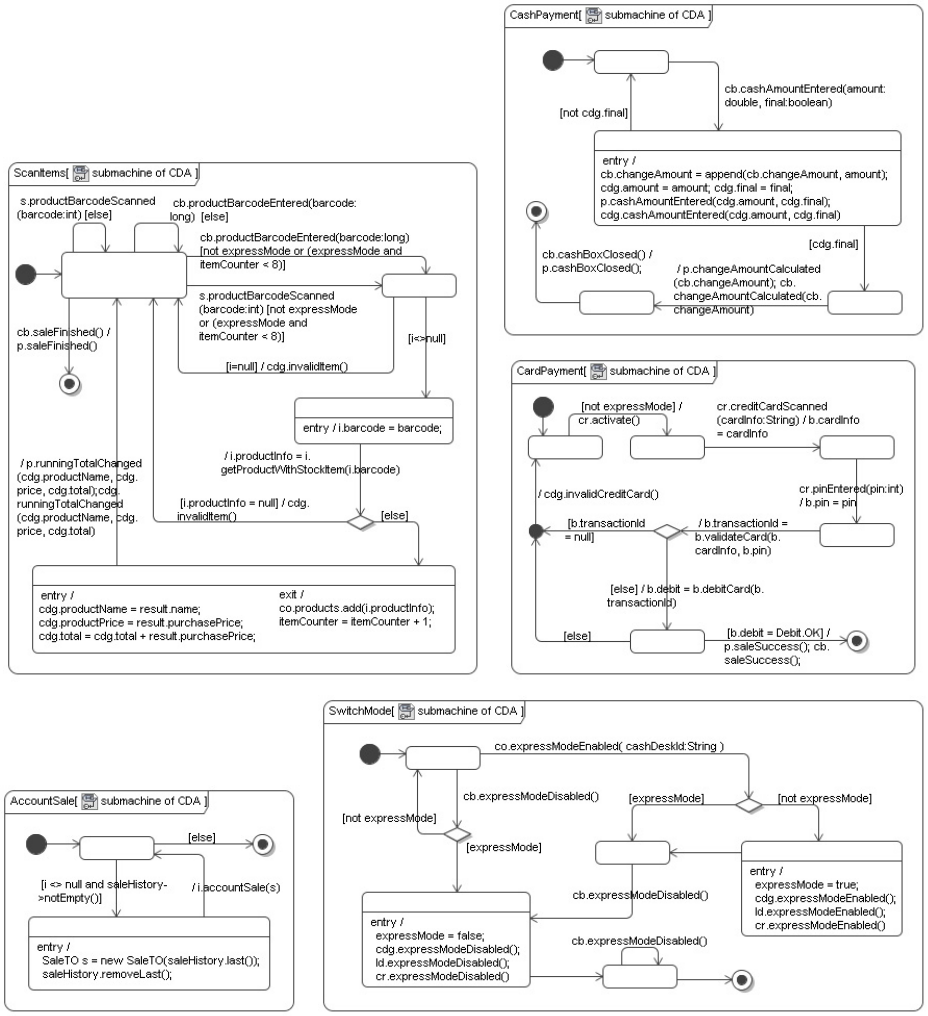


Figure 13. Submachine specifications for the CDA component (see also Fig. 12)

Since the specification of the cash desk application's behaviour is rather involved we used submachines, shown in Fig. 13, to factor out the major steps of the entire sale process: 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 such as name and price. Thereafter the payment mode must be chosen, resulting in a transition to the corresponding submachine. Note that the execution of CardPayment might be cancelled 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. Then, before the sale process is completed, the component tries to account the sale at the inventory within AccountSale. If it is not available (which is an explicit requirement of the CoCoME), 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.

Deviation. Due to the design decision to model the cash desk component with the CDA at the center, there are some minor deviations from behaviour of the CDA as given in the CoCoME. First, within the main sale process [11, Fig. 1.6] the signals saleStarted and saleFinished are sent by the cash box to the CDA only. The application notifies the printer and the GUI subsequently, instead of a direct communication between the cash-box and the other hardware devices. Similar, the signals cashBoxClosed [11, Fig. 1.6] and expressModeDisabled [11, Fig. 1.6] are distributed by the cash desk application instead of communicating along a direct linkage between the hardware components. Furthermore, the signal activate was introduced with the port behaviour of CDA-CR to cope with a deadlock possibility in the communication with the card reader and, finally the signal invalidItem was added to the communication with the GUI, specified by the port behaviour of CDA-CDG in order to signal problems during product scan or product information retrieval from the inventory. The latter is due to an exceptional process of Use Case 1 in the CoCoME.

CashBox (CB). The component CashBox, depicted in Fig. 14, declares two ports, an optional operator port of type CB-Cashier and a mandatory port of type CB-CDA. The former is a model of the possible cashier inputs, the latter specifies the behaviour w.r.t. the cash desk application. The message productBarcodeEntered at the CB-Cashier port is not mentioned in the sequence diagrams but in the standard process and also in an exceptional process of Use Case 1 of the CoCoME. It allows the cashier to manually enter a product bar code in case there are problems with the bar code scanner.Internal τ-transitions are used in both behaviour specifications. In CB-Cashier, the transition specifies an internal decision which, once triggered, indicates that the signal cashPayment is not processed any more, but the port expects the client to send disableExpressMode or startNewSale instead. In the behaviour of CB-CDA the τ-transition is used to specify a precedence of successful card payment over the internal decision to switch from card into cash payment mode. In this case, the τ-transition also shows the assignment of the port attribute mode which is used as an actual parameter in the subsequent send transition to the composite state.

The component behaviour of CashBox, shown in Fig. 15, essentially maps the cashier's input at the ca port directly to a corresponding notification at the cda port.

Deviation. We introduced the signal saleSuccess at the CB-CDA port in order to enable the cash box being notified of a successful completion of a card payment. Besides, the sequence diagrams of the CoCoME show a signal openCashBox [11, Fig. 1.6] in reaction to the reception of a message changeAmountCalculated, which does not occur explicitely in our model, because we merged the given functional component CashBox with the corresponding controller component CashBoxController. Hence the signal openCashBox disappears from the specification. However, since the passed change amount is not processed within the cash box component, this message might be re-

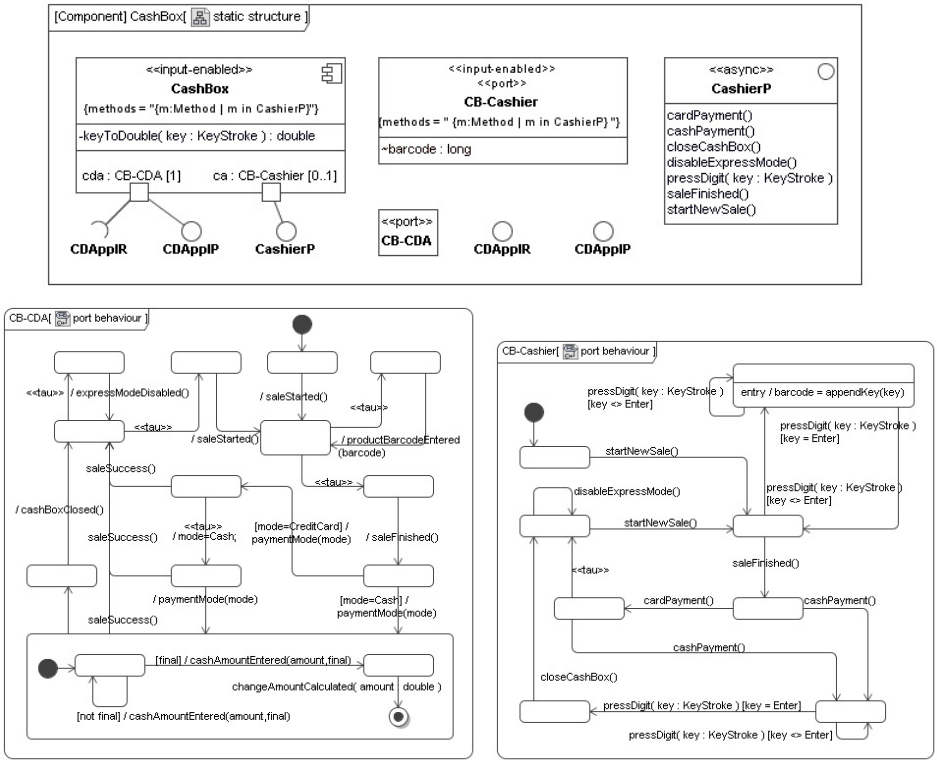


Figure 14. Static structure and port behaviour of CashBox

placed with an external signal openCashBox anyway. We refrained from doing so, in order to keep the deviation from the CoCoME specification manageable.

CardReader (CR). Figure 16 shows the static structure as well as the behaviour specifications of the component CardReader. The component declares three ports, two of them trivial operator ports (CR-Cashier and CR-Customer) which have been omitted from Fig. 16, and one of them, CR-CDA, modelling the communication with the cash desk application from the card reader's point of view. There are two main functional regions in the behaviour specification of CR-CDA. On the one hand, the card reader may be deactivated by having received a message expressModeEnabled. On the other hand, within the lower region reached by the reception of a signal activate, the port possibly engages in sending credit card information. From any of the "active" states the card reader may be again be deactivated by the message expressModeEnabled.

Deviation. We introduced the operator messages enterPin and pullCard with parameters passing the particular data in analogy to pressDigit as used in the CoCoME in the context of the cash box component. Additionally we extended the communication between cash desk application and card reader with an explicit activate signal. The Cashier may

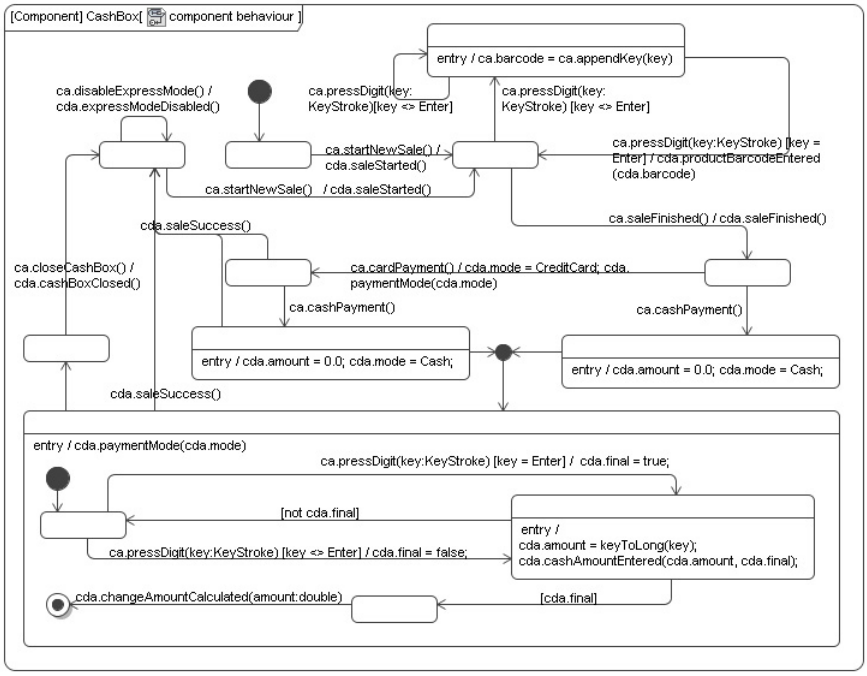


Figure 15. Component behaviour of CashBox

decide at any point in time to switch from payment mode CreditCard to Cash. Therefore, if the particular cash desk is in normal mode, the card reader might be left in any state. In order to reset the card reader to the correct state in consequence of a new sale process the signal activate was introduced. Alternatively it might be better to introduce messages such as saleStarted and saleFinished similar to the communications between CDA and the other devices also for the card reader. This would have probably simplified the port as well as the component behaviour specifications. Anyway, again we did not want to deviate too much from the provided CoCoME specifications in [11] and leave this issue as a side note.

Scanner (S). The bar code scanner is modelled by a rather simple component Scanner, depicted in Fig. 17. We omitted the trivial behaviour specifications for both of the ports showing only the specification of the component behaviour; a mere sequential combination of the port behaviours. This simple specification also illustrates the case of an optional port behaviour (ca:S-Cashier) which must not be taken into account in the component behaviour explicitly. If the port instance ca does not exist, this component simply does nothing.

Printer (P). The Printer component declares only one port using the port type P-CDA whose behaviour is specified alongside the static structure in Fig. 18. We omitted the component behaviour specification, because it is a perfect match of the port behaviour (modulo port identifier cda).

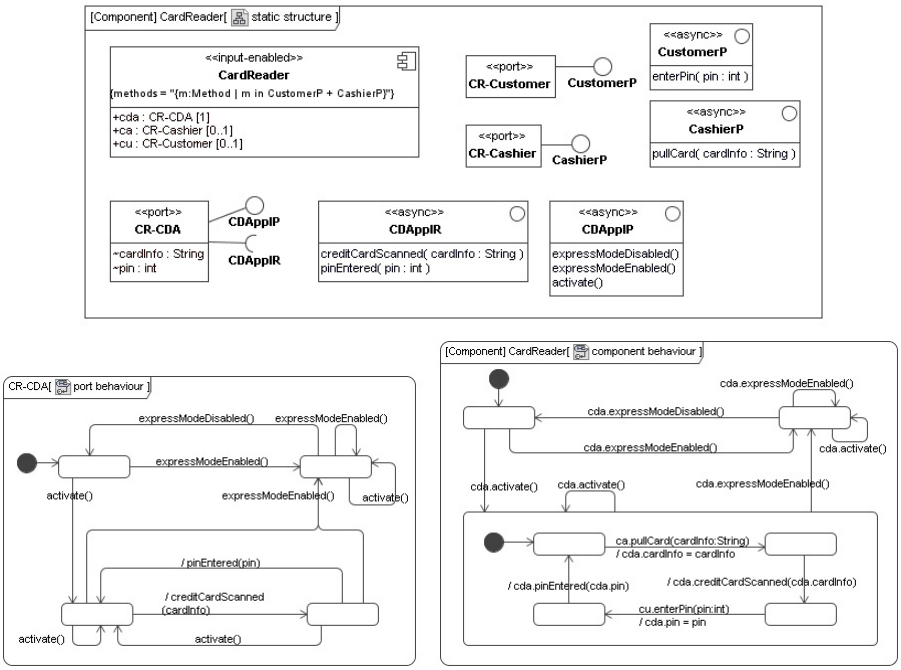


Figure 16. Static structure and behaviour specifications of CardReader

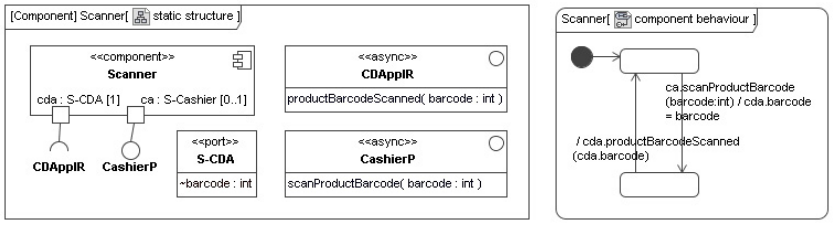


Figure 17. Static structure and behaviour of the Scanner component

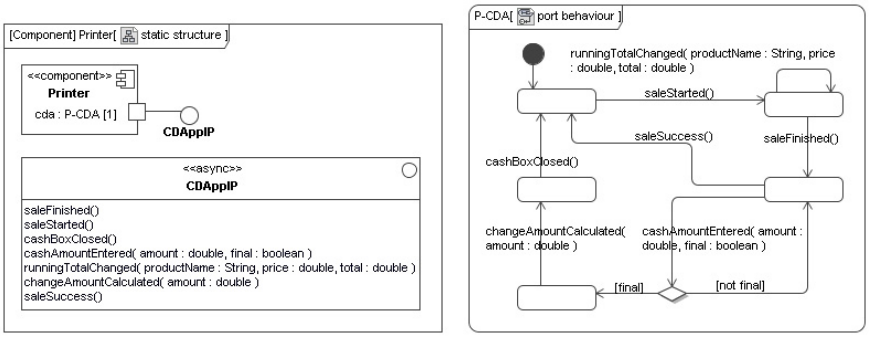


Figure 18. Static structure and port behaviour of the Printer component

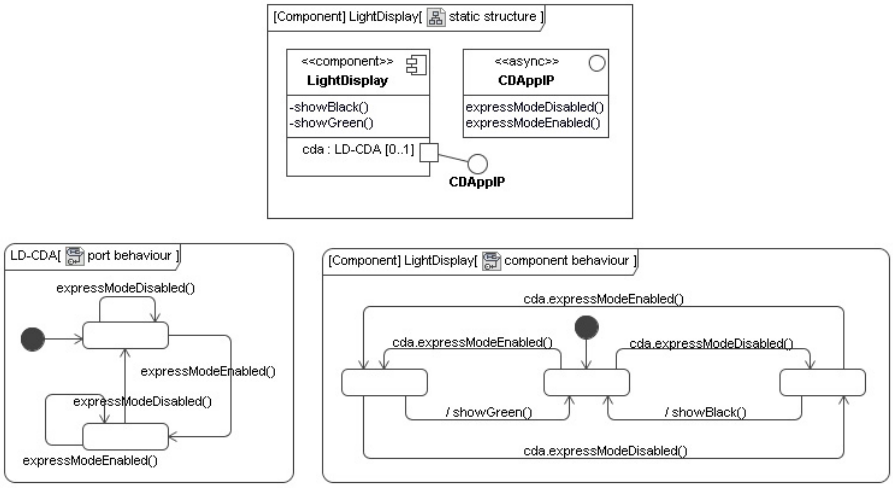


Figure 19. Static structure and behaviour of the LightDisplay component

LightDisplay (LD). Cash desks may be switched to an express mode which is signalled at the cash desks LightDisplay. As depicted in Fig. 19, there is only one port declaration whose port type shows essentially trivial behaviour. Both signals, expressModeEnabled and expressModeDisabled, are always accepted. Initially only the former triggers a state change, encoding the switch to express mode and hereafter only the latter triggers the state change to normal mode back again. The component behaviour specifies internal behaviour in form of private operations representing signals for the hardware device to show green and black lights as required by the CoCoME respectively.

CashDeskGUI (CDG). The static structure of the component CashDeskGUI modelling the graphical user interface of a cash desk is depicted in Fig. 20. We omitted any behaviour specification, since according to the reference implementation the GUI allows its clients to send messages and signals in any order, i.e., without any behavioural restriction. Note that the signal invalidItem is due to the exceptional process in Use Case 1 of the CoCoME: signal an error in case there are problems with the product item identifier entered by the cashier or scanned by the bar code scanner.

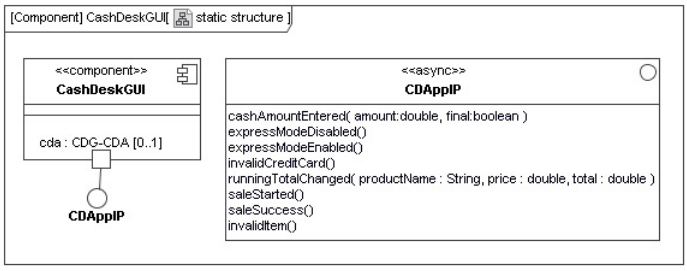


Figure 20. Static structure of the CashDeskGui component

The specification of the CashDeskGUI completes the presentation of our specifications for the parts of the composite component CashDesk (see Fig. 7). Next we complete the specification of CashDeskLine, one level up in the hierarchy, with a brief discussion of the Coordinator component.

Coordinator (C). The CashDeskLine (see Fig. 6) of a store consists of a number of cash desks and an instance of Coordinator, specified in Fig. 21, which decides on the mode of the cash desks (express or normal). The component declares its port of type C-CD

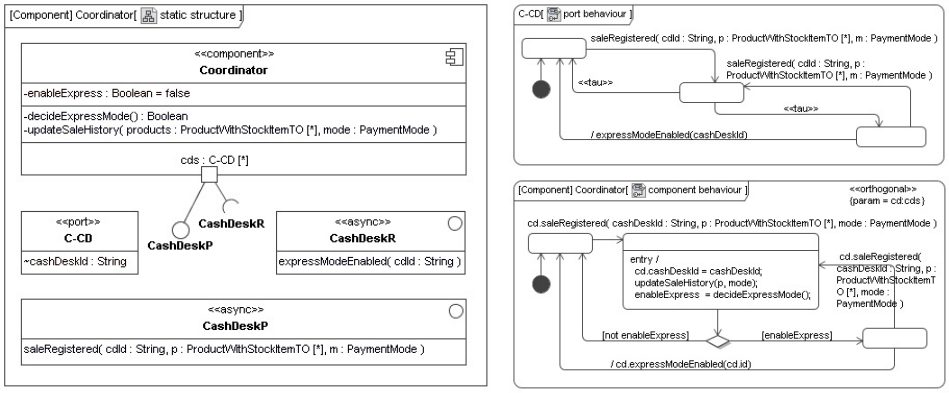


Figure 21. Static structure and behaviour specifications of Coordinator

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 to signal expressModeEnabled, 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 receives the signal saleRegistered and recomputes its internal decision.

The component behaviour shown in Fig. 21 illustrates the reification of internal actions which are hidden by τ transitions in a port behaviour specification.[4] Here, the component keeps track of the particular sale history for each cash desk and decides upon this history to signal an express mode switch for this particular cash desk. The update of the sale history is required to be synchronised (not shown in the diagram) due to the concurrent execution of the port instances cd in cds.

1.3.4 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 [11, Figs. 1.6,1.6,1.6,1.6]), which 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. Therefore, we

[4] Of course, this has been illustrated with the specification of the CDA implementation already, but there, without doubt, more difficult to comprehend due to the mere size of the specification.

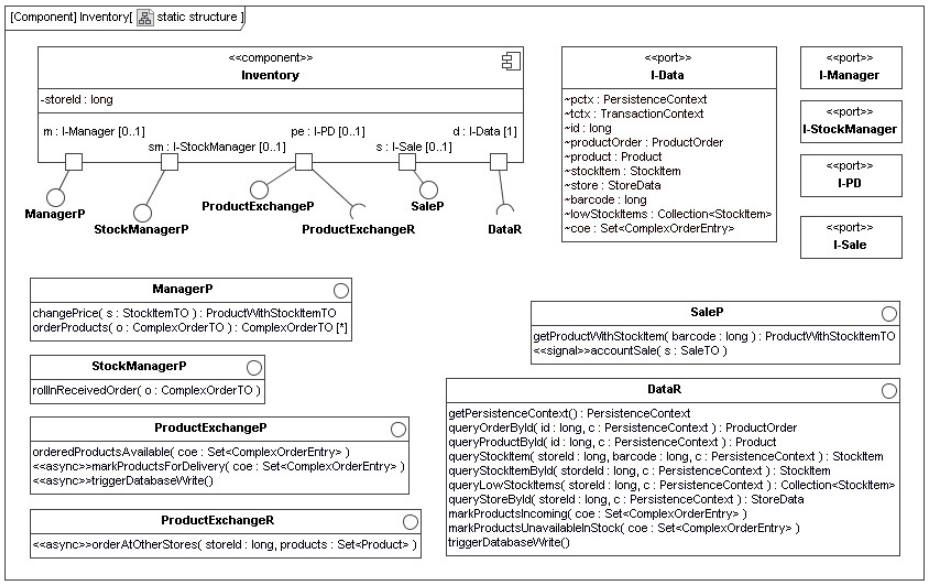


Figure 22. Static structure of the component Inventory

provide a similarly detailed discussion of its specification as we did with the cash desk application in Sect. 1.3.3. The most prominent new modelling aspect with respect to behaviour specifications discussed above is the specification of synchronous message call receptions. Afterwards, in order to complete our specifications of the static structure of this part of the trading system, we briefly describe the static structure of the components Data, Reporting, ProductDispatcher and DataBase. The particular port and component behaviour specifications would essentially mirror the behaviour as specified in the behaviours of Inventory. Also, they do not illustrate new modelling features, hence we do not provide explicit specifications thereof.

Inventory (I). The component Inventory[5] is a model of the store's portion of the application layer of the CoCoME. As depicted in the static structure of Fig. 22, 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. 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. 23 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. Finally, the component declares a

[5] Note that, as discussed in Sect. 1.3.1, Inventory models Inventory::Application::Store of [11].

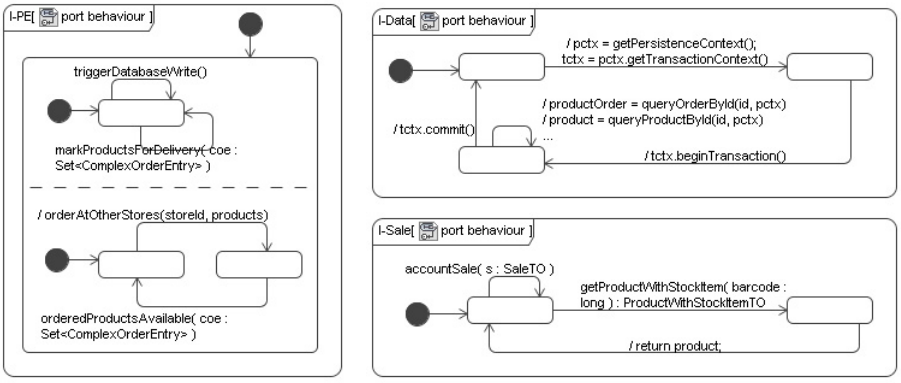


Figure 23. Port behaviour of the component Inventory (omitted trivial port behaviours)

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. 23 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 [11, Fig. 1.6]).

The component behaviour specification of Inventory (Fig. 24) comprises of six orthogonal regions, essentially each of them modelling the interaction with one possible communication partner along the ports of the component. Any communciation with the data layer, i.e., with the port d is framed by an update of the transaction context reference and the explicit transaction begin and commit. The entry actions in the second state of the top-most orthogonal region exemplify the corresponding message calls. For notational convenience we omitted these calls in the remaining states and regions and assume that the respective component behaviour with respect to d always takes the required framing into account the port behaviour of I-Data.

The first upper region specifies effects due to messages received at port s. We omitted a detailed specification of the reaction to the asynchronous message accountSale. An implementation would use the port d in order to account the sale information in the data base. In contrast, for getProductWithStockItem we show the transaction-related part of the implementation in between message reception and return in order to illustrate a kind of body specification for the implementation of synchronous message receptions. The second and third orthogonal state show the processing of the synchronous operator commands of the ports I-Manager and I-StockManager, again without further implementation details concerning the interaction with the data layer. The fourth and fifth regions specify effects on port d due to messages received at port pe, more concretely due to requests stemming from the product dispatcher in the course of executing

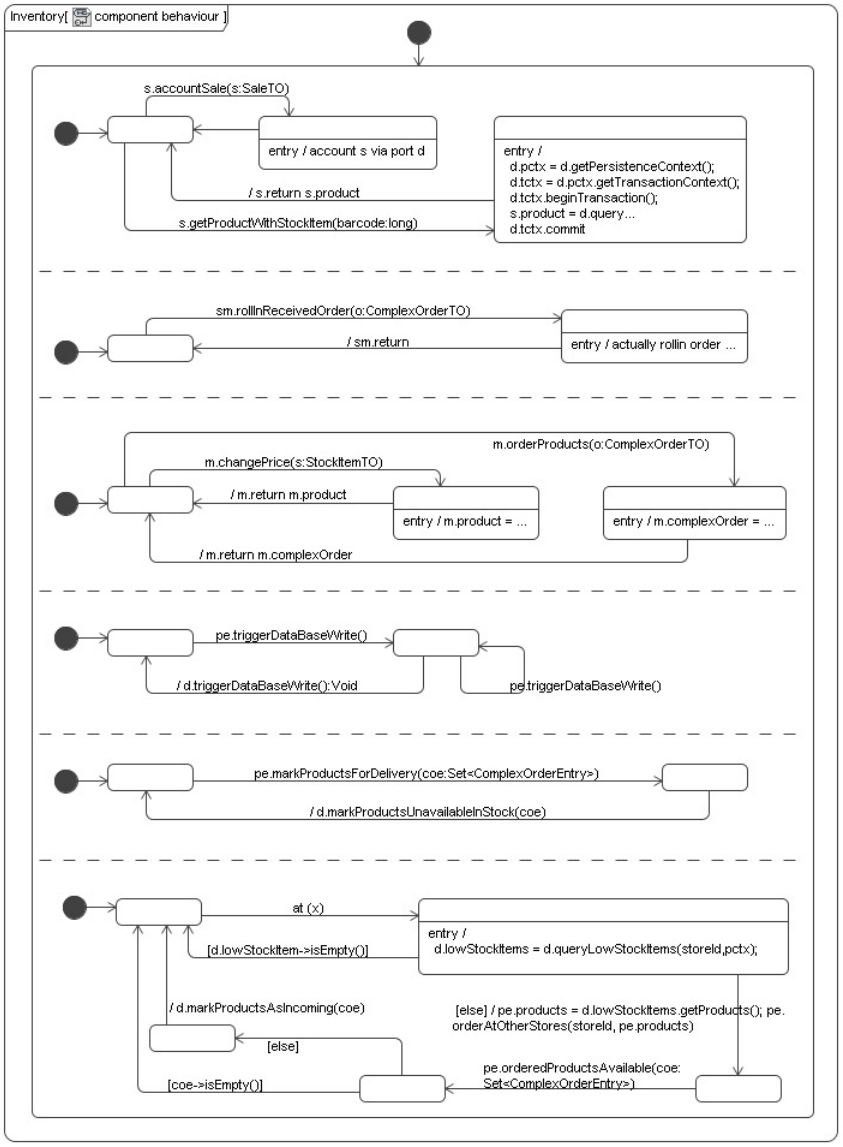


Figure 24. Component behaviour of the Inventory component

a product exchange among stores. Last, the lowermost orthogonal 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 a not further specified time period x.

Deviation. Besides the above mentioned deviation from the CoCoME's static structure (Inventory::Application::Store of [11] vs Inventory within our architecture; see Sect. 1.3.1), we used the persistence and transaction context in compliance with the

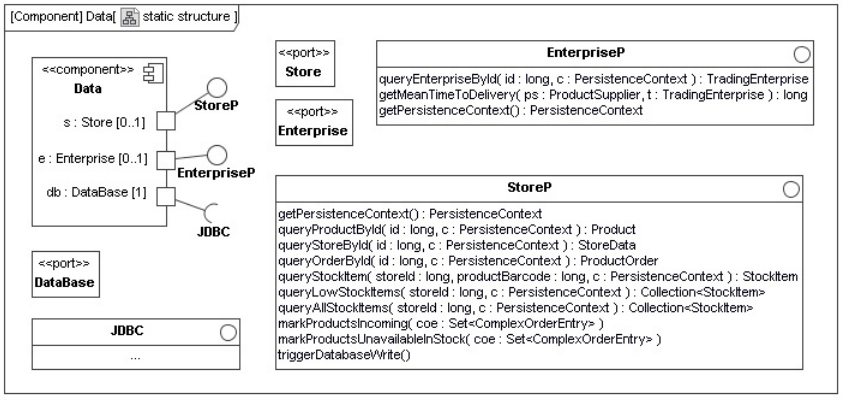


Figure 25. Static structure of the component Data

reference implementation of CoCoME. Note that this is in contrast to the sequence diagrams in [11, Figs. 1.6 – 1.6]. The latter do not show any use of PersistenceContext but of TransactionContext only. Further minor deviations are some obvious renamings of operations used in the sequence diagram [11, Fig. 1.6] modelling Use Case 8 as well as the renaming of the data type Store of the CoCoME's data model to StoreData in order to avoid confusion with the component Store of our modelling.

Data (D). The approach to a three-layered architecture for the information system part yields a component Data similar to the architecture in [11]. The component also hides the concrete data base access from the application layer but in our case this layer connects via two port declarations as specified in Fig. 25. Both ports are optional, and indeed the component is used as part within the composite component Store (see Fig 5) definitely without connecting the enterprise port e. In fact this port is relevant only in "enterprise context" namely as part of the composite component Enterprise in Fig. 5. Besides these distinct data views, the component declares a data base port db which currently shows a JDBC interface without any further details only.

Reporting (R). From the trading system's architect point of view CoCoME essentially consists of two main domain concepts, store and enterprise which both have different requirements concerning the application layer of the information system part. Whereas Inventory was designed having the requirements of store components in mind, the component Reporting (Fig. 26) was designed for the use cases specifying the enterprise requirements to the system. The static structure shows the addition of an enterprise data view to the "conventional" store view through the application of the port type DataEnterprise. The declaration m:Manager again specifies an operator port. The latter directly reflects the commands for report creation as required by Use Cases 5 and 6.

ProductDispatcher (PD). The component ProductDispatcher is the essential counterpart of the inventory in the modelling of Use Case 8, product exchange among stores of the same enterprise, of the CoCoME [11, Fig. 1.6]. Its responsibility is the implementation of heurestics and decisions concerning the processing of a store's request to

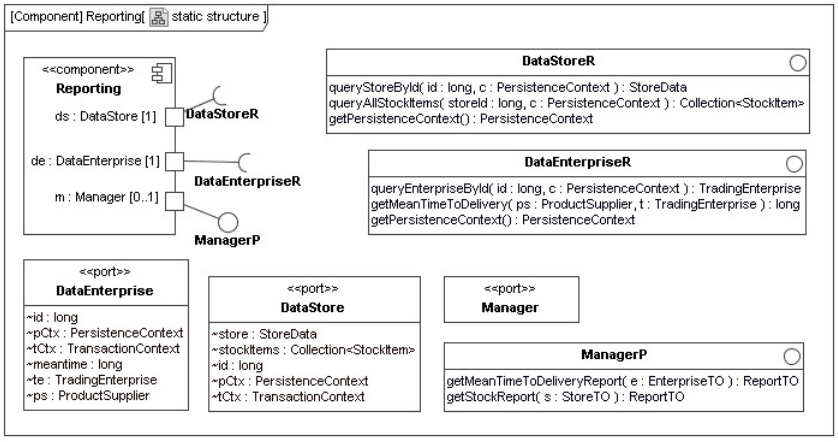


Figure 26. Static structure of the component Reporting

order products at other stores belonging to the same enterprise. The diagram in Fig. 27 shows a corresponding static structure specification. The port interfaces directly mirror the interfaces of the respective port of the Inventory component. The unbounded port multiplicity reflects the possibility to provide as many instances of the port as store components exist to allow the binary connection with the port pe (see Fig. 5.)

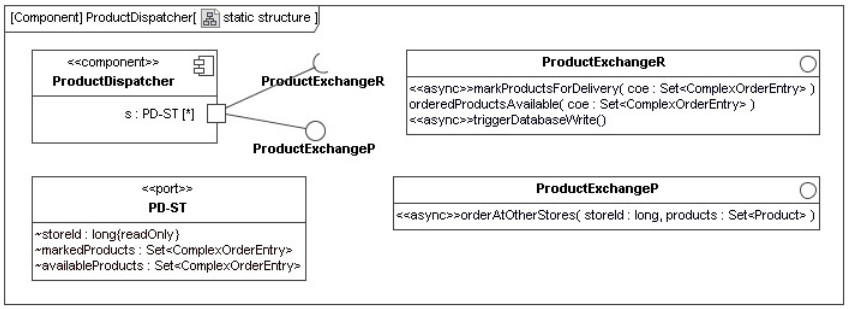


Figure 27. Static structure of the ProductDispatcher component

DataBase (DB). Since the component DataBase hides the concrete data base according to a classical three-layered architecture for information systems, there remains not to much to specify for this component, hence we omitted its explicit specification from this chapter. The static structure comprises besides a multitude of delegate functionality a port jdbcClients with an unbounded multiplicity which allows to connect as many data layer of stores and enterprises via JDBC as required (see Fig. 5).

With the component Database, the specifications of all components applied as part of the composite components Enterprise and Store (see Fig. 5), and specifically of the

root composite component TradingSystem (see Fig. 4), have been discussed, providing the basis for the formal analysis of our architecture as described in the next section.

1.4 Analysis

1.4.1 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 Section 1.3. We cosider 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 concerning message exchange and parallel execution, which are in the center 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 used 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.

Preliminaries. We first summarise the needed definitions for I/O-transition systems which are inspired by the interface automata approach of Alfaro and Henzinger [12].

I/O-transition systems. An *I/O-labelling* (I, O, T) consists of three mutually disjoint sets of *input* (or *provided*) labels I, *output* (or *required*) labels O, and *internal* labels T. Additionally, we assume that there is a special *invisible* (or *silent*) action τ.[6] An *I/O-transition system* $A = (Q, q_0, \Delta)$ over an I/O-labelling (I, O, T) is given by a set of *states* Q, an *initial state* $q_0 \in Q$ and a *transition relation* $\Delta \subseteq Q \times (I \cup O \cup T \cup \{\tau\}) \times Q$.

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

The set of *labels* of A is given by $Label(A) = I \cup O \cup T$; the set of *actions* of A is given by $Action(A) = Label(A) \cup \{\tau\}$. We define the τ-closure of Δ as $\hat{\Delta} \subseteq Q \times (I \cup O \cup T \cup \{\tau\}) \times Q$ as follows: For an $l \in Label(A)$, $(q, l, q') \in \hat{\Delta}$, if there are $q = q_0, q_1, \ldots, q_m \in Q$ and $q'_0, \ldots, q'_n = q' \in Q$ such that $(q_i, \tau, q_{i+1}) \in \Delta$ for $0 \leq i < m$, $(q_m, l, q'_0) \in \Delta$, and $(q'_j, \tau, q'_{j+1}) \in \Delta$ for $0 \leq j < n$; and $(q, \tau, q') \in \hat{\Delta}$ if there are $q = q_0, q_1, \ldots, q_n = q' \in Q$ such that $(q_i, \tau, q_{i+1}) \in \Delta$ for $0 \leq i < n$.

An *I/O-transition system* $A = (Q, q_0, \Delta)$ is *deadlock-free*, if for all states $q \in Q$ reachable from q_0 by transitions of Δ there exists a transition $(q, l, q') \in \hat{\Delta}$ with $l \in Label(A)$.

Two I/O-transition systems $A = (Q_A, q_{0,A}, \Delta_A)$ and $B = (Q_B, q_{0,B}, \Delta_B)$ over the same I/O-labelling (I, O, T) are *observationally equivalent* [13], denoted by $A \approx B$, if there exists a weak bisimulation relation R between A and B with $(q_{0,A}, q_{0,B}) \in R$. A relation $R \subseteq Q_A \times Q_B$ is a *weak bisimulation relation* between A and B, if for all $(q_A, q_B) \in R$ and all $l \in Action(A)$ the following holds:

1. $\forall q'_A \in Q_A \,.\, (q_A, l, q'_A) \in \Delta_A \supset \exists q'_B \in Q_B \,.\, (q_B, l, q'_B) \in \hat{\Delta}_B \wedge (q'_A, q'_B) \in R$
2. $\forall q'_B \in Q_B \,.\, (q_B, l, q'_B) \in \Delta_B \supset \exists q'_A \in Q_A \,.\, (q_A, l, q'_A) \in \hat{\Delta}_A \wedge (q'_A, q'_B) \in R$

Observational equivalence is compatible with deadlock-freeness.

Operators on I/O-transition systems. A *relabelling* $\lambda : L \to L'$ from an I/O-labelling $L = (I, O, T)$ to an I/O-labelling $L' = (I', O', T')$ consists of three functions $\lambda_I : I \to I', \lambda_O : O \to O'$, and $\lambda_T : T \to T'$. For simplicity, we write $\lambda(l)$ instead of $\lambda_I(l)$ if $l \in I$, and similarly if $l \in O$ or $l \in T$. Let $A = (Q, q_0, \Delta)$ be an I/O-transition system over L and let $\lambda : L \to L'$ be a relabelling. The *relabelling* of A w.r.t. λ is the I/O-transition system $A\lambda$ over L' defined by $A\lambda = (Q, q_0, \Delta\lambda)$ where $\Delta\lambda = \{(q, \lambda(l), q') \mid (q, l, q') \in \Delta \wedge l \neq \tau\} \cup \{(q, \tau, q') \mid (q, \tau, q') \in \Delta\}$. In various cases we will need a simple form of relabelling where the labels of A are just prefixed by a given name, say n. Then we write $n.A$ for the relabelling $A\lambda_n$ with $\lambda_n(l) = n.l$ for all $l \in I$, and similarly for $l \in O$ or $l \in T$ where λ_n is assumed to preserve the kinds of the labels.

The *hiding* of an I/O-labelling $L = (I, O, T)$ w.r.t. a subset $H \subseteq I \cup O \cup T$ is the I/O-labelling $L \setminus H = (I \setminus H, O \setminus H, T \setminus H)$. Let $A = (Q, q_0, \Delta)$ be an I/O-transition system over L and let $H \subseteq Label(A)$. The *hiding* of A w.r.t. H is the I/O-transition system $A \backslash H$ over $L \backslash H$ defined by $A \backslash H = (Q, q_0, \Delta \backslash H)$ where $\Delta \backslash H = \{(q, \tau, q') \mid (q, l, q') \in \Delta \wedge l \in H\} \cup \{(q, l, q') \mid (q, l, q') \in \Delta \wedge l \notin H\}$.

Two I/O-labellings $L_1 = (I_1, O_1, T_1)$ and $L_2 = (I_2, O_2, T_2)$ are *composable* if $I_1 \cap I_2 = \emptyset, O_1 \cap O_2 = \emptyset, T_1 \cap (I_2 \cup O_2 \cup T_2) = \emptyset$, and $T_2 \cap (I_1 \cup O_1 \cup T_1) = \emptyset$. The *shared* labels of L_1 and L_2, written $L_1 \cap L_2$, are given by $(I_1 \cup O_1 \cup T_1) \cap (I_2 \cup O_2 \cup T_2)$ which, if L_1 and L_2 are composable, is just $(I_1 \cap O_2) \cup (O_1 \cap I_2)$. The *product* of two composable I/O-labellings L_1 and L_2 is the I/O-labelling $L_1 \otimes L_2 = ((I_1 \cup I_2) \setminus (L_1 \cap L_2), (O_1 \cup O_2) \setminus (L_1 \cap L_2), T_1 \cup T_2 \cup (L_1 \cap L_2))$. Let $A_1 = (Q_1, q_1, \Delta_1)$ and $A_2 = (Q_2, q_2, \Delta_2)$ be two I/O-transition systems over composable I/O-labellings L_1, L_2, respectively. The *product* of A_1 and A_2 is the I/O-transition system $A_1 \otimes A_2 = (Q, q_0, \Delta)$ over $L_1 \otimes L_2$ defined as follows:[7]

[7] Intuitively, the product of A_1 and A_2 describes the parallel composition of A_1 and A_2 where coinciding input and output labels are synchronised and then become internal labels.

1. $Q = Q_1 \times Q_2$;
2. $q_0 = (q_1, q_2)$;
3. $\Delta = \{((q_1,q_2), l, (q_1',q_2)) \mid (q_1,l,q_1') \in \Delta_1 \wedge l \notin L_1 \cap L_2 \wedge q_2 \in Q_2\} \cup$
   $\{((q_1,q_2), l, (q_1,q_2')) \mid (q_2,l,q_2') \in \Delta_2 \wedge l \notin L_1 \cap L_2 \wedge q_1 \in Q_1\} \cup$
   $\{((q_1,q_2), l, (q_1',q_2')) \mid (q_1,l,q_1') \in \Delta_1 \wedge (q_2,l,q_2') \in \Delta_2 \wedge l \in L_1 \cap L_2\}$.

In several cases, for synchronisation of A_1 and A_2, some labels of A_1 and of A_2 must first be identified before the product operator is applied. For this purpose we use the synchronised product defined in the following way: Let n_1, n_2 be two names. Then $A_1 \underset{(n_1,n_2)}{\otimes} A_2$ denotes the product $A_1\lambda_\mathrm{l} \otimes A_2\lambda_\mathrm{r}$ where the relabelling λ_l maps each label of A_1 of the form $n_1.l$ to the label $(n_1,n_2).l$ and, similarly, λ_r maps each label of A_2 of the form $n_2.l$ to the label $(n_1,n_2).l$. Both relabellings are assumed to preserve the I/O nature of the labels, i.e. their inclusion in I, O and T respectively.

All operators on I/O-transition systems considered above preserve observational equivalence.

I/O-transition systems for behaviour specifications of ports and components. In our component model behaviour specifications in the form of UML state machines are attached to ports and to simple components. In order to represent them by I/O-transition systems we assume that the state machines are flattened, i.e. that any hierachical structure is resolved, which can be achieved by standard techniques. We further assume that any pseudo-states used for describing alternatives are also resolved in the standard way by attaching two outgoing transitions, one for each alternative, to the original source state.

Let us first consider how behaviour specifications of ports are represented by I/O-transition systems. As pointed out in Sect. 1.2 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. 21. The corresponding I/O-transition system, shown in Fig. 28, is directly inferred from the behaviour specification. [8] 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.

[8] For the representation of the transition systems we have used the LTSA tool (cf. Sect. 1.5) which does not support the slash symbol "/" used in the UML state machines. In order to

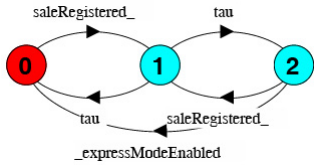


Figure 28. I/O-transition system for port C-CD

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. 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. 21). The behaviour specification uses an entry action and a pseudo-state 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. 29.

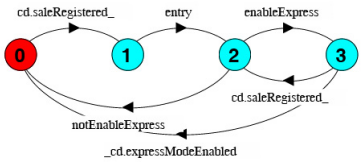


Figure 29. I/O-transition system for component Coordinator

indicate that a label i is an input label we use the visual representation $i\_$ and, symmetrically, to indicate that a label o is an output label we use the visual representation $\_o$.

1.4.2 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 specification of each of its ports and the deadlock-freeness of the behaviour specification of the component itself. Obviously, this condition is satisfied for all simple components and ports of our example.

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 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 defined in Sect. 1.4.1, $obs_p(C) = A_C \setminus H$ where H is the set of 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 $p.A_P$ (which is the copy of A_P as defined in Sect. 1.4.1). These considerations lead to the following definition of component correctness.

Definition 1. *Let C be a component and A_C the I/O-transition system representing the behaviour of C. Let $p : P$ be a port declaration of C and A_P the I/O-transition system representing the behaviour specification of P. The component C is* correct w.r.t. its port declaration $p : P$ *if $obs_p(C) \approx p.A_P$. The component C is* correct*, if it is correct w.r.t. all its port declarations.*

Let us illustrate how we can check the correctness of the component Coordinator w.r.t. its port cds : C-CD. First we consider the observable behaviour of the Coordinator at port cds : C-CD which is just the transition system shown in Fig. 21 where all labels which are not prefixed by cds are replaced by τ. If we minimise this transtion system w.r.t. observational equivalence then we obtain (up to the prefix cds) the transition system in Fig. 28 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. 1.5) that all simple components occurring in the CashDesk and CashDeskLine composite components (cf. Sect. 1.2) are correct.

The definition of the observable behaviour of a component at a particular port can be generalized 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 components 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 first 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 resp. 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.[9] To study the interaction behaviour of the two ports, let A_{P_l} be the I/O-transition system over the I/O-labelling $(I_\mathrm{l}, O_\mathrm{l}, \emptyset)$ representing the behaviour of P_l and let A_{P_r} be the I/O-transition system over the I/O-labelling $(I_\mathrm{r}, O_\mathrm{r}, \emptyset)$ representing the behaviour of P_r. According to the syntactic well-formedness condition, $O_\mathrm{l} = I_\mathrm{r}$ and $O_\mathrm{r} = I_\mathrm{l}$. Any communication between the connected ports is expressed by synchronising output labels of one port with the correponding input label of the other port according to the possible transitions of A_{P_l} and A_{P_r}. Hence, the interaction behaviour of A_{P_l} and A_{P_r} can be formally represented by the *port product* $A_{P_\mathrm{l}} \otimes A_{P_\mathrm{r}}$ where "$\otimes$" is the product operator defined in Sect. 1.4.1. Note that the transitions of the port product are marked only by internal labels (representing interactions) or by the invisible τ-action.

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. Since the interaction behaviour of two ports is represented by the transition system of the port product $A_{P_\mathrm{l}} \otimes A_{P_\mathrm{r}}$ this condition can be formalised as follows leading to the notion of behavioural port compatibility.

Definition 2. *Two ports P_l and P_r with behaviours represented by the I/O-transition systems $A_{P_\mathrm{l}}, A_{P_\mathrm{r}}$ resp. are* behaviourally compatible *if $A_{P_\mathrm{l}} \otimes A_{P_\mathrm{r}}$ is deadlock-free.*

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. 11) is shown in Fig. 30 (top) and the transition system representing the behaviour of the port C-CD was shown in Fig. 28. Hence, the interaction behaviour of the two ports is represented by the transition system of their port product which is shown in Fig. 30 (bottom). Obviously, the port product has no deadlock and therefore the two ports are behaviourally compatible.

[9] In general, one could use a more flexible condition such that the required operations of one port are included in the provided operations of the other one. However, it is technically more convenient and also sufficient for the example to use the more restrictive condition from above.

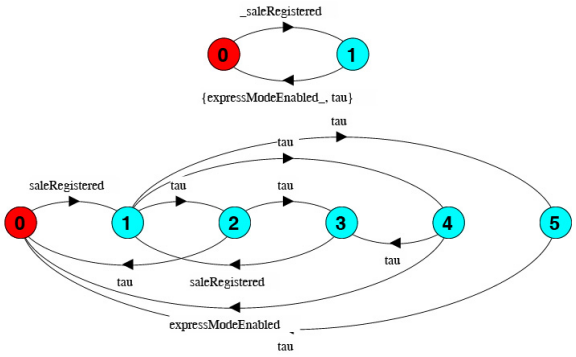


Figure 30. Port product of CDA-C and C-CD

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 port on the other side is not restricted and hence fully reflected by the interaction. It turns out that this property plays an essential role for the compositionality of behaviours that will be studied below to analyse behaviours of composite components. It can be formalised in the following way.

Definition 3. *Let $A_{P_\mathrm{l}} \otimes A_{P_\mathrm{r}}$ be the port product representing the interaction behaviour of two ports P_l and P_r (whose single behaviours are represented by the I/O-transition systems $A_{P_\mathrm{l}}, A_{P_\mathrm{r}}$ resp.). The interaction behaviour of P_l and P_r* reflects *the behaviour of P_l, if $A_{P_\mathrm{l}} \approx A_{P_\mathrm{l}} \otimes A_{P_\mathrm{r}}$ where A_{P_l} on the lefthand-side is considered as an I/O-transition system with all labels being internal (in order to fit to the labelling of the port product).*

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. 30 (bottom). After minimalisation 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. 30 (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. That is, 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. How this construction is technically performed for the case of a composite component with two connected subcomponents and with one relay port is discussed in the following. The construction can be generalised in a straightforward way to arbitrary many subcomponents, connectors and relay ports.

Let CC be a composite component which contains two component declarations $c_\mathrm{l} : C_\mathrm{l}$ and $c_\mathrm{r} : C_\mathrm{r}$. As for ports we omit multiplicities of component declarations and therefore assume that subcomponents are either declared with multiplicity 1 (as in our example) or the actions of different component instances of the same component declaration are independent from each other.[10] Assume that C_l contains two port declarations $r : R$, $p_\mathrm{l} : P_\mathrm{l}$ and C_r contains one port declaration $p_\mathrm{r} : P_\mathrm{r}$ such that $p_\mathrm{l} : P_\mathrm{l}$ and $p_\mathrm{r} : P_\mathrm{r}$ are connected and the connection is syntactically well-defined (i.e. the output labels of P_l correspond to input labels of P_r and vice versa). Moreover, assume that CC has one relay port declaration $rp : RP$ which refers to the port declaration $r : R$ of C_l. We assume that the relay port reference is syntactically well-defined, i.e. that the port type RP has the same provided and required interfaces as the port type R.

Let $obs(C_\mathrm{l})$ and $obs(C_\mathrm{r})$ be the I/O-transition systems representing the observable behaviours of C_l and C_r resp.; cf. Sect. 1.4.2. We will construct an I/O-transition system A_{CC} representing the behaviour of the composite component CC. First, let $c_\mathrm{l}.obs(C_\mathrm{l})$ be a copy of $obs(C_\mathrm{l})$ obtained by prefixing each label of $obs(C_\mathrm{l})$ with c_l (cf. Sect. 1.4.1) and, similarly, let $c_\mathrm{r}.obs(C_\mathrm{r})$ be a copy of $obs(C_\mathrm{r})$ using prefix c_r. Obviously, since component names are locally unique, the labels of $c_\mathrm{l}.obs(C_\mathrm{l})$ and $c_\mathrm{r}.obs(C_\mathrm{r})$ are disjoint. To construct A_{CC} we have to synchronise the given transition systems according to the connection of their ports which is easily achieved by using the construct for synchronised products (cf. Sect. 1.4.1) such that, for any label op of the port P_l (and hence of P_r), the labels $c_\mathrm{l}.p_\mathrm{l}.op$ and $c_\mathrm{r}.p_\mathrm{r}.op$ are identified via the shared label $(c_\mathrm{l}.p_\mathrm{l}, c_\mathrm{r}.p_\mathrm{r}).op$. Finally, the labels expressing actions of c_l on the non-connected port r must be renamed to actions on the relay port using the relabelling $\lambda_r(c_\mathrm{l}.r.op) = rp.op$ for all labels op of R (and hence of RP). The I/O-transition system A_{CC} representing the behaviour of the composite component CC is then given by $A_{CC} = (c_\mathrm{l}.obs(C_\mathrm{l}) \underset{(c_\mathrm{l}.p_\mathrm{l}, c_\mathrm{r}.p_\mathrm{r})}{\otimes} c_\mathrm{r}.obs(C_\mathrm{r}))\lambda_r$.

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 minimalisation 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.[11] In fact, our experience shows that in this way there is often not even an increment of the size of the

[10] If they would be dependent they could be simulated by introducing as many additional component declarations of the same type as instances of the original declaration are involved, provided that no reconfiguration occurs.

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

state space. For instance, the I/O-transition system of the CashDesk component has 346 states and 613 transitions while its observable behaviour has, after minimalisation, only 36 states and 66 transitions. Moving up the hierarchy, the I/O-transition system of the CashDeskLine component has, assuming one cash desk, 106 states and 252 transitions while its minimalised observable behaviour has only 8 states and 12 transitions.

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. 31 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 $i = 1, \ldots, n$.

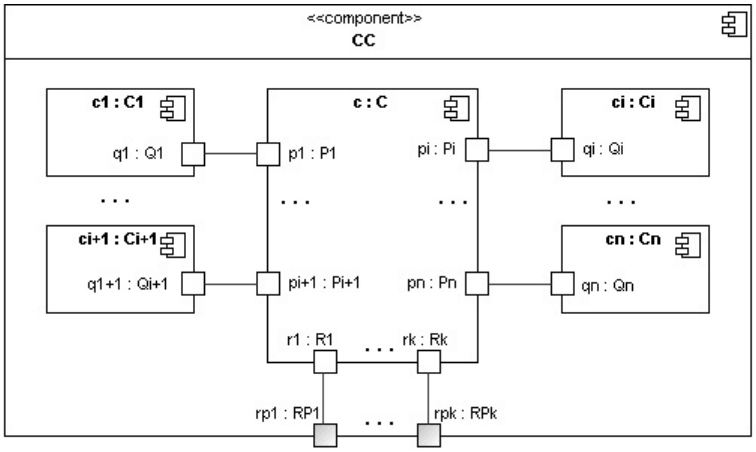


Figure 31. Star topology of composite component

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 types Q_α and Q_β properly restrict the behaviour of P_α *and* of P_β in an incompatible way.[12] This may happen, if C introduces a dependency between P_α and P_β that is

[12] 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.

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 [14]. 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 [14]) 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. 31. 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. [15] which is also motivated by the derivation of global properties from local ones, not using an explicit port concept, however. In [15] 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 [14] 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. 1.5.1) 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.

Let us now focus on the correctness of the CashDesk component w.r.t. its three relay ports which refer to ports of the CashDeskApplication with type CDA-Bank, CDA-I, and CDA-C resp. Obviously, we cannot directly conclude that the correctness of CashDeskApplication w.r.t. these ports implies the correctness of CashDesk w.r.t. the

relay ports of the same type since the behaviour of CashDeskApplication is properly restricted through the connection to the CashBox component.[13] Hence, we have to check explicitly that the connection between the ports of CashDeskApplication and CashBox has no impact on the behaviour of the ports CDA-Bank, CDA-I, and CDA-C. Since this is indeed the case the relay ports of CashDesk exhibit indeed the behaviour as it is specified for CDA-Bank, CDA-I, and CDA-C.

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 [11]. 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 [16] and recent [17] work.

1.4.3 Non-Functional Requirements

We perform quantitative analysis of the Process Sale Use Case 1 by modelling the example in the process algebra PEPA [8] and mapping it onto a Continuous-Time Markov

[13] Their connected ports are only behaviourally compatible but the connection does not reflect the behaviour of the CashDeskApplication port CDA-CB.

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 [18], 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

$$F(x) = \begin{cases} 1 - e^{-\lambda x} & \text{for } x > 0 \\ 0 & \text{for } x \leq 0 \end{cases}$$

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: $\lambda = 1/\mu$.

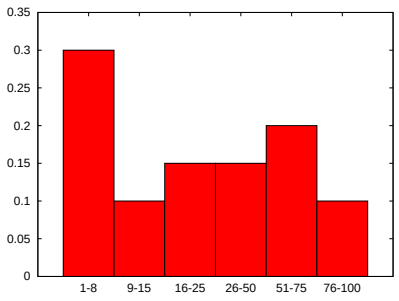


Figure 32. Probability mass function for the number of goods per customer

In the case where we only know the mean value (a case which often arises in practice) 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, what about the cases where we *do* know more about the expected distribution of times or other quantities? An example of this is the distribution of the number of goods per customer which varies according to the histogram shown in Figure 32. We assign weights to an immediate probabilistic choice in order to first choose the right range of the number of goods.

$$\begin{aligned} \mathit{Customer} \stackrel{\mathit{def}}{=} \quad & (\tau, 0.3 : \mathbf{immediate}).\mathit{Customer1to8} \\ & + (\tau, 0.1 : \mathbf{immediate}).\mathit{Customer9to15} \\ & + (\tau, 0.15 : \mathbf{immediate}).\mathit{Customer16to25} \end{aligned}$$

$$+ (\tau, 0.15 : \mathbf{immediate}).\mathit{Customer16to25}$$
$$+ (\tau, 0.2 : \mathbf{immediate}).\mathit{Customer51to75}$$
$$+ (\tau, 0.1 : \mathbf{immediate}).\mathit{Customer76to100}$$

The τ activity represents an individual choice of this component: other components cannot influence this. Within ranges we assume that the values are uniformly distributed and so we need only choose one of them, as in the example below. We write τ as a shorthand for $(\tau, 1 : \mathbf{immediate})$, making all of the choices equally weighted.

$$\begin{aligned}\mathit{Customer1to8} \stackrel{def}{=} \quad & \tau.\mathit{Customer}_1 + \tau.\mathit{Customer}_2 + \tau.\mathit{Customer}_3 \\ & + \tau.\mathit{Customer}_4 + \tau.\mathit{Customer}_5 + \tau.\mathit{Customer}_6 \\ & + \tau.\mathit{Customer}_7 + \tau.\mathit{Customer}_8\end{aligned}$$

The $\mathit{Customer}_n$ component is parameterised by the number of goods to be purchased and so this parameter decreases after every item is presented for purchase, until it reaches zero. Items either have their identifier entered manually or their barcode is scanned. The rate at which items are entered or scanned is determined by the cashier, not by the customer. The customer only passively witnesses these events (rate $\top$).

$$\mathit{Customer}_{n+1} \stackrel{def}{=} (\mathit{scanItem}, \top).\mathit{Customer}_n + (\mathit{enterItem}, \top).\mathit{Customer}_n$$

When all of their items have been presented for purchase the customer moves on to payment by cash or credit card. These are equally likely.

$$\mathit{Customer}_0 \stackrel{def}{=} \tau.\mathit{Customer}_{\mathit{payByCash}} + \tau.\mathit{Customer}_{\mathit{payByCard}}$$

Probabilistic transitions are used to select between discrete quantities in the way shown above for the number of items which the customer wishes to buy. Continuously-varying quantities such as durations are treated differently. 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 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 Figure 33 we show how 700000 values from a uniformly-distributed interpretation of the histogram for credit card validation would differ from the exponentially-distributed interpretation. We used the well-known logarithm method to sample from the exponential distribution [19] and computed the mean value of 700000 groups of samples of size 30.

In our experience, a distribution such as that shown in Figure 33 (left) is unlikely to occur in practice. For example, it has the surprising property that delays of four seconds

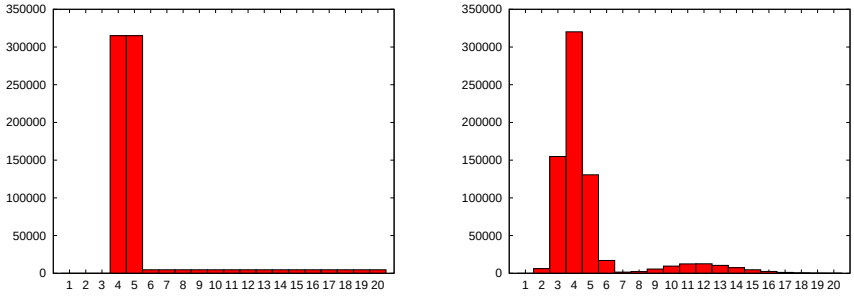


Figure 33. Specified (left) and sampled (right) distributions for credit card validation

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, which also seems surprising. In contrast, distributions as seen from our sample in Figure 33 (right) occur frequently because they are a convolution of two heavy-tailed distributions. Credit card validation requires network use where variations in network load add additional delays, leading to heavy-tailed distributions. The area under the curve is 700000 in both cases.

Other histogram-specified continuous distributions are treated similarly. We first make a weighted probabilistic choice and then delay for a exponentially-distributed time.

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.

An alternative is to find the *transient* state probability row vector $\pi(t) = [\pi_0(t), \ldots, \pi_{n-1}(t)]$ where $\pi_i(t)$ denotes the probability that the CTMC is in state i at time t. Transient and passage-time analysis of CTMCs proceeds by uniformisation [20,21]. The generator matrix, Q, is "uniformized" with: $P = Q/q + I$ where $q > \max_i |Q_{ii}|$. This process transforms a CTMC into one in which all states have the same mean holding time $1/q$.

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. In this case we can attempt to answer questions of the form "Will at least 50% of all sales go from $startNewSale$ to $handoutReceipt$ within 40 seconds?" which has as a probability bound "at least 50%", as a time bound "within 40 seconds", and as the paths through the system behaviour all paths starting with $startNewSale$ and ending with $handoutReceipt$.

We investigated how the likelihood of having completed sales varied over time, looking at periods of 10, 30, 50 and 80 seconds. Within this we looked at the passage of time from the beginning of a sale to having scanned all of the items (Figure 34 has these results). We found that in 50% of cases scanning would be completed within 30 seconds (see Figure 34(b)).

We considered how the completion of sales varies as a function of time (Figure 35 has these results). Having scanned all of the items is a necessary pre-requisite to completing the transaction by giving the receipt so the probability of completing the entire sale is, at any point in time, lower than the probability of having scanned all of the items. For example, we found that only 32% of sales would be completed within 30 seconds (see Fig. 35(b)) and that it would take at least 44 seconds to complete 50% of sales (see Fig. 35(c)).

Finally, 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 Figure 36(c)). 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 check-

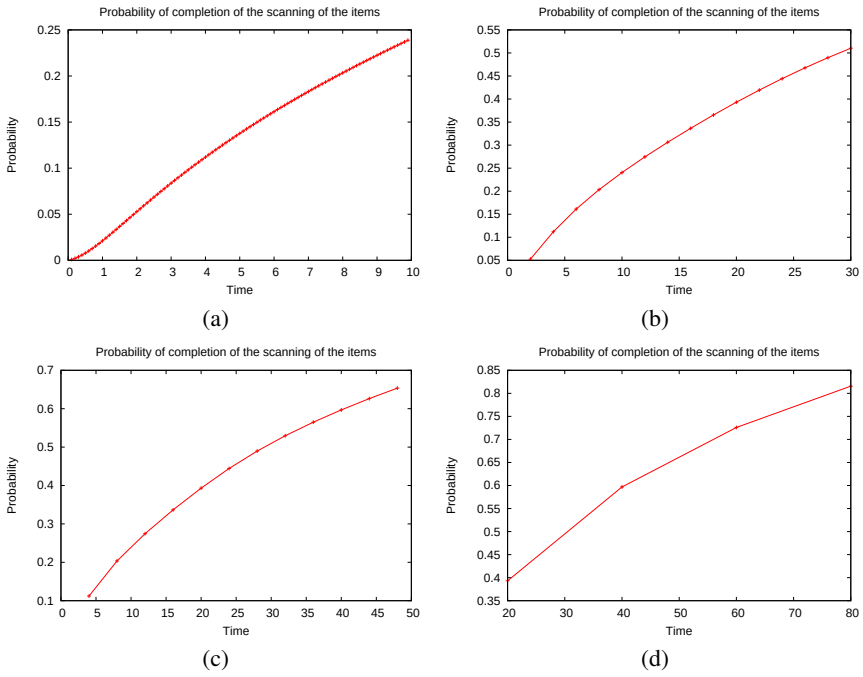


Figure 34. Graphs showing how the probability of completing scanning (from beginning the sale to scanning the last item) varies over (a) 10, (b) 30, (c) 50 and (d) 80 seconds

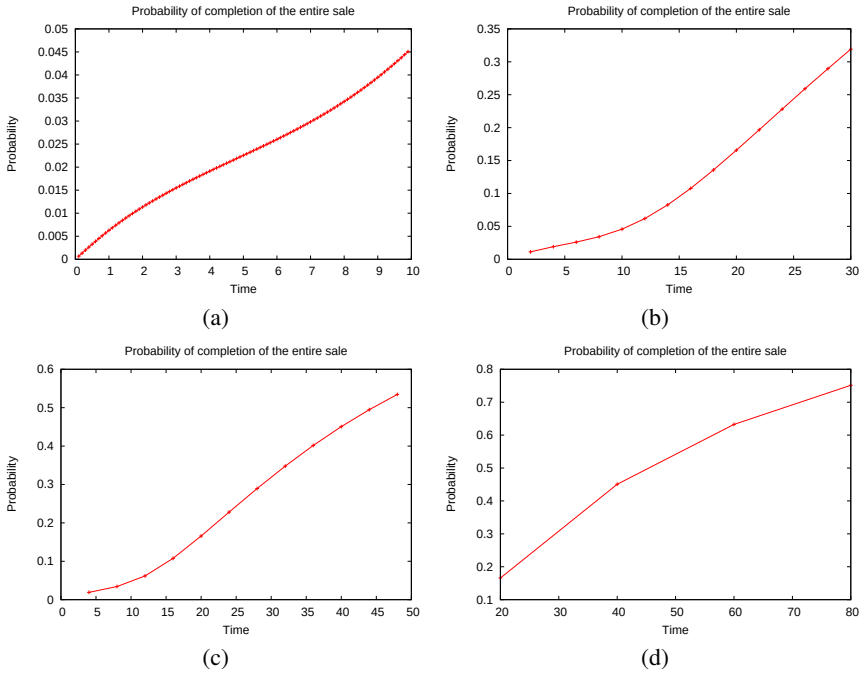


Figure 35. Graphs showing how the probability of completing a sale (from beginning to giving the receipt) varies over (a) 10, (b) 30, (c) 50 and (d) 80 seconds

out, because we have seen this happening in practice. This goes some way to explaining why the difference in the results is not larger.

1.5 Tools

1.5.1 Qualitative and Quantitative Analysis

HUGO/RT. HUGO/RT [22] 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. 38).

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. 1.5.2).

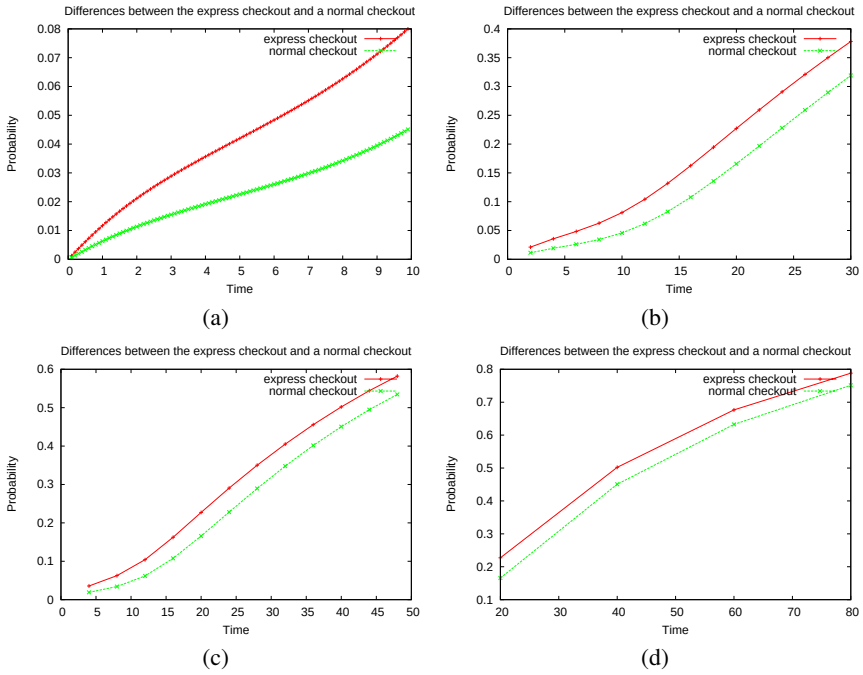


Figure 36. Graphs showing how the advantage of using the express checkout (8 items or fewer) over using a normal checkout (100 items or fewer) varies over (a) 10, (b) 30, (c) 50 and (d) 80 seconds

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 [13]). The LTSA tool supports the process algebra FSP [13] 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 [10].

PEPA. PEPA (Performance Evaluation Process Algebra [8]) is a process algebra that allows for quantitative analysis of the CoCoME using Continuous-Time Markov Chains (CTMC). For the quantitative analysis of Sect. 1.4.3 we used the IPC tool [9].

1.5.2 Architectural Programming

JAVA/A [3,23] is a Java-based architectural programming language which features syntactical constructs to express JAVA/A component model elements (see Sect. 1.2) 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. 37 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 initialisation process (which basically consists of initialising and connecting the components).

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

Figure 37. JAVA/A composite component SimplifiedStore

In Fig. 38 we give a very brief overview of the JAVA/A implementation of the component CashDeskApplication.[14] In lines 3–20 the port `CDACB`[15] 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. 1.5.1).

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 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.

[14] Of course, most of the component's body is omitted here. However, the complete implementation is available online [10].

[15] In contrast to Sect. 1.3, in the JAVA/A implementation port names are written without hyphens due to Java naming conventions.

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

Figure 38. The JAVA/A simple component CashDeskApplication

`processSaleStarted` in Fig. 38, 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 [10].

1.6 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. 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.

1. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. John Wiley & Sons, New York (1994)
2. Object Management Group: Unified Modeling Language: Superstructure, Version 2.0. Technical report, OMG (2005)
3. 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'05). Volume 160 of Elect. Notes Theo. Comp. Sci. (2006) 75–96
4. Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architecture. ACM SIGSOFT Softw. Eng. Notes **17**(4) (1992) 40–52
5. 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)
6. Aldrich, J.: ArchJava. `http://www.archjava.org`$^{(05/17/07)}$.
7. Seco, J., Caires, L.: A Basic Model of Typed Components. In Bertino, E., ed.: Proc. 14th Europ. Conf. Object-Oriented Programming (ECOOP'00). Volume 1850 of Lect. Notes Comp. Sci., Springer, Berlin (2000) 108–128
8. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press (1996)
9. Bradley, J., Clark, A., Gilmore, S.: User manual for IPC: The Imperial PEPA Compiler. `http://www.doc.ic.ac.uk/ipc`$^{(05/02/07)}$.
10. 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. `http://www.pst.ifi.lmu.de/Research/current-projects/cocome/`$^{(05/02/07)}$.
11. Reussner, R., Krogmann, K., Koziolek, H., Rausch, A., Herold, S., Klus, H., Welsch, Y., Hummel, B., Meisinger, M., Pfaller, C., Mirandola, R.: Chapter 3, CoCoME — The Common Component Modelling Example. In: CoCoME Book. (2007)
12. de Alfaro, L., Henzinger, T.A.: Interface Automata. In: Proc. 9th Ann. Symp. Foundations of Software Engineering (FSE'01), Wien, ACM Press (2001) 109–120
13. Magee, J., Kramer, J.: Concurrency — State Models and Java Programs. John Wiley & Sons (1999)

14. Hennicker, R., Janisch, S., Knapp, A.: On the Compositional Analysis of Hierarchical Components with Explicit Ports (2007) Submitted. http://www.pst.ifi.lmu.de/Research/current-projects/cocome/(06/22/07).
15. Bernardo, M., Ciancarini, P., Donatiello, L.: Architecting Families of Software Systems with Process Algebras. ACM Trans. Softw. Eng. Methodol. **11**(4) (2002) 386–426
16. 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. Volume 157/I of books@ocg.at., ÖGI (Austrian Computer Society) (2001) 600–607
17. Bidoit, M., Hennicker, R.: A Model-theoretic Foundation for Contract-based Software Components (2007) Submitted. http://www.pst.ifi.lmu.de/people/staff/hennicker/.
18. 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.: Proc. Int. Symp. Formal Methods Europe (FM'05). Volume 3582 of Lect. Notes Comp. Sci., Springer, Berlin (2005) 286–301
19. Ahrens, J., Deiter, U.: Computer methods for sampling from the exponential and normal distributions. Communications of the ACM **15**(10) (October 1972) 873–882
20. Grassmann, W.: Transient solutions in Markovian queueing systems. Computers and Operations Research **4** (1977) 47–53
21. Gross, D., Miller, D.: The randomization technique as a modelling tool and solution procedure for transient Markov processes. Operations Research **32** (1984) 343–361
22. Knapp, A.: HUGO/RT Web page. http://www.pst.ifi.lmu.de/projekte/hugo(05/02/07).
23. Hacklinger, F.: JAVA/A – Taking Components into Java. In: Proc. 13th ISCA Int. Conf. Intelligent and Adaptive Systems and Software Engineering (IASSE'04), ISCA, Cary, NC (2004) 163–169