14 CoCoME in Fractal*

Lubomír Bulej[1,2], Tomáš Bureš[1,2], Thierry Coupaye[3], Martin Děcký[1], Pavel Ježek[1], Pavel Parízek[1], František Plášil[1,2], Tomáš Poch[1], Nicolas Rivierre[3], Ondřej Šerý[1], and Petr Tůma[1]

[1] Department of Software Engineering
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, Prague 1, 11800, Czech Republic
{lubomir.bulej,tomas.bures,martin.decky,pavel.jezek,
pavel.parizek,frantisek.plasil,tomas.poch,ondrej.sery,
petr.tuma}@dsrg.mff.cuni.cz
[2] Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod Vodárenskou věží, Prague 8, 18000, CzechRepublic
{bulej,bures,plasil}@cs.cas.cz
[3] France Telecom R&D
Issy les Moulineaux, France
{thierry.coupaye,nicolas.rivierre}@orange-ftgroup.com

This chapter presents our solution to the CoCoME assignment that is based on the Fractal component model. The solution involves (i) modeling architecture in Fractal ADL, (ii) specification of component behavior via behavior protocols, (iii) checking compatibility of components, (iv) verification of correspondence between component code and behavior specification, and (v) run-time monitoring of non-functional properties. Among the issues we have faced was the need to modify the architecture - the component hierarchy was reorganized in order to improve clarity of the design and the hierarchical bus was split into two independent buses. These were modeled by primitive components, since Fractal does not support message bus as a first-class entity. Since the CoCoME assignment does not include a complete UML behavior specification (e.g. via activity diagrams and state charts), behavior protocols for all the components are based on the provided plain-English use cases, the UML sequence diagrams, and the reference Java implementation.

14.1 Introduction

14.1.1 Goals and Scope of the Component Model

Fractal [4] is a classical component model with concepts stemming from Darwin [19]. It supports components as first-class concepts and allows their hierarchical nesting. Since first published in 2002, Fractal has gained attention of the professional community and become quite popular; it is one of the key projects hosted by

* This work was partially supported by the Czech Academy of Sciences project 1ET400300504 and its results will be used in the ITEA/EUREKA project OSIRIS Σ!2023.

A. Rausch et al. (Eds.): Common Component Modeling Example, LNCS 5153, pp. 357–387, 2008.

the OW2 Consortium[1] and it has been often used as the core platform for other OW2 projects. Annual Fractal workshops take place collocated with international conferences.

Fractal aims at providing support for modeling at different levels of abstraction. It focuses not only on the design of an application, but it also provides tools and environment for development, deployment, and runtime. Fractal also tries to address the limitations in extensibility and adaptation, which are often found in other run-time supporting component systems (e.g., EJB [30], CCM [23], or .Net [22]). By providing an open set of control capabilities, Fractal allows customizing the component model with regard to the target platform (e.g. to lower memory footprint of an embedded mobile application by excluding reflection capabilities and lifecycle management).

In Fractal, a component is both design and runtime entity with explicit provided and required interfaces. Each component consists of two parts: a *content* that is responsible for application functionality (i.e. implements the component's frame), and a *membrane*, which contains a set of controllers that implement non-functional aspects. For the purpose of composition, all components are defined by their *frame* and *architecture*. A frame is formed by external interfaces of a component, while an architecture defines the internal structure of the component, i.e. its subcomponents and their composition (via binding of interfaces). Semantics of the composition is defined via behavior protocols (a specific process algebra); Fractal also supports divide and conquer via interface specification.

Component behavior is specified using behavior protocols (BP) that allow to model and verify the behavior compliance. The verification tools can verify (i) the composition correctness (both horizontal and vertical) with respect to behavior specification (in BP) independently of the implementation, and (ii) the relation between the model (in BP) and implementation (in Java).

Deployment description is not a part of the architecture and behavior specification.

14.1.2 Modeled Cutout of CoCoME

We model all aspects of the CoCoME example with the exception of extra-functional properties, which were not modeled but were monitored at runtime. In the process of modeling the example in Fractal, we modified the original architecture in order to improve clarity of the design and to cope with limitations of Fractal. In particular, the hierarchical bus was split into two independent buses that were modeled via components, since Fractal does not support message buses. We verify correctness of composition (i.e. behavior compliance) with respect to behavior for all components, and correspondence of code to behavior specification for primitive components (our approach allows modeling only such behavior that fits a regular language); the verification tools are based on the BP checker and Java PathFinder. Performance of components was monitored via a custom controller.

14.1.3 Benefit of the Modeling

The two major benefits of our approach to modeling are: (i) verification of composition correctness with respect to behavior specification, and (ii) verification of

[1] OW2 Consortium is an international community of open source middleware developers.

correspondence between code and behavior specification. Other benefits include generation of code skeletons from the model and runtime monitoring.

Usage of Fractal with behavior protocols and verification tools is quite easy; learning curves of both Fractal and behavior protocols are short and both can be used in a matter of days.

14.1.4 Effort and Lessons Learned

We have needed approximately 10 person-months to model the CoCoME example in its entirety (we treat nearly all aspects of the example). As detailed further in the text, the lessons learned include detailed understanding of the limits that our architecture model and behavior specification has, especially where support for legacy technologies that do not fit the model framework is concerned.

14.1.5 Structure of the Chapter

The rest of this paper is organized as follows. Sect. 2 introduces the Fractal component model and its behavior protocol extension. Sect. 3 presents our solution for the CoCoME assignment using Fractal. Sect. 4 presents automated transformation from FractalADL to code skeletons, which significantly helps during the implementation process. In Sect. 5, the analytic techniques for verification of behavior compliance of components are presented. In Sect. 6, the available tools supporting the verification are then summarized and accompanied with results of our experiments.

14.2 Component Model

Fractal is specified as a platform independent component model. The specification (now in version 2.0) [5] defines the key concepts of Fractal and their relations. It also defines controlling (management) functionality of components to be implemented by component controllers (Sect. 2.1). Controllers serve, e.g., for creating components, their reconfiguration, lifecycle management; the interfaces of controllers are defined in pseudo IDL with mapping to Java, C and CORBA IDL.

The platform independent specification of Fractal has been reified by a number of implementations. To the most important ones belong Julia [4] (a Java implementation with support for mobile devices), AOKell [28] (a Java implementation employing aspect oriented programming), and FractNet [7] (a .NET implementation). Moreover, there are Fractal implementations aiming at specific application domains. To these belong Plasma [17] (C++ multimedia applications), ProActive [3] (Java grid computing), and Think [31] (operating system kernel C development).

There are also a number of tools, extensions and libraries for Fractal ranging from graphical development of applications to a Swing and JMX support. For the purpose of this paper, to the most notable tools and extensions belong the FractalRMI [11] (seamless distribution using RMI), FractalADL [9] (XML-based ADL language for defining architectures of Fractal components), and FractalBPC [10] (Fractal Behavior Protocol Checker), which is an extension allowing specification of component behavior and verification of component communication compliance.

14.2.1 Static View (Metamodel)

The Fractal component model relies on components as basic building blocks. Since Fractal is a hierarchical model, components may be either composite or primitive. In the case of a composite component, the component contains a number of sub-components. An application in Fractal is represented by a single (typically composite) top-level component.

A component may have a number of interfaces ("ports" in other component systems such as Darwin) which are the points of access to the component. Each interface is an instance of its type, which states the signature of the interface, its kind, contingency and cardinality. The interface kind is either *server* or *client*, which corresponds to provided and required interfaces in Darwin.

The contingency specifies whether an interface is *mandatory* or *optional*. In the case of client interfaces, contingency is useful to express what of the component requirements are vital to its functionality, and which do not have to be addressed while still guaranteeing a consistent component behavior. In the case of server interfaces, contingency is used to express e.g. the fact that a server interface of a composite component does not have to be bound to a sub-component (effectively leaving such functionality unimplemented). The cardinality is either *singleton* or *collection*, permitting either a single or multiple interface instance(s).

An interesting concept of Fractal is *shared component*. This concept allows an instance of a component to be a part of several (non-nested) parent components. It is especially useful when modeling shared resources.

Internally, a Fractal component is formed by a *membrane* and *content*. The membrane encapsulates the component's functional "business" interfaces and also the controllers with their "management" interfaces. The content consists of several sub-components (in the case of a composite component) or implementation code, encapsulation of legacy components, etc. (in the case of a primitive component). Each of the controllers is responsible for particular management functionality. Predefined controllers include the lifecycle controller for managing the lifecycle, binding controller for binding client interfaces, content controller for introspecting and managing sub-components, etc. As mentioned in Sect. 1, Fractal is an open model, thus the set of controllers is customizable and extensible. The actual way controllers are implemented depends on a particular Fractal implementation. In Julia, a dedicated technique of combining *mixins* using the ASM tool [2] for byte-code manipulation is employed, while AOKell relies on aspect oriented programming using either AspectJ or Spoon.

Instantiation of Fractal components is performed using an API defined by Fractal specification. It defines a bootstrap component factory that serves for creating component instances. The created instances are nested and connected according to the application architecture using controller interfaces. This way of instantiation implies that an application requires a main class that instantiates and starts particular components using the API.

Besides creating such a main class manually, specifically for each application, there is also an option of using FractalADL. This tool allows defining architecture of components using an ADL. It parses the ADL description and builds the application accordingly using Fractal API.

FractalADL is in fact a declarative description of how to instantiate components. This, however, implies that FractalADL operates with component instances as opposed to components (such as found in UML 2) and that it is not possible to specify component cardinality.

More information on the development process in Fractal may be found in [21][18].

14.2.2 Behavioral View

The applications behavior is specified via behavior protocols originally employed in the SOFA component model [27]. In the Fractal platform this formalism is used by FractalBPC which is a Fractal extension allowing specification of component behavior and verification of component communication compliance.

The behavior is not captured as a whole (by a single behavior protocol). Instead, each of the application's components has one behavior protocol associated with it (*frame protocol*) that describes the component's behavior as it is observable by the environment of the component, i.e. the component's code is abstracted in terms of capturing the traces of events related to method calls crossing the component boundary. Assuming component A requires an interface I that is provided by component B and assuming the interfaces A.I and B.I are bound together, these events are: (i) issuing a method request M on component A's required interface I: !I.M↑, (ii) accepting a method request M on component B's provided interface I: ?I.M↑, (iii) sending a response to method request on component B's provided interface I: !I.M↓, (iv) accepting a response to method request issued on component A's required interface I: ?I.M↓. Component's behavior is then described by an expression (component's frame protocol), where these events can be connected together using several operators (; for sequence, + for alternative, * for repetition and | for parallel execution). To simplify expressing method calls, the following abbreviations are introduced: ?I.M (stands for ?I.M↑; !I.M↓), ?I.M{*P*} (stands for ?I.M↑; *P*; !I.M↓), where {*P*} specifies a reaction to accepting the method call (*P* is a protocol here). The abbreviations !I.M, and !I.M{*P*} have a similar meaning.

Having a set of components with formal specification of their behavior via their frame protocols, the components can be connected together and compatibility of their behavior can be verified [1] (components horizontal compliance – meaning compliance at a particular level of nesting).

Every composite component also has a hand-written frame protocol that specifies its behavior. During the development of process of a composite component, its frame protocol can be verified against the behavior of its internals [1] (the vertical compliance – meaning compliance of components on adjacent levels of nesting) – the internal behavior of a composite component is described by an *architecture protocol* of its subcomponents (this behavior protocol is however not hand-written, but it is, in a way, automatically generated from frame protocols of the components that are part of the composite component's architecture).

The verification process of the horizontal and vertical compliance assures that the application's architecture is composed correctly. In addition to it, the verification of primitive components' frame protocols against their code can be done (code model checking) [1], which guarantees that the behavior described by the top-level architecture protocol corresponds to the true behavior of the application. However, model

checking of a single component faces a significant problem, as most of today's model checkers require a complete program to verify – a so-called missing environment problem [25]. This problem can be solved by generating an artificial environment and combining it with the code of the component, forming a complete program – such a program can then be passed to a modified JPF model checker [32] and its compliance to the component's frame protocol can be verified [26].

14.2.3 Deployment View

The Fractal specification does not address issues related to deployment. As a result, each implementation of Fractal deals with deployment in a specific way. For instance, Julia, the most widely used Fractal implementation in Java, is local (i.e., it provides no support for distribution). A Fractal application in Julia is executed by a dedicated Java class that instantiates and starts the components of the application. Besides having such a class specifically created for each application, it is possible to use a generic launcher that is part of FractalADL.

Although Julia does not support distribution by itself, it is possible to use extending libraries which add this feature. This is the case of FractalRMI library, which provides distribution using RMI. FractalRMI, however, is not tied only to Julia, it introduces distribution to other Java-based implementations of Fractal (e.g. AOKell).

FractalRMI is integrated with FractalADL. Thus, it is possible to specify a target deployment node for each component in FractalADL and subsequently use the FractalADL launcher to instantiate a distributed application.

Apart from local implementations of Fractal there also exist special purpose implementations which bring the support for distribution already in their core and do not need any extensions. This is for example the case of ProActive, which aims at grid computing.

Another important issue of deployment related to the CoCoME example is the support for different communication styles (e.g., method invocation, asynchronous message delivery, etc.), which are reified by different middleware (e.g., RMI, JMS, etc). Fractal does not provide any direct support for modeling different communication styles. It addresses this issue only partially by defining so called *composite bindings*, which are in fact regular components that encapsulate the use of middleware. Such binding components can be built using the Dream framework [7] which provides Fractal components for construction of middleware.

14.3 Modeling the CoCoMe

Employing Fractal in the CoCoME assignment revealed several issues that required modifications of the architecture. These modifications are presented and justified in Sect. 3.1 (Static view). Since behavior specification using Behavior Protocols is supported by Fractal, each of the components of the Trading System was annotated with its frame protocol. As CoCoME assignment does not include complete behavior specification, these protocols are created based on the CoCoME UML specification and the reference implementation and further described in Sect. 3.2 (Behavioral view). Sect. 3.3 (Deployment view) presents deployment and distribution using

Fractal-specific means (FractalRMI and FractalADL). In Sect. 3.4 (Implementation view), we describe beside the basic Fractal implementation strategy also the details related to performance evaluation and estimation. Additionally, Sect. 3.5 presents behavior specification of two components featuring non-trivial behavior (CashDeskApplication and CeshDeskBus) in more detail. Behavior specification of the rest of the components can be found in Appendix and on the Fractal-CoCoME web page [33].

14.3.1 Static View

As the architecture of the Trading System used in Fractal differs slightly from the CoCoME assignment, this section presents the modified architecture and justifies the modifications made. In general, there are two sorts of modifications: (i) Modifications which are not directly related to Fractal and do not influence complexity of the solution, but rather contribute to the clarity of the design and the other views (in Sect. 3.2 – 3.4). (ii) Modifications directly forced by specific properties of Fractal. These modifications reveal strengths and limitations of Fractal and therefore should be taken into account in the comparison between different modeling approaches.

The (i) modifications include reorganization of the component hierarchy and explicit partitioning of EventBus into two independent buses. All primitive components are left unchanged, but the composed components GUI and Application located in the Inventory component are substituted by components StoreApplication, ReportingApplication (compare Fig. 1 and Fig. 3). The new components more clearly encapsulate the logical units featuring orthogonal functionality, whereas the old ones merely present a general three tier architecture. The StoreApplication component encapsulates the store functionality as required by the CashDeskLine component in UC1 (use case #1 in CoCoME assignment), whereas ReportingApplication encapsulates functionality for

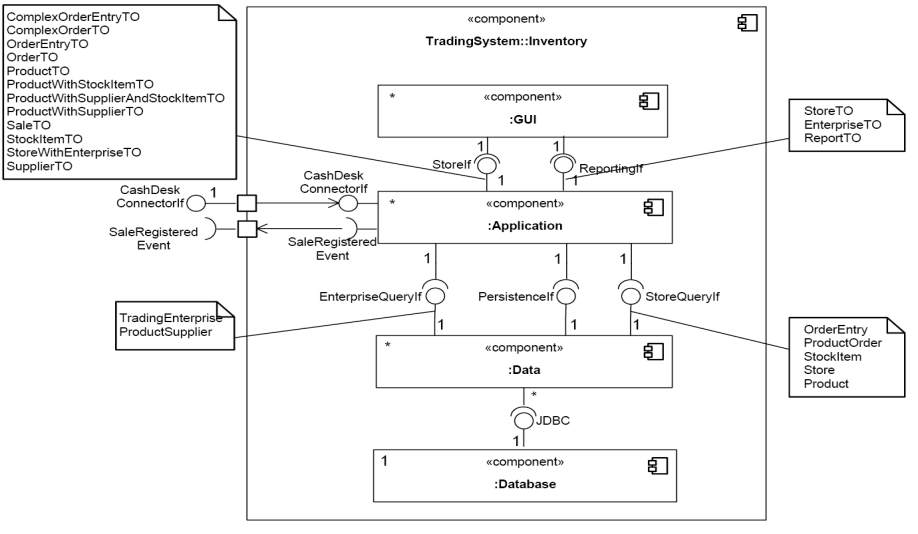

Fig. 1. The original design of the Inventory component in CoCoME

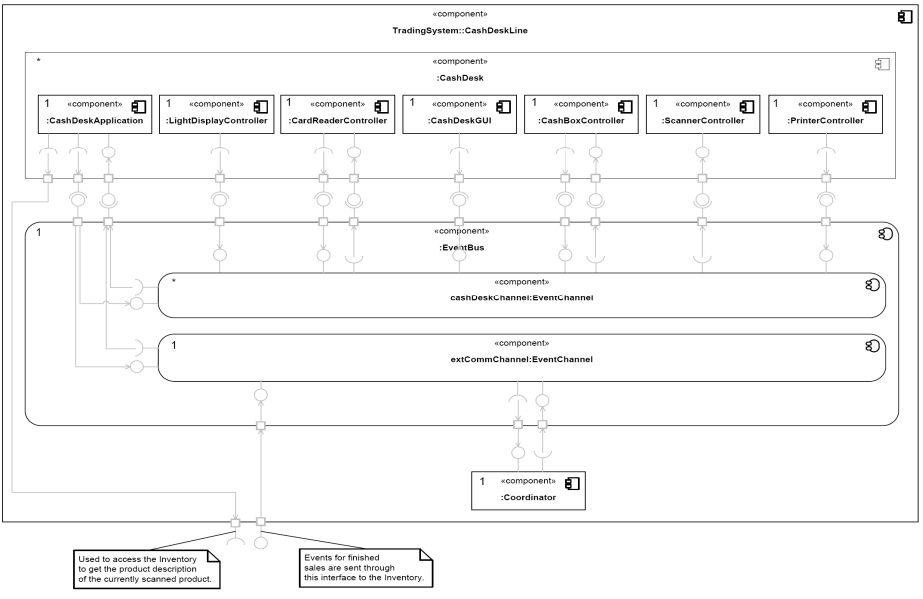

Fig. 2. The original design of the CashDeskLine component in CoCoME

managing goods as used in UC3 – UC7. The Data component is left unchanged. Second modification of the component hierarchy relates to UC8, as neither the architecture in CoCoME assignment, nor its reference implementation provides a full UC8 functionality. Specifically, UC8 expects communication among EnterpriseServer and StoreServers; however no interface for the communication is present. Moreover, the reference implementation includes UC8 functionality as a part of UC1, which, however, should be independent. The reference implementation deeply exploits the fact that it is not distributed and accesses the shared database, which would not be the case in a real-life implementation. Therefore, the new architecture is enriched by explicitly distinguishing the EnterpriseServer component and the ProductDispatcherIf and MoveGoodsIf interfaces that encapsulate UC 8 functionality (Fig. 3).

EventBus from the CoCoME assignment (Fig. 2) represents a composite of buses eventChannel and extCommChannel. As there is no apparent benefit of having the eventChannel outside the CashDesk component, EventBus is split into two independent buses CashDeskLineBus and CashDeskBus, which correspond to extCommChannel and eventChannel, respectively. Moreover, CashDeskBus is moved inside the CashDesk component where it more naturally belongs, since it mediates mostly the communication among the components and devices internal to CashDesk.

As to the (ii) modifications, Fractal does not support message bus as a first-class entity. Therefore, the CashDeskLineBus and CashDeskBus buses are modeled as primitive components, multiplexing the published messages to each of the subscribers (compare Fig. 2 and Fig. 3).

Despite the modifications made, many parts of the original design and prototype implementation are adopted even when “unrealistic”, such as the CardReader component communicating with Bank through CashDeskApplication instead of directly,

which presents a security threat with PIN code interception possibility. In order to share as much of the CoCoME assignment as possible, other parts of the design such as the data model and the transfer objects are left unmodified. The Fractal implementation is designed to use Hibernate and Derby database for persistency as is the case with the prototype implementation.

14.3.2 Behavioral View

The behavior protocols describing application's behavior are meant to be part of the specification of the application. Created at the application design stage, they allow developers to verify that the implementation is compliant with the design, or, in other words, that it really implements the specification. However, as the behavior protocols were not part of the specification of the CoCoME assignment, they had to be recreated from the description provided in it.

The provided specification contains only sequence diagrams and use-cases, which do not provide as precise and unambiguous specification of the application's behavior as it is required to formally verify the resulting implementation correctness (in order to be sufficient, the specification would have to include more complete UML description, like collaboration and activity diagrams or state machines). For this reason, we had to use the reference implementation provided also as a part of the specification and use both the UML descriptions and the reference implementation to create the behavior protocols for the application. A problem has however arisen during the behavior protocol development process – we found that the reference implementation is not fully compliant with the CoCoME UML specification as provided – there are two major differences between the reference implementation and the specification: (i) missing continuation of the payment process after erroneous credit card payment – UC1, (ii) missing implementation of UC8. We solved this problem by creating two alternatives of the protocols – the first specifying the behavior imposed by the CoCoME UML specification, and the second specifying the behavior observable in the reference implementation. As our component-based implementation of the application is based on the reference implementation (we have reused as much of the reference implementation code as possible), we show later in the text that our implementation (and the reference implementation) is not exactly following the requirements imposed by the CoCoME UML specification (by formally refuting it).

Regarding the behavior specification, it is also worth noting that we do not model the behavior of the actors (e.g. customer, cashier) specified by the UML model as we model only the software components that are part of the application's architecture. However, as the behavior of agents is observable via interfaces provided for the GUI part of the application, the behavior protocols describing the behavior of application's components also transitively impose restrictions on behavior of agents, though the actual formal verification is done against the GUI components.

We show that creating behavior protocols as part of the application specification allows precisely defining the required application's behavior early in the development process (in the application design stage). Such specification then provides not only the global view that is required to correctly create the application's architecture, but also a per component behavioral view that can serve as a precise guide for developers of

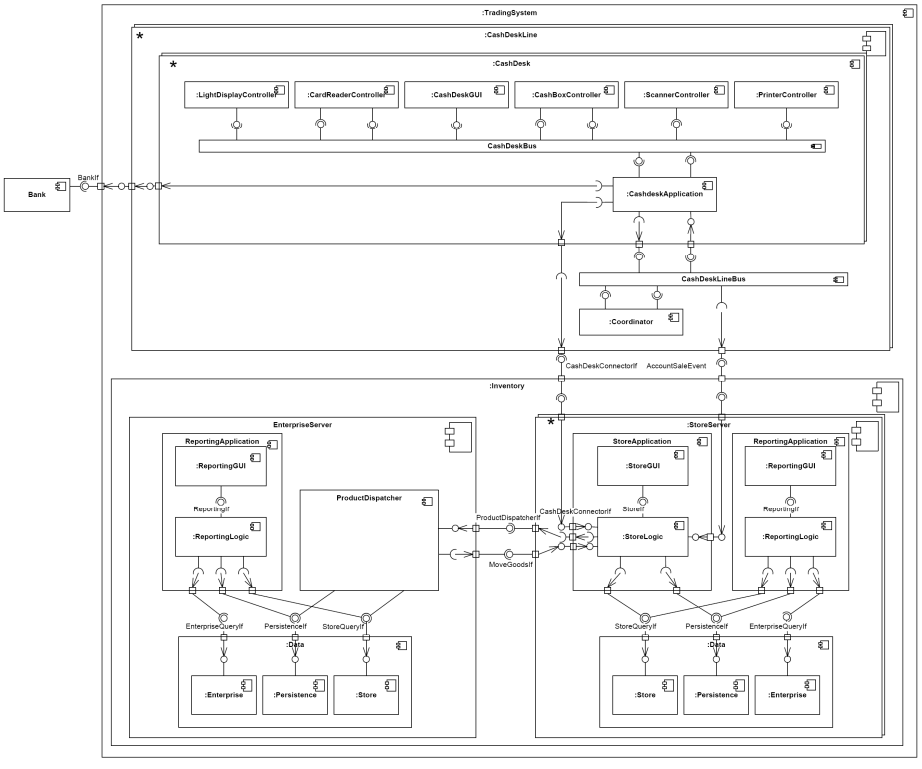

Fig. 3. Final architecture, as it is used in the Fractal modeling approach

specific component implementations. Furthermore, the specification can be used to formally verify that the implementation really complies with the specification requirements and that all the application components (although each might be implemented by a different developer) are compatible and together provide the functionality (exposed by their behavior) required by the specification.

14.3.3 Deployment View

From the deployment point of view, we introduced a few changes mainly to the middleware used in the reference architecture. These changes were motivated by the way Fractal can be distributed and by the libraries available for the distribution.

We have used FractalRMI instead of Sun RMI. FractalRMI is a library for Fractal that allows transparent distribution. The components are not aware of the fact that they communicate remotely.

In a similar fashion, we have eliminated the use of JMS, which has been used in the reference architecture for implementing buses. We have replaced each of the both busses by a component that is responsible for routing the messages. Remote communication in this case may be again transparently realized using FractalRMI.

The Fractal specification also lays out another way of solving distribution and various communication styles. It defines so called composite bindings. Each composite binding consists of a number of binding components. These components are classical components from the Fractal point of view, their responsibilities are to encapsulate or implement middleware. A significant help in implementing the composite bindings is provided by the Dream framework, which implements Fractal components that support construction of communication middleware with various communication styles, including JMS.

Our choice of FractalRMI is transparent and requires no additional implementation effort. We did not use the composite bindings and Dream also because Dream is still under development; additionally, our solution brings no restrictions to the modeled CoCoME example.

Another important aspect of deployment is the way deployment is planned and performed. In our approach, we have put the information about deployment into FractalADL. Each specified component is annotated with an element virtual-node which states the deployment node to which the component is to be deployed. The actual distribution is then realized via FractalRMI.

14.3.4 Implementation View

The Fractal implementation is based both on the Fractal architecture model of the application and the provided reference implementation. We have created a FractalADL model of the application architecture using the FractalGUI modeling tool [11], taking into account the changes mentioned in Sect. 2.3 and Sect. 3.3. The resulting model was then extended by hand to accommodate behavior protocol specification, because it is not supported by the modeling tool.

To speed up and simplify the development, we have used a tool to create component skeletons from the architecture model. More detailed description of the transformation can be found in Sect. 4. The functional part of the application was then adapted from the CoCoME reference implementation and integrated into the generated component skeletons.

14.3.4.1 Testing the Implementation against Use-Case Scenarios

To enable the testing of functional properties specified by behavior protocols, FractalBPC allows monitoring communication on the interfaces of a component C when the application is running. The runtime checker integrated in FractalBPC automatically tests whether C communicates with other components in a way that is allowed by C's frame protocol. Any violation of the frame protocol by C or one of the components communicating with C is reported.

In addition, the reference implementation of the trading system contains a small test suite for testing the behavior of the implementation against the use case scenarios described in the CoCoME assignment. The test suite, based on the jUnit [16] framework, contains a number of tests which exercise operations prescribed by the respective use cases and verify that the system responds accordingly.

As it is, however, the test suite from the reference implementation is unsuitable for testing. The key issues are testing of crosscutting concerns, test design, and insufficient automation.

The tests attempt to verify not only that the implementation functions correctly, but also impose timing constraints on the executed operations. This makes the tests unreliable, because two orthogonal aspects are tested at the same time. Combined with rather immodest resource requirements of the application arising from the use of "heavy-duty" middleware packages for database functionality, persistence, and message-based communication, the application often fails to meet the test deadlines on common desktop hardware, even though it functions correctly.

Moreover, the tests always expect to find the trading system in a specific state, which is a very strong requirement. To accommodate it, all the applications comprising the trading system are restarted and the database is reinitialized after each test run, which adds extreme overhead to the testing process.

This is further exacerbated by insufficient automation of the testing infrastructure. The trading system consists of a number of components, such as the enterprise server and clients, store server and clients, database server, etc. Starting the trading system is a long and complicated process, which can take several minutes in the best case, and fail due to insufficient synchronization between parts of the system in the worst case. Manual starting of the trading system, combined with the need for restarting the system after each test run, makes the test suite in its present form unusable.

To enable testing in a reasonably small environment, we take the following steps to eliminate or mitigate the key issues, leading to a considerable increase in the reliability of the tests as well as reduced testing time:

- We simplify the implementation of the trading system by eliminating the GUI components, leaving just the business functionality, which allows the trading system to be operated in headless mode.
- We eliminate the validation of extra-functional properties from testing; timing properties of the trading system are gathered at runtime by a monitoring infrastructure described in Sect. 3.4.2. Validation of extra-functional system properties is independent from functional testing and is based on the data obtained during monitoring.
- We improve the testing infrastructure by automating the start of the trading system. This required identifying essential and unnecessary code paths and fixing synchronization issues between various parts of the system.

14.3.4.2 Runtime Monitoring of Extra-Functional Properties

The extra-functional properties articulated in the CoCoME assignment enhance the functional specification with information related to timing, reliability, and usage profile. The specification provides two kinds of properties: assumed and required. Assumed properties reflect domain knowledge and describe the environment in which the system will be expected to operate. The required properties reflect the requirements on performance of the system within the environment.

These parameters can be used in performance analysis of the system architecture preceding the implementation of the system to verify that the proposed architecture has the potential to satisfy performance requirements. However, pure model-based performance analysis is typically used to determine principal performance behavior, such trends in response to requests, not the actual performance of a real system in a real environment.

Deciding whether a particular instance of a system satisfies the performance requirements dictated by the specification requires analyzing performance data from a real system. Runtime monitoring requires the analysis of performance data to be performed isochronously with system execution. This limits the level of detail compared to offline analysis, but provides immediate information on high-level performance attributes. On the other hand, the data from performance measurements intended for offline analysis can be used to correct the assumptions and to calibrate a generic performance model to reflect the environment and properties of a particular instance of the system.

High-level performance data suitable for monitoring are typically exported by applications using technologies such as JMX [14] and SNMP [29], for which generic monitoring tools are available. However, exporting performance data is the final step. The data has to be first obtained using either an application-specific or a generic approach.

Application-specific approach requires that an application collects performance data internally, using its own measurement infrastructure. This allows obtaining certain application and domain specific performance metrics that cannot be obtained using a generic approach, but it also requires including support for performance measurements in various places directly in the implementation of application components, which in turn requires mixing functional and non-functional aspects of implementation. This can be alleviated using aspect-oriented programming which allow separating the implementation of functional and non-functional aspects of a system.

A generic approach based on architectural aspects exploits the description of application architecture as well as the capabilities of the runtime to obtain performance data. Architectural aspects are used to instrument an application with performance data collection capabilities, but their application is less intrusive than in the case of classical aspect oriented programming, because it is performed only at the design-level boundaries exposed by the application architecture. As a result, the instrumentation is completely transparent to the developer and does not require modifications in the implementation of an application, which in turn allows the work on performance monitoring to be done in parallel with development. Another advantage of using architectural aspects is that the application source code does not have to be available, but that was not an issue in this particular case.

Selecting performance properties for monitoring

To demonstrate the concept of using architectural aspects for performance monitoring, we have identified the following extra-functional properties from the CoCoME assignment that would be suitable for monitoring:

- t13-3: Time for signaling an error and rejecting an ID
- t15b2-2: Time waiting for validation
- t14-1: Time for showing the product description, price, and running total
- t34-1: Time for querying the inventory data store

We have taken into account the importance of the properties with respect to the performance of the system, therefore the preference was mostly on required properties associated with internal actions of the system (e.g. time to execute a database query) and not external actors and hardware (e.g. time to switch a light display). We have

also taken into account assumed properties that have the potential to be covered by a Service Level Agreement (i.e. guaranteed by an external service provider, such as a bank in case of credit card validation), where monitoring can be used to ensure that an external contractor fulfills its obligations.

Performance related extra-functional properties typically specify acceptable ranges for various performance metrics. An important aspect considered during selection of properties for monitoring was also the observability of the required performance metrics on the design level of abstraction, i.e. at component boundaries represented by component interfaces. Performance metrics that could not be calculated from data collected at component boundaries would have to be explicitly supported by the implementation.

Technical implementation

To obtain performance data from a system implemented using the Fractal component model, we have taken advantage of the mixin-based construction of controllers within a component membrane (see Sect. 2.1) supported by the Julia implementation of Fractal.

We have extended the membrane to include a performance monitoring controller and an interceptor on component business interfaces. The interceptor provides events related to business method invocations to the controller, which stores these events and calculates simple statistics and durations of method invocations. When an application stops, it writes the collected data to disk. For runtime performance monitoring, the controller provides a simple JMX based interface which allows configuring what methods and events should be observed and also allows accessing the simple summary statistics.

This approach is similar to that of FractalJMX [12], which is a Fractal extension that allows exposing the functional and control interfaces of Fractal components in a JMX agent and collecting simple statistics related to method invocations We have however implemented a custom interceptor which provides low-level data to the performance monitoring controller. The data can be used both for offline analysis and performance monitoring. JMX interface is used for management of monitoring controllers and for exposing simple statistics calculated from the low-level data.

Measurement results

Of the above extra-functional properties, we have decided to focus on t15b2-2, which is the assumed time of credit card validation. Using a simple simulator of the UC1 scenario, we have collected performance data related to invocations of the validateCard() method on the BankIf interface of the Bank component. The simulation was configured for 50 cash desks, started in 1 second intervals, and each processing 50 sales. The validateCard() method in the Bank component was implemented to wait a random time according to the histogram specification associated with the t15b2-2 property.

The distribution of validateCard() invocation times calculated from the measured data is identical to the specified distribution, which served as a basic validation of the approach. Using the measured data, we have performed an analysis with the goal to determine the average load on the Bank component, expressed as the number of validateCard() method invocations during a 60-second interval. This load may be covered

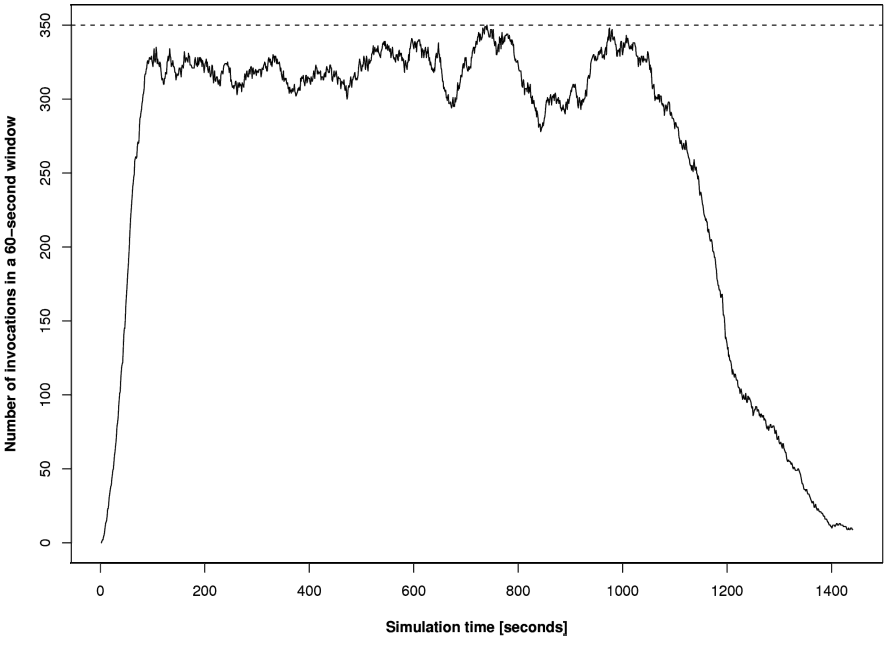

Fig. 4. The load on the card validation service of the Bank component

by a Service Level Agreement with a bank, which may only guarantee specific performance of its card validation service in response to a specific load.

The results of the analysis are shown in Fig. 4, with the dashed line denoting the maximal load on the Bank component given the above simulation parameters. The rising edge of the curve starting at time 0 corresponds to the delayed startup of individual cash desks, while the falling edge starting approx at time 1100 corresponds to closing of cash desks after they have processed 50 sales.

We would like to emphasize that the above analysis has been performed on data collected without actually modifying a single line of application code. Information about the distribution of durations of validateCard() invocations could be used to monitor the performance of the card validation service provided by a bank. On the other hand, runtime analysis (and throttling) of validateCard() invocation rate can be used to ensure that a store does not violate a Service Level Agreement.

The overhead of the measurement was 40 microseconds per method invocation, in 99% of cases, without any attempt at performance optimization. The duration of validateCard() invocation was between 4 and 5 seconds in 90% of cases. The difference between the times is 5 orders of magnitude, which in this particular case makes the overhead of the measurement insignificant. Due space constraints, we have only included the above analysis. The Distributed System Research Group project page on CoCoME in Fractal [33] provides additional measurement results.

14.3.5 Specification of Selected Components

This section is mostly focused on behavioral view of CoCoME components. More specifically, it assumes that the specification of component structure, interfaces, and overall architecture is taken over from the CoCoME assignment with the few modification mentioned in Sect. 3.1. As emphasized in Sect. 3.2, the behavior specification provided here is done in behavior protocols and stems from the CoCoME use cases and the component behavior encoded in the Java implementation provided in the CoCoME assignment. Since the behavior specification of the whole application is too large to fit into space reserved for this chapter, two "interesting" components (CashDeskApplication and CashDeskBus) were chosen to demonstrate the capabilities of behavior protocols. Interested reader may find the specification of other "interesting" components in the appendix and full specification at [33].

Demonstrating the ordinary usage of this formalism, the behavior protocol of CashDeskApplication describes the actual behavior of a cash desk. In principle, it captures the state machine corresponding to the sale process. In contrast, the behavior protocol of CashDeskBus illustrates the specific way of expressing mutual exclusion.

Since both these protocols are non-trivial, their "uninteresting" fragments are omitted in this section.

14.3.5.1 CashDeskApplication

The CashDeskApplication has application specific behavior – its frame protocol reflects the state of the current sale. It indicates what actions a cash desk allows the cashier to perform in a specific current sale state. The "interesting" parts of the protocol take the following form.

```
(
  # INITIALISED
  (
    ?CashDeskApplicationHandler.onSaleStarted
  );

  # SALE_STARTED
  (
    ?CashDeskApplicationHandler.onProductBarcodeScanned{
        !CashDeskConnector.getProductWithStockItem;
        !CashDeskApplicationDispatcher.sendProductBarcodeNotValid+
        !CashDeskApplicationDispatcher.sendRunningTotalChanged
    }
  )*; # <--- LOOP

  ?CashDeskApplicationHandler.onSaleFinished;

  # SALE_FINISHED
  (
    ?CashDeskApplicationHandler.onPaymentMode
  );

  # PAYING_BY_CASH
  (
    (
      (
        ?CashDeskApplicationHandler.onCashAmountEntered
      )*;
```

```
        # On Enter
        ?CashDeskApplicationHandler.onCashAmountCompleted{
          !CashDeskApplicationDispatcher.sendChangeAmountCalculated
        };

        ?CashDeskApplicationHandler.onCashBoxClosed{
          !CashDeskApplicationDispatcher.sendSaleSuccess;
          !CDLEventDispatcher.sendAccountSale;
          !CDLEventDispatcher.sendSaleRegistered
        }
      )
    )

  )* | (
    # Enable Express Mode
    ?CDLEventHandler.onExpressModeEnabled{
      !CashDeskApplicationDispatcher.sendExpressModeEnabled
    }
  )* | (
    # Disable Express Mode
    ?CashDeskApplicationHandler.onExpressModeDisabled
  )*
```

To communicate with each of the buses CashDeskBus and CashDeskLineBus, the component features a pair of interfaces (CashDeskApplicationHandler, CashDeskApplicationDispatcher and CDLEventHandler, CDLEventDispatcher). The interfaces contain a specific method for each event type that can occur on a bus. In addition, the interface, CashDeskInterface serves to get the data from Inventory.

The protocol specifies three parallel activities. The first one is the sale process itself, while the other two deal with cash desk mode switching. In the initial state, the sale process activity is waiting for SaleStartedEvent on the CashDeskBus (`?CashDeskApplicationHandler.onSaleStarted`). It denotes beginning of a new sale. Then (`;` operator) BarcodeScannedEvent is accepted (`?CashDeskApplicationHandler.onProductBarcodeScanned`) for each sale item. Repetition operator (*) ensures that arbitrary finite number of events can be accepted. In reaction (the expression enclosed in {}) to each BarcodeScannedEvent, the price is obtained from Inventory. (`!CashDeskConnector.getProductWithStockItem`) . Depending on the result, the rest of the CashDesk is informed about the change of total sale price (`!CashDeskApplicationDispatcher.sendRunningTotalChanged`) or, alternatively (+ operator), ProductBarcodeNotValidEvent is issued (`!CashDeskApplicationDispatcher.sendProductBarcodeNotValid`). When SaleFinishedEvent is accepted (`?CashDeskApplicationHandler.onSaleFinished`), the sale process reaches the payment phase which is specified in similar manner. When one sale is finished, the sale process activity returns to the initial state to accept another sale (repetition operator *). In parallel operators (|), the cash desk performs two other activities to process cash desk mode switching events coming from either of the buses.

This simplified version of the frame protocol does not capture paying by credit card and does not cope with events not allowed in a particular sale process state.

14.3.5.2 CashDeskBus

The particular bus behavior comprises of two different aspects – events serialization and multiplexing. While the former aspect takes part in modeling “many to one” messages, the latter aspect is related to “one to many” messages. The event passing is synchronous, meaning that if an event is emitted by a publisher component, the

component is blocked until all subscribers process the event. If there is another component wanting to emit a message when the bus is processing another message, the component is also blocked. Such behavior corresponds to the implementation using FractalRMI. This behavior might be prone to deadlocks, but fortunately, absence of deadlocks is one of properties we can verify using the behavior protocols.

As discussed in Sect. 3.1, the bus is implemented as a component. For every publisher and subscriber, it has an interface containing a method for every event type. As the bus component does not contain any application logic, its protocol can be generated using the information from the architecture – which components are involved in subscriber role, which components are involved in publisher role and what event types do they accept, resp. emit. This situation is not typical for behavior protocols.

The method used to model the serialization in behavior protocols follows the typical model of mutual exclusion in Petri nets – borrowing a token. The protocol representing the bus is accepting events from event producers in parallel, but it does not propagate them to the subscribers immediately. Instead of it, the bus protocol is waiting for the token event which is emitted by helper protocol. As the helper protocol does not produce another event until it receives response from the previous one, the bus event propagation parts are mutually excluded. Finally, the bus protocol must have empty parallel branch accepting the spare token events. Although the helper protocol in the model produces many spare token events which are just accepted by the empty parallel branch with no other use, this is not a performance issue in the implementation. In the implementation, standard Java synchronization with passive waiting is used to achieve the mutual exclusion – important is observable behavior, the means can differ in the implementation and model.

The multiplexing is straightforward – when the bus accepts an event from a producer and the token, the event is propagated to all subscribers.

The following protocol is a fragment of the CashDeskBus protocol $P_{CashDeskBus}$.

```
(?CashBoxControllerDispatcher.sendExpressModeDisabled{
  ?Helper.token{
    !CashDeskGUIHandler.onExpressModeDisabled|
    !LightDisplayControllerHandler.onExpressModeDisabled|
    !CardReaderControllerHandler.onExpressModeDisabled|
    !CashDeskApplicationHandler.onExpressModeDisabled
    }
  }
)*
|
(?CashDeskApplicationDispatcher.sendExpressModeEnabled{
  ?Helper.token{
    !CashDeskGUIHandler.onExpressModeEnabled|
    !LightDisplayHandler.onExpressModeEnabled|
    !CardReaderControllerHandler.onExpressModeEnabled
    }
  }
)*
|?Helper.token*
```

The fragment captures the synchronous delivering of ExpressModeEnabled and ExpressModeDisabled events. When the CashBoxController component emits the ExpressModeDisabled event, it is accepted by the bus (`?CashBoxController-Dispatcher.sendExpressModeDisabled`). Then, after accepting the token event, the

ExpressModeDisabled event is delivered in parallel to all subscribers (CashDeskGUI, LightDisplayController and CardReaderController). As the method calls are synchronous in behavior protocols, the bus waits until all subscribers acknowledge the event delivery. Then, the token is returned (the first closing curly brace) and finally, the CashBoxController is notified about successful delivery to all subscribers (the second closing curly brace). In the similar manner, the ExpressModeEnabled event is processed.

While the events from producers are accepted in parallel, which ensures that no producer can issue an event in a wrong moment, waiting for the equal token within the processing of distinct events ensures the mutual exclusion of the event deliveries, so the subscribers need not to care about parallelism. The final part of the fragment (`?Helper.token*`) accepts the unnecessary token events.

As there must be a token event source, the specification must be enriched by a helper protocol P_{Helper}:`!Helper.token*`. The complete frame protocol of the CashDeskBus component featuring mutual exclusion is then obtained by composing the protocols $P_{CashDeskBus}$ and P_{Helper} by the consent operator - $P_{CashDeskBus} \nabla_{\{Helper.Token\}} P_{Helper}$. It synchronizes the opposite actions (`!Helper.token` and `?Helper.token`) and replaces them by single internal action.

14.4 Transformations

In the process of implementing CoCoME components in Fractal, we have used a tool allowing for automated transformation of FractalADL specification to component code fragments. The tool runs as a backend to FractalADL and operates on the abstract syntax tree of a parsed ADL description. The implementation artifacts it can produce comprise code skeletons of component interfaces and code skeletons for primitive components.

A fragment of a code skeleton generated by the tool is provided below, showing the Coordinator component from the CoCoME example.

```
public class CoordinatorImpl implements
    BindingController, CoordinatorEventHandlerIf {

  // -------------------------------------------------------
  // Required interface CoordinatorEventDispatcherIf
  // -------------------------------------------------------
  protected                                  CoordinatorEventDispatcherIf
    CoordinatorEventDispatcherIf;

  // -------------------------------------------------------
  //         Provided         interface         CoordinatorEventHandlerIf
  // -------------------------------------------------------
  public void onSaleRegisteredEvent(
      org...cashdeskline.SaleRegisteredEvent arg0) {
    // TODO: Generated method
  }

  // -------------------------------------------------------
  // Implementation of the BindingController interface
  // -------------------------------------------------------
```

```
  public Object lookupFc(String clientItfName) ... {
    if (clientItfName.equals("CoordinatorEventDispatcherIf")) {
      return CoordinatorEventDispatcherIf;
    }
    ...
  }

  public void bindFc(String cltItfName, Object serverItf) ... {
    if (cltItfName.equals("CoordinatorEventDispatcherIf")) {
      CoordinatorEventDispatcherIf =
        (CoordinatorEventDispatcherIf) serverItf;
      return;
    }
    ...
  }
  ...
}
```

The generated code contains implementation of the binding controller, which is vital for binding required (client) interfaces. The required interfaces are reflected in the code by protected instance variables containing references to the bound provided interfaces of other components. The provided interfaces offered by the component are reflected in the *implements* clause of the generated class. The tool also generates a skeleton for each method of the provided interfaces.

14.5 Analysis

As a behavior protocol specifies behavior via allowable sequences of method calls on component's interfaces, the property to be analyzed is compliance of the behavior of components as correctness of communication on component's interfaces. In general, by correctness of communication, we mean absence of communication errors, i.e. a situation in which two or more components do not meet expectations of the others. Three types of communication errors are identified: *bad activity* – the issued event cannot be accepted, *no activity* (deadlock) – all of the ready events' tokens are prefixed by "?", and *infinite activity* (divergence) – the composed protocols "cannot reach their final events at the same time", so that the composed behavior would contain an infinite trace (only finite traces are allowed).

The compliance of behavior is of two kinds: *horizontal compliance* and *vertical compliance*. Horizontal compliance refers to correctness of communication among components on the same level of component hierarchy, whereas *vertical compliance* refers to correctness of communication on adjacent levels of component hierarchy, i.e. whether a composed component is correctly implemented by its subcomponents. The vertical compliance is therefore a kind of behavioral subtyping.

Checking of horizontal and vertical compliance makes sense only if behavior of each primitive component corresponds to its frame protocol, i.e. if each primitive component can accept and emit method calls on its external interfaces only in sequences that are determined by its frame protocol. This correspondence can be checked in two ways: (i) code model checking with the modified Java PathFinder [15] (JPF) and (ii) run-time checking.

Code model checking of primitive components with JPF allows exhaustive verification whether the implementation of each primitive component corresponds to its frame protocol. Since each primitive component is checked in isolation, the problem of missing environment has to be faced (Java PathFinder checks only complete programs) via constructing an artificial environment for a component and checking the complete program composed of the component and environment. The behavior of an environment is specified by the component's inverted frame protocol, which is derived from the frame protocol by replacing all the accept events with emit events and vice versa.

Although the well-know problem of state explosion is partially mitigated by application of code model checking to isolated primitive components (a single component has a smaller state space than the whole application), still the checking has very high time and space complexity; for highly parallel components, it may even not be feasible. We address this by optional heuristic transformations of environment's behavior specification that help reduce the complexity of a component environment, while making the checking not exhaustive (not all thread interleavings are checked if the heuristic transformations are used). Alternatively, it is also possible to use run-time checking in such a case.

The basic idea of run-time checking is to monitor method call-related events on the component's external interfaces at run-time and check whether the trace composed from the events is specified by the component's frame protocol. Since only a single run of an application is checked in this way (run-time checking is inherently not exhaustive), a violation of a frame protocol may not be detected for many runs of the application that involves the erroneous component; in this respect, the technique of run-time checking is similar to testing.

14.6 Tools and Results

Verification of an application consists of two steps. First step is checking the protocols compliance. Protocols of all components used to implement a composite component are checked against the frame protocol of the composite component. Second step is checking whether the implementation of the primitive components correspond to their protocols.

Compliance of the whole Trading System was checked using the dChecker [6] tool with positive result. The dChecker tool is based on translation of the protocols into minimized finite state machines. Then, composite state space is generated on the fly to discover a potential bad activity error or deadlock. Moreover, in order to fight the state explosion problem, dChecker supports both parallel and distributed verification, so that the full computational power of multiprocessor and multicomputer systems is exploited. For illustration, correctness of the whole architecture takes 192 seconds to be verified on a 2xDualCore at 2.3GHz with 4GB RAM PC. Specifically, the protocol of CashDeskApplication is translated into finite state machine consisting of 944 states. The composite state space of CashDesk features 398029 states and it takes 8 seconds to be verified (on the same PC).

Correspondence of the implementation of primitive components to their frame protocols is verified by the Java PathFinder (JPF) model checker. Since JPF, by default,

checks only low level properties like deadlocks and uncaught exceptions, we use JPF in combination with the behavior protocol checker (BPC) [26]. Component environment is represented by a set of Java classes that are constructed in a semi-automated way: (i) The EnvGen tool (Environment Generator for JPF) is used to generate the classes according to the behavior specification of the environment via the component's inverted frame protocol, and (ii) the generated classes are manually modified if the environment has to respect data-flow and the component's state in order to behave correctly (original behavior protocols do not model data and component's state explicitly).

By code checking implementation of the CashDeskApplication against the frame protocol created according to the reference specification of UC1, we were able to detect the inconsistency between the reference implementation and specification of UC1 that is first mentioned in Sect 3.2. Detection of this inconsistency took 2 seconds on a 2xDualCore at 2.3GHz with 4 GB RAM PC. Code checking of the implementation of CashDeskApplication against the frame protocol created according to the reference implementation has not reported any error and took 14 seconds. Nevertheless, switching between the express and normal mode is not checked, since the environment is not able to find whether the application is in the express mode or not, and thus it does not know whether it can trigger payment by credit card (forbidden in the express mode). Moreover, we also had to introduce the CashAmountCompleted event into the frame protocol and implementation of CashDeskApplication. This change was motivated by the need to explicitly denote the moment when the cash amount is completely specified (originally, the CashAmountEntered event with a specific value of its argument was used for this purpose). Were the CashAmountCompleted event not added, the environment for CashDeskApplication would exercise the component in such a way that a spurious violation of its frame protocol would be reported by JPF.

As for run-time checking, the special version of BPC is used again. The difference is that notification is not performed by JPF, but by runtime interceptors of method calls on component's external interfaces; moreover, no backtracking in BPC is needed since only a single run of the application is checked.

When using the tools, however, the state explosion problem became an issue. Some of the behavior protocols (namely CashDeskBus and Data) originally featured prohibitively large state space. Thus, in order to fight the state explosion problem, heuristics were employed. First, CashDeskBus protocol is separated into multiple protocols (as if for multiple components), so that it can be represented by multiple smaller finite state machines in contrast to a single unfeasibly large state machine. Second, method calls inside behavior protocol of the Data component are explicitly annotated by the thread number. This is again in order to the fight state explosion as this makes the protocol more deterministic while preserving the same level of parallelism. For these reasons, protocols on the CoCoME Fractal web page differ from the protocols described in Sect. 3.5 and Appendix, as they include also the heuristics.

14.7 Summary

In this chapter, we presented our solution to the CoCoME assignment that is based on the Fractal component model extended with support for component behavior specification.

Several issues in the UML specification and reference implementation were discovered and solved during implementation of the CoCoME assignment in the Fractal component model. Most notably, the component hierarchy was reorganized in order to improve clarity of the design and the hierarchical bus was split into two independent buses. These were modeled by primitive components, since Fractal does not support message bus as a first-class entity.

Behavior of all components of the Trading System is specified via behavior protocols. Since the CoCoME assignment does not include a complete UML behavior specification (e.g. via activity diagrams and state charts), behavior protocols for all the components are based on the provided plain-English use cases, the UML sequence diagrams, and the reference Java implementation. By application of code model checking to our implementation (based on the reference implementation), we were able to detect inconsistency between the specification and reference implementation of UC1 (details in Sect. 3.2. and Sect. 5). Consequently, we have created two versions of behavior protocols for several components – one version corresponds to the UML specification and the second to the reference implementation.

For deployment and distribution, we have used Fractal-specific means (FractalRMI and FractalADL). Since the buses are represented by primitive components that route the message, use of JMS was eliminated.

One limitation of our approach is only partial support for extra-functional properties via monitoring (no static analysis is employed). In particular, performance is monitored by custom component controllers for the Julia implementation of Fractal.

Very useful is support for verification of primitive component's code against the behavior specification (behavior protocols). Using that, it was possible to check whether the implementation corresponds to the behavior specification created at design time.

Acknowledgements

The authors would like to thank Marc Leger (France Telecom R&D) for providing the transformation tool that generates primitive component skeletons from FractalADL.

References

1. Adamek, J., Bures, T., Jezek, P., Kofron, J., Mencl, V., Parizek, P., Plasil, F.: Component Reliability Extensions for Fractal Component Model (2006), `http://kraken.cs.cas.cz/ft/public/public_index.phtml`
2. ASM, `http://asm.objectweb.org/`
3. Baude, F., Baduel, L., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, Composing, Deploying for the Grid. In: Cunha, J.C., Rana, O.F. (eds.) GRID COMPUTING: Software Environments and Tools. Springer, Heidelberg (January 2006)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.B.: The FRACTAL component model and its support in Java. Softw., Pract. Exper. 36(11-12) (2006)
5. Bruneton, E., Coupaye, T., Stefani, J.B.: Fractal Component Model, version 2.0-3 (February 2004)
6. dChecker, `http://dsrg.mff.cuni.cz/projects.phtml?p=dchecker`

7. Dream, http://dream.objectweb.org/
8. FRACTNET, http://www-adele.imag.fr/fractnet/
9. Fractal ADL, http://fractal.objectweb.org/fractaladl/index.html
10. Fractal BPC, http://fractal.objectweb.org/fractalbpc/index.html
11. Fractal GUI, http://fractal.objectweb.org/fractalgui/
12. Fractal JMX, http://fractal.objectweb.org/fractaljmx/
13. Fractal RMI, http://fractal.objectweb.org/fractalrmi/index.html
14. Java Management Extensions (JMX) Specification, version 2.0, JSR 255, http://jcp.org/en/jsr/detail?id=255
15. Java PathFinder, http://javapathfinder.sourceforge.net/
16. JUnit, http://www.junit.org/
17. Layaida, O., Hagimont, D.: PLASMA: A Component-based Framework for Building Self-Adaptive Applications. In: Proceedings of SPIE/IS&T Symposium On Electronic Imaging, Conference on Embedded Multimedia Processing and Communications, San Jose, CA, USA (January 2004)
18. Loiret, F., Servat, D., Seinturier, L.: A First Experimentation on High-Level Tooling Support upon Fractal. In: Proceedings of the 5th International ECOOP Workshop on Fractal Component Model (Fractal 2006), Nantes, France (July 2006)
19. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying Distributed Software Architecture, Proceeding of the 5th European Software Engineering Conference (ESE'C 1995). In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, pp. 137–153. Springer, Heidelberg (1995)
20. Mencl, V., Bures, T.: Microcomponent-Based Component Controllers: A Foundation for Component Aspects. In: Proceedings of APSEC 2005, Taipei, Taiwan, December 2005, IEEE CS, Los Alamitos (2005)
21. Mencl, V., Polak, M.: UML 2.0 Components and Fractal: An Analysis. In: Proceedings of the 5th International ECOOP Workshop on Fractal Component Model (Fractal 2006), Nantes, France (July 2006)
22. Microsoft .NET Framework, http://www.microsoft.com/net/
23. Object Management Group, Corba Components, version 3.0 (June 2002), http://www.omg.org/docs/formal/02-06-65.pdf
24. OMG, Object Management Group: UML Profile for Schedulability, Performance and Time (2005), http://www.omg.org/cgi-bin/doc?formal/2005-01-02
25. Parízek, P., Plášil, F.: Modeling Environment for Compoment Model Checking from Hierarchical Architecture. In: Proceedings of Formal Aspects of Component Software (FACS 2006), Prague, Czech Republic (September 2006)
26. Parízek, P., Plášil, F., Kofroň, J.: Model checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. In: Proceedings of 30th IEEE/NASA Sofrware Engineering Workshop (SEW-30), January 2007, pp. 133–141. IEEE Computer Society, Los Alamitos (2007)
27. Plášil, F., Višňovský, S.: Behavior Protocols for Software Components. IEEE Transactions on Software Engineering 28(11) (November 2002)
28. Seinturier, L., Pessemier, N., Duchien, L., Coupaye, T.: A Component Model Engineered with Components and Aspects. In: Gorton, I., Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063. Springer, Heidelberg (2006)
29. Simple Network Management Protocol (SNMP), RFC (1157), http://www.faqs.org/rfcs/rfc1157.html
30. Sun Microsystems, JSR 220: Enterprise JavaBeansTM,Version 3.0

31. THINK, http://think.objectweb.org
32. Visser, W., Havelund, K., Brat., G., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering Journal 10(2) (April 2003)
33. Fractal CoCoME, http://dsrg.mff.cuni.cz/cocome/fractal

Appendix

This section contains the full behavior specification of the CashDeskApplication and CashDeskBus components discussed in the Sect. 3.5, the CashDeskBox component, and the StoreApplication component. In comparison to the protocols published on the project site [33], the interface names are slightly abbreviated for brevity. In a similar vein, the CashDeskBus protocol is presented for simplicity in a form of a single protocol containing a number of parallel activities - as mentioned in Sect. 6, the CashDeskBus protocol used for compliance checking was split into smaller parts to overcome technical difficulties with state explosion.

TradingSystem::CashDeskLine::CashDesk::CashDeskApplication

```
(
# INITIALISED
(
 ( #Accept and throw away events that are not expected in this phase
  ?CashDeskApplicationHandler.onSaleFinishedEvent+
  ?CashDeskApplicationHandler.onPaymentModeEvent+
  ?CashDeskApplicationHandler.onCashAmountEnteredEvent+
  ?CashDeskApplicationHandler.onCashBoxClosedEvent+
  ?CashDeskApplicationHandler.onProductBarcodeScannedEvent+
  ?CashDeskApplicationHandler.onCreditCardScannedEvent+
  ?CashDeskApplicationHandler.onPINEnteredEvent
 )*;
 ### The important part:
 ?CashDeskApplicationHandler.onSaleStartedEvent
);


# SALE_STARTED
(
 ### The important part:
 ?CashDeskApplicationHandler.onProductBarcodeScannedEvent {
  # ExpressMode & products.size == 8
  NULL
  +
  (
   !CashDeskConnectorIf.getProductWithStockItem;

   !CashDeskApplicationDispatcher.sendProductBarcodeNotValidEvent
   +
   !CashDeskApplicationDispatcher.sendRunningTotalChangedEvent
  )
 ###
 } +
 ?CashDeskApplicationHandler.onSaleStartedEvent+
 ?CashDeskApplicationHandler.onPaymentModeEvent+
 ?CashDeskApplicationHandler.onCashAmountEnteredEvent+
 ?CashDeskApplicationHandler.onCashBoxClosedEvent+
 ?CashDeskApplicationHandler.onCreditCardScannedEvent+
 ?CashDeskApplicationHandler.onPINEnteredEvent
)*;  # <--- LOOP
```

```
(
 ### The important part:
 ?CashDeskApplicationHandler.onSaleFinishedEvent;
 (
  ?CashDeskApplicationHandler.onSaleStartedEvent+
  ?CashDeskApplicationHandler.onSaleFinishedEvent+
  ?CashDeskApplicationHandler.onCashAmountEnteredEvent+
  ?CashDeskApplicationHandler.onCashBoxClosedEvent+
  ?CashDeskApplicationHandler.onCreditCardScannedEvent+
  ?CashDeskApplicationHandler.onPINEnteredEvent+
  ?CashDeskApplicationHandler.onProductBarcodeScannedEvent
 )*
);

# SALE_FINISHED
(
 ### The important part:
 ?CashDeskApplicationHandler.onPaymentModeEvent;
 ###
 (
  ?CashDeskApplicationHandler.onSaleStartedEvent+
  ?CashDeskApplicationHandler.onSaleFinishedEvent+
  ?CashDeskApplicationHandler.onCashBoxClosedEvent+
  ?CashDeskApplicationHandler.onPaymentModeEvent+
  ?CashDeskApplicationHandler.onProductBarcodeScannedEvent+
  ?CashDeskApplicationHandler.onPINEnteredEvent
 )*
);

# PAYING_BY_CASH
(
 (
  (
   ?CashDeskApplicationHandler.onSaleStartedEvent+
   ?CashDeskApplicationHandler.onSaleFinishedEvent+
   ?CashDeskApplicationHandler.onCashBoxClosedEvent+
   ?CashDeskApplicationHandler.onPaymentModeEvent+
   ?CashDeskApplicationHandler.onProductBarcodeScannedEvent+
   ?CashDeskApplicationHandler.onPINEnteredEvent+
   ?CashDeskApplicationHandler.onCreditCardScannedEvent+
   ### The important part:
   ?CashDeskApplicationHandler.onCashAmountEnteredEvent
  )*;

  # On Enter
  ### The important part:
  ?CashDeskApplicationHandler.onCashAmountEnteredEvent {
   !CashDeskApplicationDispatcher.sendChangeAmountCalculatedEvent
  };

  (
   ?CashDeskApplicationHandler.onSaleStartedEvent+
   ?CashDeskApplicationHandler.onSaleFinishedEvent+
   ?CashDeskApplicationHandler.onPaymentModeEvent+
   ?CashDeskApplicationHandler.onCashAmountEnteredEvent+
   ?CashDeskApplicationHandler.onProductBarcodeScannedEvent+
   ?CashDeskApplicationHandler.onPINEnteredEvent+
   ?CashDeskApplicationHandler.onCreditCardScannedEvent
  )*;
  ### The important part:
  ?CashDeskApplicationHandler.onCashBoxClosedEvent {
   !CashDeskApplicationDispatcher.sendSaleSuccessEvent;
   !CashDeskDispatcher.sendAccountSaleEvent;
   !CashDeskDispatcher.sendSaleRegisteredEvent
  }
 )
```

```
+

# PAYING_BY_CREDITCARD
(
 (
  ?CashDeskApplicationHandler.onCreditCardScannedEvent;


  # CREDITCARD_SCANNED
  (
   ?CashDeskApplicationHandler.onPINEnteredEvent {
    !BankLock.lock;
    !BankIf.validateCard;
    (
     !CashDeskApplicationDispatcher.sendInvalidCreditCardEvent
     +
     (
      !BankIf.debitCard;
      !CashDeskApplicationDispatcher.sendInvalidCreditCardEvent
     )
    );
    !BankLock.unlock
   }
   +
   ?CashDeskApplicationHandler.onSaleStartedEvent+
   ?CashDeskApplicationHandler.onSaleFinishedEvent+
   ?CashDeskApplicationHandler.onPaymentModeEvent+
   ?CashDeskApplicationHandler.onCashAmountEnteredEvent+
   ?CashDeskApplicationHandler.onProductBarcodeScannedEvent+
   ?CashDeskApplicationHandler.onCreditCardScannedEvent+
   ?CashDeskApplicationHandler.onCashBoxClosedEvent
  )*;
  ?CashDeskApplicationHandler.onPINEnteredEvent {
   !BankLock.lock;
   !BankIf.validateCard;
   !BankIf.debitCard;
   !CashDeskApplicationDispatcher.sendInvalidCreditCardEvent;
   !BankLock.unlock
  }
 )*;


 ?CashDeskApplicationHandler.onCreditCardScannedEvent;

 # CREDITCARD_SCANNED
 (
  ?CashDeskApplicationHandler.onSaleStartedEvent+
  ?CashDeskApplicationHandler.onSaleFinishedEvent+
  ?CashDeskApplicationHandler.onPaymentModeEvent+
  ?CashDeskApplicationHandler.onCashAmountEnteredEvent+
  ?CashDeskApplicationHandler.onProductBarcodeScannedEvent+
  ?CashDeskApplicationHandler.onCreditCardScannedEvent+
  ?CashDeskApplicationHandler.onCashBoxClosedEvent
 )*;

 ### The important part:
 ?CashDeskApplicationHandler.onPINEnteredEvent {
  !BankLock.lock;
  !BankIf.validateCard;
  !BankIf.debitCard;
  !BankLock.unlock;
  !CashDeskApplicationDispatcher.sendSaleSuccessEvent;
  !CashDeskDispatcher.sendAccountSaleEvent;
  !CashDeskDispatcher.sendSaleRegisteredEvent
 }
)

)
```

```
  )* | (
   # Enable Express Mode
   ?CashDeskHandler.onExpressModeEnabledEvent {
    !CashDeskApplicationDispatcher.sendExpressModeEnabledEvent
   }

  )* | (
   # Disable Express Mode
   ?CashDeskApplicationHandler.onExpressModeDisabledEvent
  )*
```

TradingSystem::CashDeskLine::CashDesk::CashDeskBus

```
(!Helper.token)*

sync{Helper.token}

(?CashBoxControllerDispatcher.sendCashAmountEnteredEvent{
 ?Helper.token{
  !CashDeskApplicationHandler.onCashAmountEnteredEvent|
  !PrinterControllerHandler.onCashAmountEnteredEvent|
  !CashDeskGUIHandler.onCashAmountEnteredEvent
  }
 })*
|
(?CashBoxControllerDispatcher.sendCashBoxClosedEvent{
 ?Helper.token{
  !CashDeskApplicationHandler.onCashBoxClosedEvent|
  !PrinterControllerHandler.onCashBoxClosedEvent
  }
 })*
|
(?CardReaderControllerDispatcher.sendCreditCardScannedEvent{
 ?Helper.token{
  !CashDeskApplicationHandler.onCreditCardScannedEvent
  }
 })*
|
(?CashBoxControllerDispatcher.sendExpressModeDisabledEvent{
 ?Helper.token{
  !CashDeskGUIHandler.onExpressModeDisabledEvent|
  !LightDisplayControllerHandler.onExpressModeDisabledEvent|
  !CardReaderController.onExpressModeDisabledEvent|
  !CashDeskApplicationHandler.onExpressModeDisabledEvent
  }
 })*
|
(?CashDeskApplicationDispatcher.sendExpressModeEnabledEvent{
 ?Helper.token{
  !CashDeskGUIHandler.onExpressModeEnabledEvent|
  !LightDisplayControllerHandler.onExpressModeEnabledEvent|
  !CardReaderController.onExpressModeEnabledEvent
  }
 })*
|
(?CashDeskApplicationDispatcher.sendChangeAmountCalculatedEvent{
  !CashDeskGUIHandler.onChangeAmountCalculatedEvent|
  !PrinterControllerHandler.onChangeAmountCalculatedEvent|
  !CashBoxController.onChangeAmountCalculatedEvent
  }
)*
|
(?CashDeskApplicationDispatcher.sendInvalidCreditCardEvent{
  !CashDeskGUIHandler.onInvalidCreditCardEvent
  }
)*
```

```
|
(?CashBoxControllerDispatcher.sendPaymentModeEvent{
 ?Helper.token{
  !CashDeskApplicationHandler.onPaymentModeEvent
  }
 })*
|
(?CardReaderControllerDispatcher.sendPINEnteredEvent{
 ?Helper.token{
  !CashDeskApplicationHandler.onPINEnteredEvent
  }
 })*
|
(?CashDeskApplicationDispatcher.sendProductBarcodeNotValidEvent{
  !CashDeskGUIHandler.onProductBarcodeNotValidEvent
  }
)*
|
(?ScannerControllerDispatcher.sendProductBarcodeScannedEvent{
 ?Helper.token{
  !CashDeskApplicationHandler.onProductBarcodeScannedEvent
  }
 })*
|
(?CashDeskApplicationDispatcher.sendRunningTotalChangedEvent{
  !CashDeskGUIHandler.onRunningTotalChangedEvent|
  !PrinterControllerHandler.onRunningTotalChangedEvent
  }
)*
|
(?CashBoxControllerDispatcher.sendSaleFinishedEvent{
 ?Helper.token{
  !CashDeskApplicationHandler.onSaleFinishedEvent|
  !PrinterControllerHandler.onSaleFinishedEvent
  }
 })*
|
(?CashBoxControllerDispatcher.sendSaleStartedEvent{
 ?Helper.token{
  !PrinterControllerHandler.onSaleStartedEvent |
  !CashDeskApplicationHandler.onSaleStartedEvent|
  !CashDeskGUIHandler.onSaleStartedEvent
  }
 })*
|
(?CashDeskApplicationDispatcher.sendSaleSuccessEvent{
  !PrinterControllerHandler.onSaleSuccessEvent|
  !CashDeskGUIHandler.onSaleSuccessEvent
  }
)*
|
# accept spare tokens
?Helper.token*
```

TradingSystem::CashDeskLine::CashDesk::CashBox protocol

This component is an example of simple bus event producer and subscriber containing no internal state information. In response to the cashier actions, which are not modeled, it is sending events and in parallel it is able to receive the ChangeAmountCalculated event.

```
(
 !CashBoxControllerDispatcherIf.sendCashAmountEnteredEvent
 +
```

```
 !CashBoxControllerDispatcherIf.sendCashBoxClosedEvent
 +
 !CashBoxControllerDispatcherIf.sendExpressModeDisabledEvent
 +
 !CashBoxControllerDispatcherIf.sendPaymentModeEvent
 +
 !CashBoxControllerDispatcherIf.sendSaleFinishedEvent
 +
 !CashBoxControllerDispatcherIf.sendSaleStartedEvent
)* |
?CashBoxController.onChangeAmountCalculatedEvent*
```

TradingSystem::Inventory::StoreApplication protocol

This component is an example of a component from the Inventory part of the application which does not directly communicate with a bus.

```
(
 (
  ?CashDeskConnectorIf.getProductWithStockItem  {
   !PersistenceQueryIf_1.getPersistenceContext_1;
   !StoreQueryIf_1.queryStockItem_1
  }
 )
 +
 (
  ?AccountSaleEvent.bookSale {
   !PersistenceQueryIf_1.getPersistenceContext_1;
   !StoreQueryIf_1.queryStockItemById_1*;

   !PersistenceQueryIf_1.getPersistenceContext_1;
   !StoreQueryIf_1.queryLowStockItems_1;
   !StoreQueryIf_1.queryStoreById_1;
   !ProductDispatcherIf.orderProductsAvailableAtOtherStores;
   (!StoreQueryIf_1.orderProductsAvailableAtOtherStores_1 + NULL)
  }
 )
)*
|
(
 !PersistenceQueryIf_1.getPersistenceContext_2;
 (
  (
   !StoreQueryIf_1.queryProductById_2*;
   !StoreQueryIf_1.queryStoreById_2*
  )
  +
  (# Fig. 20
   !StoreQueryIf_1.queryOrderById_2;
   !StoreQueryIf_1.queryStockItem_2*
  )
  +
  ( # Fig. 21
   !StoreQueryIf_1.queryStoreById_2;
   !StoreQueryIf_1.queryAllStockItems_2
  )
  +
  ( # Fig. 23
   !StoreQueryIf_1.queryStockItemById_2
  )
  +
  (
   !StoreQueryIf_1.queryLowStockItems_2
  )
  +
  (
```

```
   !StoreQueryIf_1.queryProducts_2
  )
 )
)*
|
(
 ?MoveGoodsIf.queryGoodAmount{
  !PersistenceQueryIf_1.getPersistenceContext_3;
  !StoreQueryIf_1.queryProductById_3*
 }
 +
 ?MoveGoodsIf.sendToOtherStore{
  !PersistenceQueryIf_1.getPersistenceContext_3;
  !StoreQueryIf_1.queryStockItem_3*
 }
)*
|
?MoveGoodsIf.acceptFromOtherStore{
 !PersistenceQueryIf_1.getPersistenceContext_4;
 !StoreQueryIf_1.queryProductById_4*
}*
```