15 CoCoME in SOFA*

Tomáš Bureš[1,2], Martin Děcký[1], Petr Hnětynka[1], Jan Kofroň[1,2], Pavel Parízek[1], František Plášil[1,2], Tomáš Poch[1], 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
{tomas.bures,martin.decky,petr.hnetynka,jan.kofron,
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, Czech Republic
{bures,kofron,plasil}@cs.cas.cz

Abstract. This chapter presents our solution to the CoCoME assignment that is based on the SOFA 2.0 (SOFtware Appliances) hierarchical component model. The solution involves (i) modeling architecture in SOFA meta-model, (ii) specification of component behavior via extended behavior protocols, (iii) checking behavior compliance of components, (iv) verification of correspondence between selected component Java code and behavior specification, (v) deployment to SOFA run-time environment (using connectors that support RMI and JMS), and (vi) modeling of performance and resource usage via layered queueing networks. We faced several issues during implementation of the CoCoME assignment in SOFA 2.0. Most notably, the architecture was modified in order to improve clarity of the design – in particular, the hierarchical bus was replaced by two separate buses and the Inventory component was restructured. Extended behavior protocols for all the components are based on the provided plain-English use cases, the UML sequence diagrams, and the reference Java implementation (the assignment does not include a complete UML behavior specification e.g. via activity diagrams and state charts).

15.1 Introduction

15.1.1 Goals and Scope of the Component Model

SOFA 2.0 [9] is a component model employing hierarchically composed components. It is a direct successor of the SOFA component model [24], which has provided the following features: ADL-based design, behavior specification using behavior protocols [25], automatically generated connectors supporting seamless and transparent distribution of applications, and distributed runtime environment with dynamic update of components.

* This work was partially supported by the Grant Agency of the Czech Republic project 201/06/0770, the results will be used in the ITEA/EUREKA project OSIRIS Σ!2023.

A. Rausch et al. (Eds.): Common Component Modeling Example, LNCS 5153, pp. 388–417, 2008.
© Springer-Verlag Berlin Heidelberg 2008

From its predecessor, SOFA 2.0 has inherited the core component model, which is enhanced in the following way: (i) the component model is defined by means of its meta-model; (ii) it allows for dynamic reconfiguration of component architecture and for accessing components under the SOA concepts; (iii) via connectors, it supports not only plain method invocation, but in fact any communication style [8]; (iv) it clearly separates and makes extensible the control (extra-functional) part of component implementations. Similar to its predecessor, SOFA 2.0 is not only a tool for modeling, but it provides a complete framework [28] supporting all the stages of an application lifecycle from development to execution.

In SOFA 2.0, a component is primarily treated as a black-box with well-defined interfaces and exists at design time, deployment time, and run time. Components are defined using their *frame* and *architecture*. A frame provides a black-box view of a component via defining component's interfaces. An architecture implements at least one frame and defines internal structure of the component, i.e. subcomponents and their composition. Semantics of the composition is defined via Extended Behavior Protocols; SOFA 2.0 also supports divide and conquer via interface specification.

The component specification is separated from the implementation and is defined using models (based on the SOFA 2.0 meta-model). Extra functional properties (EFP) are specified by separate annotations and resource model.

Component behavior is specified using Extended Behavior Protocols (EPB). EBPs allow to model and verify the behavior compliance and LTL_{-X} properties. The verification tools can verify the component architecture independently from the implementation, and the relation of the model and implementation.

Deployment-related features are specified separately from the architecture specification in a deployment plan.

15.1.2 Modeled Cutout of CoCoME

We model (and verify) nearly all aspects of the CoCoME example; regarding EFPs, however, we do not model other properties than resource and performance related.

To allow modeling of the example in SOFA 2.0, we have introduced minor changes to the original architecture. In particular, these include replacement of buses, introducing Enterprise server, and restructuring StoreServer. We verify behavior compliance of all components of the example and the results of performance prediction are compared with benchmarks of the reference implementation. For fully automatic behavior verification, we use a tool chain consisting of EBP to Promela translator and the Spin model checker; for performance prediction we use the Carleton LQN solver.

15.1.3 Benefit of the Modeling

The biggest advantages of our approach are namely (i) behavior verification, (ii) performance and resource usage performance prediction at design time, and (iii) the potential of checking whether a given use case is really implemented.

On the other hand, our approach cannot treat behavior that cannot be modeled by a regular language (e.g. recursion). Together with the manual preparation of the resource usage model, this remains the weakest part of our approach for now.

Usage of SOFA 2.0 and the verification and prediction tools and approaches is quite easy; an average computer science student takes approximately five days to learn about SOFA 2.0 itself and another five days to write behavior specification of simple components.

15.1.4 Effort and Lessons Learned

We have spent approximately 10 person-months to model the CoCoME example (we treat nearly all its aspects). Besides the modeling itself, this includes also modifications of the prototype implementation and collection of performance data on the prototype, required for comparison with the performance modeling results.

Major lessons learned from the modeling effort include the number of details that need to be covered to model a project in its entirety, as opposed to modeling only selected aspects. Having the fully modeled CoCoME example at hand is also extremely useful for further research into what other properties can be modeled and checked.

The chapter continues as follows. Section 2 provides an in-depth description of SOFA 2.0, as well as a brief comparison of SOFA 2.0 and other contemporary component models. In Section 3, we present our modeling of the CoCoME example in SOFA 2.0 and Section 4 provides short note about used transformation. In Section 5, we analyze modeling results while Section 6 presents used tools and achieved results. Section 7 concludes the paper.

15.2 Component Model

15.2.1 Static View

SOFA 2.0 uses a hierarchical component model with connectors, which are also first-class entities like components. The component model is defined using a meta-model [22]. In comparison to an ADL-based component model definition (like in the previous SOFA version), or even just plain language description, such an approach has many advantages like support of MDD, the possibility of automated generation of meta-data repositories with a standard interface, a standard format for data exchange among repositories, support for automated generation of model editors, etc. As the particular technology for defining the meta-model and generating a meta-data repository, we have been using EMF [10]. The meta-model is depicted in Appendix A; a brief description of main entities of the meta-model follows (a more complete description is available in [9]).

The *NamedEntity* and *VersionedEntity* classes[1] are reused multiple times in the meta-model. All other classes featuring a name inherit from *NamedEntity*. The *VersionedEntity* class further extends *NamedEntity* by adding a version (the versioning model used in SOFA is described in [9]).

A black-box view of a component is defined by the *Frame* class (it inherits from the *VersionedEntity*). The provided, resp, required, interfaces of a frame are modeled

[1] Like other elements in the meta-model, they are meta-classes but for better readability we omit the meta- prefix of meta-classes, meta-associations, etc. in the rest of the paper.

by the *provideInterface,* resp. *requiredInterface,* association with the *Interface* class, which is further associated with the *InterfaceType* class defining the real type of the interface. Also, *Frame* is associated with the *Property* class, which defines the name-value properties used to parameterize components (these values are specified at the deployment time).

A gray-box view of a component is defined by the *Architecture* class. The component's architecture implements at least one frame captured by the association between the *Frame* and *Architecture* classes (the option of multiple frames for an architecture allow for taking different views on the component behavior) and contains subcomponents and connections among them. If the architecture is empty, then the component is *primitive* and is directly implemented. Architectures can also add other properties (again captured via the association with the *Property* class), and/or can expose subcomponents' properties as their own (captured by the association with the *MappedProperty* class).

Connections among subcomponents are realized via connectors. At the meta-model level, connectors are just links among components' interfaces and they are captured by the *Connection* and *Endpoint* classes. The communication style of a connector and its non-functional features, which it has to provide at runtime (like secure connection, etc.) are defined by the *Feature* class, which is associated with *Interface*.

The dynamic reconfigurations [15] are allowed through well-defined reconfiguration patterns. Currently, SOFA 2.0 supports three patterns: factory pattern, removal pattern, and utility interface pattern. The factory pattern allows adding new components to the architecture at runtime; the removal pattern is complementary to the former and allows removing components at runtime. In the meta-model, the *Factory* class (which inherits from *Annotation*) can be used to mark an interface that it can create new component instances. The last pattern on the list introduces the concept of *utility* interfaces (in the meta-model the *connectionType* attribute of the *Interface*), which stems from SOA and allows accessing interfaces across the component boundaries (orthogonally to component hierarchy). In more detail, a provided utility interface can be accessed by any component at any level of nesting or from a completely different application (even non-component based), and the reference to such an interface can be freely passed among components. Thus, it serves as a generally accessible service. In a similar vein, a required utility interface can be connected across the application hierarchy. The concept of utility interfaces combines the advantages of component-based design and SOA.

15.2.2 Behavior View

For modeling behavior of SOFA 2.0 components, *Extended Behavior Protocols (EBP)* [16] are used; they have been derived from the original behavior protocols [2] by addition of enumeration data types and synchronization of multiple events [16].

Behavior protocols describe the behavior of software components as a set of traces of events appearing on component interfaces (method calls requests and returns from the calls (responses)). A behavior protocol is an expression built up from event tokens combined by classical regular (';', '+', '*') and the operator expressing parallel composition by event interleaving ('|'). For parallel composition, there is also another

operator ∇ (*consent*), which allows to detect communication errors as explained further in this section. As an example, consider the following protocol:

(?i.open ; (?i.read + ?i.write) ; ?i.close) | ?ctrl.status**

This behavior protocol generates the set of those traces starting with the *open* method and ending with the *close* method on the *i* interface with an arbitrary sequence of repeating the *read* and *write* methods on the same interface between them. The traces may be interleaved with an arbitrary number of the *status* method calls on the *ctrl* interface. As an aside, syntactically, this protocol is composed by means of the abbreviations 8-10 mentioned below.

The Extended Behavior Protocols (EBP) are able to describe the derived behavior more precisely than BP, since method parameters and local variables (e.g. for capturing component modes [12]) become a part of the specification.

A *frame protocol* is associated with each component frame; it describes the behavior of the component as the sequence of events appearing on the component frame. Furthermore, the *architecture protocol* is a parallel composition of frame protocols of the first-level-of-nesting subcomponents of a composite component.

As an example frame protocol, consider the following specification of the LightDisplay component. In the example, the behavior specification of the component is divided into three parts: (1) type definitions (**types**), (2) local variable definitions (**vars**), and (3) behavior definition (**behavior**) containing an extended behavior protocol (the entities introduced in (1) and (2) are employed here).

```
 component LightDisplay {

  types {
   states = {LIGHT_ENABLED, LIGHT_DISABLED}
  }

  vars {
   states state = LIGHT_ENABLED
  }

  behavior {
   ?LDispCtrlEventHandlerIf.onEvent(EVENT ExpModeEnabledEvent){
    state <- LIGHT_ENABLED
   }*
   |
   ?LDispCtrlEventHandlerIf.onEvent(EVENT ExpModeDisableEvent){
    state <- LIGHT_DISABLED
   }*
  }
}
```

Extended behavior protocol is an expression defining the set of allowed sequences of events (method call requests and responses) appearing on the frame of a component. It is formed using event tokens, operators, and control statements. Event tokens are of seven forms:

1. ?interface.method(type1 arg1, type2 arg2, ...)↑
2. ?interface.method(val1, val2, ...)↑

3. ?interface.method↓
4. !interface.method(val1, val2, ...) ↑
5. !interface.method↓
6. @multisynchronization_event
7. local_var <- symbolic_value

Furthermore, the following abbreviations are defined:

8. ?interface.method(...) for ?interface.method(...)↑ ; !interface.method↓
9. !interface.method(...) for !interface.method(...)↑ ; ?interface.method↓
10. ?interface.method(...) {expr} for
    ?interface.method(...)↑ ; expr; !interface.method↓

The event tokens (1) – (10) represent primitive terms, which can be combined into expressions via the operators: ';' (sequencing), '+' (alternative), '*' (repetition), and '|' (parallel composition) as in the example above.

Each of the event tokens (1) – (7) represents an atomically occurring event (events do not overlap). The event token (1) stands for accepting a method call request (↑) and assigning the values provided by the caller to variables arg1 of type1, arg2 of type2, etc. The event token (2) is similar to (1), but it represents acceptance of a method call request only if it has been emitted with parameter the values val1, val2, etc. The event token (3) denotes an event representing acceptance of a response (↓) from a method call. The event token (4) stands for emitting a method call request providing val1, val2, ... as the parameters – the type of each parameter must correspond to the values declared at the callee (server) side. The event token (5) denotes emitting of a method call response event. The multisynchronization event (6), e.g. @x, is a blocking event taking place only if the all protocols in a parallel composition which contain @x are able to execute it at once. Then, it is executed as a single event atomically and simultaneously by all the protocols. The event token (7) denotes the event of assignment of a value to the local variable. The last two event tokens are not associated with a method call events.

Additionally, the behavior of a component may depend on the value of a local variable or a parameter via using the *switch* statement. The semantics of the *switch* statement is the same as in common programming languages. As an example, consider the following specification fragment:

```
switch (local_var) {
 value1: { protocol1 }
 value2: { protocol2 }
}
```

Finally, a *while* loop can be used for modeling finite repetitions that are done while a condition holds. The syntax of a *while* cycle follows:

```
while (local_var == value) {
 repeated-part
}
```

Another protocol operator ∇ (*consent*) is a specific parallel composition which, in addition to classical event interleavings and joining two complementary events (?x

and !x) into a single internal τ-event (visible but no more composable with another event), detects the following communication errors:

(i) *bad activity* – the inability of components to accept an emit event at the time the event is emitted;
(ii) *no activity* – deadlock.

The *divergence* composition error, detected in the original behavior protocols, is omitted here because of two reasons. (i) Its implementation is of a very high complexity, and (ii) our experience shows that it occurs very rarely.

Having all components of an application specified using extended behavior protocols, two types of component compliance relations can be verified:

1) Horizontal compliance captures the correctness of communication among protocols of first-level-of-nesting subcomponents of a composite component, i.e., the absence of errors inside of an architecture protocol.
2) Vertical compliance captures the compliance of a composite component and its subcomponents, i.e., verifies the component frame against architecture protocols.

15.2.3 Deployment View

The *SOFAnode* is a distributed runtime environment, which consist of a single *repository* and set of *deployment docks*. The repository serves as storage for both component meta-data and code and is generated from the meta-model. The deployment dock is a container inside which the components are instantiated and running.

From the implementation view, components have two parts – *functional* and *control*. While the functional part provides the business functionality of a component and is directly implemented or in the case of a composite component composed of subcomponents, the control part controls the non-functional features (like managing lifecycle and bindings, intercepting calls, etc.) of a component and is composed of so called *micro-components* [19]. Micro-components allow for a modular fully extensible way to build the control part of a component; individual extensions are applied as aspects (using AOP techniques). Also, micro-components can expose themselves via *control interfaces*.

A SOFA application has the following lifecycle. First, a developer creates new components and uploads them to the repository and/or reuses already existing components from the repository. The next stage is assembly, where subcomponents defined using frames are "refined" by corresponding architectures. The assembly process starts with the top-level component (which represents the whole application) and recursively continues till primitive architectures. Finally, the assembled application can be deployed and launched. A deployer (i.e. a person responsible for deploying) chooses and assigns components of the application to particular deployment docks in the SOFAnode and sets values for components' properties. Also at this stage, the deployer can choose which control aspects have to be applied in the application. As a next part of the deployment process, connectors are generated. The connector generator [8] takes as an input the "development-time" connectors (edges with non-functional properties and communication style assigned – see 2.1) and current assignment of components to docks and automatically generates the code of these connectors, which transparently connects the components and have all required properties.

All the deployment information (i.e. assignment of components to particular docks, connectors and other information mentioned in the paragraph above together with a reference to the application architecture) is stored in a *deployment plan*, which serves as a recipe for launching the application. Note that besides driving the application launch process, the deployment plan is also used for performance modeling.

At runtime, code of the components is automatically obtained by deployment docks from the repository. SOFA also supports versioning of components; every component can exist in multiple versions and these can be simultaneously used (if desired) even in a single application and/or deployed in a single dock.

15.2.4 Performance View

Previously described views can be used individually or in combination with other views to reason about properties such as security, quality of service, etc. Some models can be derived directly from the behavior model, but to describe quality of service needed e.g. for service level agreements, a separate performance model is needed. The reasoning about service times and related properties cannot be based simply on the output of the behavior model (such as the number of method invocations) – it has to take into account the real execution time.

In general, the performance modeling process deals with predicting common performance attributes, such as roundtrip and throughput. Here, however, we consider precise prediction of individual performance attributes to be a secondary goal. Our primary goal is using the performance modeling process to predict how the performance attributes change when the scale of the application changes. This choice of priorities is motivated by perceived practical relevance of the results – when building an application, if it turns out that its real throughput differs from the predicted one by a constant, it is often possible to adjust the real throughput simply by installing more powerful hardware – but when the difference impacts scalability, no such adjustment is possible.

Typical approaches to performance modeling of component systems [32, 33] start with the behavior model of the system. This model is transformed into one of the well known formal performance models such as Layered Queueing Networks (LQN) [31] or Stochastic Petri Nets (SPN) [13]. The performance model is then populated by the performance attributes of the primitive components and solved to predict the performance.

A problem of this approach is that the performance attributes of the primitive components may change when the primitive components are composed, simply because the composed components share resource such as processor cores or memory caches. The typical approaches tackle this problem by including the resources in the performance model, which requires detailed knowledge of shared resources and in the end leads to excessively complex models [12].

To keep the performance model simple while taking into account the effect of shared resources, we have adopted an iterative approach, in which the usage of shared resources forms a basis for calculating the performance attributes of the primitive components. The performance attributes are fed to the performance model, which predicts the performance but also provides feedback to adjust resource usage. The entire cycle is repeated until stable resource usage and performance results are obtained.

When building the resource usage model, we have to take into account that even though it is the primitive components that consume resources, the fact that resources are consumed in the first place is not caused by the individual primitive components, but by the interaction of multiple primitive components.

Most primitive components are passive and consume resources only when some of their methods are invoked and only as much as those invocations dictate. In the resource usage model, we capture this behavior by annotating invocations with *resource demand hints*, which carry information related to resource usage, and by annotating components with *resource usage rules*, which describe how to calculate resource usage from resource demand hints. The choice of the annotations depends on the resources whose usage impacts performance, and can be determined by prior experience or by benchmarking experiments with primitive components.

The allocation of resources in a specific usage scenario whose performance is to be modeled is described by the deployment model, which assigns the components to the deployment docks and connects the components using the connectors, all at the level of individual instances. In the scenario, the interaction of primitive components can be captured by event traces generated from EBP. These event traces are used to direct the propagation of resource demand hints to components whose resource usage rules should apply.

To summarize, our performance view models the performance aspects of both component interaction and resource usage. As the output of the performance modeling process, we not only predict the common performance attributes such as roundtrip and throughput, but especially predict the trends of how these attributes change when the scale of the application changes.

15.2.5 Comparison with Other Component Models

Darwin [18] is a classical component model based on an architecture description language. It has influenced almost all of other component models. It employs hierarchical components without connectors and allows for describing dynamic architectures, but in a quite limited way and without any possibility to control the dynamic reconfiguration. Also, it is mainly just a specification language (ADL), not providing any runtime environment; at the same time, it includes the option to specify behavior of primitive components by means of finite state processes (FSP) [17]. Behavior specification of composite component is constructed as a parallel composition of the behaviors of subcomponents.

Another classical ADL is Wright [4]. It also uses hierarchical component but with connectors. Even more, it allows describing behavior of components and connectors formally using a CSP-like notation. But like Darwin, it is just a specification language without any runtime support.

The contemporary industry-supported component models like EJB [30] and CORBA Component Model [21], are just flat component models (no hierarchical component composition) and focus mainly on providing a stable and mature runtime environment. An exception is Koala [23] developed by Philips – it uses a hierarchical component model also significantly inspired by Darwin. It aims at being a development platform for embedded software for TVs, set-top-boxes, etc. Koala strongly

focuses on component design and optimizations of an architecture; the Koala compiler (a tool which from ADL generates skeletons of implementations) allows for removing unused components and further architectural optimization. But the runtime options are quite limited, since the model is targeted to an embedded environment.

Fractal [7] is a component model very close to SOFA 2.0; it defines a number of abstractions and their interfaces in support of their existence at run-time (there are several Fractal implementations). Fractal defines a hierarchical component model without connectors (if the connectors are required, the Fractal specification instructs to simulate them using components which, however, leads to rather unclean architectures, mixing different levels of abstractions). Like SOFA, Fractal also separates functional and control parts of components. In addition, it supports shared components, i.e. a single component is subcomponent of several composite components. Such an approach allows for easy managing of dynamic reconfigurations but it breaks component encapsulation. An integral part of Fractal is Fractal ADL, which is an XML-based language for designing components and architectures; its usage is optional though since components can be built at runtime using Fractal API. Even though Fractal does not provide any formal behavioral descriptions of components, there are two projects adding it [1].

Like SOFA, the Fractal implementations (e.g., Julia, AOkell [26]) also provide complete environments for not only designing and modeling components, but also executing applications composed of them.

Both SOFA 2.0 and most of the Fractal implementations are built over the Java platform, being conceptually just code libraries. A different approach is taken by the ArchJava [3] and Java/A [5] component models, which modify the Java language by introducing new constructs for creating components. According to their authors, this approach prevents uncontrolled architecture modifications at runtime ("architecture erosion"). Nevertheless, the SOFA runtime (and also the Fractal implementations) prevents this completely.

15.3 Modeling the CoCoME

15.3.1 Static View

The architecture applied to the SOFA component model is based on the original architecture as defined in the CoCoME assignment. The modifications include (i) replacement of the hierarchical bus by two separated buses, (ii) interfaces and bindings necessary for the UC8 implementation, and (iii) restructuring of the Inventory component. All these modifications are justified and explained below.

As to (i), the hierarchical bus in the CashDesk was replaced by buses CashDeskLine and CashDeskBus, since this modification better reflects the orthogonal activities of CashDeskApplication and Coordinator with respect to other components in the CashDesk devices (Figure 1). This also elegantly reflects the fact that the number of instances of CashDesk and cashDeskChannel has to be the same, which is impossible to capture by the UML diagram from the CoCoME assignment (Figure 12, in the assignment). The direction of communication with buses is determined by their push semantics.

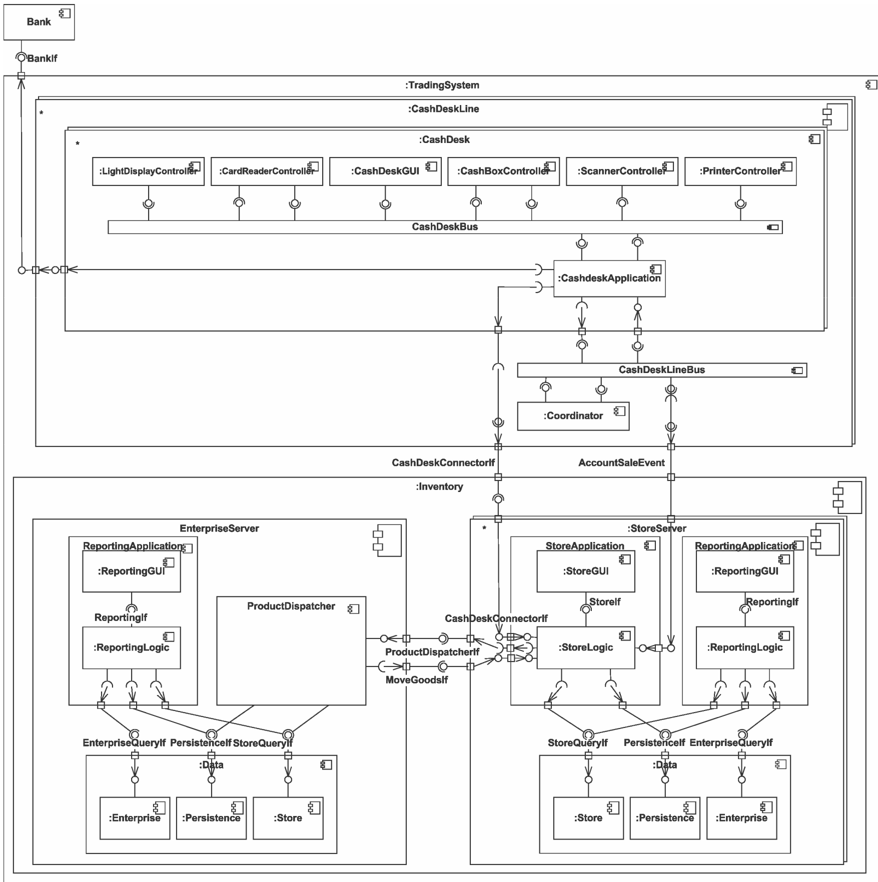


Fig. 1. SOFA architecture of the CoCoME assignment

As for (ii), this change relates to the Inventory component. The most visible modification here is the introduction of enterprise server and store server as explicit entities in support of UC8 implementation (EnterpriseServer and StoreServer components were brought up). According to the deployment diagram in the CoCoME assignment, the enterprise server is just a simplification of the store server, but UC8 requires additional functionality to capture transportation among different stores. This functionality is implemented by the EnterpriseServer component, in particular by its internal component ProductDispatcher. The rest of EnterpriseServer uses the same components as StoreServer which corresponds to the original idea of component reuse.

Finally, the modification (iii) comprises the replacement of two composite components TradingSystem::Inventory::Application and TradingSystem::Inventory::GUI by two other composite components – StoreApplication and ReportingApplication. The new componens group the primitive components in orthogonal way. The modification was motivated by the fact that Store and Reporting, the two internal components of the GUI (resp. Application) component are independent and communicate only with

the corresponding component from the Application (resp. GUI) component over the binding from the higher level of the hierarchy (Figure 2 – there is no binding among the primitive components Store and Reporting). The fact that the behavior of Store and Reporting is orthogonal is much better captured by the modification shown in Figure 1. Here, the design functionality is covered by the components StoreApplication and ReportingApplication, each of them composed of two communicating subcomponents – one reflecting GUI and the second the business logic. This way, the communication between StoreGUI and StoreLogic is hidden within the StoreApplication component. Similar argument holds for ReportingApplication. Since the Data component provides interfaces needed by both StoreApplication and ReportingApplication, it is on the same level of nesting. Finally, the Database component from the CoCoME assignment is not in the diagram, as it is accessed via JDBC, not the SOFA framework. Since this is another level of abstraction it is not modeled in the diagram.

15.3.2 Behavioral View

Although the design of SOFA 2.0 supports different behavior specification backends, the default specification language is EBP (Extended Behavior Protocols) already described in Sect. 2.2. When specifying Trading System using EBP, the basic decision is whether to base the specification on the CoCoME assignment UML specification or rather on the reference implementation. On one hand, the CoCoME assignment UML specification does not constitute an unambiguous complete specification. The sequence diagrams specify only individual runs of the application, in contrast to the overall behavioral interplay of components. Unfortunately, other UML means for behavior specification (collaboration diagrams, activity diagrams, state charts) are not used. Moreover, when it comes to UC8, the CoCoME assignment UML specification is ambiguous, as the component diagrams lack interfaces for EnterpriseServer to StoreServer communication specified in UC8. Therefore, an analysis of the reference implementation provides additional behavioral information. On the other hand, the reference implementation conflicts with UC1 and UC8, introducing additional ambiguities. In UC1, the reference implementation differs during CreditCard payment and does not allow manual BarCode entry. UC8 is implemented as a part of UC1 exploiting direct access to the shared database).

Finally, the EBP specification is based partly on the provided use cases and sequence diagrams (UC3 – UC8) and partly on the reference implementation (UC1, UC2). Additionally, the EBP specification related to UC8 is also influenced by the architectural modifications mentioned in Sect. 3.1.

The actual EBP behavior specification is created per component at a particular level of nesting, i.e. both primitive and composed components are annotated by their frame protocol. This way, the EBP specification provides a better picture than the sequence diagrams, which follow the calls ignoring the component hierarchy (i.e. orthogonal to the hierarchy). Moreover, an EBP protocol describes the overall interaction, in contrast to a single application run.

The actors (Customer, Cashier, StockManager, and Manager) were not directly modeled; however, their behavior is modeled as an autonomous activity of the UI components (CashDeskGUI, CashBoxController, CardReaderController, ScannerController, StoreGUI, and ReportingGUI).

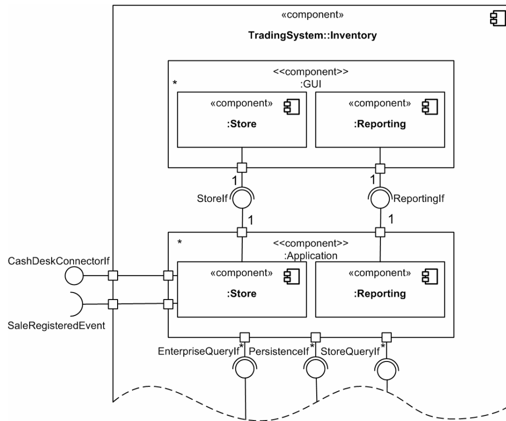


Fig. 2. Fragment of the original architecture

15.3.3 Deployment View

From the deployment point of view, the most important aspects of the CoCoME example have been the way deployment is described and managed, and the way distribution is addressed.

With regard to the description of deployment and its management, we have benefited from the full fledged deployment and runtime environment of SOFA 2.0. We have specified the implementation artifacts and the actual placement of components in a deployment plan. Subsequently, we used the SOFA 2.0 tools to launch deployment docks and start the application according to the deployment plan.

The actual distribution of the application in SOFA 2.0 has been performed via connectors. They encapsulate middleware and realize the distribution transparently to component. Since they support different communication style, we were able to model method invocations (realized by RMI in the reference architecture) as well as busses (realized by JMS in the reference architecture).

The connectors in SOFA 2.0 are automatically generated, thus a component developer is not forced to prescribe a particular middleware during design. The choice of middleware is performed at deployment time and it is done automatically by the connector generator (by constraint programming techniques). The generator also automatically creates an implementation of connectors to be used at runtime to actually address the distribution.

15.3.4 Implementation View

The implementation in fact completely follows the original reference implementation; several changes of the application architecture are described in Sect. 3.1. Connections among components are implemented via connectors (Sect. 3.3).

In SOFA 2.0, an application is represented by its model. Technically, the model is expressed a set of files containing architecture specification in SOFA 2.0 ADL. All the ADL files related to the CoCoME in SOFA application in are available on [27].

Further, the ADL files are processed by the *Cushion* development tool (Sect. 6.3), which validates them for syntactical correctness and semantical compliance and enters the model determined by them into the SOFA repository. Moreover, Cushion also controls verification of the component behavior specification.

15.3.5 Performance View

For our chosen approach, which iterates between modeling resource usage and modeling performance, it is necessary that the resource usage model provides performance attributes of primitive components to the performance model, and that the performance model provides feedback on resource usage to the resource usage model.

In CoCoME, we have decided to adopt LQN as the performance model – the feedback from LQN takes the form of queue length and processor utilization values. Another choice would be adopting SPN as the performance model – the feedback from SPN would take the form of numbers of tokens in selected places. Both LQN and SPN were reported to achieve good results when modeling enterprise information systems [5].

Our resource usage model in CoCoME is a hybrid model that uses benchmarking of primitive components under varying resource usage conditions to provide basic understanding of how resource usage influences the performance attributes. Additional benchmarking under specific resource usage conditions is coupled with modeling of resource usage through resource demand hints and resource usage rules to provide input to the performance model.

Because of the presence of the resource usage model, building the performance model does not need to go beyond the level of detail captured in the deployment plan and the behavior model. As described below, we have built the LQN model mechanically and in fact even introduced additional simplifications. We hope to eventually generate the performance model from the deployment plan and the behavior model, potentially simplifying it manually afterwards.

The first layer of LQN contains tasks generating the external requests on the system. These are the customers present at each store and coming to a cash desk at a given average rate, store managers accessing store clients for ordering stock items and enterprise managers accessing enterprise clients for generating reports.

The requests of the customers for processing their sale are handled by the cash desks. The multiplicity represents the number of cash desks in a store and the internal description of the cash desk activity (a complex probabilistic behavior description) represents the properties of a typical sale (the figures were taken from the extrafunctional properties of CoCoME). The load generated by the cash desks corresponds to UC1.

Each store also has a store client and a store server, and the system also includes an enterprise client and an enterprise server. All these tasks serve the purpose of generating disruptions of the standard load by the other use cases.

The database task represents the server part of the architecture which is actually performing the data lookups and is the place where resources are shared. The utilization of the processor running the database task represents the amount of concurrency.

The most challenging issue with using LQN was the representation of multiple stores. A trivial approach – using just the multiplicity of tasks – might not be appropriate because the multiplicity does not model the queueing in a realistic way, but it has the semantics of choosing any of the available cash desk by any customer regardless of the store he or she is in.

Our choice was to create all the separate instances of stores and cash desks inside them individually. Moreover, the LQN model is created by a generator script which can be thus seen as a meta-model. One instance of the model is shown on Figure 3.

Our benchmarking of primitive components under varying resource usage conditions has suggested that the resource whose sharing affects the performance attributes most significantly is – perhaps not surprisingly – memory of the StoreApplication and Data components. For sake of brevity, we limit ourselves to these components, which are in fact pivotal to the usage scenario most relevant to customers (UC 1).

The implementation of StoreApplication uses Hibernate, which caches data separately for each transaction. The memory usage of StoreApplication therefore grows linearly (i) with the number of transactions executing simultaneously and (ii) with the size of the data fetched in each transaction. Neither the number of simultaneous transactions nor the size of fetched data is limited - the former depends on the number of invocations coming to StoreApplication over RMI, which uses a thread pool of unlimited size, while the latter depends on the number of items in queries executed by StoreApplication, which can span all application data.

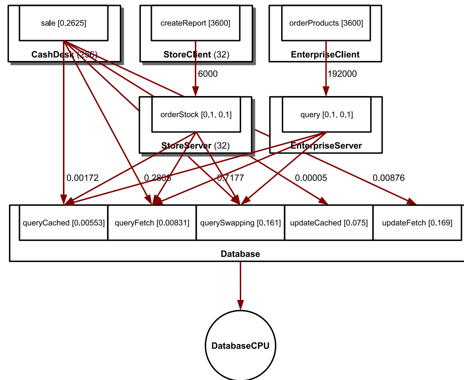


Fig. 3. Simplified LQN model for UC1

The implementation of Data uses Derby, which keeps separate context for each connection and caches pages for all transactions together. The memory usage of Data therefore grows linearly (i) with the number of connections opened simultaneously and (ii) with the size of the data cached for all transactions. The number of connections opened simultaneously is limited, as each instance of StoreApplication uses Hibernate, which has a connection pool of limited size. Similarly, the size of the data cached for all transactions is limited, as Derby uses a page cache of limited size.

The above description forms the basis for the resource usage rules of the StoreApplication and Data components. We get the complete resource usage rules by adding

the information on static memory consumption, collected through benchmarking, and the information on the number of concurrent invocations, collected from the performance model in the iteration loop:

$mem_{\text{StoreApplication}} = base_usage + sum_per_invocation_type\ (average_number_of_concurrent_invocations_of_this_type * query_size_per_invocation * memory_usage_per_unit_query_size)$

$mem_{\text{data}} = base_usage + \min(average_number_of_connections_{\text{StoreApplication}}, connection_pool_limit) * \#\text{StoreApplications} * memory_usage_per_connection + \min(page_cache_size_limit, database_size_in_items * page_cache_occupation_per_item)$

The resource demand hints used along the event trace, supply us with the query size per invocation, which is one for getProductWithStockItem queries and average shopping cart size for bookSale updates. The resource demand hints can be deduced from the complete event trace of UC1, which was generated from the behavior protocols. The key resource demand hints are the query and update sizes per invocations – one item for getProductWithStockItem queries and average shopping cart size for bookSale updates. The resource demand hints are associated individually with all relevant invocations along the event trace.

We have also determined the effect of memory usage on performance attributes by benchmarking, which shows that there are two milestones in memory usage that impact the roundtrip time as the performance attribute of choice. First, the duration of queries depends on whether the queries are satisfied from the page cache. We approximate this behavior by adjusting the mean roundtrip time of queries with the probability that a query is satisfied from the page cache:

$\text{P(query_cached)} = \min(1, max_cache_size\ /\ data_size)$
$T_{\text{query}} = T_{\text{query_cached}} * \text{P(query_cached)} + T_{\text{query_from_disk}} * (1 - \text{P(query_cached)})$

Second, the duration of all operations depends on whether the memory used by the operations has been paged out and therefore needs paging in. To approximate this behavior, we would need to know (among other things) the memory access patterns of individual operations, which would make the model too complex. We therefore simplify the model by taking the sizes of the individual components to be correlated with the sizes of their memory access patterns (we can afford to do this especially because with any but insignificant amount of swapping, the system becomes overloaded – the estimate of the very onset of swapping, rather than exact performance under swapping, is therefore of interest):

$\text{P(swapping_needed)} = occupied_physical_memory\ /\ occupied_virtual_memory$
$T_{\text{operation}} = T_{\text{operation_without_swapping}} + T_{\text{swapping_overhead}} * operation_component_size * \text{P(swapping_needed)}$

Other milestones in memory usage that could potentially impact the roundtrip time are situations when the processor cache becomes exhausted and situations when the file cache becomes exhausted. On contemporary processors, the size of the processor cache is way below the memory usage of CoCoME, diminishing the importance of the first milestone. Similarly, as the memory usage increases, the size of the file cache decreases way below the size of the page cache, diminishing the importance of the second milestone.

15.3.6 Specification of CashDeskApplication

To give a rough idea about the resulting behavior specification, frame protocols of CashDeskApplication and CashDeskBus in EBP formalism (Sect. 3.7) – being probably the two most interesting components from the behavioral point of view – are presented below (the rest of EBPs can be found in the appendix and on [27]).

As the state transitions of the CashDeskApplication component can be easily expressed as a state chart (states representing different phases during a single sale), it is very important that the behavior specification were as easy to comprehend as the state chart, while containing additional information on the method calls interplay. In this respect, the frame protocol of CashDeskApplication shown bellow satisfies this requirement and shows how EBP can cope with specifying this kind of behavior.

Since EBP can explicitly model component's internal state (via local state variables), expressing CashDeskApplication's behavior is straightforward, as can be seen on the CashDeskApplication frame protocol; here, the interface and method names are shorten for the sake of brevity. In our view, having the states explicitly expressed in the behavior specification of a component and revealing it to the outer world contributes to the clarity of the EBP specification.

```
component CashDeskApplication {
  types {
    states = { INITIALIZED, SALE_STARTED, SALE_FINISHED,
      PAYING_BY_CREDITCARD, CREDIT_CARD_SCANNED,
      PAYING_BY_CASH, PAID }
  }
  vars { states state = INITIALIZED }

  behavior {
    (
      ?CDAppEvHandler.onEvent(SaleStarted) {
        switch (state) {
          INITIALIZED:
            { state <- SALE_STARTED }
          default:
            { NULL }
        }
      } +
      ?CDAppEvHandler.onEvent(ProductBarcodeScanned) {
        switch (state) {
          SALE_STARTED: {
            !CDConnector.getProductWithStockItem;
            (
              !CDAppEvDisp.send(ProductBarcodeNotValid) +
              !CDAppEvDisp.send(RunningTotalChanged)
            )}
          default:
            { NULL }
        }
      } +
```

```
?CDAppEvHandler.onEvent(SaleFinished) {
  switch (state) {
    SALE_STARTED:
      { state <- SALE_FINISHED }
    default:
      { NULL }
    }
  } +
  ?CDAppEvHandler.onEvent(CashAmountEntered) {
    switch (state) {
      PAYING_BY_CASH: {
        NULL + (
          !CDAppEvDisp.send(ChangeAmountCalculated);
          state <- PAID
        )}
      default:
        { NULL }
    }
  } +
  ?CDAppEvHandlerIf.onEvent(CashBoxClosed) {
    switch (state) {
      PAID: {
        !CDAppEvDisp.send(SaleSuccess);
        !CashDeskEventDisp.send(AccountSale);
        !CashDeskEventDisp.send(SaleRegistered);
        state <- INITIALIZED }
      default:
        { NULL }
    }
  } +
  ?CDAppEvHandlerIf.onEvent(CreditCardScanned) {
    switch (state) {
      PAYING_BY_CREDITCARD:
        { state <- CREDIT_CARD_SCANNED }
      CREDIT_CARD_SCANNED:
        { state <- CREDIT_CARD_SCANNED }
      default:
        { NULL }
    }
  } +
  ?CDAppEvHandlerIf.onEvent(PINEntered) {
    switch (state) {
      CREDIT_CARD_SCANNED: {
        !BankIf.validateCard; (
        !CDAppEvDisp.send(InvalidCreditCard) +
        (
          !BankIf.debitCard; (
            !CDAppEvDisp.send(InvalidCreditCard); (
             NULL +
             state <- PAYING_BY_CREDITCARD) + (
               !CDAppEvDisp.send(SaleSuccess);
               !CDEvDisp.send(AccountSale);
```

```
                     !CDEvDisp.send(SaleRegistered);
                     state <- INITIALIZED
                   )
                 )
               )
             ) }
           default:
             { NULL }
         }
       }
     )* |  (
       ?CDEvHandler.onEvent(ExpressModeEnabled) {
         !CDAppEvDisp.send(ExpressModeEnabled)
       }
     )* |  (
       ?CDAppEvHandler.onEvent(ExpressModeDisabled)
     )*
   }
 }
}
```

The EBP specification above consists of (1) definition of types via enumerating their values, (2) definition of component's local variables, the `state` variable representing state of a single sale, and (3) the actual behavior. The CashDeskApplication accepts and publishes events from/through two buses, CashDeskBus and CashDeskLineBus, via its interfaces `CDAppEvDisp`, `CDAppEvHandler`, `CDEvDisp`, and `CDEvHandler`. Behavior of CashDeskApplication during a single sale (state transtion) is specified in the first and longest one of three parallel subprotocols, separated by the parallel operator "|". The other two subprotocols specify switching on and off the express mode as a reaction on the ExpressModeEnable and ExpressModeDisable events.

In more detail, the first subprotocol specifies different reactions, note the alternative operator "+", on sale-related events depending on the actual event received from the CashDeskBus (e.g. `SaleStarted` as a parameter of the `onEvent` method). The reaction then typically depends on the current state, stored in the `state` local variable (the `switch` statement), and consists of communication over the buses and/or switching transition to another state using the assignment statement "<-".

15.3.7 Specification of Component CashDeskBus

The second EBP snippet is protocol of the CashDeskBus component. The component implements a classical publisher-subscriber pattern, i.e. the message publishing method `send()` blocks until all subscribers receive the published message via the `onEvent()` method.

The key feature of the CashDeskBus component is that it synchronizes delivery of notification messages with next event acceptance, i.e. a notification message is delivered to all subscribers before another event message is accepted. To model this behavior, a technique which resembles the way a mutual exclusion is expressed in Petri nets. There is an auxiliary component Token which repeatedly (and sequentially)

produces tokens via the `Helper.token` method. The CashDeskBus accepts the token and, while still "holding it" (until return from the `Helper.token` method), CashDeskBus distributes the corresponding notification message to all subscribers. This way, processing of other messages is blocked until the token is released. As a technicality, the CashDeskBus protocol can always accept the token when there are no event messages to accept.

```
component Token {
  behavior {
    !Helper.token*
  }
}

component CashDeskBus {
  behavior {
    ?CashBoxControllerEvDisp.send(CashAmountEntered) {
      ?Helper.token {
        !CDAppEvHandler.onEvent(CashAmountEntered) |
        !PrinterControllerEvHandler.onEvent(CashAmountEntered) |
        !CDGUIEvHandler.onEvent(CashAmountEntered)
      }
    }*
    |
    ?CashBoxControllerEvDisp.send(CashBoxClosed) {
      ?Helper.token {
        !CDAppEvHandler.onEvent(CashBoxClosed) |
        !PrinterControllerEvHandler.onEvent(CashBoxClosed)
      }
    }*
    |
    ?CardReaderControllerEvDispatcher.send(CreditCardScanned) {
      ?Helper.token {
        !CDAppEvHandler.onEvent(CreditCardScanned)
      }
    }*
    |
    . . .
    |
    ?Helper.token*
  }
}
```

It is fair to say that the presented technique takes advantage of the fact that notification messages are handled synchronously with event acceptance. In contrast, if the handling was asynchronous (i.e. message buffering was necessary), the modeling would be much harder (and obviously impossible for unbounded buffer).

15.4 Transformations

The modeling process of SOFA can be viewed as a series of transformations. The most notable transformations include the deployment process, which represents a

transformation from the component model into the deployment plan, and the performance modeling process, which represents a transformation from the deployment plan and the behavior model into the resource usage model, and a transformation from the component model and the behavior model into the performance model.

The transformation from the component model into the deployment plan is mostly done manually, with some parts which can be automated. The decisions are particularly based on information like geographical properties of the deployment instance, hardware configuration, which is available, etc. Partial automation can be achieved by defining well-known deployment patterns in the component model (e.g. if some components are connected with a connector representing a bus, they should be deployed in the same deployment dock). This transformation is one-way.

The creation of the performance and resource models starts with a decision what use cases of the modeled system are actually interesting. This heavily depends on the SLA and QoS expectations of the real implementation (e.g. the real time duration of some use cases might be of very low importance for the service customers). Also some components from the component model can be manually eliminated or merged for the purpose of performance and resource modeling, when it is clear that their performance impact is either negligible (which is assumed) or fatal (which would make the performance model instantly useless).

After this selection and elimination, the performance model is derived automatically from the component model (components becoming tasks of the performance model) and behavior model (the behavior describes both the methods of the tasks and the kind and probability of their interaction). The decisions and transformation can be done iteratively when the performance model is too complex (difficult to solve) or oversimplified (captures only trivial interaction).

After the manual decisions on the importance have been made, the transformation from the deployment plan and behavior model into resource usage model can be also automatic. Basically the assignment of the tasks to processing units directly reflects the assignment of components to deployment docks. The behavior description is used to distinguish different types of processing units – tasks which act as pure clients (generating load) use unlimited processing units (with unbounded parallelism), whilst other tasks use processing units with a given multiplicity (which is derived from the number of instances of the given component in the deployment plan). This transformation is usually one-way.

15.5 Analysis

15.5.1 Compliance Both Vertical and Horizontal

Both the vertical and the horizontal compliance verification is done using a tool chain. The parts of the chain are EBP2PR (Extended Behavior Protocols to Promela Translator) and Spin [29] (described in detail in Sect. 6). Basically, the EBP2PR tool translates the specification in EBP into the Promela language. After that, this output is verified by the Spin model checker. Here, each EBP is modeled as a Promela process and the absence of communication errors is transformed (by a technical trick) into absence of deadlocks.

The tool chain was applied to the frame protocols associated with all of the components in the CoCoME application (Figure 1). The data on the time the tool chain spent on verification in this process is available at the project web site [27]. For illustration, Tab.1 below provides data on the time spent on the verification of the horizontal compliance of the CashDeskApplication and CashDeskBus frame protocols. In the first column, the size of the state space generated by the frame protocols' composition is shown. The other columns show the time spent in particular parts of the verification process – *EBP2PR* is the time required for transformation of the specification in EBP into Promela, *Verification* represents the time Spin spent on the verification process. As the composition of the CashDeskApplication and CashDeskBus frame protocols generates the most demanding state space in the context of the CoCoME application, the total time spent on behavior verification of the other parts of the application was shorter.

This particular horizontal compliance verification was performed on the following hardware and software configuration: PC 2x Intel Core2 Duo (dual core) processor with 4 MiB L2 cache and 4 GiB operational memory running the Linux (kernel version 2.6.19), and Spin version 4.2.9.

Table 1. The result of vertical compliance verification of the CashDesk component

# of states	EBP2PR [s]	Verification [s]	Total time [s]
3 335 950	41.5	46.1	95,6

Vertical compliance evaluation starts with inversion of the frame protocol of the composite component (the emitting and accepting events are swapped, e.g. !i.a↑ is replaced by ?i.a↑ and vice versa). Via the consent operator, the actual vertical compliance is verified by composing the inverted frame protocol with the architecture protocol capturing the composed behavior of the subcomponents; as an aside, this way vertical compliance is converted into horizontal compliance [2]. The complete data on verification are listed on [27].

15.5.2 Verification of Code against Frame Protocols

Checking of horizontal and vertical compliance makes sense only if implementation of each primitive component in a particular architecture obeys (is compliant with) the component's frame protocol. In order to check this property, we apply the technique of code model checking to individual primitive components. However, an isolated primitive component cannot be checked by a typical code model checker (like Java PathFinder), since such a component does not form a complete Java program required by a model checker – an environment that forms a complete program together with the component is needed. The environment is constructed from its behavior specification (component's inverted frame protocol) in such a way that forces the model checker to verify all reasonable control-flow paths in the component's implementation.

Although the well-known problem of state explosion is partially mitigated by application of model checking to a single component (having a smaller state space than the whole application), still the technique has very high time and space complexity

since it exhaustively verifies all possible runs of the code; for highly parallel components, model checking may even be infeasible. In such a case, we recommend to use run-time checking to check the property of obeying a frame protocol at least partially (i.e. not exhaustively).

15.5.3 Run Time Checking against Code

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 run-time checking can verify only a single trace recorded during a single run of an application and is therefore not exhaustive, an error (violation of a frame protocol) may not be detected for many runs of the application; in this respect, the technique of run-time checking is similar to testing.

15.5.4 Performance Analysis

The basis for our performance analysis is the CoCoME reference implementation, which was benchmarked to provide input for the resource usage and performance models. We have run several types of benchmarks on the implementation to identify those parts of code that have significant performance impact and resource consumption. Our further reasoning about scalability was based on these benchmarks (for detailed benchmark results, see [27]).

We have focused our analysis on a simple but practical configuration for both benchmarking and modeling performance, which is a repeated execution of Use Case 1. While quite simple, it represents a typical load on the system generated in a production environment. Other use cases serve the purpose of generating disruptions (the Reporting components in both Enterprise and Store can submit queries that can trash the database cache occasionally, but it would not make sense to consider a continuous reporting use case).

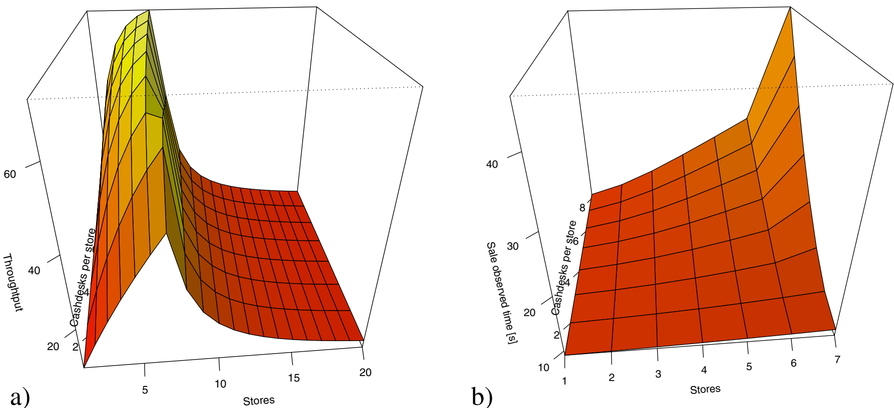


Fig. 4. Calculated a) throughput and b) average service time of Use Case 1

It is our assumption that a cash desk by itself cannot be overloaded (the throughput is strictly limited by the cashier interacting with the cash desk – the rate of his or her requests is significantly below any reasonable value which might be considered a high load) and its resource consumption is constant. We can therefore focus on the performance of the inventory components under the aggregated load by requests generated by several cash desks in several stores.

The results of our modeling are on Figure 4a, which shows the dependency of the overall enterprise throughput in terms of item queries processed per second on the number of stores in the enterprise, and on Figure 4b, which shows the dependency of the time to perform a single sale on the number of stores in the enterprise. We see that when the infrastructure is saturated or overloaded by requests from the cash desks, insufficient performance is noticeable even on the cash desk side, since the cash desks are unable to get answers to item queries within a reasonable time (however, the operations performed in the cash desks still take the same constant time, independent on the state of other components).

To evaluate the results of our modeling, we have measured the performance of the modeled scenario on the prototype implementation. The benchmark, as well as the benchmark experiments used to obtain the average durations of the atomic actions, have used an Intel Pentium 4 Xeon 2.2 GHz machine with 512 MB RAM running Fedora Core 6 and the database cache of 10000 pages for the server machine, and a dual Intel Core 2 Quad Xeon 1.8 GHz machine with 8 GB RAM running Fedora Core 6 for the client machine. The relatively low amount of memory on the server machine was used deliberately to allow the manifestation of swapping with a reasonably small number of cash desks and stores.

The results on Figure 5 suggest that our approach is reasonably precise in predicting the item query throughput and single sale roundtrip values. We have also predicted getting within 10% of the maximum throughput around 2 stores, when the measurement shows this happening around 3 stores, and the degradation in performance due to swapping around 8 stores, when the measurement shows this happening

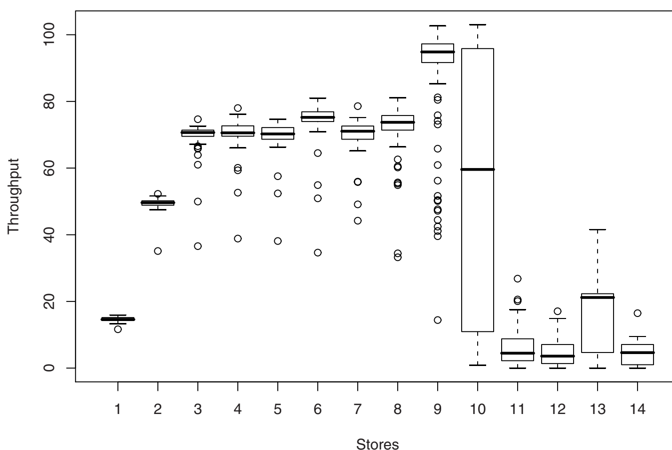


Fig. 5. Benchmarked throughput of Use Case 1

around 10 stores. Some difference in these results can be explained by our inability to determine the precise memory requirements of the individual components, something that is difficult to do with current tools.

15.6 Tools and Results

15.6.1 BP Checker

The EBP2PR tool is used to transform the specification in EBP into Promela (input language of the Spin model checker). On its input, this tool accepts the frame protocol of a composite component and the architecture protocol capturing behavior of its subcomponents. The output is a corresponding Promela model. This section describes the transformation process in detail.

As the semantics of the Promela language and EBP differ in many aspects (e.g. in handling nondeterminism), a straightforward translation of EBP into Promela is not possible. Therefore, the following sequence of transformations is applied: Each EBP on its input is first transformed into a corresponding nondeterministic finite automaton (NFA) (local variables and method parameters are not considered). This NFA is transformed into an equivalent deterministic finite automaton (DFA) that is minimized. Finally, Promela code modeling the minimal DFA is generated and the local variables and method parameters (ignored in the preceding steps) are added as variables of a Promela enumeration type.

As to communication errors (bad activity and no activity), bad activity is modeled as Promela deadlock in the following way: Each time a Promela process (recall that each EBP is modeled as a Promela process) emits an event *e*, it first acquires a lock (shared variable) and then emits *e*. If no other process is ready to accept *e*, a deadlock occurs since, due to the lock, no other event may be either accepted or emitted before accepting *e*. Technically, an event is represented by a dedicated shared variable. No activity is modeled naturally as a deadlock when the lock is not acquired.

If a communication error is discovered during the verification process, the verification is stopped, and the Promela error trace is used to guide the Spin simulation (i.e., only the error trace is executed) to generate the corresponding error trace in the EBP.

15.6.2 Modified JPF and BP Checker

As indicated in Sect. 5.2, we use the technique of code model checking to verify that a primitive component obeys its frame protocol. Since the SOFA component model is strictly Java-based, we use the Java PathFinder (JPF) tool, which is a highly extensible model checker for Java byte code, in combination with the old version of behavior protocol checker (BPC) that supports only old behavior protocols – there is no Java-based checker for new behavior protocols.

Technically, JPF and BPC cooperate while traversing their own state spaces and since both checkers work at different levels of abstraction, we defined a mapping from the JPF state space into the state space of BPC to make such cooperation possible. The mapping is implemented via a JPF plug-in that traces traversal of JPF state space and drives BPC in traversal of its own state space; for each executed byte code

instruction related to method calls on the component's external interfaces, JPF plug-in tells BPC what transition it has to take in its state space. If such a transition does not exist in the BPC state space, a violation of a frame protocol is detected and reported. Environment for a primitive component is constructed in a semi-automated way: (i) the EnvGen tool generates stub implementations of component's required interfaces and skeleton of a driver program, and (ii) the driver program is manually modified so that the environment behaves correctly (with respect to data-flow and component's state).. Translation of EBP into the old behavior protocols (BP) is performed in an automated way. EBP-specific features are translated into constructs supported by the old BP with the possible loss of information (some behaviors may be added).

Run-time checker is also based on the old Java-based version of BPC. However, in this case, the state space traversal in BPC is driven by run-time notification performed by micro-components that intercept method calls on the component's external interfaces; moreover, no coordination of backtracking is needed since only a single run of the application is checked.

By application of our tool to the implementation of CashDeskApplication and its 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 CashDeskApplication against the frame protocol based on the reference implementation (i.e. with no UC1-related inconsistency) has not reported any error and took 14 seconds.

Nevertheless, correctness of 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 does not know whether it can trigger payment by credit card (which is being forbidden in the express mode). Moreover, we added the CashAmountCompleted event into the frame protocol and implementation of CashDeskApplication in order 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.

15.6.3 Cushion Development Support Tool

As mentioned in Sect. 3.4, Cushion (available as a part of the SOFA 2.0 implementation [28]) is a tool supporting development of SOFA 2.0 applications. It allows adding new ADL specification of interfaces, frames, and architectures to the repository, retrieving already existing specifications from the repository for further development (multiple versions are supported), etc. Also, it controls behavior validation of components. Further, Cushion serves as a build tool; it supports compilation of the code of primitive components, preparation of deployment plan, etc.

By its design, Cushion is fully extensible. In fact, by itself, Cushion provides just a core; all the above described functionality is implemented as extensions.

In addition to Cushion (command-line tool by nature), SOFA 2.0 application can be developed in a graphical development environment (designed as an Eclipse IDE plugin); unfortunately is has been still under the development at the time of writing, not being ready for a no-trivial use such as CoCoME.

15.7 Summary

In this paper, we presented our experience and results of using the SOFA 2.0 component model for specification and implementation of components of the CoCoME application. The architecture was modified in order to improve clarity of the design – in particular, the hierarchical bus was replaced by two separated buses and the Inventory component was restructured. All these modification, as well as the motivation for them, were described in Sect. 3.1. To demonstrate the modeling power of SOFA 2.0 in terms of behavior specification at the ADL level, Extended Behavior Protocols described in Sect. 2.2 were used. They further extend the expressive power of Behavior protocols via introducing local variables of enumeration type, method parameters of these types, and add simple control structures (loop, switch, if). By taking advantage of the EBP specification, the behavior compliance of communicating components was verified (Sect. 5.1). Furthermore, compliance of a part of the implementation (the CashDeskApplication component) against its EBP specification was also verified. This illustrated that the question whether the implementation fulfills the specification in EBP can be answered in an automated way at an early stage of the application development.

The deployment of the CoCoME application was done by the runtime and deployment environment of SOFA 2.0. Thanks to the presence of connectors in SOFA 2.0, RMI for remote method invocation and JMS for modeling buses could be used for implementation of the communication among components. For modeling the performance and resource usage, Layered Queueing Networks coupled with an extra resource usage model were used. Specifically, the performance of a particular enterprise under a varying spectrum of load conditions was modeled. By comparing the results with measurements on the prototype implementation, our model was shown to successfully predict changes in performance due to resource sharing and resource exhaustion.

As a future work, we plan to integrate the verification tool chain into the SOFA 2.0 framework and to predict performance of a component application by using a combination of formal methods and benchmarking.

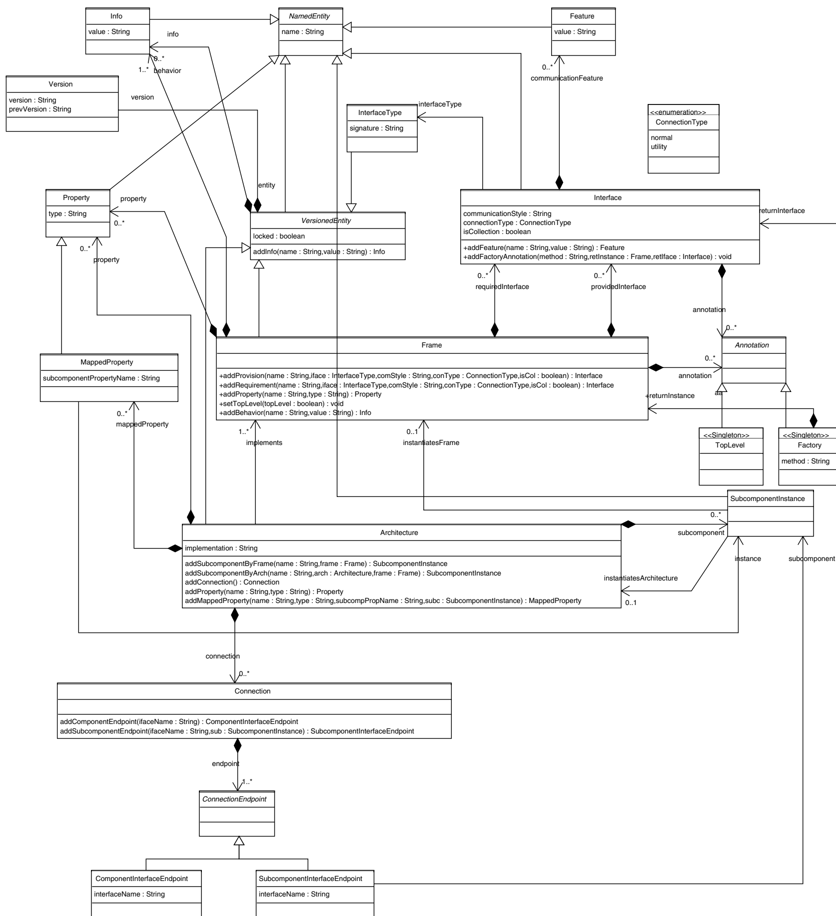
References

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. Adamek, J., Plasil, F.: Component Composition Errors and Update Atomicity: Static Analysis. Journal of Software Maintenance and Evolution: Research and Practice 17(5) (September 2005)
3. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting Software Architecture to Implementation. In: Proc. of ICSE 2002, Orlando, USA (May 2002)
4. Allen, R.J.: A Formal Approach to Software Architecture, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University (May 1997)
5. Balsamo, S., DiMarco, A., Inverardi, P., Simeoni, M.: Model-based Performance Prediction in Software Development. IEEE Transactions on Software Engineering 30(5) (May 2004)

6. Baumeister, H., Hacklinger, F., Hennicker, R., Knapp, A., Wirsing, M.: A Component Model for Architectural Programming. In: ENTCS, Vol. 160 (August 2006)
7. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: The Fractal Component Model and Its Support in Java. Software Practice and Experience, Special issue on Experiences with Auto-adaptive and Reconfigurable Systems 36(11-12) (2006)
8. Bures, T.: Generating Connectors for Homogeneous and Heterogeneous Deployment, Ph.D. Thesis, Dep. of SW Engineering, Charles University, Prague (September 2006)
9. Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In: Proc. of SERA 2006, Seattle, USA, August 2006, IEEE CS, Los Alamitos (2006)
10. Eclipse Modeling Framwork, `http://www.eclipse.org/emf/`
11. Enterprise Java Beans specification, version 2.1, Sun Microsystems (November 2003)
12. Ghosh, A., Givargis, T.: Cache optimalization for embedded processor cores: An analytical approach. ACM Transactions on Design Automation of Electronic Systems (TODAES), 9(4) (October 2004)
13. Haas, P.J.: Stochastic Petri Nets: Modelling. Springer, New York (2002)
14. Hirsch, D., Kramer, J., Magee, J., Uchitel, S.: Modes for Software Architectures. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344. Springer, Heidelberg (2006)
15. Hnetynka, P., Plasil, F.: Dynamic Reconfiguration and Access to Services in Hierarchical Component Models. 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)
16. Kofron, J.: Extending Behavior Protocols With Data and Multisynchronization, Tech. Report No. 2006/10, Dep. of SW Engineering, Charles University in Prague (October 2006)
17. Magee, J., Kramer, J.: Concurrency State Models & Java Programs. Wiley, Chichester (1999)
18. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: Proc. of FSE 2004, San Francisco, USA (October 1996)
19. Mencl, V., Bures, T.: Microcomponent-Based Component Controllers: A Foundation for Component Aspects. In: Proc. of APSEC 2005, Taipei, Taiwan, December 2005, IEEE CS, Los Alamitos (2005)
20. OMG: UML Profile for Schedulability, Performance and Time, OMG document formal/2005-01-02 (January 2005)
21. OMG: CORBA Components, v 3.0, OMG document formal/02-06-65 (June 2002)
22. OMG: MDA Guide, v. 1.0.1, OMG document omg/03-06-01 (Jun 2003)
23. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala Component Model for Consumer Electronics Software. IEEE Computer 33(3) (March 2000)
24. Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In: Proc. of ICCDS 1998, Annapolis, USA. IEEE CS Press, Los Alamitos (May 1998)
25. Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components. IEEE Transactions on Software Engineering 28(11) (November 2002)
26. 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)
27. SOFA CoCoME (2007), http://dsrg.mff.cuni.cz/cocome/sofa
28. SOFA 2.0 implementation, http://sofa.objectweb.org/
29. Spin, http://www.spinroot.com/

30. Sun Microsystems, JSR 220: Enterprise JavaBeansTM, Version 3.0
31. Woodside, C.M., Neron, E., Ho, E.D.S., Mondoux, B.: An Active-Server Model for the Performance of Parallel Programs Written Using Rendezvous. Journal of Systems and Software (1986)
32. Xu, J., Oufimtsev, A., Woddside, C.M., Murphy, L.: Performance Modeling and Prediction of Enterprise JavaBeans with Layered Queueing Network Templates. ACM SIGSOFT Software Engineering Notes 31(2) (March 2006)
33. Wu, X.P., Woodside, C.M.: Performance Modeling from Software Components. In: Proc. of WOSP 2004, Redwood Shores, USA (January 2004)

Appendix A: SOFA 2.0 Meta-model



Appendix B: Specification of Selected Components

The full version of CoCoME EBP specification is available at the SOFA CoCoME project web site [27]. As the space allotted for this chapter is limited, this appendix contains EBP specification only of selected, “representative” components CashDesk-Line and CardReader – those which are discussed elsewhere in the text.

```
component CashDeskLine {
  behavior {
    (
      !CDConnector1.getProductWithStockItem*;
      ( !BankIf1.validateCard*;
        !BankIf1.debitCard
      )*;
      !CDLineEvDisp1.send(AccountSale);
      !CDLineEvDisp2.send(SaleRegistered)
    )*
    |
    (
      !CDConnector2.getProductWithStockItem*;
      (
        !BankIf2.validateCard*;
        !BankIf2.debitCard
      )*;
      !CDLineEvDisp2.send(AccountSale);
      !CDLineEvDisp2.send(SaleRegistered)
    )
  }
}

component CardReader {
  types {
    states = {CARD_READER_ENABLED, CARD_READER_DISABLED}
  }
  vars {
    states state = CARD_READER_ENABLED
  }
  behavior {
    (
      !CardReaderControllerEvDispatcher.send(PINEntered) +
      !CardReaderControllerEvDispatcher.send(CreditCardScanned)
    )*
    |
    ?CardReaderController.onEvent(ExpressModeDisabled){
      state <- CARD_READER_DISABLED
    }*
    |
    ?CardReaderController.onEvent(ExpressModeEnabled) {
      state <- CARD_READER_ENABLED
    }*
  }
}
```