A Case Study in Parallel Verification of Component-Based Systems

N. Beneš, I. Černá, J. Sochor, P. Vařeková and B. Zimmerova[1,2]

Faculty of Informatics, Masaryk University
Brno, Czech Republic

Abstract

In large component-based systems, the applicability of formal verification techniques to check interaction correctness among components is becoming challenging due to the concurrency of a large number of components. In our approach, we employ parallel LTL-like model checking to handle the size of the model. We present the results of the actual application of the technique to the verification of a complex model of a real system created within the *CoCoME Modelling Contest* [18]. In this case study, we check the validity of the model and the correctness of the system via checking various temporal properties. We concentrate on the component-specific properties, like local deadlocks of components, and correctness of given use-case scenarios.

Keywords: Component-based systems, formal verification, parallel model checking.

1 Introduction

During the last decade, software industry has seriously started to take advantage of component-based software development as an alternative to existing software development techniques. Component-based development proposes to assemble software systems from reusable components, possibly in a hierarchical manner. This helps to significantly reduce development costs, but brings the issue of correctness of such system, especially if components are delivered by different vendors.

In this paper, we present a practical application of verification techniques to a large component-based system designed within the *CoCoME Modelling Contest* [15]. In the contest, a number of teams were asked to create a detailed model of a common component-based system to make their modelling approaches comparable. While in [18], we present our model of the CoCoME system, this paper

[1] Email: {xbenes3,cerna,sochor,xvareko1,zimmerova}@fi.muni.cz

[2] The work has been supported by the grants No. 1ET400300504 and No. 1ET408050503.

complements the work by verifying the model. In verification, we concentrate on properties of the final model like correctness of given use-case scenarios, local deadlocks of components, and response properties. Besides these we demonstrate how the verification helped us to check the validity of the model during modelling.

As a modelling language for component-based systems we use *Component-Interaction automata* (or *CI automata* for short) [6,8] which allow very precise and detailed description of communication among system components. System properties are specified in an extended version of the action-based linear time logic LTL, called *CI-LTL*. For verification itself we use the automata-based model checking algorithms implemented in the parallel model checking tool *DiVinE* [4,10]. We advocate the choice of a parallel tool by a tremendous size of the model given by concurrency of components in the system.

A short description of the CoCoME Modelling Contest is given in Section 2 followed by an outline of CI automata modelling language and CI-LTL logic in Section 3. Section 4 introduces the model we have created within the contest, and Section 5 lists required properties and use-case scenarios including their verification. Finally, Section 6 discusses the results and experience gained during the verification.

2 CoCoME Modelling Contest

In order to leverage component-based system design to build correct and dependable component-based systems, researchers have developed various formal and semi-formal component models which concentrate on different yet related aspects of component modelling [13,7,5,12,2,11]. The main goal of the *CoCoME (Common Component Modelling Example) Modelling Contest* [15] was to evaluate and compare the practical appliance of existing component modelling approaches and techniques on a common modelling example, which was designed to comprise a large number of various aspects and modelling issues that can be identified in different types of component-based systems.

The modelling example, called *Trading System*, serves to handle sales in a chain of supermarkets. Its functionality includes the interaction with the cashier at the cash desk, like product scanning, price lookup, cash/card payment, and bill printing, as well as accounting the sale at the inventory, or determining whether an express cash desk is needed in the store. Furthermore, the Trading System deals with ordering goods from wholesalers, and generating various kinds of reports.

The Trading System was implemented as a Java application where components correspond to packages in the source code. The Java source code (125 Java classes in total) served as a detailed specification of the system. The component structure of the application up to depth four is depicted in Figure 1. The system is an open system, designed to interact with external components representing users of the system (cashiers and managers) and a bank application.

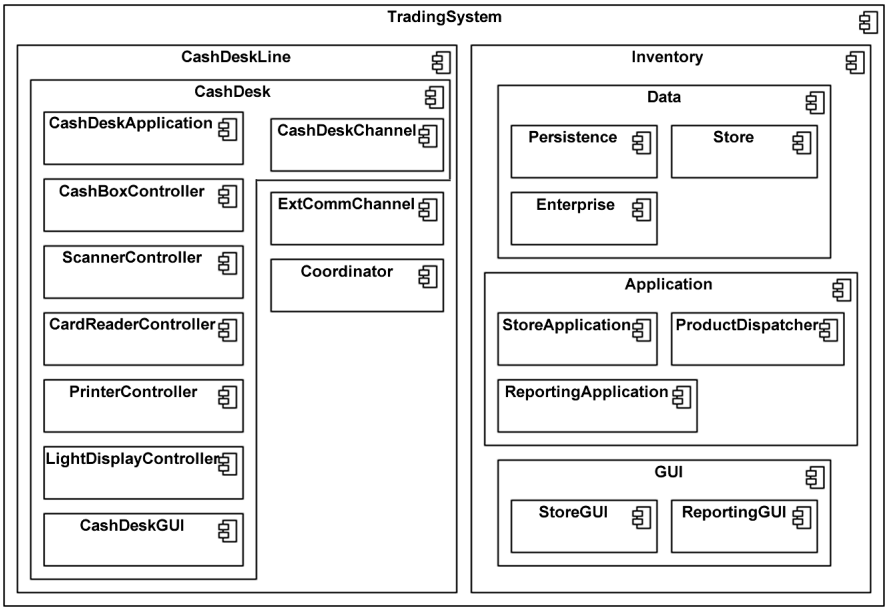

Fig. 1. CoCoME Trading System overview

3 Foundations

3.1 Modelling language

To model behaviour of component-based systems we use the *CI automata* language [6,8]. The language models each component as a labelled transition system with structured labels and a hierarchy of component names. The transition label articulates which components communicate on an action, and the hierarchy of names represents the architectural structure of the component.

A *CI automaton* is a 5-tuple $\mathcal{C} = (Q, Act, \delta, I, H)$ where Q is a finite set of states, Act is a finite set of *actions*, $\Sigma = ((S_H \cup \{-\}) \times Act \times (S_H \cup \{-\})) \setminus (\{-\} \times Act \times \{-\})$ is a set of *labels*, $\delta \subseteq Q \times \Sigma \times Q$ is a finite set of *labelled transitions*, $I \subseteq Q$ is a nonempty set of *initial states*, and H is a structured tuple representing a hierarchy of component names where the set of component names is denoted S_H.

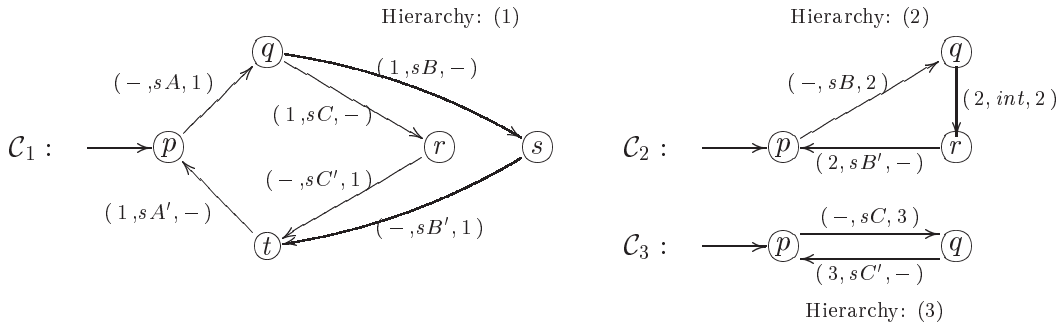

Fig. 2. Three examples of CI automata

The labels have semantics of input, output, or internal, based on their structure. In the triple, the middle item represents an action name, the first item represents

a name of the component that outputs the action, and the third item represents a name of the component that inputs the action. Examples of three CI automata are in Figure 2. Each of them represents a model of behaviour of a basic component. For example, $(-, sA, 1)$ in $\mathcal{C}_1$ signifies that the component with numerical name 1 inputs an action sA (a request for a service `sA()`), and $(1, sA', -)$ in $\mathcal{C}_1$ signifies that the component 1 outputs an action sA' (a response for the service `sA()`).

To compose components into a higher-level component a composition operator is defined. Automata can be composed together using a parametrizable composition operator $\otimes^{\mathcal{F}}$, which composes a given finite set of automata with respect to the set of *feasible labels* $\mathcal{F}$. Given a set of labels $\mathcal{F}$, the operator composes the set of CI automata into a product automaton allowing only those transitions from the product that have labels from $\mathcal{F}$. In the product, the components cooperate either by interleaving of their original transitions, or by simultaneous execution of two complementary transitions (with labels $(n_1, a, -)$, $(-, a, n_2)$) which results into a new internal transition (with label (n_1, a, n_2)). An example of a composite automaton is in Figure 3. A wider range of composition operators is defined in [6,8].

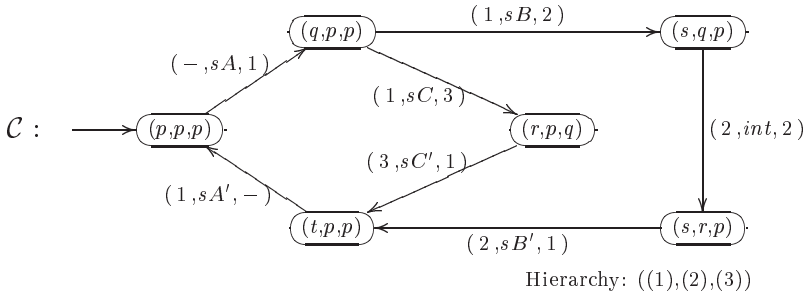

Fig. 3. A composite CI automaton $\mathcal{C} = \otimes^{\mathcal{F}}\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ where $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ are in Fig. 2, and $\mathcal{F} = \{(-, sA, 1), (1, sA', -), (1, sB, 2), (2, sB', 1), (2, int, 2), (1, sC, 3), (3, sC', 1)\}$

3.2 Temporal logic

For property specification, we use a slightly modified version of the linear temporal logic LTL [14] which we refer to as *CI-LTL*. CI-LTL is designed to express properties about occurring component interaction (i.e. labels in automata), but also about possible component interaction (i.e. label enabledness).

Syntax. For a given set of labels, formulas of CI-LTL are defined as

(1) $\mathcal{P}(l)$ and $\mathcal{E}(l)$ are formulas, where l is a label.

(2) If Φ and Ψ are formulas, then also $\Phi \wedge \Psi$, $\neg\, \Phi$, $\mathcal{X}\, \Phi$ and $\Phi\ \mathcal{U}\ \Psi$ are formulas.

(3) Every formula can be obtained by a finite number of applications of steps (1) and (2).

Other operators can be defined as shortcuts: $\Phi \vee \Psi \equiv \neg\, (\neg\, \Phi \wedge \neg\, \Psi)$, $\Phi \Rightarrow \Psi \equiv \neg\, (\Phi \wedge \neg\, \Psi)$, $\mathcal{F}\, \Phi \equiv \text{true}\ \mathcal{U}\ \Phi$ (Future), $\mathcal{G}\, \Phi \equiv \neg\, \mathcal{F} \neg\, \Phi$ (Globally).

Semantics. Let $\mathcal{C} = (Q, Act, \delta, I, H)$ be a CI automaton. We define a *run* of $\mathcal{C}$ as

an infinite sequence $\sigma = q_0, l_0, q_1, l_1, q_2, \ldots$, where $q_i \in Q$, and $\forall i \,.\, (q_i, l_i, q_{i+1}) \in \delta$. We further define:

- $\sigma(i) = q_i$ (i-th state of σ)
- $\sigma^i = q_i, l_i, q_{i+1}, l_{i+1}, q_{i+2}, \ldots$ (i-th sub-run of σ)
- $\mathcal{L}(\sigma, i) = l_i$ (i-th label of σ)

CI formulas are interpreted over runs and the satisfaction relation $\models$ is defined as

$$
\begin{array}{ll}
\sigma \models \mathcal{E}(l) & \iff \exists q \,.\, \sigma(0) \xrightarrow{l} q \\
\sigma \models \mathcal{P}(l) & \iff \mathcal{L}(\sigma, 0) = l \\
\sigma \models \Phi \wedge \Psi & \iff \sigma \models \Phi \text{ and } \sigma \models \Psi \\
\sigma \models \neg\, \Phi & \iff \sigma \not\models \Phi \\
\sigma \models \mathcal{X}\, \Phi & \iff \sigma^1 \models \Phi \\
\sigma \models \Phi\ \mathcal{U}\ \Psi & \iff \exists j \in \mathbb{N}_0 \,.\, \sigma^j \models \Psi \text{ and } \forall k \in \mathbb{N}_0, k < j \,.\, \sigma^k \models \Phi
\end{array}
$$

Informally, formula $\mathcal{E}(l)$ is true in all states of the system where the interaction represented by the label l can possibly happen. Formula $\mathcal{P}(l)$ is true for a run whenever the interaction represented by the label l is actually happening as the very first transition of the run.

3.3 Model checking and verification tool

For model checking CI-LTL properties, the automata-based algorithm [17] is slightly modified in the way how a formula is translated into a Büchi automaton. Automaton has a special alphabet formed by doubles (set of labels, label). The items correspond to the two operators $\mathcal{E}(l)$ and $\mathcal{P}(l)$. Apart from that, the model checking algorithm remains the same as in the case of standard LTL (accepting cycle detection) and therefore it has the same complexity.

The tool DiVinE, which we use for the verification, provides several LTL model checking algorithms. In our case study, the algorithm OWCTY [3] is employed. The verifications presented in this paper have been performed on a cluster of ten 2.60 GHz Intel Pentium 4 Linux workstations with 3800 MB of RAM, interconnected with a fast 100Mbps Ethernet and using Message Passing Interface (MPI) library. The chosen number of computers is explained in Section 6.

4 Model of the Trading System

Within the *CoCoME Modelling Contest* [15], we have created a detailed model of the Trading System in terms of component interaction using CI automata [18]. The model in a textual notation is available at [16]. The model consists of 140 primitive automata (59 in the CashDeskLine part, and 81 in the Inventory part), composed hierarchically into 34 composite automata up to 6 levels of depth. The Trading System model is complemented by several models of cashiers and managers, who interact with the system, and specify various usage profiles under which properties of the system are checked.

From the number of usage scenarios we have experimented with, in this case study, we employ a usage scenario describing one sale assisted by a cashier. This scenario represents the most complex usage profile described in [15], and it is connected to a large number of component-specific properties that can be checked on the behaviour of the system that is implied by the scenario. In the scenario, the cashier first starts the sale, then scans items (in a loop), finishes the sale and receives the payment. It can select cash or card payment, where the cash payment is followed by entering received amount and returning change, and the card payment with scanning the card and entering PIN.

Besides the users, the system interacts with a bank application to exchange information during card payments. For the bank we suppose that it can perform any correct scenario, i.e. the bank is anytime able to receive requests and for each request it returns a response. We simulate this situation by leaving the communication with the bank open.

Size of the model. As mentioned above, the Trading System model is composed out of 140 primitive automata hierarchically assembled into 34 composite automata. Even if the size (number of states) of individual primitive automata is moderate, the size of the complete state space is immense due to the concurrency in component behaviour. An attempt to generate the complete state space on a cluster of twenty computers finished with 322 millions of states demanding for 60 GB of memory in total. However, for the verification of the model, the key properties are dependent on usage scenarios performed by the user of the system. For the verification of the model under the given usage scenario, the model is composed with an automaton representing the user. This restricts the possible behaviours and decreases the state space. The size of the model with the cashier mentioned earlier is 749 340 reachable states and 3 181 473 reachable transitions.

5 Verification of the model

In this section, we discuss some of the properties that were checked on the model, and present verification results. We concentrate on the properties that are specific to component-based systems and emerged from the requirements on the Trading System. In the CoCoME Modelling Contest, a number of requirements were specified in terms of use-case scenarios. Use-case scenarios define a behaviour of the system in response to a given usage profile. Verification of use-case scenarios is studied at the end of this section, and is followed by discussion on the importance of formal verification, to check the validity of the model during the modelling process.

5.1 Basic properties

As the basic properties, we present two properties demonstrating the capability of the *CashDeskChannel* component in the Trading System to broadcast events to the components that subscribed for them.

Property 1 (Unwanted duplicity). When the *CashDeskChannel (200)* receives a request to broadcast the *SaleSuccessEvent* via $(100, publishSaleSuccessEvent, 200)$, the event is going to be delivered to all subscribers $(200, onEventSaleSuccess, X)$ at most once. In the property, as well as in the following properties, action names are shortened to the sequence of first letters of their sub-words, e.g. *publishSaleSuccessEvent* becomes *pSSE*.

$$(a)\quad \mathcal{G}\,\big(\mathcal{P}(100,pSSE,200) \Rightarrow \neg\,[\neg\,\mathcal{P}(100,pSSE,200)\ \mathcal{U}\ (\mathcal{P}(200,oESS,142) \wedge \mathcal{X}\,[\neg\,\mathcal{P}(100,pSSE,200)\ \mathcal{U}\ \mathcal{P}(200,oESS,142)])]\big)$$

$$(b)\quad \mathcal{G}\,\big(\mathcal{P}(100,pSSE,200) \Rightarrow \neg\,[\neg\,\mathcal{P}(100,pSSE,200)\ \mathcal{U}\ (\mathcal{P}(200,oESS,162) \wedge \mathcal{X}\,[\neg\,\mathcal{P}(100,pSSE,200)\ \mathcal{U}\ \mathcal{P}(200,oESS,162)])]\big)$$

property	states	transitions	memory	time	result
prop1a	749 340	3 181 473	533 MB	68 s	holds
prop1b	749 340	3 181 473	534 MB	67 s	holds

The data in the table refer to the model composed with the appropriate property automaton. "memory" means all memory used in verification of the property. Note that the number of states of the model composed with the property is, in this case, equal to the number of states of the original model. This interesting fact is explained in Section 6.

Property 2 (Guaranteed delivery). Whenever the *CashDeskChannel (200)* receives a request to broadcast the *SaleSuccessEvent*, the event is going to be delivered to all subscribers $(200, onEventSaleSuccess, X)$ at least once, or an exception occurs $(200, exceptionPublishSaleSuccessEvent, 100)$.

$$\mathcal{G}\,[\mathcal{P}(100,pSSE,200) \Rightarrow ([BOTH \wedge \neg\,EXC] \vee [NONE \wedge EXC])]$$

where

$$BOTH = [\neg\,\mathcal{P}(100,pSSE,200)\ \mathcal{U}\ \mathcal{P}(200,oESS,142)] \wedge [\neg\,\mathcal{P}(100,pSSE,200)\ \mathcal{U}\ \mathcal{P}(200,oESS,162)]$$
$$NONE = (\neg\,[\neg\,\mathcal{P}(100,pSSE,200)\ \mathcal{U}\ \mathcal{P}(200,oESS,142)]) \wedge (\neg\,[\neg\,\mathcal{P}(100,pSSE,200)\ \mathcal{U}\ \mathcal{P}(200,oESS,162)])$$
$$EXC = \neg\,\mathcal{P}(100,pSSE,200)\ \mathcal{U}\ (200,ePSSE,100)$$

property	states	transitions	memory	time	result
prop2	749 340	3 181 473	533 MB	68 s	holds

5.2 Local deadlocks of components

In component-based systems, many components coexist in parallel. Hence deadlock of some of them cannot be detected as halting of the whole system. We understand a *local deadlock* of a component as a state from which the component *cannot* move

further. This situation requires the *enabledness* $\mathcal{E}$ operator, otherwise we could only express that it *does not* move further. The following two properties describe a local deadlock of a component on a particular service call, and the third property specifies a local deadlock with respect to any action.

Property 3 (Local deadlock on one action). It cannot happen that the *StoreApplication (610)* is ready to call `getTransactionContext()` but never can do so because its counterpart *Persistence (511)* is never ready to accept the call.

$$[\mathcal{F}\,\mathcal{P}(610, gTC, -)] \vee \mathcal{G}\,[\mathcal{E}(610, gTC, -) \Rightarrow \mathcal{F}\,\mathcal{E}(610, gTC, 511)]$$

property	states	transitions	memory	time	result
prop3	778 100	3 298 237	538 MB	73 s	holds

This property helped us to evaluate one of our modelling decisions. As the service `getTransactionContext()` activates a new instance of the component *TransactionContextImpl*, where only a limited number of instances can be active at any time, this property allows us to check that the bound on the number of instances that are ready to be activated is sufficient.

Note that this property requires the existence of the $(610, getTransactionContext, -)$ label in the model. However it is omitted in our model because we suppose it must synchronize with its counterpart and be removed from the model. Therefore for the purpose of verification of this property, we modify the model in a way that this label is not omitted. Surprisingly, this does not influence the state-space traversed during verification because the property automaton forces traversal of only the runs with no $(610, getTransactionContext, -)$ on them.

Property 4 (Local deadlock on one action). It cannot happen that the *CashDeskApplication (100)* is ready to send a notification to the *CashDeskChannel (200)* saying that it received the *SaleStartedEvent*, but the *CashDeskChannel* is never ready to accept the notification.

$$[\mathcal{F}\,\mathcal{P}(100, oESS'', -)] \vee \mathcal{G}\,[\mathcal{E}(100, oESS'', -) \Rightarrow \mathcal{F}\,\mathcal{E}(100, oESS'', 200)]$$

property	states	transitions	memory	time	result
prop4	749 343	3 181 479	533 MB	67 s	holds

The *CashDeskChannel (200)* in the system is not allowed to accept notifications before it delivers events to all subscribers. If some of the subscribers would be constantly refusing to accept the event, it could block other components that already accepted the event and want to notify the channel. As the property is valid, this

cannot happen in the system (on the *SaleStartedEvent*).

Property 5 (Local deadlock on any action). It cannot happen that the *Persistence (511)* for StoreApplication becomes deadlocked (cannot make any action).

$$\mathcal{G}\,\mathcal{F}\,(ENABLED_{511})$$

where $ENABLED_{511} = \mathcal{E}(610, gPC, 511) \vee \mathcal{E}(620, gPC, 511) \vee \ldots \vee \mathcal{E}(511, eIA, 620)$, that is a disjunction of formulas of type $\mathcal{E}(label)$ for all labels the *Persistence (511)* participates in.

property	states	transitions	memory	time	result
prop5	1 498 679	7 805 074	690 MB	561 s	does not hold

The violation of the property means that the system gets into a state from which the component is no more able to perform any computation. This can happen for three reasons: (1) it gets stuck in its internal computation, (2) the environment refuses to accept its calls, or (3) the environment does not wish the component to compute anything for it any more. In our model, the last case is true, because in the usage profile, we suppose that only one sale is accomplished. Hence the system is not supposed to execute forever.

5.3 Blocking of components

Here we study a more strict version of local deadlocks, which is temporary *blocking* of a component because of non-readiness of its counterpart to accept its calls. This property is considered the core issue of correctness of component-based systems in several component-based models (SOFA [1], Interface automata [9]).

Property 6. It cannot happen that the *StoreApplication (610)* wants to begin a transaction $(610, beginTransaction, -)$ calling the *TransactionContextImpl (511)*, which is not right in the current state ready to accept it.

$$[\mathcal{F}\,\mathcal{P}(610, bT, -)] \;\vee\; \mathcal{G} \neg [\mathcal{E}(610, bT, -) \wedge \neg\, \mathcal{E}(610, bT, 511)]$$

property	states	transitions	memory	time	result
prop6	749 340	3 181 473	536 MB	71 s	holds

Note that we require the existence of the $(610, beginTransaction, -)$ label in the model. For the purpose of this verification, we modify the model in a way similar to the case with property 3. Even here, the resulting state space does not change, due to the nature of the property automaton.

Property 7. It cannot happen that the *CashDeskApplication (100)* is ready to send a notification to the *CashDeskChannel (200)* saying that it received the *SaleStartedEvent*, but the *CashDeskChannel* is not right in the current state ready to accept the notification.

$$[\mathcal{F}\ \mathcal{P}(100, oESS'', -)] \vee \mathcal{G} \neg [\mathcal{E}(100, oESS'', -) \wedge \neg\, \mathcal{E}(100, oESS'', 200)]$$

property	states	transitions	memory	time	result
prop7	1 498 671	6 362 935	689 MB	546 s	does not hold

The property is a more strict version of the property 4. While the property 4 shows that the *CashDeskChannel (200)* always sends all copies of the *SaleStartedEvent* and gets into the state where it is ready to start accepting notifications, this property shows that it may take a while before the channel gets ready. However, this is not an error in the system. It correctly reflects the nature of the channel.

5.4 Loop issues

In our model, many cycles/loops can be found. Each loop can complete a run that enters it but never exits. In software systems, however, most of the loops in models result from `for` or `while` cycles that are traversed only finitely many times. This can cause non-realistic results of properties verification. The properties should be verified only on the runs that follow selected loops only finitely many times.

Property 8. Whenever the *ProductDispatcher (630)* call `queryStoreById()` on the *Store* for *ProductDispatcher (523)* via $(630, queryStoreById, 523)$, it gets a response $(523, queryStoreById', 630)$ at some point in the future.

$$\mathcal{G}\ [\mathcal{P}(630, qSBI, 523) \Rightarrow \mathcal{F}\ \mathcal{P}(523, qSBI', 630)]$$

property	states	transitions	memory	time	result
prop8	750 684	3 186 705	534 MB	262 s	does not hold

In the counterexample, one of the components gets into a loop that it never exits. This does not report a real situation in the system and hence a modification of the property is necessary.

Property 9. Whenever the *ReportingApplication (620)* calls `queryStoreById()` on the *Store (522)* for *ReportingApplication* via $(620, queryStoreById, 522)$, it gets a response $(522, queryStoreById', 620)$ at some point in the future.

$$\mathcal{G}\ [\mathcal{P}(620, qSBI, 522) \Rightarrow \mathcal{F}\ \mathcal{P}(522, qSBI', 620)]$$

property	states	transitions	memory	time	result
prop9	749 340	3 181 473	531 MB	69 s	holds

The validity of this property may seem surprising as it is analogical to the previous one. Further verification shows that the reason for the validity is that $(620, queryStoreById, 522)$ is not reachable in the model with selected usage profile.

Property 10. Whenever the *ProductDispatcher (630)* calls `queryStoreById()` on the *Store (523)* for *ProductDispatcher*, it gets a response at some point in the future, if the progress of the system is forced by transitions of the *Store (523)*, which cannot get into invalid infinite loop.

$$\mathcal{G}\,[(\mathcal{P}(630, qSBI, 523) \wedge \mathcal{G}\,\mathcal{F}\ MOVE_{523}) \Rightarrow \mathcal{F}\,\mathcal{P}(523, qSBI', 630)]$$

where $MOVE_{523} = \mathcal{P}(610, qLSI, 523) \vee \mathcal{P}(620, qASI, 523) \vee \ldots \vee \mathcal{P}(630, qSI, 523)$, that is a disjunction of formulas of type $\mathcal{P}(label)$ for all labels the *Store (523)* participates in.

property	states	transitions	memory	time	result
prop10	750 684	3 186 705	532 MB	69 s	holds

What remains is to make sure that there is a run in the model which satisfies the premise of this property's implication, in order to prevent the same thing that has happened in verification of property 8. This has been verified and such run has been successfully found.

5.5 Use-case scenarios

In the verification of use-case scenarios, we are given an assumption on the usage profile of the system, and we want to guarantee that a particular behaviour is present in the response of the system. A use-case scenario is defined as a sequence of interactions (labels). It can be either complete (all labels are listed) or partial (given labels can be interleaved with other labels). In component-based systems, where the searched behaviour can be interleaved by behaviour of independent components in the system, the partial scenarios are of higher interest. This section presents results of verification of the three most complex (partial) scenarios defined in [15].

In contrast with the other verified properties, the use-case scenarios do not state that for all paths, some property holds (as is usual in the LTL model checking), but they state that there is a path, along which some property holds (namely the property representing the sequence of labels). This can be verified with the same methods, just by negating the property. Note that the properties representing the use-case scenarios are so large that we do not give their formal representation here. However, they are a part of the model, which is available at [16].

UC scenario 1. CashPayment The scenario reflects cooperation of system components to successfully accomplish purchase of goods finished with cash payment.

UC scenario 2. Unsuccessful CardPayment The scenario describes system reactions to a sale finished with card payment that is refused by the bank.

UC scenario 3. Successful CardPayment The scenario describes component interaction following a successful sale finished with card payment.

property	states	transitions	memory	time	result
uc1	13 689 354	58 190 231	3 093 MB	3 712 s	scenario found
uc2	11 670 924	49 165 124	2 696 MB	3 946 s	scenario found
uc3	11 680 736	49 202 320	2 695 MB	3 147 s	scenario found

5.6 Validity of the model

During modelling, we needed to abstract from aspects of the system that could make the size of the model unmanageable, while staying confident about the safety of the abstractions. Two types of abstractions were considered: simplification of the internal behaviour of primitive components, and simplification of the communicational scheme. Regarding the communication among components, we evaluated serialization of selected parallel service calls and changing of some asynchronous calls to synchronous. The serialization was considered both on required (calling services) and provided (serving calls) side. This significantly reduced the state space, while causing no harm when the service calls were independent and their ordering had no effect on further behaviour of the system. Verification helped us to evaluate a number of serialization and synchronisation decisions via checking the validity of the model after the modification.

When checking the validity of the model, we worked with a set of properties based mainly on the use-case scenarios and test cases defined in the CoCoME Modelling Contest. We also tested the model for deadlocks, because we experienced that violation of the model validity often results in deadlock situations, either global or local.

6 Experience and discussion

In this section, we share our modelling and verification experience, discussing some of the results and observations we have achieved.

Characteristics of the model. As the number of components in the Trading System is quite large, and our modelling language expresses component concurrency

through interleaving, the model suffers from state space explosion. However, the size of the reachable state space does not grow evenly during the hierarchical composition of components, but it changes dramatically. The reason for the irregular changes of the state space is that a composite automaton does not need to be larger than the automata it is composed of. We have observed cases, where the number of reachable states has been dramatically reduced by the composition. This is due to the parametrized operator that can delimit possible behaviour in the composition. This fact can complicate the estimation of the number of states for a given model. But on the other hand, it can be exploited to produce a smaller model out of a large one, as was demonstrated in this case study, where the large Trading System model has been restricted by adding the cashier.

Deadlocks in the model. After deciding on the model for verification, in the validation phase, we have checked the model for global and local deadlocks. We have learned that the existence of deadlock states often signals a modelling error. A few *global* deadlocks were found. By careful investigation, we found that these deadlocks correspond to a behaviour reflecting that two components decide to receive messages from one of the event channels sent to them in an incorrect order, thus blocking each other. As we were not provided with the implementation of the event channels, we can treat this finding in two ways. Either the deadlock reveals an error in the system, or it reflects an unrealistic behaviour, i.e. the system guards that the components receive messages in the right order. We decided to treat the runs leading to the deadlock states as unrealistic, and ignore them during verification. This is done implicitly in our verification method, because it verifies *infinite* runs only.

Local deadlocks and component-blocking properties. Interesting observations were made in verifying the *local* deadlocks and their more strict form, the component-blocking properties. We have verified many pairs of such properties and we have found a strong relation between the two kinds. Mostly, it was either the case that both properties were satisfied, or none of them was. The reasons are similar to those explained after property 5, that is, the environment does not wish the components to compute anything any more. We have, though, found a few cases, when the *local deadlock* property holds, but the *blocking* property does not, and we have presented one of them. Note that both kinds of properties take advantage of the *enabledness* $\mathcal{E}$ operator without which they could not be formulated.

Size of the model/property composition. As may be noticed in Section 5, in some of the presented cases the state-space size remains (nearly) the same when the model is composed with the property automaton. This interesting fact deserves an explanation. The property automata are generated with the effort to make the resulting composition as small as possible. Then in case of some properties (such as safety and request/response properties), for every state of the model in the composition, there is a unique state in the property automaton. Hence the composition with the property does not influence the size of the model.

Cluster. The experiments presented in this paper were run on a cluster of ten computers. This choice was justified by a number of experiments on various numbers of computers. A smaller number of workstations would suffice, but the verification would get substantially slower in the case of larger property automata (e.g. the use-case scenarios). On the other hand, larger number of workstations causes inadequate memory overhead in the case of small property automata. The choice of ten computers is a reasonable compromise.

7 Conclusion and future work

In this paper, we give a practical application of the presented CI-LTL verification technique to a large component-based system using a parallel model checking tool DiVinE. We briefly introduce our modelling language as well as the temporal logic CI-LTL, a modification of the action based LTL. We have verified a multitude of properties of the Trading System. Thirteen of them that are of particular interest within the component-oriented software engineering society, are presented here together with the results of the verification and their discussion. The presented properties include two basic properties describing the broadcasting ability of the event-channel components, three properties concerning the possibility of a local deadlock, two properties addressing the component blocking problem, and three properties dealing with the problems caused by cycles in the model. The last three properties are different from the previous. They are used for checking the correctness of the use-case scenarios. Finally, we discuss how the model checking helped us in creation of the model, and we summarize the experience obtained during verification, including discussion of some of the results.

The study confirms that the CI automata modelling language suits well both to capture various types of interactions among individual components in component-based systems, and to formally verify interaction properties. This distinguishes our modelling approach from others presented in the *CoCoME Modelling Contest* [15] and brings a new value to the area of component-based software engineering. As the very significant feature of component-based systems is the concurrent behaviour of individual components and consequently the enormous size of the state space, distributed and parallel verification techniques are a need for handling these type of systems. They allowed us to verify very complex properties of the Trading System when restricted to a usage profile. But still, we were not able to verify the Trading System with no usage profile added — this means any usage possible with any number of users — as our hardware capacity did not suffice.

In future, we aim at extending our verification techniques with various reduction methods to allow us to verify even larger systems. Currently, we explore the possibilities of two existing reduction techniques, the partial-order reduction and the symmetry reduction. However, their application in our framework is not straightforward, due to the nature of the temporal logic we use. We also try to find new reduction methods taking advantage of component-specific features.

References

[1] Adamek, J. and F. Plasil, *Behavior protocols capturing errors and updates*, in: *Proceedings of the ETAPS Workshop on Unanticipated Software Evolution (USE'03)* (2003), pp. 17–25.

[2] Allen, R. J., "A Formal Approach to Software Architecture," Ph.D. thesis, Carnegie Mellon University, School of Computer Science, USA (1997).

[3] Barnat, J., L. Brim and I. Černá, *Distributed Analysis of Large Systems*, in: *Proc. of the 4th International Symposium on Formal Methods for Components and Objects (FMCO 05)*, LNCS **2006** (2006), pp. 259–279.

[4] Barnat, J., L. Brim, I. Černá, P. Moravec, P. Ročkai and P. Šimecek, *Divine – a tool for distributed verification*, in: *Proceedings of the Computer Aided Verification conference (CAV'06)* (2006), pp. 278–281.

[5] Becker, S., H. Koziolek and R. Reussner, *Modelbased performance prediction with the palladio component model*, in: *Proceedings of the International Workshop on Software and Performance (WOSP'07)* (2007), pp. 54–65.

[6] Brim, L., I. Černá, P. Vařeková and B. Zimmerova, *Component-Interaction automata as a verification-oriented component-based system specification*, in: *Proceedings of the ESEC/FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS'05)* (2005), pp. 31–38, published also in ACM SIGSOFT Software Engineering Notes, Volume 31, Issue 2 (March 2006).

[7] Bruneton, E., T. Coupaye, M. Leclercq, V. Quéma and J.-B. Stefani, *The fractal component model and its support in java*, Software: Practice and Experience **36** (2006), pp. 1257–1284.

[8] Černá, I., P. Vařeková and B. Zimmerova, *Component-interaction automata modelling language*, Technical Report FIMU-RS-2006-08, Masaryk University, Faculty of Informatics, Brno, Czech Republic (2006).

[9] de Alfaro, L. and T. A. Henzinger, *Interface-based design*, in: *Proceedings of the 2004 Marktoberdorf Summer School* (2005), pp. 1 – 25.

[10] *DiVinE project web page.*
URL http://anna.fi.muni.cz/divine/

[11] Garlan, D., R. T. Monroe and D. Wile, "Foundations of Component-Based Systems," Cambridge University Press, USA, 2000 ISBN 0-521-77164-1.

[12] Magee, J., J. Kramer and D. Giannakopoulou, *Behaviour analysis of software architectures*, in: *Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA'99)* (1999), pp. 35–50.

[13] Plasil, F. and S. Visnovsky, *Behavior protocols for software components*, IEEE Transactions on Software Engineering **28** (2002), pp. 1056–1076.

[14] Pnueli, A., *The temporal logic of programs*, in: *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science* (1977), pp. 46–57.

[15] Rausch, A., R. Reussner, R. Mirandola and F. Plasil, editors, "The Common Component Modeling Example: Comparing Software Component Models," To appear in LNCS, 2007.
URL http://www.cocome.org

[16] The CoIn Team, *The complete CoIn model of the Trading System* (2007).
URL http://anna.fi.muni.cz/coin/cocome/

[17] Vardi, M. Y., *An automata-theoretic approach to linear temporal logic*, in: *Logics for Concurrency: Structure versus Automata*, LNCS **1043** (1996), pp. 238 – 266.

[18] Zimmerova, B., P. Vařeková, N. Beneš, I. Černá, L. Brim and J. Sochor, "The Common Component Modeling Example: Comparing Software Component Models, chapter Component-Interaction Automata Approach (CoIn)," To appear in LNCS, 2007 .