Reflecting Creation and Destruction of Instances in CBSs Modelling and Verification*

Barbora Zimmerova and Pavlína Vařeková

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{zimmerova,xvareko1}@fi.muni.cz

Abstract. In the paper, we present our solution to the issue of modelling and verification of communicational behaviour in Component-Based Systems (CBSs) allowing creation and destruction of component instances. We first introduce a modelling technique for capturing each component type and component instance as a finite-state transition system, and define the system model as a collection of those. Then we present a verification technique we have defined for a similar type of systems [1], and discuss application of the technique to the systems with dynamic instantiation of components at run time.

1 Introduction

Component-Based Systems (CBSs) are software systems assembled of autonomous components that are usually developed by third parties. In such systems, high accent is on the correctness of interaction among components and on the communicational behaviour of the whole systems. For the purpose of verification, the behaviour of components is usually captured via finite-state models, which are composed together into the model of the system. If the number of components is given in advance, the model is again finite-state, which allows direct application of existing finite-state verification techniques. These are usually very effective comparing to verification of infinite-state systems. A problem arises when the number of components in the system is not known in advance. This is the case in systems that support dynamic instantiation of components at run-time, as usual in Object-Oriented Systems. If there is no information about the maximal number of instances that can be created, the model of the system is infinite in general.

In [1] we have proposed a general verification approach for verifying systems with unknown number of components using finite-state verification. However in the approach, we did not treat explicit creation and destruction of component instances. In this paper we focus on this issue in detail. First we introduce a modelling technique for capturing creation and destruction of instances (section 4) in a way that models of all components remain finite. However there are infinitely many of them. Then we discuss application of the technique from [1] to verify the model (section 5).

* Supported by the Czech Science Foundation within the project No. 102/05/H050.

2 Related work

There are two main directions to follow when addressing the issue of verification of systems with creation and destruction of component instances. The first one is to create an infinite-state model of the system, and apply a verification technique that is able to deal with it. The main representative of this direction is the π-calculus [2]. But there are also other approaches, like [3], where systems are first modelled as infinite-state and then reduced to finite-state by an adjusted verification technique. The second option to deal with this issue is creation of a finite-state model early in the modelling phase. One way is to create a parametric model of the system where the parameter stating the number of dynamic components is evaluated as soon as the number of components can be estimated [4]. However in the case of component instantiation this value cannot be guessed even at run-time. This was our motivation to introduce the approach in [1].

3 Running example

The core idea of our approach can be explained already on a very simple system, with only one type of instances that can be created and destructed at run-time. Hence suppose a system that consists of a number of *instances* of a common type, and one stable component, called *handler*, that is supposed to handle creation and destruction of the instances. Such a system can be in fact only a sub-system of a more complex system. This extends our approach also to the systems, that can be decomposed into sub-systems, each of them handling a different type of instances.

As the running example of this paper, we use a part of the *CoCoME (Common Component Modelling Example)* system, the *Trading System* [5]. As our sub-system, we have chosen the *Coordinator* component, which is in the *Trading System* used for managing express checkouts. For this purpose, it keeps a list of sales that were done during last 60 minutes, which helps it to decide whether an express cash desk is needed. The component consists of two sub-components, the *CoordinatorEventHandler* and the *Sale*. Anytime a new sale arrives, the *CoordinatorEventHandler* creates a new instance of the *Sale* component to represent it in the list. Whenever the sale represented by an instance expires, the *CoordinatorEventHandler* removes the instance from the list which causes its destruction. A simplified Java code implementing the *Coordinator* is in appendix A [6]. We use the code to derive the behavioural models of the components.

In the models, name of the method `getNumberofItems()` is shortened to *gNI*, `getPaymentMode()` to *gPM*, `getTimeofSale()` to *gTS*, `updateStatistics()` to *uS*, and `isExpressModeNeeded()` to *iEMN*.

4 Modelling

In this section, we first present the language that we use for capturing communicational behaviour of component-based systems. Then we present our technique for modelling the systems with creation and destruction of instances.

4.1 Modelling language

We use *Component-interaction automata* [7, 5] as a language for modelling behaviour of components. The language captures each component as a labelled transition system with structured labels (to remember components which communicated on an action) and a hierarchy of component names (which represents the architectural structure of the component). In this section, we sketch the essential definitions and present them on examples. For full definitions see [7].

A *component-interaction automaton* (or a *CI automaton* for short) 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.

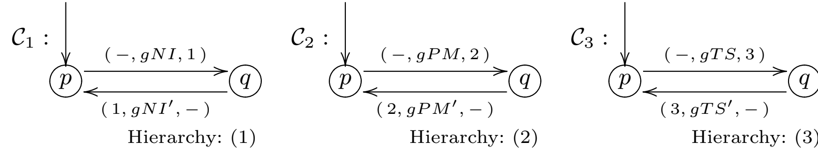

Fig. 1. CI automata $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ modelling the methods `getNumberofItems()`, `getPaymentMode()`, `getTimeofSale()` from appendix A [6]

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 1. Each of them represents a model of behaviour of one basic method. For example, $(-, gNI, 1)$ in $\mathcal{C}_1$ signifies that a component

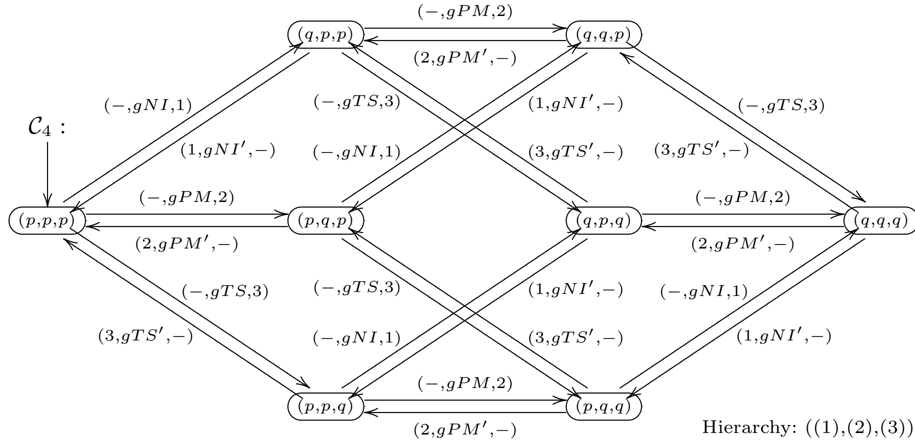

Fig. 2. Composition $\mathcal{C}_4 = \otimes^{\mathcal{F}}\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ where $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ are in Fig. 1 and $\mathcal{F}$ consists of all labels $\mathcal{F} = \{(-, gNI, 1), (1, gNI', -), (-, gPM, 2), (2, gPM', -), (-, gTS, 3), (3, gTS', -)\}$

(method) with a numerical name 1 inputs an action gNI (a request for a method `getNumberofItems()`), and $(1, gNI', -)$ in $\mathcal{C}_1$ signifies that a component 1 outputs an action gNI' (a response for the method `getNumberofItems()`).

Such automata can be composed together using a parameterizable 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 a given set of CI automata into a product automaton and removes all transitions from the product that have labels outside $\mathcal{F}$. In the product, the components cooperate either by interleaving of their original transitions, or 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)). Examples of two composite automata are in Figure 2 and 3.

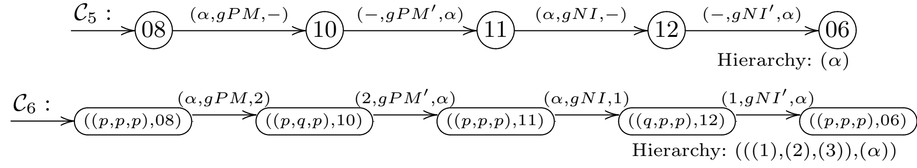

Fig. 3. Composition $\mathcal{C}_6 = \otimes^{\mathcal{F}}\{\mathcal{C}_4, \mathcal{C}_5\}$ where $\mathcal{C}_4$ is in Fig. 2, $\mathcal{C}_5$ is a part of the automaton in Fig. 4, and $\mathcal{F} = \{(\alpha, gPM, 2), (2, gPM', \alpha), (\alpha, gNI, 1), (1, gNI', \alpha)\}$

As the composition allows us to compose only a finite set of components, we more define a *dynamic system model* (full definition in [1]), which is able to capture a system as a collection of infinite number of components. A *dynamic system model* is a triple $(\mathcal{C}_0, \{\mathcal{C}_i\}_{i\in\mathbb{N}}, \mathcal{F})$ of the stable component $\mathcal{C}_0$ (called *provider* in the definition), an infinite set of unified components $\mathcal{C}_i, i \in \mathbb{N}$ (called *clients*), and $\mathcal{F}$ used to compose a finite subset of them in a way that the clients do not communicate with each other and the provider handles all clients equally.

4.2 Modelling technique

In our system, we have two types of components, a *handler* (represented by the *CoordinatorEventHandler* component), and *instances* (represented by the *Sale* component). For each of them we create a CI automaton. In both cases we first create models of all services (represented by the methods) and then compose them together into a model of the component. The model of each service is a loop that starts with receive of the service request and finishes with sending the response. Each call to external service is modelled as a sequence of two transitions, output of the request and input of the response.

Model of the handler. The model of the *CoordinatorEventHandler* (code in appendix A [6]) is in Fig. 4. In addition to standard service calls, we also model actions that cause the creation and destruction of instances. In the case of our example, the code `new Sale(...)` in *CoordinatorEventHandler* is represented by the sequence $(\alpha, Sale, -), (-, Sale', \alpha)$, in the case of destruction, the code `i.remove()` is represented by the sequence $(\alpha, SaleD, -)$, $(-, SaleD', \alpha)$.

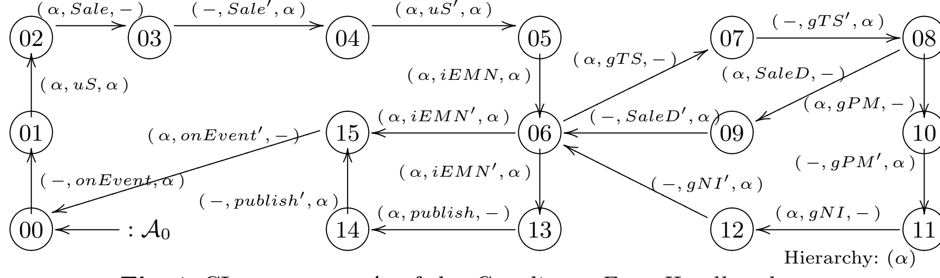

Fig. 4. CI automaton $\mathcal{A}_0$ of the *CoordinatorEventHandler class*

Model of the instance. First, we create the model of the instance type. For our example, the model of the *Sale* type is in fact the automaton $\mathcal{C}_4$ in Fig. 2, after renaming of all component names to 0 and making the hierarchy primitive (0), see Fig. 5 on the left. Then the model of an instance is created as an extension of the model of the type with an initial activation part, see Fig. 5 on the right. The path $0 \to 1 \to (p,p,p)$ represents a call of the constructor of the component, and the way back $(p,p,p) \to 2 \to 0$ represents the destruction, which could start also from other states than (p,p,p). However it should not be possible for it to start in states 0 or 1 to guarantee that the system can destruct only the instances that have been created before. Note that if the *Sale* performed some calls inside its constructor or destructor, these calls could be included in the paths $0 \to 1 \to (p,p,p)$ and $(p,p,p) \to 2 \to 0$ as one may intuitively think.

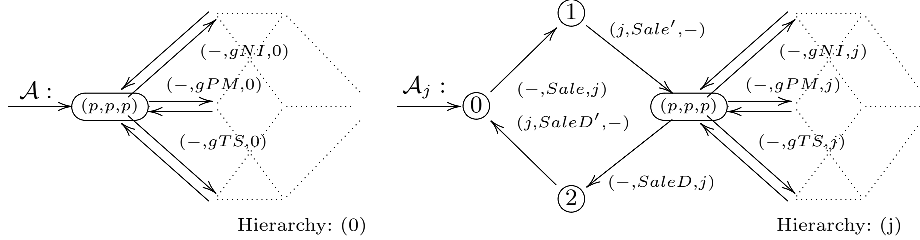

Fig. 5. CI automaton $\mathcal{A}$ a of the *Sale type* and $\mathcal{A}_j, j \in \mathbb{N}$ of the *Sale instances*

Model of the composite system. The model of the system is the dynamic system model $\mathcal{D} = (\mathcal{A}_0, \{\mathcal{A}_j\}_{j\in\mathbb{N}}, \mathcal{F})$ where $\mathcal{F} = \{(\alpha, uS, \alpha), (\alpha, uS', \alpha), (\alpha, iEMN, \alpha),$ $(\alpha, iEMN', \alpha), (-, onEvent, \alpha), (\alpha, onEvent', -), (\alpha, publish, -), (-, publish', \alpha)\} \cup$ $\bigcup_{j\in\mathbb{N}}\{(\alpha, Sale, j), (j, Sale', \alpha), (\alpha, SaleD, j), (j, SaleD', \alpha), (\alpha, gNI, j), (j, gNI', \alpha), (\alpha, gPM, j),$ $(j, gPM', \alpha), (\alpha, gTS, j), (j, gTS', \alpha)\}$. $\mathcal{F}$ realizes the handshake-like synchronization of the instances with the handler. In any composition $\mathcal{D}_n = \otimes^{\mathcal{F}}\{\mathcal{A}_i\}_{0\le i\le n}$ (handler and n instances), all the instance automata start in the state 0, which means that they are inactive at the beginning – the *CoordinatorEventHandler* cannot access their behaviour which starts to be available in state (p,p,p). However, when the *CoordinatorEventHandler* calls a constructor of the *Sale* $(\alpha, Sale, -), (-, Sale', \alpha)$, one of the instances synchronizes with it and moves to the state that makes its functionality available. When the *CoordinatorEventHandler* removes the *Sale* from its list $(\alpha, SaleD, -), (-, SaleD', \alpha)$, it returns back to the state 0 and waits for another call of its constructor.

5 Verification

Each snapshot of our dynamic system $\mathcal{D}_n = \otimes^{\mathcal{F}}\{\mathcal{A}_i\}_{0\leq i\leq n}$ for concrete $n \in \mathbb{N}$ is finite-state and hence verifiable with existing formal verification techniques for finite-state systems, like model checking [8]. However we do not know the maximal number of instances in advance, hence we need to verify all $\mathcal{D}_0, \mathcal{D}_1, \mathcal{D}_2, ...$ to guarantee that the answer is correct.

In [1], we have proposed a verification approach that for a dynamic system model $\mathcal{D}$ and a temporal property finds a number k such that if we positively verify $\mathcal{D}_0, \mathcal{D}_1, ..., \mathcal{D}_k$, we can conclude that the property holds on any $\mathcal{D}_l, l \in \mathbb{N}_0$. We apply this technique to our dynamic system model even if it was designed for a slightly different type of systems. In [1], the stable component is supposed to be a passive *provider* of some functionality, and the dynamic components to be active *clients* that access this functionality. In the case of our kind of systems, the provider, now called *handler*, is the active one, and the clients, now called *instances*, are passive. The activity of the handler brings two issues that need to be considered.

Issue 1. Each $\mathcal{D}_n$ contains only n activable instances, which introduces non-realistic deadlock states into the composite model. These are reached when the handler wants to activate a new instance when all are in use.

Solution: We can easily ignore the deadlock states during verification by considering only infinite runs of the system.

Issue 2. When the set of observable labels X contains all labels necessary for verification like in [1], the number of instances regarded by the handler w.r.t. X (the active instances) cannot be limited by any constant, which implies $k = \infty$.

Solution: We can resolve this with the optimization [9] based on a modification of the system model and minimization of X, which helps to make k finite.

5.1 Properties for verification.

We focus on verification of the properties expressed in CI-LTL [5] (see appendix B [6]), a state/action-based extension of the linear temporal logic LTL [10]. Each property is captured with a harmonized sequence of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$, each φ_i corresponding to a given $\mathcal{D}_i$ (model with i instances), that make no distinction among instances.

Property 1. If the handler enters the cycle inside execution of `isExpressMode-Needed()` (states $06-12$ in Fig. 4), it exits the cycle eventually.

This property can be expressed by a harmonized set of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$:
$\varphi_i = G\ [\mathcal{P}(\alpha, iEMN, \alpha) \Rightarrow\ F\ \mathcal{P}(\alpha, iEMN', \alpha)]$.

Property 2. An activated instance of Sale cannot get deadlocked.

This property can be expressed by a harmonized set of formulas $\{\varphi_i\}_{i\in\mathbb{N}_0}$:
$\varphi_i = \bigwedge_{j\leq i} \varphi(j)$ where $\varphi(j) = \neg F\ (\mathcal{P}_{(j, Sale', \alpha)} \wedge [\neg\mathcal{P}_{(\alpha, SaleD, j)}\ U\ G \bigwedge_{l\in L_j} \neg\mathcal{E}(l)])$

where $L_j = \{(\alpha, Sale, j), (j, Sale', \alpha), (\alpha, SaleD, j), (j, SaleD', \alpha), (\alpha, gNI, j), (j, gNI', \alpha), (\alpha, gPM, j), (j, gPM', \alpha), (\alpha, gTS, j), (j, gTS', \alpha)\}$.

5.2 Verification technique

Now we informally present the steps of the verification technique (see [1] for formal definitions, theorems and proofs) including examples of verification of selected properties on our model $\mathcal{D} = (\mathcal{A}_0, \{\mathcal{A}_j\}_{j\in\mathbb{N}}, \mathcal{F})$ from section 4.2.

Complexity of the temporal property. For the temporal property $\{\varphi_i\}_{i\in\mathbb{N}}$, we syntactically (based on the structure of the formulas) compute the minimal number of clients in the dynamic system $\mathcal{D}$ that is necessary to exhibit a path violating the property. We denote this value $|\{\varphi_i\}_{i\in\mathbb{N}}|_{\mathcal{D}}$, and use $Property(\mathcal{D}, m)$ to denote the set of all harmonised sequences of formulas $\{\varphi_i\}_{i\in\mathbb{N}}$ such that $|\{\varphi_i\}_{i\in\mathbb{N}}|_{\mathcal{D}} \leq m$. As the instances (clients) show the same behaviour and do not communicate with each other, the violation of the property can be confirmed also while observing only specifically selected m instances.

Property 1. For any $i \in \mathbb{N}_0$ and an infinite run π starting in an initial state of the automaton $\mathcal{D}_i$ satisfying $\pi \not\models \varphi_i$, it holds that for confirming this violation it suffices to observe the handler, no instances need to be involved. Hence $\{\varphi_i\}_{i\in\mathbb{N}_0} \in Property(\mathcal{D}, 0)$.

Property 2. For any $i \in \mathbb{N}_0$ and an infinite run π starting in an initial state of the automaton $\mathcal{D}_i$ satisfying $\pi \not\models \varphi_i$, there must be a number $j \in \mathbb{N}$ such that on this run the formula $\varphi(j)$ is not valid. For confirming this violation it suffices to observe the handler and the *Sale* instance with the numerical name j. Hence $\{\varphi_i\}_{i\in\mathbb{N}_0} \in Property(\mathcal{D}, 1)$.

Complexity of the dynamic system model. For the dynamic system model $\mathcal{D}$, we estimate the maximal number of instances that the handler is able to manipulate concurrently at any moment. This number is denoted $|\mathcal{D}|_X$ and is more dependent on the set of labels X that are necessary for verification of the property. The value $|\mathcal{D}|_X$ is defined in a way that any path of the model with an arbitrary number of activable instances is in some sense equivalent to a path in the model with at most $|\mathcal{D}|_X$ instances.

Property 1. For this property it suffices to observe labels $X = \{(\alpha, iEMN, \alpha), (\alpha, iEMN', \alpha)\}$. Then the handler regards at most one instance $|\mathcal{D}|_X = 1$ because when focusing on labels from X and ignoring names of instances, each run of the system with any number of instances is equivalent to some run of the system with one instance only.

Property 2. For this property and any X containing all labels necessary for verification of the property, $|\mathcal{D}|_X = \infty$. Hence we need to use the optimization introduced in [9]. It in this case means that we select one instance, change its component name to β and move it inside the handler component. By this, we create a modified system $\overline{\mathcal{D}}$. In such settings, $X = L_\beta$ and $|\overline{\mathcal{D}}|_X = 1$.

A number of instances needed for verification. Having a dynamic system model with $|\mathcal{D}|_X = n$ and a property in $Property(\mathcal{D}, m)$ then the main theorem of [1] (see appendix C [6]) states that it suffices to verify the dynamic system model with $0, 1, \ldots, k{=}n{+}m$ instances to conclude if the property holds on the model no matter how many instances are going to be part of it.

Property 1. Here $k = 1 + 0 = 1$, thus after verifying the models $\mathcal{D}_0$ and $\mathcal{D}_1$ we can conclude on the general validity of the formula. In this case the formula does not hold which is confirmed by the model $\mathcal{D}_1$.

Property 2. Here $k = 1 + 1 = 2$ and after verifying the models $\mathcal{D}_0$, $\mathcal{D}_1$ and $\mathcal{D}_2$ we can conclude that the property always holds.

6 Conclusion

In the paper, we propose a modelling technique that allows us to capture component instances as finite-state models, and we use an adjusted verification technique [1, 9] to verify models of systems with dynamically created and destructed instances modelled by this way. Throughout the text, we present all steps of our technique on a simple running example, a part of the *CoCoME Trading System*. The verification is practically studied on two selected temporal properties. One of them is proved true, the other one false.

References

1. Vařeková, P., Moravec, P., Černá, I., Zimmerova, B.: Effective verification of systems with a dynamic number of components. In: Proceedings of the ESEC/FSE Workshop SAVCBS'07, Dubrovnik, Croatia (2007) 3–13
2. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge University Press, USA (1999)
3. Rensink, A., Schmidt, Á., Varró, D.: Model Checking Graph Transformations: A Comparison of Two Approaches. In: ICGT'04, Springer-Verlag (2004) 226–241
4. Adámek, J.: Addressing Unbounded Parallelism in Verification of Software Components. In: SNPD. (2006) 49–56
5. Zimmerova, B., Vařeková, P., Beneš, N., Černá, I., Brim, L., Sochor, J.: Component-Interaction Automata Approach (CoIn). In: The Common Component Modeling Example: Comparing Software Component Models. To appear in LNCS (2007)
6. Zimmerova, B., Vařeková, P.: Appendix of this paper (MEMICS'07). Electronically on http://anna.fi.muni.cz/coin/documents/MEMICS-07_appendix.pdf (2007)
7. Černá, I., Vařeková, P., Zimmerova, B.: Component-Interaction Automata Modelling Language. Technical Report FIMU-RS-2006-08, Masaryk University, Faculty of Informatics, Brno, Czech Republic (2006)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000)
9. Vařeková, P., Zimmerova, B.: Challenge problem: Subject-observer specification with component-interaction automata. In: Proceedings of the ESEC/FSE Workshop SAVCBS'07, Dubrovnik, Croatia (2007) 73–79
10. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science, IEEE Press (1977) 46–57