Model-based Compatibility Checking of System Modifications[*]

Arnd Poetzsch-Heffter, Christoph Feller, Ilham W. Kurnia, and Yannick Welsch

University of Kaiserslautern, Germany
{poetzsch,c_feller,ilham,welsch}@cs.uni-kl.de

Abstract. Maintenance and evolution of software systems require to modify or exchange system components. In many cases, we would like the new component versions to be backward compatible to the old ones, at least for the use in the given context. Whereas on the program level formal techniques to precisely define and verify backward compatibility are under development, the situation on the system level is less mature. A system component C has not only communication interfaces to other system components, but also to human users or the environment of the system. In such scenarios, compatibility checking of different versions of C needs more than program analysis:

- The behavior of the users are not part of the program, but needs to be considered for the overall system behavior.
- If the user interaction in the new version is different from the old one, the notion of compatibility needs clarification.
- Analyzing the user interface code makes checking technically difficult.

We suggest to use behavioral software models for compatibility checking. In our approach, the underlying system, the old and new component, and the nondeterministic behavior of the environment are modeled with the concurrent object-oriented behavioral modeling language ABS. Abstracting from implementation details, the checking becomes simpler than on the program level.

1 Introduction

Software systems play a key role in the infrastructure for modern societies. The size and cost of these systems forbid to create them “de novo” time and again. Thus, we need to systematically evolve systems and adapt them to new requirements. A typical evolution step is the exchange of a component C by a new version C'. We say that C' is *backward compatible* with C if the behaviors of C are also provided by C'. Backward compatibility is a central notion for quality assurance in software evolution and has different variants. Weaker forms of backward compatibility ensure that some well-defined properties are maintained during evolution steps, but not necessarily all behaviors. Another line of variation is with respect to the contexts in which backward compatibility should be guaranteed. For example, we can require that a component is backward compatible in all possible contexts or just for the use in certain systems.

[*] This research is partly funded by the EU project FP7-231620 HATS (Highly Adaptable and Trustworthy Software using Formal Models) and the German Research Foundation (DFG) under the project MoveSpaci in the priority programme RS3 (Reliably Secure Software Systems).

In this paper, we investigate compatibility of components that have a *bipartite context* consisting of, on the one hand, interactions with users or the environment and, on the other hand, communication with an underlying system. A common example is an application component C with a GUI that talks to an underlying database. Our goal is to show that a new version C', having possibly a very different GUI, can be used instead of C. More precisely, we want to make sure that users of C' can trigger the same interactions with the underlying system as in the old version. That is, we allow modifying the interactions with users or the environment[1], but want to maintain the behavior at the interface to the underlying system. Checking this kind of compatibility is challenging:

- Usual program analysis techniques are not sufficient, because we have to also take the user behavior into account.
- As the user interactions in the new version might be quite different from the old one, we have to be able to compose user and component behavior to derive the behavior at the interface to the underlying system.
- We have to abstract from the complexities of GUI software.

The central contribution of this paper is a new method for reasoning about compatibility of components with bipartite contexts. The method is based on the following framework:

- An executable behavioral modeling technique: Software components and users are modeled using the concurrent, object-oriented modeling language ABS [8]. ABS models abstract from implementation details (e.g., event handling and layout management in GUIs) and capture the concurrent behavior among possibly distributed components. They can faithfully reflect the software structure, simulate the implementation[2] and allow validation of models. ABS also supports modeling internal nondeterminism, which is, e.g., crucial to model the possible behavior of users.
- Component transition systems: The semantics of each component of the ABS model is represented by a component transition system (CTS) receiving and sending messages. In contrast to ABS which is very good for modeling, the CTS-level simplifies composition and reasoning. The consistency between an ABS component and a CTS can be verified by programming logics (see, e.g., [4]).
- A reasoning technique for compatibility with bipartite contexts based on CTS composition and simulation proofs.

In this paper, we describe the framework, use it to model a system with a GUI, and demonstrate a typical evolution step for such systems, in which the GUI and the possible user interactions are modified. Then, we define compatibility of components with bipartite contexts and describe how to check compatibility.

Overview. Section 2 presents the executable behavioral modeling technique, the language ABS, and our running example. Section 3 describes evolution steps and defines compatibility. Section 4 introduces CTSs, their composition, and compatibility checking. Finally, Sects. 5 and 6 discuss related work and present conclusions.

[1] For brevity, we will only consider user interactions in the following. However, we claim that our approach can also be used in settings in which sensors and actors are used to communicate with a modeled environment.

[2] ABS also supports code generation.

2 Modeling Software Systems

This section describes our behavioral modeling technique. It is more abstract than implementations, e.g., by abstracting from the event handling mechanisms of GUIs, but still reflects the structure and communication behavior of implementations which is important for component-based reasoning. We illustrate the modeling technique by an example that will also be used to explain our reasoning technique in subsequent sections.

2.1 ABS Modeling

To model software systems, we use the modeling language ABS, an object-oriented language with a concurrency model based on *concurrent object groups* (COGs). COGs follow the actor paradigm [6] and are developed to avoid data races and the complexity of multithreading. COGs are the unit of concurrency and distribution. During execution, each object is a member of exactly one COG for its entire lifetime. This is similar to the Java RMI setting where objects belong to certain JVM instances, which may run distributed on different machines. Groups can be created dynamically and work concurrently. Execution within a single group is sequential. Communication between groups is asynchronous. This concurrency model is used in the abstract behavioral specification language ABS [8] and in JCoBox [11], a Java based realization of COGs.

ABS supports object-oriented concepts using a Java-like syntax and immutable recursive datatypes in the style of functional languages. In ABS, the creation of COGs is related to object creation. The creation expression specifies whether the object is created in the current COG (using the standard **new** expression) or is created in a fresh COG (using the **new cog** expression). Communication in ABS between different COGs happens via *asynchronous method calls* which are indicated by an exclamation mark (!). A reference in ABS is *far* when it targets an object of a different COG, otherwise it is a *near* reference. Similar to the E programming language [10], ABS restricts synchronous method calls (indicated by the standard dot notation) to be made only on near references.

2.2 Example: Flight Booking System

As an example, we consider a simple flight booking system. It follows a two-tier architecture with an application accessing an underlying repository. The application has a GUI with several state-dependent views. The system consists of the two main components:

- Agent, modeling the application and graphical user interface layer, and
- Server, modeling the database upon which the actual booking takes place.

In addition to these two software components, we provide an explicit user model in ABS. All three components are modeled as COGs. The runtime structure of the system is given in Fig. 1. As the details of the server behavior are not relevant for our model, we represent that part of the system by a generic server object and a session object that handles the connection to the agent. The agent component consists of a main agent object that does the actual booking and various view objects that are used to present the steps of the booking process to the user. For example, the airlineView object presents

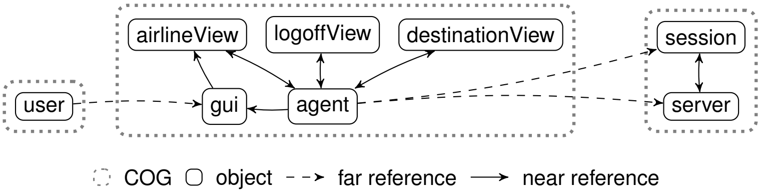

Fig. 1. Runtime structure of ABS model

```
interface Server {
    Session^N createSession(); }
interface Session {
    Price checkAndReserve(Airline al,
      Destination dest);
    Unit buy();
    Unit cancel();
    Unit close(); }
interface Agent {
    View^N getView(); }
interface GUI extends View {
    Unit setView(View^N v); }
interface View {
    Unit clickOn(Int position);
    String viewContent();
    Int getNbt(); }
```

Fig. 2. Interfaces of ABS model

the user with choices of the bookable airlines and the buttons to select the airline. The gui object is used as a proxy for the various views of the agent. Initially it delegates to the airlineView, but over time, it may delegate to other views. Encapsulating the various views by the gui proxy allows the agent to control which views are presented to the user.

The gui object together with the view objects realize the graphical user interface of the system where the gui object represents the part corresponding to a GUI toolkit. The design of our very simple GUI model is based on two principles:

- The communication between users and the GUI is asynchronous and triggered by the user, and the used methods are application-independent.
- The presentation of views is controlled by the software system.

Asynchronous communication is obtained by using ABS; application independency is achieved by realizing the communication over a generic interface. In our simple model, the interface View (see Fig. 2) allows users to click on buttons (method clickOn where the parameter identifies the button) and to inspect the view: method viewContent returns the shown content and getNbt yields the number of enabled buttons. The user interacts with the system by invoking these methods (the software system cannot call methods on the user). The gui object is the boundary between the software and the user. It delegates calls to the currently visible view and allows the software system to change between views using the setView method (see Fig. 2). Based on the illustrated principles, one can develop more realistic GUI models by providing sufficiently elaborate view interfaces.

The interfaces of the two main components (see Fig. 2) describe a small two-tier system. The server object implements the Server interface which provides a method to create new sessions; session objects implement the Session interface which provides methods to do the actual booking. The method checkAndReserve inquires the price for a certain flight to a certain destination. Note that String, Price, Airline and Destination are *data types* and represent immutable data instead of objects. The agent object implements the Agent interface. For reference types, the type annotations N and F provide enhanced type information, namely whether references of this type point to near or far objects (i.e., objects in the same COG or not). For example, the session objects returned by the server are in the same COG as the server, which the type Session^{N} illustrates for the createSession method. This enhanced typing information can either be manually specified or automatically inferred by the ABS tools [17]. The architecture in Fig. 1 is configured in a *main* COG (that is the reason why all references are far (F)):

```
1  Server^F s = new cog ServerImpl();
2  Agent^F a = new cog AgentImpl(s);
3  Fut<View^F> vfut = a!getView();
4  View^F v = vfut.get;
5  new cog User(v);
```

The server is passed to the agent in the constructor (line 2). The view is obtained from the agent and passed to the user to enable the interaction with the agent. Asynchronous calls like the call to getView directly return a future. The value of a future is accessed by **get** (line 4) which blocks execution until the futures is resolved. Note that the agent is a component with a bipartite context: It interacts with the user and the server COG.

Users are modeled as nondeterministic and active components. In ABS, active components are described by classes with a run method which is automatically called when a new COG is created. Thus, user behavior is described in the run method of class User (see below). A user looks at the view content (calls viewContent) and at the number of buttons (calls getNbt) and then randomly clicks on one of the available buttons (line 9). The lines 5, 7, and 9 are asynchronous method calls and represent the communication from the user to the agent. The user waits for the futures to the first two calls to be resolved at line 6 and 8, representing the communication from the agent to the user.

```
1  class User(View^F v) {
2     Unit run() {
3        Bool abort = False;
4        while( ~abort ) {
5            Fut<String> contentfut = v!viewContent();
6            String content = contentfut.get; // look at content
7            Fut<Int> crtNmbBtfut = v!getNbt();
8            Int crtNmbBt = crtNmbBtfut.get;
9            if ( crtNmbBt > 0 ) { v!clickOn(random(crtNmbBt)); } else { abort = True; }
10 } } }
```

The agent COG includes the View objects (see Fig. 1 and below). When a user calls a method on the GUI object, the GUI object delegates the call to the current view (e.g., AirlineView). This specific view can then call the Agent (e.g., slctAirline, slctDestination or buyOffer) which might lead to change the current view (e.g., gui.setView(destinationView) in line 17). Initially, the GUI object delegates to the AirlineView object, which allows the user to select an airline. After the selection (call of the method clickOn), the AirlineView object calls slctAirline method on the AgentImpl object, which then tells the GUI object to *switch* to the DestinationView. After buying a ticket, the connection to the server is closed and the view is changed to the logoffView that does not react to user inputs.

```
class AgentImpl(Server^F myserv)
             implements Agent, ... {
 View^N aView;
 View^N dView;
 GUI^N gui;
 Session^F session;
 ... {
  aView = new AirlineView(this);
  dView = new DestinationView(this);
  ...
  gui = new GUIImpl();
  gui.setView(aview);
 }
 ...
 Unit slctAirline(Airline al) {
  slctdAl = al;
  gui.setView(dView);
 }
 ...
 Unit buyOffer() {
  Fut<Unit> f = session!buy();
  f.get;
  session!close();
  gui.setView(logoffView);
 }
}
```

We finish this section on modeling with summarizing the five aspects of the modeling technique that are important for our method:

1. The models should be read and written by software developers that might not master formal reasoning. They should be executable for validation.
2. The models should be sufficiently close to realistic implementations, particularly in reflecting the component structure and interfaces. This eases the conformance checking with implementations when they are not generated from the models.
3. The models should express the behavior of the software system and the users/environment in order to define and analyze the overall system behavior.
4. The models should allow for abstraction (e.g., in our example, we abstract from the details of GUI implementations).
5. To allow reasoning, the models need a precise formal semantics that also covers the concurrency aspects.

In the following, we consider evolution steps for components with bipartite contexts.

3 Evolution of Systems

Evolvable systems must be open to change. Often, new component versions should not change the overall behavior of the system. For example, we might want to change the implementation of the agent but still guarantee that the same kind of flight booking operations are possible. Compatibility of the new agent implementation with the old one then means that the *observable* behavior of the system remains the same.

There are different ways to define what should be considered as observable. If we consider all interactions of the agent as its behavior, we could not modify the GUI, because GUI modification in general changes the interactions with the user. Thus, we focus on the communication with the underlying system, namely on the communication between the agent and the server component. This is the communication that leads to the

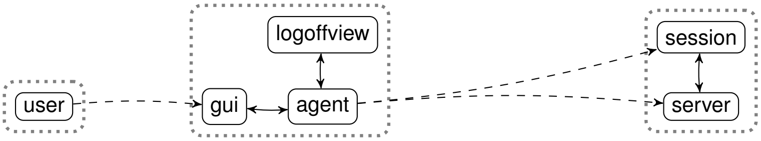

Fig. 3. Runtime structure of ABS model after evolution

actual flight bookings. More precisely, we allow new components to change the views (different content and buttons) and the way the views are presented and how they react to button clicks. But, we want to guarantee that every behavior at the interface to the underlying system that could be achieved with the old component can also be achieved with the new version. We formalize behavior as traces, that is, sequences of interactions.

Definition 1 (Backward compatibility). *Let U be a nondeterministic user model, C be an application component with a GUI, and D be a component such that Sys=(U,C,D) is a closed system. A component C' with a GUI is* backward compatible *with C if Sys'=(U,C',D) is a closed system and the traces between C and D in Sys are a subset of the traces between C' and D in Sys'.*

To illustrate this definition by our example, let us consider a second implementation of the agent sketched in Fig. 3. For this implementation, the airline view and destination view are combined into a single view that is directly implemented by the agent object itself. This means that the agent class also implements the View interface:

```
class AgentImpl(Server^F myserv)
        implements Agent, View ... { ...
  { ...
    gui = new GUIImpl();
    gui.setView(this);
  } ...
}
```

The new view allows the selection of airlines and destinations in one view and in any order[3]. Thus, the user interactions in the two versions are very different. Nevertheless, the new implementation should allow users to make the same bookings as in the old version. Thus, the new version is backward compatible,although the communication between the user and the agent component is very different. To make this more tangible, consider a concrete trace of events between the agent and the server s:

```
⟨ f1 = s!createSession(), f1!(sess), f2 = sess!checkAndReserve(al,dest),
  f2!(price), f3 = sess!buy(), f3!(), sess!close() ⟩
```

Here, f!(x) denotes the resolution of future f by value x. One can now see that the order in which the airline al and the destination dest have to be selected is irrelevant for the trace because these choices are transmitted by only one message. So this trace will be a trace for both the old and the new version of the system.

[3] For brevity, we do not show the complete ABS description of it.

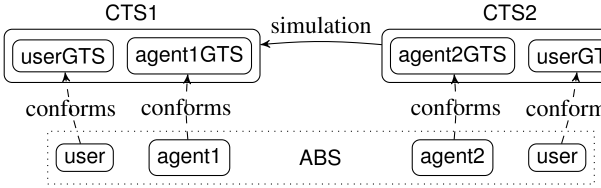

Fig. 4. Reasoning approach

4 Reasoning Approach

At the end of the last section, we postulated that two agent components are backward compatible. In this section, we describe our approach to checking compatibility for such components. The approach is based on three well-known techniques:

- Finite representation of behavior by rule-based transition systems
- Composition of transition systems
- Proving compatibility by using simulation

The ABS language makes it easy to read and write behavioral models of concurrent and distributed software systems. A logic to prove properties about ABS models is under development ([4] presents such a logic for a subset of ABS). As there is no logic to directly prove the compatibility of two COGs, we use a two-step approach:

1. First, we represent the behavior of a COG by a suitable kind of transition systems that we call *group transition systems* (GTS). Logics such as the one in [4] allow us to prove that a GTS faithfully represents the semantics of the corresponding COG.
2. Second, we compose and enclose the GTSs into shells called *component transition systems* (CTS) in order to produce the trace semantics of the component. Then we use composition and simulation techniques for CTSs to reason about compatibility.

Figure 4 illustrates the approach using the flight-booking example: For the ABS model of the user and the two agent models agent1 and agent2, we derived the GTSs userGTS, agent1GTS, and agent2GTS, respectively. We compose each agent GTS with the user GTS to produce CTS1 and CTS2, and show that CTS2 simulates CTS1 in the context of the underlying server. The following subsections explain these steps in more detail.

4.1 Group Transition Systems

GTSs are special labeled transition systems to abstractly represent the behavior of COGs written in ABS. A COG processes an incoming message by executing a corresponding method. The execution outputs a number of messages until the method reaches its end or the COG needs to wait for a future to be resolved. The way a COG processes an incoming message also depends on its current state. Thus, a GTS state should contain an abstract representation of the internal COG state and a bag of incoming messages that the COG must process. The transitions represent an interleaving semantics of the COG, matching nicely the asynchronous nature of the messages. Transitions are labeled by the outgoing messages that the component produces in that transition. As such, the incoming message being processed is obtained from the state information. In general, the state space of GTSs is infinite. To specify GTSs finitely, we utilize first-order logic.

The GTS is based on two sets, namely $\mathbf{O}$ as the set of all object and future instances, or simply names, and $\mathbf{M}$ as the set of all messages that can be produced. A message can be either an asynchronous method call $\mathsf{o!mtd}(\overline{p})$ to object o calling method mtd with parameters $\overline{p}$ or a future resolution $\mathsf{f!(v)}$ of the future f with value v. In asynchronous method calls, the last parameter is a future name that is used to return the result of the method call. Future names are freshly produced by the sender. Given a message m, the function $target(m)$ extracts the target object o or future f from the message. In this paper, we assume that futures are not passed as parameters and that all COGs are created during program start up. In particular, we do not consider messages for dynamic COG creation.

Definition 2 (GTS). *A* group transition system *is a quadruple* $T = (L, S, R, s_0)$ *where*

- $L \subseteq \mathbf{O}$ *is the set of object and future names local to the group,*
- $S \subseteq Bag(\mathbf{M}) \times \mathbf{O} \times Q$ *is the set of states consisting of a message bag, a set of exposed local names and the set of (abstract) local states,*
- $R \subseteq S \times Bag(\mathbf{M}) \times S$ *is a transition relation describing the processing of a message in the incoming message bag by the group, and*
- s_0 *is the initial state.*

The message bag stores the incoming messages that the COG is yet to process. Each transition is labeled with a bag of output messages, sent by the COG when it processes a message. We write $(M,q) \xrightarrow{Mo} (M',q')$ to represent a transition in R, where M, M' and Mo are message bags, and q and q' are local states. The locality of the objects and futures can be guaranteed using the ownership type system as mentioned in Sect. 2.2.

To ensure that a GTS captures the behavioral properties satisfied by all ABS COGs, we enforce a simple *well-formedness* criterium on the states and relations. For this purpose, we need a projection function $M{\downarrow_L}$. This projection function on a message bag M with respect to local names L produces the message bag $M_L \subseteq M$ where each message is targeted to some local object in L or a future resolution of a future in L.

Definition 3 (Well-formed GTS). *A GTS* $T = (L, S, R, s_0)$ *is* well-formed *if*

1. $\forall (M,q) \in S \bullet M{\downarrow_L} = M$*, and*
2. $\forall (M,q) \xrightarrow{Mo} (M',q') \bullet \exists m \in M \bullet M' = M \cup Mo{\downarrow_L} - \{m\}$[4].

The first item states that the message bag contains only incoming messages. The second states when a transition is taken, the processed incoming message m is taken out from the message bag, while messages produced by the COG directed to the COG are added to the message bag. In other words, every transition of a GTS is a reaction to a method call or a future resolution. Thus, GTSs can represent the behavior of reactive COGs and active COGs that receive messages from other COGs (like User), but they cannot model active COGs that generate infinitely many messages without expecting any response.

We use rules of the form $m_{in} : P \longrightarrow P' \succ Mo$ to describe the transition relation of a GTS where:

[4] We take the union operator on bags as adding all elements from one bag to the other and the difference operator as removing corresponding elements from one bag.

- m_{in} is an incoming message contained within the message bag of the COG before the transition occurs and $target(m_{in})$ is in the set of local names L,
- P and P' are boolean expressions over the local state and the message parameters,
- Mo is the bag of outgoing messages resulting from the transition. For each outgoing asynchronous method call, there is always a future created by the component which is represented by **new** f.

A transition $(M,q) \xrightarrow{Mo} (M',q')$ satisfies a rule if $M' = M \cup Mo{\downarrow}_L - \{m_{in}\}$, P evaluates to true for the current state q and the parameters of m_{in}, P' evaluates to true for the post-state q' and the parameters of the messages in Mo. The rules describe the largest transition relation R where each transition satisfies at least one of the rules. Moreover, the transition relation is such that each future name is created fresh. In GTSs that are consistent with ABS models, futures will be resolved at most once.

As an example, let us take a look at the User COG. Users have an application-independent behavior. Local states are pairs of a control state and the GUI reference v. The user looks at the view content and sees a number of buttons. Then, he clicks on some random button (i.e., the nb-th button), unless no buttons are present, indicating the end of the interaction. In the expressions, we use special variable \$ to denote the local state:

$$\begin{array}{llcll}
\text{u!run()} & : \$=(\text{u0,v}) & \longrightarrow & \$=(\text{u1,v}) & \succ \text{v!viewContent}(\textbf{new}\ \text{f1}) \\
\text{f1!(s)} & : \$=(\text{u1,v}) & \longrightarrow & \$=(\text{u2,v}) & \succ \text{v!getNbt}(\textbf{new}\ \text{f2}) \\
\text{f2!(n)} & : \$=(\text{u2,v}) \land \text{n}>0 & \longrightarrow & \$=(\text{u3,v}) \land (0 \leq \text{nb} < \text{n}) & \succ \text{v!clickOn}(\text{nb},\textbf{new}\ \text{f3}) \\
\text{f3!()} & : \$=(\text{u3,v}) & \longrightarrow & \$=(\text{u1,v}) & \succ \text{v!viewContent}(\textbf{new}\ \text{f1}) \\
\text{f2!(n)} & : \$=(\text{u2,v}) \land \text{n}=0 & \longrightarrow & \$=(\text{u4,v}) & \succ \epsilon
\end{array}$$

For the overall understanding of our method, the details of the GTS construction are not so important. Important are the following three points:

1. GTSs are appropriate for formal analysis, but are not a good language for designing realistic software models (see the requirements mentioned at at the end of Sect. 2).
2. The general theory to verify that a COG, considered as a program, conforms to a GTS, considered as a specification, is available. A specialization of the theory to the particular setting considered here is under development (cf. [4]).
3. Techniques for composition and compatibility checking of transition systems are in general well-developed. A specialization to our setting will be discussed next.

4.2 Component Transition Systems

We introduced GTSs as faithful representations of COGs. In the following, we construct transition systems that exhibit exactly the observable traces that we need for compatibility checking. In particular, they allow us to prove trace inclusion by simulation methods. Technically, we construct a *component transition systems* CTS_T from a set T of GTSs. CTS_T hides internal messages. By composing two CTSs C_1 and C_2, written as $C_1 \mid C_2$, we obtain a closed system that generates the traces at the boundary between C_1 and C_2. Construction and composition can be fully automated.

To illustrate the approach, let us again consider our example with GTS_U being the GTS for the user, GTS_A and $GTS_{A'}$ for the first and second version of the agent, and GTS_D for the underlying system. Compatibility checking is then realized as follows:

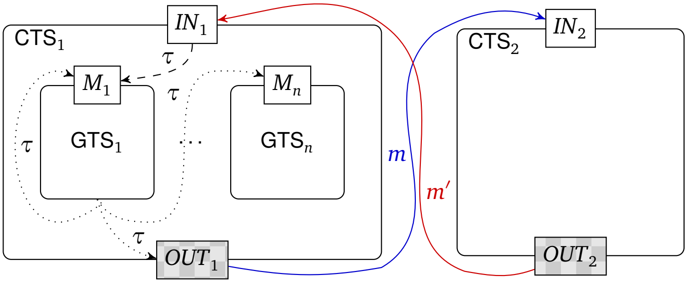

Fig. 5. Component transition systems and their composition

- construct $CTS_{\{U,A\}}$ from GTS_U and GTS_A ,
- construct $CTS_{\{U,A'\}}$ from GTS_U and $GTS_{A'}$,
- construct $CTS_{\{D\}}$ from GTS_D , and
- check that $CTS_{\{U,A'\}} \mid CTS_{\{D\}}$ simulates $CTS_{\{U,A\}} \mid CTS_{\{D\}}$.

In summary, construction essentially puts GTSs together and hides internal messages. Composition yields an executable system in which traces at the boundary of the two[5] components are observable, allowing the checking of compatibility as defined in Def. 1. Construction and composition are illustrated in Fig. 5.

Definition 4 (CTS construction). *Let $T = \{T_1, \ldots, T_n\}$ be a set of GTSs where $T_i = (L_i, S_i, R_i, s_{0i})$, $i \in \{1, \ldots n\}$ and L_i pairwise disjoints. Let $s_i = (M_i, q_i), s'_i = (M'_i, q'_i)$ be states of T_i. The* component transition system *of T is $CTS_T = (L, S, R, s_0)$ where*

- $L = \bigcup_{i=1}^{n} L_i$, $\qquad S \subseteq Bag(\mathbf{M}) \times S_1 \times \ldots \times S_n \times Bag(\mathbf{M})$, $\qquad s_0 = (\emptyset, s_{01}, \ldots, s_{0n}, \emptyset)$,
- $R \subseteq S \times (\mathbf{M} \cup \{\tau\}) \times S$ *is the smallest relation such that*
  1. $\forall i, s_i \xrightarrow{Mo}_i s'_i \bullet (I, s_1, \ldots, s_n, O) \xrightarrow{\tau} (I, s'_1, \ldots, s'_n, O \cup Mo{\downarrow}_{\mathbf{O}-L})$ *where* $\forall j \neq i \bullet s'_j = (M_j \cup Mo{\downarrow}_{L_j}, q_j)$;
  2. $\forall m \bullet \forall (I \cup \{m\}, s_1, \ldots, s_n, O) \in S \bullet (I \cup \{m\}, s_1, \ldots, s_n, O) \xrightarrow{\tau} (I, s'_1, \ldots, s'_n, O)$ *where if* $target(m) \in L_i$ *then* $s_i = (M_i \cup \{m\}, q_i)$ *otherwise* $s_i = s'_i$;
  3. $\forall m \bullet \forall (I, s_1, \ldots, s_n, O \cup \{m\}) \in S \bullet (I, s_1, \ldots, s_n, O \cup \{m\}) \xrightarrow{m} (I, s_1, \ldots, s_n, O)$.

Similar to GTS, a CTS is described by a set of local names L, a set of states S, a labeled transition relation R and an initial state s_0. L is a union of all local names of the GTSs. S includes not just the states of all GTSs, but also two sets of message bags IN and OUT, which act as input and output ports of the component, respectively. s_0 is the empty message bags with the initial states of the GTSs. The labels on R can be a message m that is transmitted out of the CTS or an internal transition τ. The relation is built from the transition relation of the GTSs in the following way. For each transition in some T_i (Case 1), we insert a relation labeled with τ to R by distributing the output messages Mo to the corresponding message bags. If a message $m \in Mo$ is targeted to some local name

[5] Composition can be easily generalized to a finite set of components.

in L_j, that message is inserted into the incoming message bag M_j. Otherwise, m goes to the output port *OUT* of the CTS, as portrayed in the figure by the dotted lines. The second rule (Case 2) states the nondeterministic distribution of messages in the input port to the appropriate incoming message bag. As this is a hidden step, it is labeled with τ (represented by the dashed line). The last rule (Case 3) dispenses nondeterministically the messages in the output port one by one as long as *OUT* is not empty. This message sending is visible from outside of the component, thus the transition is labeled with the corresponding message m (the solid line in Fig. 5). This treatment of output guarantees that outputs from different enclosed GTSs can be interleaved.

To represent CTSs, we use the same rule format as for GTSs. For example, the transition corresponding to the button click in $CTS_{\{User,Agent\}}$ can be formulated as follows where the underscore is a wildcard for any agent state:

```
f2!(n) : $=((u2,v),_) ∧ n>0  ⟶  $=((u3,v),_) ∧ (0≤nb<n) ≻ v!clickOn(nb,new f3)
```

A CTS represents an open system that needs a context in which it can work. We use another CTS as a context and observe the communication traces between them:

Definition 5 (CTS composition). *Let* $C_i = (L_i, S_i, R_i, s_{0i})$ *for* $i = \{1,2\}$ *be two CTSs. The* composed *CTS of* C_1 *and* C_2 *is* $C = (C_1 \mid C_2) = (L, S, R, s_0)$ *where*

- $L = L_1 \cup L_2, \qquad S \subseteq S_1 \times S_2, \qquad s_0 = (s_{01}, s_{02}),$
- $R \subseteq (S_1 \times S_2) \times (\mathbf{M} \cup \{\tau\}) \times (S_1 \times S_2)$ *such that*
  1. $\forall s_1 \in S_1 \bullet \forall s_2 \xrightarrow{\tau}_2 s_2' \bullet (s_1, s_2) \xrightarrow{\tau} (s_1, s_2'),$
  2. $\forall s_2 \in S_2 \bullet \forall s_1 \xrightarrow{\tau}_1 s_1' \bullet (s_1, s_2) \xrightarrow{\tau} (s_1', s_2),$
  3. $\forall (I_2, s_{21}, \ldots, s_{2n}, O_2) \in S_2 \bullet \forall s_1 \xrightarrow{m}_1 s_1' \bullet$
     $(s_1, (I_2, s_{21}, \ldots, s_{2n}, O_2)) \xrightarrow{m} (s_1', (I_2', s_{21}, \ldots, s_{2n}, O_2))$ *where*
     $I_2' = I_2 \cup \{m\}$ *if* $target(m) \in L_2$, *otherwise* $I_2' = I_2$, *and*
  4. $\forall (I_1, s_{11}, \ldots, s_{1n}, O_1) \in S_1 \bullet \forall s_2 \xrightarrow{m}_2 s_2' \bullet$
     $((I_1, s_{11}, \ldots, s_{1n}, O_1), s_2) \xrightarrow{m} ((I_1', s_{11}, \ldots, s_{1n}, O_1), s_2')$ *where*
     $I_1' = I_1 \cup \{m\}$ *if* $target(m) \in L_1$, *otherwise* $I_1' = I_1$.

The composition of two CTSs C_1 and C_2 corresponds to the interaction illustrated in Fig. 5. The local names L are combined from the respective components, the same with the states S and the initial state s_0. The transition relation R lifts up all internal transitions of the components (Cases 1, 2), and for each non-internal transition (Cases 3, 4), we update the input port of the corresponding component if the outgoing message is targeted to that component. The standard computation of the composed CTS provides the necessary ingredients for producing the traces of the components.

Definition 6 (Computation and trace of composed CTS). *A* computation *of a composed CTS* $C = (L, S, R, s_0)$ *is a sequence*

$$s_0 \xrightarrow{m_1} s_1 \xrightarrow{m_2} s_2 \xrightarrow{m_3} s_3 \ldots .$$

A trace *of the composed CTS is the sequence of the non-internal labels of a computation.*

For our flight-booking system, the traces of $CTS_{\{U,A\}} \mid CTS_{\{D\}}$ and $CTS_{\{U,A\}} \mid CTS_{\{D\}}$ are the ones we need for compatibility checking according to in Def. 1.

4.3 Checking Compatibility

As explained in Sect. 3, compatibility is defined based on the traces at the boundary between the combined user and application component on the one side and the underlying system on the other side. The CTSs give us a finite representation of the infinite trace sets and allow us to formally prove compatibility using simulation techniques (cf. [3]).

Essentially, we have to find a simulation relation and show that from two related states we can make a step in both systems and end up again in related states. Simulations for CTSs slightly deviate from the standard simulations because we have to account for the τ-transitions. We write $s \overset{m}{\Rightarrow} s'$, $m \neq \tau$, if there is a sequence of states $s_0 \dots s_n$ with $s = s_0$, $s' = s_n$, and an $i < n$ such that

$$s_0 \xrightarrow{\tau} \dots \xrightarrow{\tau} s_i \xrightarrow{m} s_{i+1} \xrightarrow{\tau} \dots \xrightarrow{\tau} s_n$$

Definition 7 (Simulation relation on composed CTSs). *Let $C_a = (L_a, S_a, R_a, s_{0a})$ and $C_b = (L_b, S_b, R_b, s_{0b})$ be two composed CTSs. We call $SR \subseteq S \times S'$ a* simulation relation on composed CTSs *iff*

- $(s_{0a}, s_{0b}) \in SR$*, and*
- $\forall (s_a, s_b) \in SR \bullet \forall s'_a \bullet\ s_a \overset{m}{\Rightarrow}_a s'_a$ *implies* $\exists s''_a, s'_b \bullet s_a \overset{m}{\Rightarrow}_a s''_a \wedge s_b \overset{m}{\Rightarrow}_b s'_b \wedge (s''_a, s'_b) \in SR.$

If there exists such a simulation relation, we say that C_b simulates *C_a.*

The initial states have to be in the simulation relation. If a pair (s_a, s_b) of states in C_a and C_b, respectively, is in the relation, then for every computation in C_a starting in s_a that emits a message m there must be corresponding computation in C_b emitting m such that they end up in related states. The following theorem states that we can use simulation to prove compatibility:

Theorem 1. *Let C_a and C_b be two composed CTS, Tr_a and Tr_b the set of traces of C_a and C_b, respectively. If C_b simulates C_a, $Tr_a \subseteq Tr_b$.*

Proof. We show by induction that for all $t \in Tr_a$, $t = m_1 \dots m_n$, of length n there are sequences of states $s_{0a} \dots s_{na}$ and $s_{0b} \dots s_{nb}$ with

$$s_{0a} \overset{m_1}{\Longrightarrow} \dots \overset{m_n}{\Longrightarrow} s_{na} \quad \text{and} \quad s_{0b} \overset{m_1}{\Longrightarrow} \dots \overset{m_n}{\Longrightarrow} s_{nb}$$

and $(s_{ia}, s_{ib}) \in SR$ for all $i \in \{0, \dots, n\}$. In particular, $t \in Tr_b$. This is obviously true for the empty trace and the induction step is directly obtained from the definition of simulation relations.

For our flight-booking system, the simulation relation can include the initial state pairs, the state pairs when the session is requested, the session is returned, checking and reserving a route, getting the price, buying the ticket and closing the session. All the internal state changes of the different versions are hidden.

5 Related Work

Several techniques for modeling the behavior of object-oriented systems are available (e.g., VDM++ [5] or Object-Z [12]). ABS has the advantage that it is easier to handle for programmers, that it is executable, and that it supports a important form of concurrency.

In contrast to the novel approach taken here, where we use an abstract model to reason about compatibility (see Sect. 4). Compatibility or equivalence of components has also been studied directly at the code level [7, 9, 15]. Using our sound and complete simulation techniques developed in [15] which relate two different implementations, we have made first steps towards automated verification of compatibility for object-oriented libraries [16]. We believe that using a more abstract model like the one in Sect. 4 can further improve the level of automation.

GTS and CTS are a variant to the well-known concept of labeled transition systems tailored to the Actor model [1]. Having all outgoing messages as the label of a single transition in GTS is similar to the big-step semantics of Specification Diagrams [13] without the internal operations. Because the states contain incoming message bags, there is no need to synchronize the messages in the composition. Furthermore it allows lifting up the specification of the subcomponents' GTS to form the specification of the CTS.

Another way to obtain a trace-based interpretation of GTS is by building an independent relation between messages caused by transitions which is produced by different COGs during the composition of GTSs. One can then extract trace equivalence classes that represent the complete behavior of the system as shown in [14]. Coming up with this independent relation, however, is not a trivial task.

6 Conclusion and Future Work

Software maintenance and evolution steps modify some properties of a system and should maintain others. In this paper, we presented a method to reason about modifications of components that have both an interface to users (or an environment) and an interface to other system parts. Whereas changes at the interface to the users should be allowed, we wanted to maintain the behavior at the interface to other system parts. Our approach addresses three challenges:

1. Modeling of user behavior, i.e., of behavior that is not represented by software
2. Abstraction from technical complexities such as GUI frameworks
3. Reasoning about compatibility of two component versions

The first two challenges were met by using the behavioral modeling language ABS. To solve the third challenge, we developed CTS, a special form of transition systems. A CTS finitely represents the semantics of ABS components and is suitable for composition and verifying compatibility of components using simulation proofs.

Our central goal for the future is to develop tools supporting the presented method. We would like to use model mining techniques [2] for automating the constructing of the GTS for a COG. Where full automation is not possible, we need verification support to prove conformance. In addition, we need tools helping with the simulation proofs.

References

1. Agha, G., Mason, I. A., Smith, S. F., Talcott, C. L.: A Foundation for Actor Computation. In: J. Funct. Program. 7.1 (1997), pp. 1–72.
2. Ammons, G., Bodík, R., Larus, J. R.: Mining Specifications. In: POPL. New York, NY, USA: ACM, 2002, pp. 4–16.
3. Baier, C., Katoen, J.: Principles of Model Checking. Vol. 950. MIT press, 2008.
4. Din, C. C., Dovland, J., Johnsen, E. B., Owe, O.: Observable Behavior of Distributed Systems: Component Reasoning for Concurrent Objects. In: Journal of Logic and Algebraic Programming 81.3 (2012), pp. 227–256.
5. Dürr, E., Katwijk, J.: VDM++, A Formal Specification Language for Object Oriented Designs. In: COMP EURO. IEEE, 1992, pp. 214–219.
6. Hewitt, C., Bishop, P., Steiger, R.: A Universal Modular ACTOR Formalism for Artificial Intelligence. In: IJCAI. 1973, pp. 235–245.
7. Jeffrey, A., Rathke, J.: Java Jr.: Fully Abstract Trace Semantics for a Core Java Language. In: ESOP. 2005, pp. 423–438.
8. Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: FMCO 2010. Vol. 6957. LNCS. Springer-Verlag, 2011, pp. 142–164.
9. Koutavas, V., Wand, M.: Reasoning About Class Behavior. In: Informal Workshop Record of FOOL. 2007.
10. Miller, M. S., Tribble, E. D., Shapiro, J. S.: Concurrency Among Strangers. In: TGC. Vol. 3705. LNCS. Springer, 2005, pp. 195–229.
11. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing Active Objects to Concurrent Components. In: ECOOP. Vol. 6183. LNCS. Springer, 2010, pp. 275–299.
12. Smith, G.: The Object-Z Specification Language. Kluwer Academic Publishers, 2000.
13. Smith, S. F., Talcott, C. L.: Specification Diagrams for Actor Systems. In: Higher-Order and Symbolic Computation 15.4 (2002), pp. 301–348.
14. Vasconcelos, V. T.: Trace Semantics for Concurrent Objects. MA thesis. Keio University, Mar. 1992.
15. Welsch, Y., Poetzsch-Heffter, A.: Full Abstraction at Package Boundaries of Object-Oriented Languages. In: SBMF. LNCS. Springer, 2011, pp. 28–43.
16. Welsch, Y., Poetzsch-Heffter, A.: Verifying Backwards Compatibility of Object-Oriented Libraries Using Boogie. In: FTfJP. Beijing, China, 2012.
17. Welsch, Y., Schäfer, J.: Location Types for Safe Distributed Object-Oriented Programming. In: TOOLS Europe. LNCS. Springer, 2011, pp. 194–210.