Engineering Support for UML Activities by Automated Model-Checking — An Example

Frank Alexander Kraemer, Vidar Slåtten, and Peter Herrmann

NTNU, Department of Telematics, N-7491 Trondheim, Norway.
{kraemer,vidarsl,herrmann}@item.ntnu.no

Abstract. In our approach for the engineering of reactive services, we specify systems as collaborations by means of UML 2.0 activities. In automated and correctness-preserving steps, the collaborative models are transformed into executable code. The semantics of the activities are defined using temporal logic. This formal fundament can be utilized to prove that the collaborations fulfill certain general well-formedness properties which can be verified by the model checker TLC. This is quite relevant since communication delays in the interactions between the participants realizing a collaboration aggravate the design of correct collaborative behavior. The well-known state space explosion problem of model checkers is mitigated by using special external state machines which define the interface behavior of sub-activities. The generation of the formal input for TLC from the activities is completely automated, so that the engineers working on the activities do not need to be experts in temporal logic and model checking. In this paper, we describe the utilization of TLC to detect and correct design errors by means of an example.

1 Introduction

In our engineering approach for reactive services SPACE [1–5], system specifications are composed of building blocks that model functionality related to a certain task. The building blocks are collaborations covering several components. In addition to the necessary interactions, they also define the local behavior of all the participating components. We use UML 2.0 activities to describe the behavior of collaborations. Activities can be divided into several partitions, each identifying the tasks of the individual participating components. Control flows are represented explicitly and may be synchronized by a number of control nodes. Moreover, activities can be decomposed into sub-activities, so that systems may be built from already existing building blocks.

Enabling entire collaborations as the structuring units of service specifications is beneficial in various respects. First, services usually involve several participating components. Describing them by collaborations gives a holistic view of the service which can be understood without combining all the component descriptions. Second, the degree of reuse is potentially higher since a collaboration solves only a certain subtask and is therefore more likely to be useful in

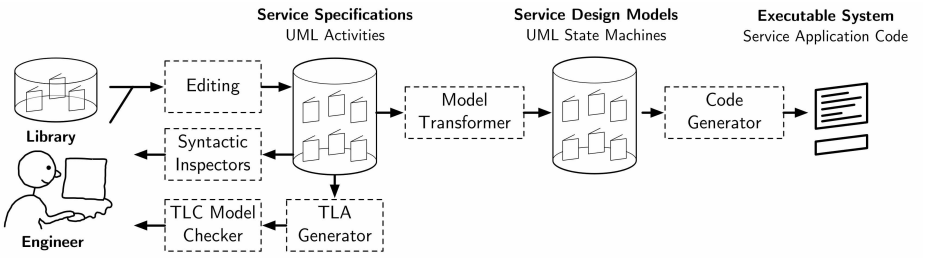

Fig. 1. Tool Support for the SPACE engineering approach

other applications than entire components that typically combine several tasks making them very specific (see for example [5]).

Figure 1 outlines the development process along with the tools supporting it. An engineer works on collaborative service specifications, using a library of reusable building blocks providing solutions to reoccurring problems. The building blocks can be composed together with additional "glue" logic using an editor for activities. For the execution of the services, however, descriptions of the system components are needed. We hereby follow a specification-driven approach, in which the service specifications composed of the collaborations are automatically transformed to component-oriented service design models in the form of UML 2.0 state machines, as described in [3]. This has the benefit that consistency between the different development stages is ensured, and engineers just have to maintain the service specifications. The state machines are then the input for our code generators that produce executable code for various platforms (see [4]).

For such an approach and its tools to be correct, formal reasoning is needed to guarantee that properties described by the individual collaborative building blocks are preserved by the composed system. Furthermore, the properties must be maintained through the model transformation to state machines and the implementation on the various execution platforms. For this, we use the compositional Temporal Logic of Actions (cTLA, [6]). We formalized both the service specifications in terms of activities [7] as well as the state machines [4]. The coupling principle of cTLA supports the property of superposition [8], in which properties of a part of the system (i.e., the individual building blocks) are also valid for the composed system. This makes it possible to map the composition of activities and state machines directly to the cTLA couplings. The model transformation and code generation correspond to refinement steps. Thus, we can use cTLA refinement proofs to verify that these steps are correct (see [3, 4]).

This approach is already beneficial for specification quality, since the abstraction level of the models is higher which allows for a better understanding of the behavior. Coding errors cannot be introduced due to the automatic translation. Nevertheless, the created models need to be correct as well. While some properties may be ensured by a purely syntactic analysis, others require us to consider entire behaviors, for example, that interface events of building blocks have to occur in a certain order. This is usually hard to guarantee manually as behav-

ior involving several components can get quite complex due to the unavoidable delays of the communication medium connecting them. To assure correctness of such behaviors, model checking (i.e., the examination of all reachable states a behavioral description implies) can be used. Model checking, however, needs a certain amount of expertise in formal reasoning, which we do not want to claim from the engineers using our approach. A possibility to overcome this situation is, as Rushby suggests in "Disappearing Formal Methods" [9], to wrap formal techniques within tools so that they are not seen as difficult anymore, and to increase their user-friendliness. The idea behind this is that a user does not necessarily need to understand the details of a formal technique and model-checking, if an automated checking tool gives understandable feedback addressing the problem in the language of the engineer's domain.

To follow such an approach, we developed in [10] an automatic transformation tool from UML 2.0 activities to TLA$^+$, the language of the Temporal Logic of Actions (TLA, [11]). For this language, the model-checker TLC [12] is available which can check a specification for various temporal properties that are stated in form of theorems. For each activity, we generate a set of theorems automatically which claim certain properties to be kept by activities in general. Examples for these properties are the correct usage of building blocks within the activity as well as that the activity itself satisfies a certain externally visible behavior. When TLC finds that a theorem is violated, it produces an error trace displaying the state sequence that leads to the violation. This trace can be given in terms of easily comprehensible token markings within an activity as well. So, an engineer using our tools does not have to write or understand the temporal logic formulas.

The presented approach for model checking makes use of the compositional nature of our service specifications. As described in [7], a system composed of collaborations guarantees the properties of the single collaborations to be maintained. This follows directly from the semantics based on cTLA [6] and the principle of superposition. The activities describing the complete behavior of collaborations may be specified in a more abstract form by means of special state machines that refer to externally visible events dedicated for composition. When model checking a composite specification, only the abstract specification has to be taken into account, which reduces the state space. Thus, we check each collaboration separately and do not consider the entire hierarchy which effectively mitigates the likelihood of state space explosions.

After discussing some related work done on formal checking of UML models, we give an introduction to temporal logic as well as the model checker TLC in Sect. 3. We proceed by introducing an example specification based on activities, and explain the semantics of activities in temporal logic in Sect. 4. Thereafter, we use our tools in Sect. 5 to develop an example, starting with a naive solution that gets corrected based on the feedback of the model checking. We close in Sect. 6 with some concluding remarks.

2 Related Work

Formal checks on UML models are done as part of OMEGA [13], FUJABA [14] and HUGO [15]. However, these approaches mainly concentrate on state machines or sequence diagrams, but not on activities as in our case. In [16], UML activities are translated into PROMELA, the input language for the SPIN model checker [17]. In [18], a mapping from UML 2.0 activities to Colored Petri Nets is described enabling the usage of Petri Net tools for analysis. In [19], UML activities are transformed into the π-calculus where safety and liveness properties can be expressed using the modal mu-calculus and checked using the MWB tool [20]. Eshuis [21] uses NuSMV, a symbolic model verifier to check the consistency of activity diagrams. The difference of these approaches to ours lies mainly on the domain that activities are used for and the chosen semantics. While they focus on activities more from a perspective of business processes assuming a central clock or synchronous communication, we need for our activities reactive semantics [7] reflecting the transmission of asynchronous messages between distributed components. This semantics enables us to generate the executable state machines defined in [4].

3 The Temporal Logic of Actions

Leslie Lamport's Temporal Logic of Actions (TLA, [22]) is a linear-time temporal logic in which semantics is expressed by infinite state sequences. The corresponding syntax is TLA$^+$ that enables describing system behavior by special state transition systems and additional fairness properties. Fig. 2 is an example of a TLA$^+$ specification. After a frame containing the module name (i.e., *HotelWakeUpSystem*), it uses the expression EXTENDS *Naturals* describing the import of a module including definitions, operators and axioms to model the natural numbers. The states of the state transition system are modeled by variables (here i, t, h and a) which are, in general, non-typed. The predicate *Init* specifies the set of values the variables shall have in the initial state. The transitions are described by actions each specifying a pair of a current state and its successor state. Here, the current state is referred to by variable identifiers in a simple form while the next state is modeled by primed variable identifiers. An example is the action *initial* which may be executed if the variable i has the value 1 and h has the value "off". After its execution, i will carry the value 0 which is described by $i' = 0$. In addition, h will have the value "started" in the following state while the two other variables a and t do not change their values during the execution of the action. The set of system transitions is modeled as the disjunction of the system actions which is expressed by the definition *Next*, the so-called next-state relation. The overall system description is modeled by the canonical formula *Spec*. The first conjunct of this temporal formula defines that the predicate *Init* holds in the first state of every state sequence modeled by *Spec*. The second conjunct uses the temporal operator $\Box$ ("always") specifying that the rest of the conjunct is valid in all states of all state sequences describing the behavior of the system. The TLA expression $[Next]_{\langle i,t,h,a\rangle}$ determines

```
MODULE HotelWakeupSystem
EXTENDS Naturals
VARIABLES i, t, h, a

Init ≜
  ∧ i = 1 ∧ t = 0
  ∧ h = "off" ∧ a = "off"

initial ≜
  ∧ i = 1 ∧ i' = 0
  ∧ h = "off" ∧ h' = "started"
  ∧ UNCHANGED ⟨a, t⟩

startAlert ≜
  ∧ h = "started" ∧ h' = "alerting"
  ∧ a = "off"  ∧ a' = "active"
  ∧ UNCHANGED ⟨i, t⟩

stopAlert ≜
  ∧ h = "alerting" ∧ h' = "stopped"
  ∧ a = "active" ∧ a' = "off"
  ∧ UNCHANGED ⟨i, t⟩

aborted ≜
  ∧ h = "stopped" ∧ h' = "off"
  ∧ t = 0 ∧ t' = 1
  ∧ UNCHANGED ⟨i, a⟩

confirmed ≜
  ∧ h = "stopped" ∧ h' = "off"
  ∧ t = 0  ∧ t' = 1
  ∧ UNCHANGED ⟨i, a⟩

timeout ≜
  ∧ t = 1 ∧ t' = 0
  ∧ h = "off" ∧ h' = "started"
  ∧ UNCHANGED ⟨i, a⟩

Next ≜
  ∨ initial  ∨ startAlert ∨ stopAlert
  ∨ aborted ∨ confirmed ∨ timeout

Spec ≜ Init ∧ □[Next]⟨i, t, h, a⟩

t0 ≜ □((i = 1) ⇒ (h = "off"))
t1 ≜ □((h = "stopped") ⇒ (t = 0))
t2 ≜ □((h = "started") ⇒ (a = "off"))
t3 ≜ □((h = "alerting") ⇒ (a = "active"))
t4 ≜ □((t = 1) ⇒ (h = "off"))
```

Fig. 2. TLA Module

that a state transition has to be either a stuttering step in which all variables listed in the subscript maintain their values or satisfies the condition *Next*. Thus, every state sequence begins with a state fulfilling *Init* and corresponds only to state transitions which either meet one of the system actions or are stuttering steps. Further conjuncts may be used to describe liveness properties by fairness assumptions on actions which, however, is not discussed in this paper.

The second paragraph of the specification contains a list of properties $t0$ to $t4$ which shall be kept by the system. As they all start with the always operator, they state invariant behavior (e.g., if variable i has value 1, h must be "off"). To verify an invariant, one has to prove that it holds in the initial condition *Init* and that it is preserved by every system action.

The compositional Temporal Logic of Actions (cTLA [6]) mentioned in the introduction is a derivative of TLA. It resolves a shortcoming of TLA which is limited to compositions based on joined variables [23]. In contrast, cTLA combines modules by defining joined system actions as simultaneously executed module actions which is a prerequisite for constraint-oriented models [24]. There, one specifies not single physical components but properties describing partial system behavior which spans several components. As the UML 2.0 collaboration and activity-based models used in our approach demand this particular specification style, we used cTLA instead of TLA to define their semantics [7]. cTLA uses a

Fig. 3. Reception Panel

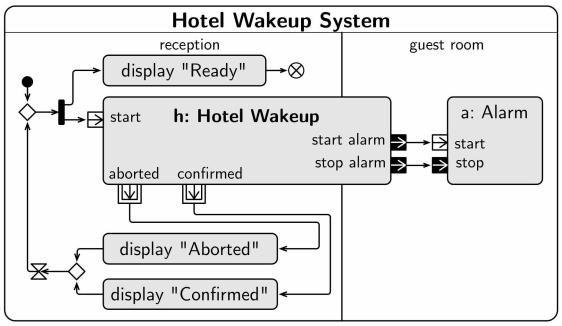

Fig. 4. Activity for the entire system

process-like specification style which encompasses both simple and compositional process descriptions. As the compositional process models can be transferred to simple ones (see [6]) and the simple processes are basically defined by the same canonical formulas as TLA$^+$, it was not a major problem to transform the UML activities to TLA$^+$ modules like the one depicted in Fig. 2. This is done by the tool introduced in [10] such that we can use the model checker TLC [12] to automatically prove that the activities fulfill certain properties since TLC uses TLA$^+$ specifications as input. TLC performs an exhaustive exploration of all reachable system states and verifies that invariant properties are maintained by every checked state[1]. In the case of a failure, a path of states leading to the one not fulfilling a property is shown which facilitates the search for the error and can be visualized in the UML activities.

4 UML 2.0 Activities in the SPACE Approach

In order to study an intricate problem in isolation, we consider a system to carry out wake-up alarms for guests of a hotel. The system is partly automated, as the requests for wake-up alarms are noted manually by the receptionist in a book. The guests prefer to be woken by an alarm instead of a direct phone call, to avoid contact with the personnel at an early morning hour. To convince the receptionist that they really are awake, they confirm the alarm by pressing a button. The reception has a control panel with two buttons and a display for each of the guest rooms, illustrated in Fig. 3. At wake-up time, the receptionist pushes the alert button which sounds the alarm in the guest room. If the guest confirms, the display shows *Confirmed* for some seconds so that the receptionist knows that the guest is actually awake. If the guest does not confirm, the receptionist can abort the alert after some time, upon which he or she may visit the room and rouse the guest with more drastic measures.

[1] For liveness proofs not introduced here, TLC checks sequences of states.

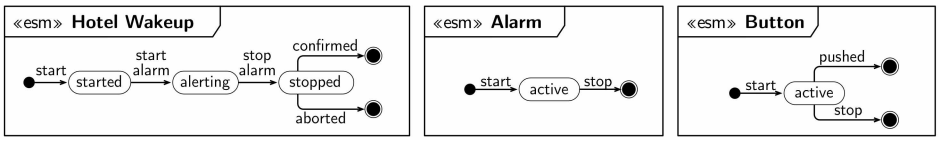

Fig. 5. External state machines

4.1 Informal Explanation of Activities

The behavior of the example system is described by the UML 2.0 activity shown in Fig. 4. It is divided into two activity partitions, one denoting the hotel reception and one for a guest room[2]. On the reception side, the activity contains three operations to control the display by printing the messages *Ready*, *Aborted* and *Confirmed*. On the side of the guest room, an alarm device is represented by a so-called *call behavior action*. This is a node that may refer to other activities (in the following referred to as *sub-activities*) and be used for decomposition. In the system here, we do not know about the internals of the alarm, just that it can be started by a token entering via *start* and stopped by a token via *stop*. Similarly, h refers to another activity realizing the protocol between the reception and the hotel room[3]. In contrast to the building block for the alarm, h spans over both activity partitions and as such describes a collaboration between the reception and the guest room.

The system activity starts on the side of the reception at the initial node. A token is emitted upon system startup and moved to a fork node, where it is duplicated. One of the tokens continues to operation *display Ready*, causing the display to show that the system is ready. Afterwards, it ends at a flow final node. The other token leaves the fork and moves into the call behavior action h via input pin *start*. This activates the *Hotel Wakeup* sub-activity. On this level, we just need to know about its externally visible behavior, described by the state machine *Hotel Wakeup* in Fig. 5. The stereotype ≪esm≫ applied to it marks that the diagram denotes an external state machine (ESM, [25]) for the sub-activity. Its transitions refer to the input and output pins of the corresponding sub-activity, describing in which sequence tokens may be passed. We see that after *start*, event *start alarm* will eventually happen, followed by *stop alarm*. Thereafter, the sub-activity terminates as either *aborted* or *confirmed*, depending on the behavior of the guest. On the side of the guest room, the flow leaving *start alarm* and *stop alarm* of h is connected to *start* resp. *stop* of the call behavior action a modeling the alarm. On the reception side, the display informs the receptionist about the outcome via two distinct display messages once sub-activity h terminates. As soon as the display messages *Confirmed* or *Aborted* appear, a timer is started waiting for a certain time, so that the message

[2] To keep the discussion simple, we only consider one room. Using the mechanisms presented in [1], this design can easily be expanded to multiple rooms.

[3] The decision to put the alarm and the display outside of the *Hotel Wakeup* h was here mainly to ease the presentation of the contents of h as shown in Sect. 5.

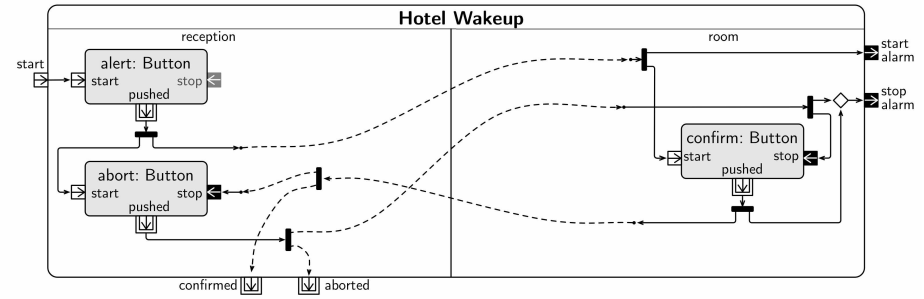

Fig. 6. Solution 1

can be read. Upon a timeout, the display is reset to *Ready* and the hotel wake-up can be used again.

A first (naive) solution for the internals of the call behavior action *h: Hotel Wakeup* is shown in Fig. 6. Note that the dashed lines are not a concept of UML activities but are here used to illustrate the preliminary state of the model which we will replace later, based on the findings of the model checking. The flows in solid lines remain stable throughout all solutions. The activity is composed from three buttons *alert*, *confirm* and *abort* from our library of reusable building blocks [26]. Their external behavior is described by ≪esm≫ *Button* in Fig. 5. There, a button is activated via *start*. In this state, it may be pushed by the user, which causes its termination via *pushed*. It may also be stopped by a token through *stop*, whereupon any pushes by the user are ignored.

When the *Hotel Wakeup* collaboration is started, the alert button is activated immediately. Once it is pressed, a token is emitted via *pushed*, activating the abort button. At the same time, the flow continues towards the partition for the guest room. As the partitions will be implemented by different, physically remote components, we assume a buffered communication between activity partitions. Therefore, a token waits for an arbitrary time in a virtual queue place where a flow crosses partition borders. This corresponds to the transmission through a physical medium. When the flow from the alert button is received by the guest room partition, the confirm button is activated, and a token is branched off towards the output node *start alarm* to notify the alarm device. If the confirm button is pushed, the alarm is stopped via output node *stop alarm* and a confirmation is routed back to the reception partition where the collaboration terminates via output pin *confirmed*. If the receptionist presses the abort button, the guest room is notified to switch off the alarm and the confirmation button, and the collaboration is terminated via *aborted*.

4.2 Semantics of UML 2.0 Activities in Temporal Logic

Formally, UML activities are based on Petri Nets and describe as such a state transition system. In [7] we defined the semantics of activities in terms of cTLA, which can be easily mapped to TLA$^+$, the input language for the model checker TLC, as discussed in Sect. 3. The transformer from UML 2.0 activities to

TLA$^+$ [10] uses UML activity models stored in the UML2 repository of Eclipse as input. Roughly speaking, the tool maps each token movement of an activity to an action in a TLA$^+$ formula in which stateful nodes such as timers, sub-activities, joins and accept signal actions are represented by their own variables. The buffering of flows that cross activity partitions is formalized by queue variables which are bags of tokens. Whenever a token leaves a source partition, it is added to the corresponding queue place. In a second action, it is removed from the queue place and continues the flow in the target partition.

As an example, the specification in Fig. 2 displays the TLA$^+$ code generated for the system activity depicted in Fig. 4. It consists of six actions[4], each modeling a token movement. The module declares a variable for each stateful node of the activity, that is, the initial node by variable i, the timer by t as well as the sub-activities for the wake-up h and the alert a. For both the timer and the initial node, we use simply an integer to store the number of tokens that are resting in them. Initially, there is one token in the initial node (which means the activity is ready to start) and no token in the timer (i.e., the timer is idle). This is expressed with the initial predicate *Init* by $i = 1 \wedge t = 0$. The variables for the sub-activities store the current states of the ESMs that represent their externally visible behavior. Initially, both ESMs are in their initial state, so that value "off" is assigned to h and a by *Init*. The six actions model the token movements within the activity. Action *initial* specifies the start of the activity. The token resting in the initial node is removed from it ($i' = 0$) and enters h via input pin *start*. The ESM of h (according to its definition in Fig. 5) makes a transition to state *started*[5]. When h is in state "started", action *startAlert* is enabled. It models the emission of a token from h via *startAlert* activating the alarm ($a' =$"active"). Eventually, the alarm will be deactivated again by the execution of *stopAlert*. After that, the two actions *aborted* and *confirmed* are enabled, modeling the termination of sub-activity h (by $h' =$"off"). Due to the merge node, both of these actions start timer t (by $t' = 1$), enabling action *timeout*, which restarts sub-activity h.

4.3 Theorems for Well-Formed Activities

An important property of our activity specifications is that the events of the sub-activites are invoked in the order specified by their ESMs. This means for example that, whenever a token attempts to enter *start* of sub-activity h, then h must not yet be activated, i.e., $h =$"off". A token can be released from the initial node whenever it has a token, i.e., $i = 1$. So, we want to be sure that whenever there is a token in the initial node, the sub-activity is not yet active. Formally, this is an implication $i = 1 \Rightarrow h =$ "off". As this property must always hold, our tool writes the theorem as an invariant $t0 \triangleq \Box((i = 1) \Rightarrow (h =$ "off")). The further theorems describe the other cases in which the ESM of a sub-activity

[4] We adjusted the automatically chosen variable and action names for readability.

[5] The token is further forked into operation *display Ready*, which we can ignore here since no stateful node is reached.

must not be violated by its environment. For example, *t4* ensures that whenever the timer is active ($t = 1$), sub-activity h may be started again ($h =$ "off"). The violation of ESMs is only one major source for errors. Thus, the current transformation tool also writes theorems to check the boundedness of queues as well as assertions on the execution of operations that can be added with additional stereotypes [10]. This is, however, not discussed here.

5 Developing and Model Checking the Example

The use of model checking to correct activity-based service specifications is outlined by discussing the improvements of the hotel wakeup system. We start by applying our transformation tool and create the TLA$^+$ specification of the system activity listed in Fig. 4. The outcome is the TLA module introduced in Fig. 2 which is checked by TLC. The model checker notifies that 5 distinct states were generated and that no errors were found. Given the theorems that are included in our automatically generated formal specification, this means that the contracts of the used building blocks h and a are obeyed. Thus, we can proceed by checking the design of the *Hotel Wakeup* activity.

5.1 Solution 1: A Naive Start

As an initial solution, we consider the activity introduced in Fig. 6. On a first glance, it looks quite straightforward. When the alarm button is pushed, the guest room is notified to activate the confirmation button. A push on their button by either the receptionist or the guest stops the alarm and the respective other button. However, when we model check this activity, TLC says that temporal properties are violated and prints a trace of states that describes the behavior up to the moment when the violation took place. This trace may be projected onto the activity, as illustrated in Fig. 7. Hereby, the transfer queues are shown as token places where the flows cross partitions, and the activity and its sub-activities are amended with boxes showing the current state of their ESM.

State 1. The activity is not yet active and its ESM is in state *off*. The queues *a*, *b* and *c* are empty, and all sub-activities are in state *off* as well.

State 2. A token was moved via the input node of the activity and activated the alarm button, which is now in state *active*.

State 3. After the alarm button was pressed, a token was forwarded into queue *a* and the abort button is now active. In this state, TLC reports that a theorem is violated. This theorem states that whenever the abort button is active (and may therefore emit a token at any time), the ESM of Hotel Wakeup is in state *stopped*, as an outgoing token from *abort* would pass through parameter node *aborted* (see Fig. 5). So, in the current state, the active abort button could terminate the entire activity through flow *x1* and contradict the ESM. In practice this means that the system using *Hotel Wakeup* could assume the alarm to be aborted after the abort button was pressed, although the alarm was never

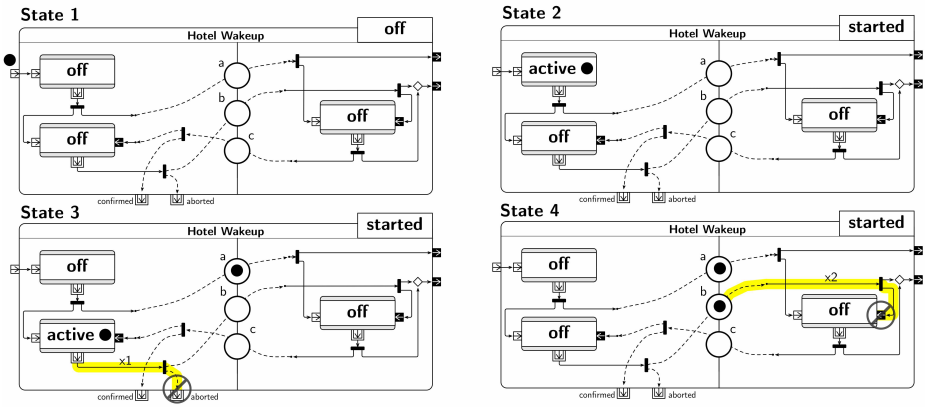

Fig. 7. Error trace of solution 1

started. To check for further errors before a redesign, the tool allows us to ignore this error for a moment and let the abort button be pushed.

State 4. By pushing the abort button, a token was emitted via *aborted* and another one is placed in queue *b*. In this state, the guest room may decide to consume the token from queue *b*, which would then be moved via *x2* into the confirm button that is in state *off*, which is is against the ESM of the button (see Fig. 5). Obviously, the activity in Fig. 6 does not regard that due to the transfer medium, an abort flow may overtake the alarm flow.

5.2 Solution 2: Improved Version with a Sequencer

The problem found in state 4 of solution 1, where the confirm button could be stopped before it was even started, can be solved by adding a building block of type *Sequencer* from our library [26] to the new activity in Fig. 8. It controls two flows arriving in any order at *i1* and *i2* such that their respective outputs may only happen in the order *o1* followed by *o2*. The problem found in state 3 of the previous solution, according to which the ESM of Hotel Wakeup was violated, can be solved by an additional flow *f* that returns from the hotel room after the alarm was started. A new run of TLC on the activity in Fig. 8 reveals, however, that there are still flaws in the system. Figure 9 shows the new error trace. The two first states are omitted as they correspond to the ones of Fig. 7.

State 3. The alert button has been pressed and a token is waiting to cross from partition *reception* to partition *room* in queue *a*. The abort button has also received a token and is in state *active*.

State 4. The token waiting in queue *a* has passed through the sequencer and activated the confirm button. The token was also forked so that a copy left the activity via *start alarm* causing the ESM of Hotel Wakeup (Fig. 5) to change from *started* to *alerting*. Both buttons are now waiting to be pushed.

State 5. The confirm button has been pushed sending a token via *stop alarm* changing the state of the ESM to *stopped*. The token was also forked into the

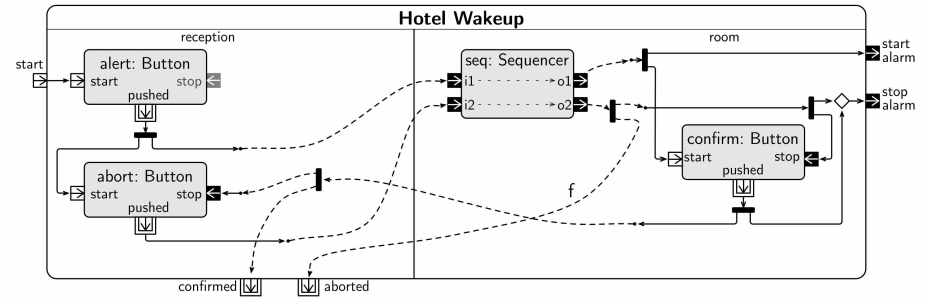

Fig. 8. Solution 2

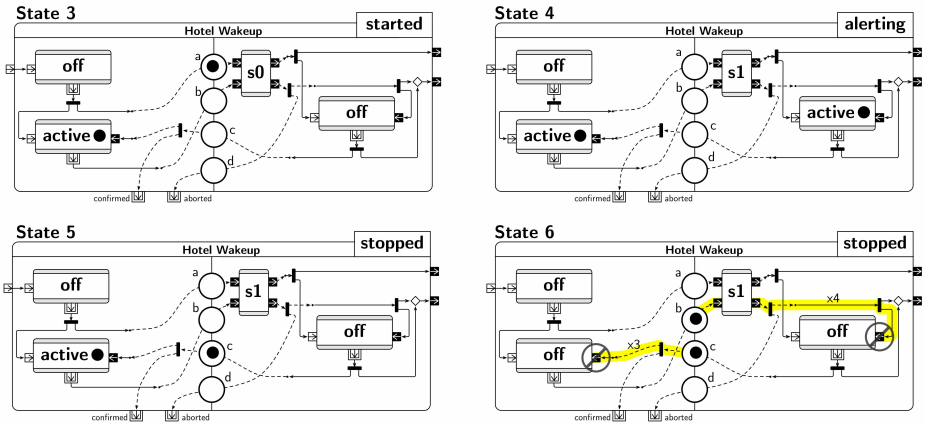

Fig. 9. Error trace of solution 2

queue c where it is waiting to enter the reception partition. The confirm button has returned to state *off*.

State 6. The receptionist pushed the abort button, which switched to *off* and emitted a token into queue b, so that there is now one token in each of the queues b and c. This harms, however, two theorems that protect the contracts of the buttons. The confirm and stop button are both in state *off*, but tokens are placed in the queues that flow into the stop pin of the buttons via flows *x3* and *x4*, which would violate their ESMs.

5.3 Solution 3: A Building Block for Mixed Initiatives

State 6 of the trace in Fig. 9 reveals an intrinsic peculiarity of the system: Due to the communication delay between the reception and the hotel room, both, an abort and a confirmation, can be in progress simultaneously. This is since during the alerting phase, both the receptionist and the hotel guest may take their initiative at nearly the same time. Although not always recognized, this situation occurs frequently in reactive systems, and has several names such as *conflicting* [27] or *mixed initiative* [28] as well as *non-local choice* [29]. As the problem is quite general, our library of building blocks contains a collaboration

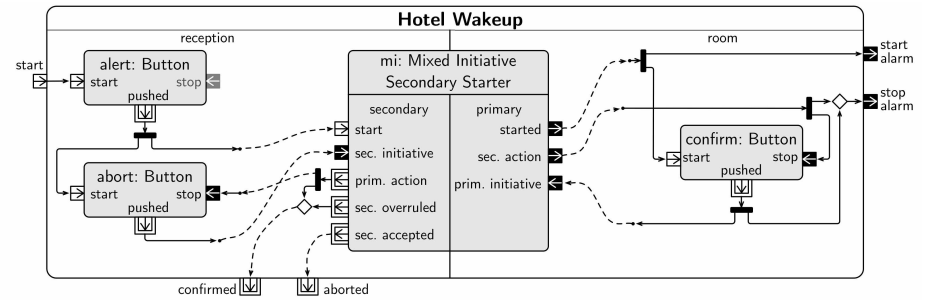

Fig. 10. Correct solution with a building block to handle mixed initiatives

to handle mixed initiatives. This collaboration has two participants, a *primary* and a *secondary* one. These names reflect which of the sides gets priority over the other if both sides take initiative contemporaneously. Two variants of the building block exist, one where the primary participant starts the collaboration, and one where the secondary one starts. In our system, we use the latter one and assign the primary role to the guest room, so that a confirmation from a hotel guest has priority over the abort from the reception. Fig. 10 shows the building block already embedded into the new solution while the ESM showing the detailed interleaving of its events is given in Fig. 11. For the sake of brevity, we look here just at the externals of the block, as an engineer would do when reusing it. The internals are similar to the building block *Tour Request* introduced in [1].

After the start of the collaboration via *start* on the secondary side, *started* notifies the primary side that the state is reached in which it may trigger an initiative. We couple this action with the start of the alarm. Input pin *prim. initiative*, denoting an initiative taken by the primary participant, is coupled with the pushing of the confirmation button. As the primary side has priority, we know that the confirmation will succeed, and can therefore stop the alarm right away. If the secondary side takes initiative (input pin *sec. initiative*), the primary side gets notified via *sec. action*, which is used to stop both the alarm as well as the confirmation button.

On the secondary side we have to take into account that an initiative from the abort button can be overruled by the confirmation of the guest room. Besides the nodes to start the collaboration and to take initiative, the secondary side has therefore three terminating output pins, from which only one will eventually release a token.

- Pin *primary action* releases a token if the primary side took initiative, and the secondary remained passive, i.e., only the guest confirmed. This leads to stop the abort button and to terminate via *confirmed*.
- Pin *sec. overruled* models that both initiatives have been taken, from which only the primary prevails. It is sensible to distinguish this case from the first one, as the reception in this case does not have to switch off the abort button, which already terminated because of its initiative.

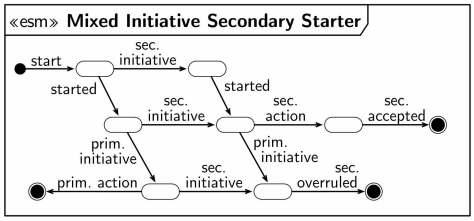

Fig. 11. ESM for Mixed Initiative Secondary Starter

– Pin *sec. accepted* emits a token if the secondary initiative was the only one, and the primary side did not start an initiative on its own, i.e., the alarm was aborted without the confirmation button being pressed.

When we translate this activity into TLA$^+$ and start TLC, we get the message that all properties are fulfilled now. Thus, the activity handles all the incorporated building blocks as prescribed by their respective ESMs. Moreover, it respects its own ESM and can be correctly used within the system described in Fig. 4. After checking the activity realizing the call behavior action *a* modeling alarms we know that the overall service specification is well-formed and can use it as input for the transformation steps producing executable code.

6 Concluding Remarks

We presented our service development approach SPACE that uses collaborations as building blocks. Their behavior is described by UML 2.0 activities which we can transform automatically into temporal formulas and a number of theorems expressing relevant properties to be fulfilled by an activity. The correctness of these theorems is model checked by TLC and its error messages lead to stepwise improvements of the models. The approach works both bottom/up and top/down. Sub-services may be arranged and their composition to larger services may be checked. Vice-versa, as done for the hotel wakeup, we may first assume a certain external behavior and then realize the internals of the service. Of course, many real systems are more extensive than the example used for the discussion here. The larger scale of these system results, however, mostly in a higher number of collaborations to be executed than in more complicated interactions. Thus, we will have a higher number of decomposition levels (see, for instance [5]), while the complexity of the models describing individual collaborations will remain of manageable size.

Once a collaboration between components in form of activities is model checked, it can be used in other systems without further proof efforts. This is feasible as the building blocks may be abstracted by their ESMs describing their external behavior. Thus, if we check an activity containing a sub-activity, we only have to consider the ESM of the sub-activity which hides the internal states, such that the state space of the model checked activity is reduced. In consequence, model checking is never done on the entire system with all its details, but it is enough to successively check activities on their decomposition level

separately. In this way, services and their compositions from sub-services may be verified in a compositional way which effectively rules out state explosions.

With the automatic formulation of the temporal formulas and theorems we created the base for user-friendly model checking of the service specifications based on UML activities. In future versions, we may offer more advanced feedback to the user that may explain error situations further and suggest typical improvements. This work will be performed as part of the research and development project *Infrastructure for Integrated Services* ISIS, funded by the Research Council of Norway, where we develop methods, tools and building blocks for services in the domain of home automation.

References

1. Kraemer, F.A., Bræk, R., Herrmann, P.: Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In Gaudin, E., Najm, E., Reed, R., eds.: SDL 2007. LNCS 4745. Springer–Verlag (2007) 166–185
2. Kraemer, F.A., Herrmann, P.: Service Specification by Composition of Collaborations — An Example. In: Proc. 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology). (2006) 129–133, Hong Kong.
3. Kraemer, F.A., Herrmann, P.: Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In Ehring, K., Giese, H., eds.: Proc. 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007). Volume 7 of Electronic Communications of the EASST. (2007)
4. Kraemer, F.A., Herrmann, P., Bræk, R.: Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In Meersmann, R., Tari, Z., eds.: Proc. 8th International Symposium on Distributed Objects and Applications (DOA), 2006, Montpellier, France. LNCS 4276, Springer–Verlag (2006) 1613–1632
5. Herrmann, P., Kraemer, F.A.: Design of Trusted Systems with Reusable Collaboration Models. In Etalle, S., Marsh, S., eds.: IFIP International Federation for Information Processing. Volume 238., IFIP, Springer (2007) 317–332
6. Herrmann, P., Krumm, H.: A Framework for Modeling Transfer Protocols. Computer Networks **34**(2) (2000) 317–337
7. Kraemer, F.A., Herrmann, P.: Formalizing Collaboration-Oriented Service Specifications using Temporal Logic. In: Networking and Electronic Commerce Research Conference 2007 (NAEC). (2007) (to appear).
8. Back, R.J.R., Kurki-Suonio, R.: Decentralization of Process Nets with Centralized Control. Distributed Computing **3** (1989) 73–87
9. Rushby, J.: Disappearing Formal Methods. In: High-Assurance Systems Engineering Symposium, Albuquerque, ACM (2000) 95–96
10. Slåtten, V.: Model Checking Collaborative Service Specifications in TLA with TLC. Project Thesis (2007) Norwegian University of Science and Technology.
11. Lamport, L.: The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems **16**(3) (1994) 872–923
12. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA$^+$ Specifications. In Pierre, L., Kropf, T., eds.: Proc. 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99). LNCS 1703, Springer-Verlag (1999) 54–66

13. Hooman, J.: Towards Formal Support for UML-based Developement of Embedded Systems. In: Proceedings PROGRESS 2002 Workshop, STW. (2002)
14. Burmester, S., Giese, H., Hirsch, M., Schilling, D.: Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In: Proc. of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004. (2004) 1–20
15. Balser, M., Bäumler, S., Knapp, A., Reif, W., Thums, A.: Interactive Verification of UML State Machines. In Davies, J., Schulte, W., Barnett, M., eds.: Proc. International Conference on Formal Engineering Methods. LNCS 3308, Springer–Verlag (2004) 434–448
16. Guelfi, N., Mammar, A.: A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In: APSEC '05: Proc. 12th Asia-Pacific Software Engineering Conference (APSEC'05), Washington, DC, USA, IEEE Computer Society (2005) 283–290
17. Holzmann, G.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading, Massachusetts (2003)
18. Störrle, H.: Semantics and Verification of Data Flow in UML 2.0 Activities. In: ENTCS 127. Elsevier. (2005) 35 – 52
19. Dong, Y., Shensheng, Z.: Using π-Calculus to Formalize UML Activity Diagram for Business Process Modeling. In: Proceedings 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, Huntsville, AL, USA (2003) 47 – 54
20. Victor, B., Moller, F.: The Mobility Workbench — A Tool for the π-Calculus. In Dill, D., ed.: CAV'94: Computer Aided Verification. LNCS 818, Springer–Verlag (1994) 428–440
21. Eshuis, R.: Symbolic Model Checking of UML Activity Diagrams. ACM Transactions on Software Engineering and Methodology **15**(1) (2006) 1–38
22. Lamport, L.: Specifying Systems. Addison-Wesley (2002)
23. Abadi, M., Lamport, L.: Conjoining Specifications. ACM Transactions on Programming Languages and Systems **17**(3) (1995) 507–535
24. Vissers, C.A., Scollo, G., van Sinderen, M., Brinksma, H.: Specification Styles in Distributed System Design and Verification. Theoretical Computer Science **89** (1991) 179–206
25. Kraemer, F.A.: UML Profile and Semantics for Service Specifications. Avantel Technical Report 1/2007 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway (2007)
26. Kraemer, F.A.: Building Blocks, Patterns and Design Rules for Collaborations and Activities. Avantel Technical Report 2/2007 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway (2007)
27. Bræk, R., Haugen, Ø.: Engineering Real Time Systems: An Object-Oriented Methodology Using SDL. The BCS Practitioner Series. Prentice Hall (1993)
28. Floch, J.: Towards Plug-and-Play Services: Design and Validation using Roles. PhD thesis, Norwegian University of Science and Technology (2003)
29. Ben-Abdallah, H., Leue, S.: Syntactic Detection of Process Divergence and Non-Local Choice in Message Sequence Charts. In: Proc. of the 2nd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97). (1997)